# MSP430 Optimizing C/C++ Compiler v 3.3

# User's Guide


Literature Number: SLAU132E
July 2010

# Contents

# List of Figures



# List of Tables

# Read This First

## About This Manual

The *MSP430 Optimizing C/C++ Compiler User's Guide* explains how to use these compiler tools:

- Compiler
- Library-build process
- C++ name demangler

The compiler accepts C and C++ code conforming to the International Organization for Standardization (ISO) standards for these languages. The compiler supports the 1989 version of the C language and the 1998 version of the C++ language.

This user's guide discusses the characteristics of the C/C++ compiler. It assumes that you already know how to write C programs. The C Programming Language (second edition), by Brian W. Kernighan and Dennis M. Ritchie, describes C based on the ISO C standard. You can use the Kernighan and Ritchie (hereafter referred to as K&R) book as a supplement to this manual. References to K&R C (as opposed to ISO C) in this manual refer to the C language as defined in the first edition of Kernighan and Ritchie's The C Programming Language.

## Notational Conventions

This document uses the following conventions:

- Program listings, program examples, and interactive displays are shown in a `special typeface`. Interactive displays use a bold version of the special typeface to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).

  Here is a sample of C code:

```
#include <stdio.h>
main()
{    printf("hello, cruel world\n");
}
```

- In syntax descriptions, the instruction, command, or directive is in a **bold typeface** and parameters are in an *italic typeface*. Portions of a syntax that are in bold should be entered as shown; portions of a syntax that are in italics describe the type of information that should be entered.
- Square brackets ( [ and ] ) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets. Unless the square brackets are in the **bold typeface**, do not enter the brackets themselves. The following is an example of a command that has an optional parameter:

**cl430** [*options*] [*filenames*] [**--run_linker** [*link_options*] [*object files*]]

- Braces ( { and } ) indicate that you must choose one of the parameters within the braces; you do not enter the braces themselves. This is an example of a command with braces that are not included in the actual syntax but indicate that you must specify either the --rom_model or --ram_model option:

**cl430 --run_linker** {**--rom_model | --ram_model**} *filenames* [**--output_file=** *name.out*]
**--library=** *libraryname*

- In assembler syntax statements, column 1 is reserved for the first character of a label or symbol. If the label or symbol is optional, it is usually not shown. If it is a required parameter, it is shown starting against the left margin of the box, as in the example below. No instruction, command, directive, or parameter, other than a symbol or label, can begin in column 1.

| *symbol* **.usect** "*section name*", *size in bytes*[, *alignment*] |
|---|

- Some directives can have a varying number of parameters. For example, the .byte directive. This syntax is shown as [, ..., *parameter*].

## Related Documentation

You can use the following books to supplement this user's guide:

**ANSI X3.159-1989, Programming Language - C (Alternate version of the 1989 C Standard)**, American National Standards Institute

**C: A Reference Manual (fourth edition)**, by Samuel P. Harbison, and Guy L. Steele Jr., published by Prentice Hall, Englewood Cliffs, New Jersey

**ISO/IEC 9899:1989, International Standard - Programming Languages - C (The 1989 C Standard)**, International Organization for Standardization

**ISO/IEC 9899:1999, International Standard - Programming Languages - C (The C Standard)**, International Organization for Standardization

**ISO/IEC 14882-1998, International Standard - Programming Languages - C++ (The C++ Standard)**, International Organization for Standardization

**Programming Embedded Systems in C and C++**, by Michael Barr, Andy Oram (Editor), published by O'Reilly & Associates; ISBN: 1565923545, February 1999

**Programming in C**, Steve G. Kochan, Hayden Book Company

**The C Programming Language (second edition)**, by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice-Hall, Englewood Cliffs, New Jersey, 1988

**The C++ Programming Language (second edition)**, Bjarne Stroustrup, published by Addison-Wesley Publishing Company, Reading, Massachusetts, 1990

**The Annotated C++ Reference Manual**, Margaret A. Ellis and Bjarne Stroustrup, published by Addison-Wesley Publishing Company, Reading, Massachusetts, 1990

**Tool Interface Standards (TIS) DWARF Debugging Information Format Specification Version 2.0**, TIS Committee, 1995

**DWARF Debugging Information Format Version 3**, DWARF Debugging Information Format Workgroup, Free Standards Group, 2005 (http://dwarfstd.org)

## Related Documentation From Texas Instruments

You can use the following books to supplement this user's guide:

**SLAU012**— ***MSP430x3xx Family User's Guide***. Describes the MSP430x3xx™ CPU architecture, instruction set, pipeline, and interrupts for these ultra-low power microcontrollers.

**SLAU049**— ***MSP430x1xx Family User's Guide***. Describes the MSP430x1xx™ CPU architecture, instruction set, pipeline, and interrupts for these ultra-low power microcontrollers.

**SLAU056**— ***MSP430x4xx Family User's Guide***. Describes the MSP430x4xx™ CPU architecture, instruction set, pipeline, and interrupts for these ultra-low power microcontrollers.

**SLAU131**— ***MSP430 Assembly Language Tools User's Guide***. Describes the assembly language tools (the assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, object file format, and symbolic debugging directives for the MSP430 devices.

**SLAU134**— ***MSP430FE42x ESP30CE1 Peripheral Module User's Guide***. Describes common peripherals available on the MSP430FE42x and ESP430CE1 ultra-low power microcontrollers. This book includes information on the setup, operation, and registers of the ESP430CE1.

**SLAU144**— ***MSP430x2xx Family User's Guide***. Describes the MSP430x2xx™ CPU architecture, instruction set, pipeline, and interrupts for these ultra-low power microcontrollers.

**SLAU208**— ***MSP430x5xx Family User's Guide***. Describes the MSP430x5xx™ CPU architecture, instruction set, pipeline, and interrupts for these ultra-low power microcontrollers.

**SPRAAB5**— ***The Impact of DWARF on TI Object Files.*** Describes the Texas Instruments extensions to the DWARF specification.

# Introduction to the Software Development Tools

The MSP430 is supported by a set of software development tools, which includes an optimizing C/C++ compiler, an assembler, a linker, and assorted utilities.

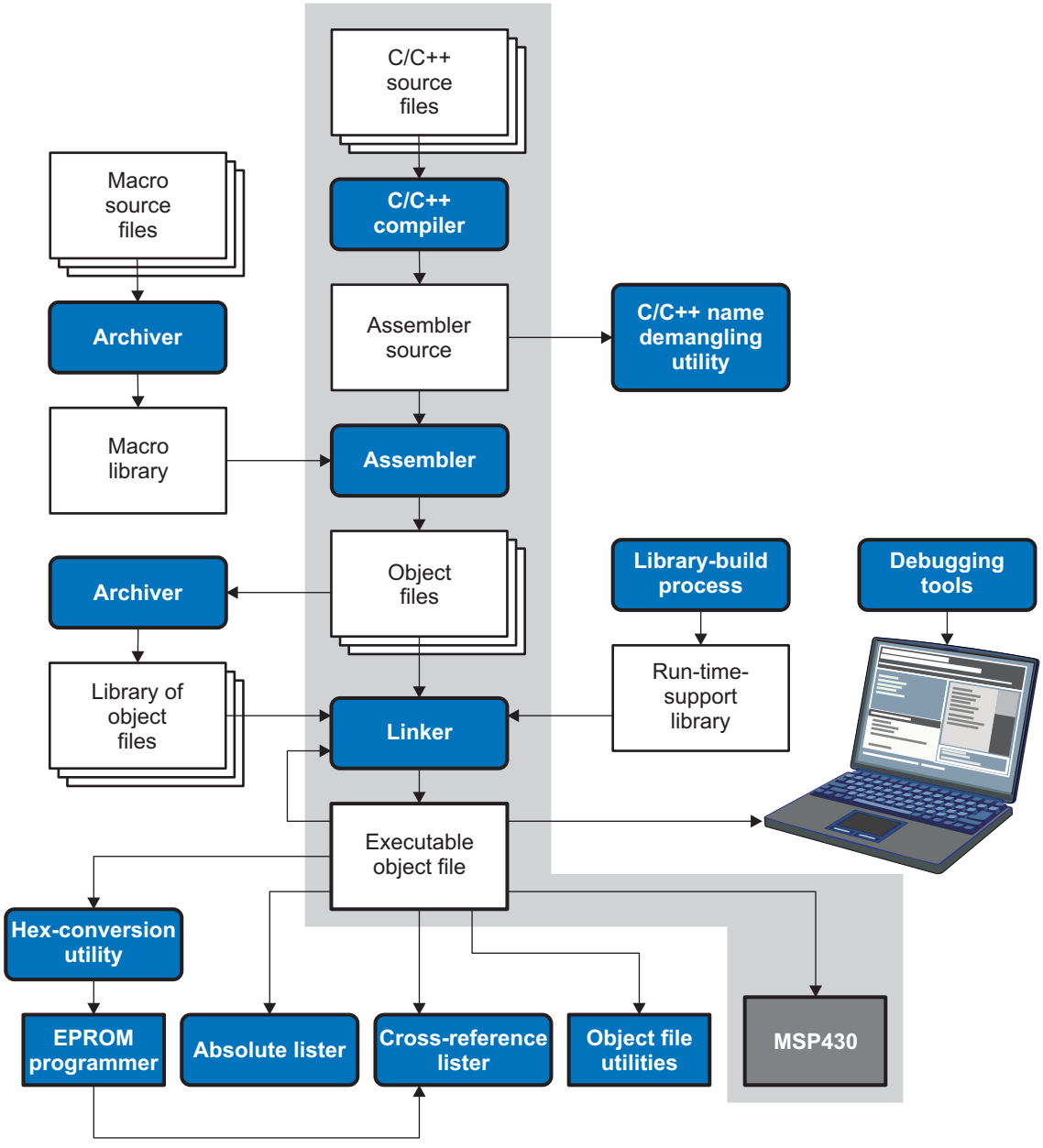
This chapter provides an overview of these tools and introduces the features of the optimizing C/C++ compiler. The assembler and linker are discussed in detail in the *MSP430 Assembly Language Tools User's Guide*.

**Topic** **Page**

## 1.1 Software Development Tools Overview

Figure 1-1 illustrates the software development flow. The shaded portion of the figure highlights the most common path of software development for C language programs. The other portions are peripheral functions that enhance the development process.

**Figure 1-1. MSP430 Software Development Flow**



The following list describes the tools that are shown in Figure 1-1:

- The **compiler** accepts C/C++ source code and produces MSP430 assembly language source code. See Chapter 2.
- The **assembler** translates assembly language source files into machine language object modules. The *MSP430 Assembly Language Tools User's Guide* explains how to use the assembler.
- The **linker** combines object files into a single executable object module. As it creates the executable module, it performs relocation and resolves external references. The linker accepts relocatable object files and object libraries as input. See Chapter 4. The *MSP430 Assembly Language Tools User's Guide* provides a complete description of the linker.
- The **archiver** allows you to collect a group of files into a single archive file, called a *library*. Additionally, the archiver allows you to modify a library by deleting, replacing, extracting, or adding members. One of the most useful applications of the archiver is building a library of object modules. The *MSP430 Assembly Language Tools User's Guide* explains how to use the archiver.
- You can use the **library-build process** to build your own customized run-time-support library. See Section 7.4. Standard run-time-support library functions for C and C++ are provided in the self-contained rtssrc.zip file.

  The **run-time-support libraries** contain the standard ISO run-time-support functions, compiler-utility functions, floating-point arithmetic functions, and C I/O functions that are supported by the compiler. See Chapter 7.
- The **hex conversion utility** converts an object file into other object formats. You can download the converted file to an EPROM programmer. The *MSP430 Assembly Language Tools User's Guide* explains how to use the hex conversion utility and describes all supported formats.
- The **absolute lister** accepts linked object files as input and creates .abs files as output. You can assemble these .abs files to produce a listing that contains absolute, rather than relative, addresses. Without the absolute lister, producing such a listing would be tedious and would require many manual operations. The *MSP430 Assembly Language Tools User's Guide* explains how to use the absolute lister.
- The **cross-reference lister** uses object files to produce a cross-reference listing showing symbols, their definitions, and their references in the linked source files. The *MSP430 Assembly Language Tools User's Guide* explains how to use the cross-reference utility.
- The **C++ name demangler** is a debugging aid that converts names mangled by the compiler back to their original names as declared in the C++ source code. As shown in Figure 1-1, you can use the C++ name demangler on the assembly file that is output by the compiler; you can also use this utility on the assembler listing file and the linker map file. See Chapter 8.
- The **disassembler** disassembles object files. The *MSP430 Assembly Language Tools User's Guide* explains how to use the disassembler.
- The main product of this development process is a module that can be executed in a **MSP430** device.

## 1.2 C/C++ Compiler Overview

The following subsections describe the key features of the compiler.

### 1.2.1 ANSI/ISO Standard

These features pertain to ISO standards:

- **ISO-standard C**

  The C/C++ compiler conforms to the ISO C standard as defined by the ISO specification and described in the second edition of Kernighan and Ritchie's *The C Programming Language* (K&R). The ISO C standard supercedes and is the same as the ANSI C standard.
- **ISO-standard C++**

  The C/C++ compiler supports C++ as defined by the ISO C++ Standard and described in Ellis and Stroustrup's *The Annotated C++ Reference Manual* (ARM). The compiler also supports embedded C++. For a description of *unsupported* C++ features, see Section 5.2.
- **ISO-standard run-time support**

  The compiler tools come with an extensive run-time library. All library functions conform to the ISO C/C++ library standard. The library includes functions for standard input and output, string manipulation, dynamic memory allocation, data conversion, timekeeping, trigonometry, and exponential and hyperbolic functions. Functions for signal handling are not included, because these are target-system specific. For more information, see Chapter 7.

### 1.2.2 Output Files

These features pertain to output files created by the compiler:

- **COFF object files**

  Common object file format (COFF) allows you to define your system's memory map at link time. This maximizes performance by enabling you to link C/C++ code and data objects into specific memory areas. COFF also supports source-level debugging.

### 1.2.3 Compiler Interface

These features pertain to interfacing with the compiler:

- **Compiler program**

  The compiler tools include a compiler program that you use to compile, optimize, assemble, and link programs in a single step. For more information, see Section 2.1
- **Flexible assembly language interface**

  The compiler has straightforward calling conventions, so you can write assembly and C functions that call each other. For more information, see Chapter 6.

### 1.2.4 Utilities

These features pertain to the compiler utilities:

- **Library-build process**

  The library-build process lets you custom-build object libraries from source for any combination of run-time models. For more information, see Section 7.4.
- **C++ name demangler**

  The C++ name demangler (dem430) is a debugging aid that translates each mangled name it detects to its original name found in the C++ source code. For more information, see Chapter 8.
- **Hex conversion utility**

  For stand-alone embedded applications, the compiler has the ability to place all code and initialization data into ROM, allowing C/C++ code to run from reset. The COFF files output by the compiler can be converted to EPROM programmer data files by using the hex conversion utility, as described in the *MSP430 Assembly Language Tools User's Guide*.

# Using the C/C++ Compiler

The compiler translates your source program into machine language object code that the MSP430™ can execute. Source code must be compiled, assembled, and linked to create an executable object file. All of these steps are executed at once by using the compiler.

**Topic** **Page**

## 2.1 About the Compiler

The compiler lets you compile, assemble, and optionally link in one step. The compiler performs the following steps on one or more source modules:

- The **compiler** accepts C/C++ source code and assembly code, and produces object code.
  You can compile C, C++, and assembly files in a single command. The compiler uses the filename extensions to distinguish between different file types. See Section 2.3.8 for more information.
- The **linker** combines object files to create an executable object file. The linker is optional, so you can compile and assemble many modules independently and link them later. See Chapter 4 for information about linking the files.

By default, the compiler does not invoke the linker. You can invoke the linker by using the --run_linker compiler option.

For a complete description of the assembler and the linker, see the *MSP430 Assembly Language Tools User's Guide*.

## 2.2 Invoking the C/C++ Compiler

To invoke the compiler, enter:

**cl430** [*options*] [*filenames*] [**--run_linker** [*link_options*] *object files*]]

| | |
|---|---|
| **cl430** | Command that runs the compiler and the assembler. |
| *options* | Options that affect the way the compiler processes input files. The options are listed in Table 2-2 through Table 2-26. |
| *filenames* | One or more C/C++ source files, assembly language source files, linear assembly files, or object files. |
| **--run_linker** | Option that invokes the linker. The --run_linker option's short form is -z. See Chapter 4 for more information. |
| *link_options* | Options that control the linking process. |
| *object files* | Name of the additional object files for the linking process. |

The arguments to the compiler are of three types:

- Compiler options
- Link options
- Filenames

The --run_linker option indicates linking is to be performed. If the --run_linker option is used, any compiler options must precede the --run_linker option, and all link options must follow the --run_linker option.

Source code filenames must be placed before the --run_linker option. Additional object file filenames can be placed after the --run_linker option.

For example, if you want to compile two files named symtab.c and file.c, assemble a third file named seek.asm, and link to create an executable program called myprogram.out, you will enter:

```
cl430 symtab.c file.c seek.asm --run_linker --library=lnk.cmd
      --library=rts430.lib --output_file=myprogram.out
```

## 2.3 Changing the Compiler's Behavior With Options

Options control the operation of the compiler. This section provides a description of option conventions and an option summary table. It also provides detailed descriptions of the most frequently used options, including options used for type-checking and assembling.

For an online summary of the options, enter **cl430** with no parameters on the command line.

The following apply to the compiler options:

- Options are preceded by one or two hyphens.
- Options are case sensitive.
- Options are either single letters or sequences of characters.
- Individual options cannot be combined.
- An option with a *required* parameter should be specified with an equal sign before the parameter to clearly associate the parameter with the option. For example, the option to undefine a constant can be expressed as --undefine=*name*. Although not recommended, you can separate the option and the parameter with or without a space, as in --undefine *name* or -undefine*name*.
- An option with an *optional* parameter should be specified with an equal sign before the parameter to clearly associate the parameter with the option. For example, the option to specify the maximum amount of optimization can be expressed as -O=3. Although not recommended, you can specify the parameter directly after the option, as in -O3. No space is allowed between the option and the optional parameter, so -O 3 is not accepted.
- Files and options except the --run_linker option can occur in any order. The --run_linker option must follow all other compile options and precede any link options.

You can define default options for the compiler by using the MSP430_C_DIR environment variable. For a detailed description of the environment variable, see Section 2.4.1.

Table 2-2 through Table 2-26 summarize all options (including link options). Use the references in the tables for more complete descriptions of the options.

**Table 2-1. Basic Options**

| Option | Alias | Effect | Section |
|---|---|---|---|
| −−silicon_version={msp\|mspx} | -v | Selects the instruction set | Section 2.3.3 |
| --code_model={large\|small} | | Specifies the code memory model | Section 2.3.3 |
| --data_model={restricted\|large\|small} | | Specifies the data memory model | Section 2.3.3 |
| --symdebug:dwarf | -g | Enables symbolic debugging | Section 2.3.4<br>Section 3.9 |
| --symdebug:none | | Disables all symbolic debugging | Section 2.3.4 |
| --symdebug:skeletal | | Enables minimal symbolic debugging that does not hinder optimizations (default behavior) | |
| --opt_level[=0-4] | -O | Optimization level (Default:2) | Section 3.1 |
| --opt_for_speed=*n* | -mf | Optimizes for speed over space (0-5 range) (Default is 4.) | Section 3.10 |

**Table 2-2. Control Options**

| Option | Alias | Effect | Section |
|---|---|---|---|
| --compile_only | -c | Disables linking (negates --run_linker) | Section 4.1.3 |
| --help | -h | Prints (on the standard output device) a description of the options understood by the compiler. | Section 2.3.1 |
| --run_linker | -z | Enables linking | Section 2.3.1 |
| --skip_assembler | -n | Compiles or assembly optimizes only | Section 2.3.1 |

**Table 2-3. Symbolic Debug Options**

| Option | Alias | Effect | Section |
|---|---|---|---|
| --symdebug:dwarf | -g | Enables symbolic debugging | Section 2.3.4<br>Section 3.9 |
| --symdebug:none | | Disables all symbolic debugging | Section 2.3.4 |
| --symdebug:skeletal | | Enables minimal symbolic debugging that does not hinder optimizations (default behavior) | Section 2.3.4 |

**Table 2-4. Language Options**

| Option | Alias | Effect | Section |
|---|---|---|---|
| --cpp_default | -fg | Processes all source files with a C extension as C++ source files. | Section 2.3.6 |
| --embedded_cpp | -pe | Enables embedded C++ mode | Section 5.13.3 |
| --exceptions | | Enables C++ exception handling | Section 5.6 |
| --extern_c_can_throw | | Allow extern C functions to propagate exceptions | |
| --fp_mode={relaxed\|strict} | | Enables or disables relaxed floating-point mode | Section 2.3.2 |
| --gcc | | Enables support for GCC extensions | Section 5.14 |
| --gen_asp_raw | -pl | Generates a raw listing file | Section 2.10 |
| --gen_acp_xref | -px | Generates a cross-reference listing file | Section 2.9 |
| --keep_unneeded_statics | | Keeps unreferenced static variables. | Section 2.3.2 |
| --kr_compatible | -pk | Allows K&R compatibility | Section 5.13.1 |
| --multibyte_chars | -pc | Enables multibyte character support. | - |
| --no_inlining | -pi | Disables definition-controlled inlining (but --opt_level=3 (or -O3) optimizations still perform automatic inlining) | Section 2.11 |
| --no_intrinsics | -pn | Disables intrinsic functions. No predefinition of compiler-supplied intrinsic functions. | - |
| --program_level_compile | -pm | Combines source files to perform program-level optimization | Section 3.3 |
| --relaxed_ansi | -pr | Enables relaxed mode; ignores strict ISO violations | Section 5.13.2 |
| --rtti | -rtti | Enables run time type information (RTTI) | – |
| --static_template_instantiation | | Instantiate all template entities with internal linkage | – |
| --strict_ansi | -ps | Enables strict ISO mode (for C/C++, not K&R C) | Section 5.13.2 |
| --check_misra={all\|required\|advisory\|none\|*rulespec*} | | Enables checking of the specified MISRA-C:2004 rules | Section 2.3.2 |
| --misra_advisory={error\|warning\|remark\|suppress} | | Sets the diagnostic severity for advisory MISRA-C:2004 rules | Section 2.3.2 |
| --misra_required={error\|warning\|remark\|suppress} | | Sets the diagnostic severity for required MISRA-C:2004 rules | Section 2.3.2 |

### Table 2-5. Parser Preprocessing Options

| Option | Alias | Effect | Section |
|---|---|---|---|
| --preproc_dependency[=*filename*] | -ppd | Performs preprocessing only, but instead of writing preprocessed output, writes a list of dependency lines suitable for input to a standard make utility | Section 2.6.7 |
| --preproc_includes[=*filename*] | -ppi | Performs preprocessing only, but instead of writing preprocessed output, writes a list of files included with the #include directive | Section 2.6.8 |
| --preproc_macros[=*filename*] | -ppm | Performs preprocessing only. Writes list of predefined and user-defined macros to a file with the same name as the input but with a .pp extension. | Section 2.6.9 |
| --preproc_only | -ppo | Performs preprocessing only. Writes preprocessed output to a file with the same name as the input but with a .pp extension. | Section 2.6.3 |
| --preproc_with_comment | -ppc | Performs preprocessing only. Writes preprocessed output, keeping the comments, to a file with the same name as the input but with a .pp extension. | Section 2.6.5 |
| --preproc_with_compile | -ppa | Continues compilation after preprocessing | Section 2.6.4 |
| --preproc_with_line | -ppl | Performs preprocessing only. Writes preprocessed output with line-control information (#line directives) to a file with the same name as the input but with a .pp extension. | Section 2.6.6 |

### Table 2-6. Predefined Symbols Options

| Option | Alias | Effect | Section |
|---|---|---|---|
| --define=*name*[=*def*] | -D | Predefines *name* | Section 2.3.1 |
| --undefine=*name* | -U | Undefines *name* | Section 2.3.1 |

### Table 2-7. Include Options

| Option | Alias | Effect | Section |
|---|---|---|---|
| --include_path=*directory* | `-I` | Defines #include search path | Section 2.6.2.1 |
| --preinclude=*filename* | | Includes *filename* at the beginning of compilation | Section 2.3.2 |

### Table 2-8. Diagnostics Options

| Option | Alias | Effect | Section |
|---|---|---|---|
| --compiler_revision | | Prints out the compiler release revision and exits | -- |
| --diag_error=*num* | -pdse | Categorizes the diagnostic identified by *num* as an error | Section 2.7.1 |
| --diag_remark=*num* | -pdsr | Categorizes the diagnostic identified by *num* as a remark | Section 2.7.1 |
| --diag_suppress=*num* | -pds | Suppresses the diagnostic identified by *num* | Section 2.7.1 |
| --diag_warning=*num* | -pdsw | Categorizes the diagnostic identified by *num* as a warning | Section 2.7.1 |
| --display_error_number=*num* | -pden | Displays a diagnostic's identifiers along with its text | Section 2.7.1 |
| --issue_remarks | -pdr | Issues remarks (nonserious warnings) | Section 2.7.1 |
| --no_warnings | -pdw | Suppresses warning diagnostics (errors are still issued) | Section 2.7.1 |
| --quiet | -q | Suppresses progress messages (quiet) | -- |
| --set_error_limit=*num* | -pdel | Sets the error limit to *num*. The compiler abandons compiling after this number of errors. (The default is 100.) | Section 2.7.1 |
| --super_quiet | -qq | Super quiet mode | -- |
| --tool_version | -version | Displays version number for each tool | -- |
| --verbose | | Display banner and function progress information | -- |
| --verbose_diagnostics | -pdv | Provides verbose diagnostics that display the original source with line-wrap | Section 2.7.1 |
| --write_diagnostics_file | -pdf | Generates a diagnostics information file. Compiler only option. | Section 2.7.1 |

**Table 2-9. Run-Time Model Options**

| Option | Alias | Effect | Section |
|---|---|---|---|
| --code_model={large\|small} | | Specifies the code memory model | Section 6.1.1 |
| --data_model={restricted\|large\|small} | | Specifies the data memory model | Section 6.1.2 |
| --fp_reassoc={on\|off} | | Enables or disables the reassociation of floating-point arithmetic | Section 2.3.3 |
| --gen_func_subsections | | Puts each function in a separate subsection in the object file | Section 4.2.2 |
| --large_memory_model | -ml | Uses a large memory model when compiling for the MSP430X. (Deprecated) | Section 2.3.3 |
| --near_data={globals\|none} | | Specifies location of global read/write data | Section 2.3.3 |
| --optimize_with_debug | | Reenables optimizations disabled --symdebug:dwarf | Section 3.9 |
| --plain_char={signed\|unsigned} | -mc | Changes variables of type char from unsigned to signed | Section 2.3.3 |
| --printf_support={full\|minimal\|nofloat} | | Enables support for smaller limited versions of printf. | Section 2.3.3 |
| --sat_reassoc={on\|off} | | Enables or disables the reassociation of saturating arithmetic | |
| --silicon_version={msp\|mspx} | -v | Selects the instruction set | Section 2.3.3 |
| --small_enum | | Uses the smallest possible size for the enumeration type | Section 2.3.3 |

**Table 2-10. Optimization Options[(1)]**

| Option | Alias | Effect | Section |
|---|---|---|---|
| --opt_level=0 | -O0 | Optimizes register usage | Section 3.1 |
| --opt_level=1 | -O1 | Uses -O0 optimizations and optimizes locally | Section 3.1 |
| --opt_level=2 | -O2 or -O | Uses -O1 optimizations and optimizes globally (default) | Section 3.1 |
| --opt_level=3 | -O3 | Uses -O2 optimizations and optimizes the file | Section 3.1<br>Section 3.2 |
| --opt_level=4 | -O4 | Invokes link-time optimization | Section 3.4 |
| --auto_inline=[*size*] | -oi | Sets automatic inlining size (--opt_level=3 only). If *size* is not specified, the default is 1. | Section 3.7 |
| --call_assumptions=0 | -op0 | Specifies that the module contains functions and variables that are called or modified from outside the source code provided to the compiler | Section 3.3.1 |
| --call_assumptions=1 | -op1 | Specifies that the module contains variables modified from outside the source code provided to the compiler but does not use functions called from outside the source code | Section 3.3.1 |
| --call_assumptions=2 | -op2 | Specifies that the module contains no functions or variables that are called or modified from outside the source code provided to the compiler (default) | Section 3.3.1 |
| --call_assumptions=3 | -op3 | Specifies that the module contains functions that are called from outside the source code provided to the compiler but does not use variables modified from outside the source code | Section 3.3.1 |
| --gen_opt_info=0 | -on0 | Disables the optimization information file | Section 3.2.2 |
| --gen_opt_info=1 | -on1 | Produces an optimization information file | Section 3.2.2 |
| --gen_opt_info=2 | -on2 | Produces a verbose optimization information file | Section 3.2.2 |
| --opt_for_speed=*n* | -mf | Optimizes for speed over space (0-5 range) | Section 3.10 |
| --optimizer_interlist | -os | Interlists optimizer comments with assembly statements | Section 3.8 |
| --remove_hooks_when_inlining | | Removes entry/exit hooks for auto-inlined functions | Section 2.13 |
| --single_inline | | Inlines functions that are only called once | |
| --std_lib_func_defined | -ol1 or -oL1 | Informs the optimizer that your file declares a standard library function | Section 3.2.1 |
| --std_lib_func_not_defined | -ol2 or -oL2 | Informs the optimizer that your file does not declare or alter library functions. Overrides the -ol0 and -ol1 options (default). | Section 3.2.1 |
| --std_lib_func_redefined | -ol0 or -oL0 | Informs the optimizer that your file alters a standard library function | Section 3.2.1 |
| --aliased_variables | -ma | Assumes variables are aliased | Section 3.5 |

(1) **Note:** Machine-specific options (see Table 2-9) can also affect optimization.

### Table 2-11. Entry/Exit Hook Options

| Option | Alias | Effect | Section |
|---|---|---|---|
| --entry_hook[=*name*] | | Enables entry hooks | Section 2.13 |
| --entry_parm={name\|address\|none} | | Specifies the parameters to the function to the --entry_hook option | Section 2.13 |
| --exit_hook[=*name*] | | Enables exit hooks | Section 2.13 |
| --exit_parm={name\|address\|none} | | Specifies the parameters to the function to the --exit_hook option | Section 2.13 |

### Table 2-12. Library Function Assumptions Options

| Option | Alias | Effect | Section |
|---|---|---|---|
| --std_lib_func_defined | -ol1 or -oL1 | Informs the optimizer that your file declares a standard library function | Section 3.2.1 |
| --std_lib_func_not_defined | -ol2 or -oL2 | Informs the optimizer that your file does not declare or alter library functions. Overrides the -ol0 and -ol1 options (default). | Section 3.2.1 |
| --std_lib_func_redefined | -ol0 or -oL0 | Informs the optimizer that your file alters a standard library function | Section 3.2.1 |
| --printf_support={full\|nofloat\|minimal} | | Enables support for smaller, limited versions of the printf and sprintf run-time-support functions. | |

### Table 2-13. Assembler Options

| Option | Alias | Effect | Section |
|---|---|---|---|
| --keep_asm | -k | Keeps the assembly language (.asm) file | Section 2.3.10 |
| --asm_listing | -al | Generates an assembly listing file | Section 2.3.10 |
| --c_src_interlist | -ss | Interlists C source and assembly statements | Section 2.12<br>Section 3.8 |
| --src_interlist | -s | Interlists optimizer comments (if available) and assembly source statements; otherwise interlists C and assembly source statements | |
| --absolute_listing | -aa | Enables absolute listing | Section 2.3.10 |
| --asm_define=*name*[=*def*] | -ad | Sets the *name* symbol | Section 2.3.10 |
| --asm_dependency | -apd | Performs preprocessing; lists only assembly dependencies | Section 2.3.10 |
| --asm_includes | -api | Performs preprocessing; lists only included #include files | Section 2.3.10 |
| --asm_undefine=*name* | -au | Undefines the predefined constant *name* | Section 2.3.10 |
| --copy_file=*filename* | -ahc | Copies the specified file for the assembly module | Section 2.3.10 |
| --cross_reference | -ax | Generates the cross-reference file | Section 2.3.10 |
| --include_file=*filename* | -ahi | Includes the specified file for the assembly module | Section 2.3.10 |
| --no_const_clink | | Stops generation of .clink directives for const global arrays. | |
| --output_all_syms | -as | Puts labels in the symbol table | Section 2.3.10 |
| --syms_ignore_case | -ac | Makes case insignificant in assembly source files | Section 2.3.10 |

### Table 2-14. File Type Specifier Options

| Option | Alias | Effect | Section |
|---|---|---|---|
| --asm_file=*filename* | -fa | Identifies *filename* as an assembly source file regardless of its extension. By default, the compiler and assembler treat .asm files as assembly source files. | Section 2.3.6 |
| --c_file=*filename* | -fc | Identifies *filename* as a C source file regardless of its extension. By default, the compiler treats .c files as C source files. | Section 2.3.6 |
| --cpp_file=*filename* | -fp | Identifies *filename* as a C++ file, regardless of its extension. By default, the compiler treats .C, .cpp, .cc and .cxx files as a C++ files. | Section 2.3.6 |
| --obj_file=*filename* | -fo | Identifies *filename* as an object code file regardless of its extension. By default, the compiler and linker treat .obj files as object code files. | Section 2.3.6 |

### Table 2-15. Directory Specifier Options

| Option | Alias | Effect | Section |
|---|---|---|---|
| --abs_directory=*directory* | -fb | Specifies an absolute listing file directory | Section 2.3.9 |
| --asm_directory=*directory* | -fs | Specifies an assembly file directory | Section 2.3.9 |
| --list_directory=*directory* | -ff | Specifies an assembly listing file and cross-reference listing file directory | Section 2.3.9 |
| --obj_directory=*directory* | -fr | Specifies an object file directory | Section 2.3.9 |
| --temp_directory=*directory* | -ft | Specifies a temporary file directory | Section 2.3.9 |

### Table 2-16. Default File Extensions Options

| Option | Alias | Effect | Section |
|---|---|---|---|
| --asm_extension=[.]*extension* | -ea | Sets a default extension for assembly source files | Section 2.3.8 |
| --c_extension=[.]*extension* | -ec | Sets a default extension for C source files | Section 2.3.8 |
| --cpp_extension=[.]*extension* | -ep | Sets a default extension for C++ source files | Section 2.3.8 |
| --listing_extension=[.]*extension* | -es | Sets a default extension for listing files | Section 2.3.8 |
| --obj_extension=[.]*extension* | -eo | Sets a default extension for object files | Section 2.3.8 |

### Table 2-17. Command Files Options

| Option | Alias | Effect | Section |
|---|---|---|---|
| --cmd_file=*filename* | -@ | Interprets contents of a file as an extension to the command line. Multiple -@ instances can be used. | Section 2.3.1 |

### Table 2-18. Precompiled Header Options

| Option | Alias | Effect | Section |
|---|---|---|---|
| --create_pch=*filename* | | Creates a precompiled header file with the name specified | Section 2.5 |
| --pch | | Creates or uses precompiled header files | Section 2.5 |
| --pch_dir=*directory* | | Specifies the path where the precompiled header file resides | Section 2.5.2 |
| --pch_verbose | | Displays a message for each precompiled header file that is considered but not used | Section 2.5.3 |
| --use_pch=*filename* | | Specifies the precompiled header file to use for this compilation | Section 2.5.2 |

The following tables list the linker options. See the *MSP430 Assembly Language Tools User's Guide* for details on these options.

**Table 2-19. Linker Basic Options Summary**

| Option | Alias | Description |
|---|---|---|
| | | **Basic Options** |
| --output_file=*file* | -o | Names the executable output module. The default filename is a.out. |
| --map_file=*file* | -m | Produces a map or listing of the input and output sections, including holes, and places the listing in *filename* |
| --heap_size=*size* | -heap | Sets heap size (for the dynamic memory allocation in C) to *size* bytes and defines a global symbol that specifies the heap size. Default = 128 bytes |
| --stack_size=*size* | -stack | Sets C system stack size to *size* bytes and defines a global symbol that specifies the stack size. Default = 128 bytes |
| --use_hw_mpy[={16\|32\|F5}] | | Replaces all references to the default integer/long multiply routine with the version of the multiply routine that uses the hardware multiplier support. |

**Table 2-20. Command File Preprocessing Options Summary**

| Option | Alias | Description |
|---|---|---|
| --define | | Predefines *name* as a preprocessor macro. |
| --undefine | | Removes the preprocessor macro *name*. |
| --disable_pp | | Disables preprocessing for command files |

**Table 2-21. Diagnostic Options Summary**

| Option | Alias | Description |
|---|---|---|
| --diag_error | | Categorizes the diagnostic identified by *num* as an error |
| --diag_remark | | Categorizes the diagnostic identified by *num* as a remark |
| --diag_suppress | | Suppresses the diagnostic identified by *num* |
| --diag_warning | | Categorizes the diagnostic identified by *num* as a warning |
| --display_error_number | | Displays a diagnostic's identifiers along with its text |
| --issue_remarks | | Issues remarks (nonserious warnings) |
| --no_demangle | | Disables demangling of symbol names in diagnostics |
| --no_warnings | | Suppresses warning diagnostics (errors are still issued) |
| --set_error_limit | | Sets the error limit to *num*. The linker abandons linking after this number of errors. (The default is 100.) |
| --verbose_diagnostics | | Provides verbose diagnostics that display the original source with line-wrap |
| --warn_sections | -w | Displays a message when an undefined output section is created |

**Table 2-22. File Search Path Options Summary**

| Option | Alias | Description |
|---|---|---|
| --library | -l | Names an archive library or link command *filename* as linker input |
| --search_path | `-I` | Alters library-search algorithms to look in a directory named with *pathname* before looking in the default location. This option must appear before the --library option. |
| --disable_auto_rts | | Disables the automatic selection of a run-time-support library |
| --priority | -priority | Satisfies unresolved references by the first library that contains a definition for that symbol |
| --reread_libs | -x | Forces rereading of libraries, which resolves back references |

**Table 2-23. Linker Output Options Summary**

| Option | Alias | Description |
|---|---|---|
| --output_file | -o | Names the executable output module. The default filename is a.out. |
| --map_file | -m | Produces a map or listing of the input and output sections, including holes, and places the listing in *filename* |
| --absolute_exe | -a | Produces an absolute, executable module. This is the default; if neither --absolute_exe nor --relocatable is specified, the linker acts as if --absolute_exe were specified. |
| --generate_dead_funcs_list | | Writes a list of the dead functions that were removed by the linker to file fname. |
| --mapfile_contents | | Controls the information that appears in the map file. |
| --relocatable | -r | Produces a nonexecutable, relocatable output module |
| --run_abs | -abs | Produces an absolute listing file |
| --xml_link_info | | Generates a well-formed XML *file* containing detailed information about the result of a link |

**Table 2-24. Symbol Management Options Summary**

| Option | Alias | Description |
|---|---|---|
| --entry_point | -e | Defines a global symbol that specifies the primary entry point for the output module |
| --globalize | | Changes the symbol linkage to global for symbols that match *pattern* |
| --hide | | Hides global symbols that match *pattern* |
| --localize | | Changes the symbol linkage to local for symbols that match *pattern* |
| --make_global | -g | Makes *symbol* global (overrides -h) |
| --make_static | -h | Makes all global symbols static |
| --no_sym_merge | -b | Disables merge of symbolic debugging information in COFF object files |
| --no_sym_table | -s | Strips symbol table information and line number entries from the output module |
| --scan_libraries | -scanlibs | Scans all libraries for duplicate symbol definitions |
| --symbol_map | | Maps symbol references to a symbol definition of a different name |
| --undef_sym | -u | Places an unresolved external *symbol* into the output module's symbol table |
| --unhide | | Reveals (un-hides) global symbols that match *pattern* |

**Table 2-25. Run-Time Environment Options Summary**

| Option | Alias | Description |
|---|---|---|
| --heap_size | -heap | Sets heap size (for the dynamic memory allocation in C) to *size* bytes and defines a global symbol that specifies the heap size. Default = 128 bytes |
| --stack_size | -stack | Sets C system stack size to *size* bytes and defines a global symbol that specifies the stack size. Default = 128 bytes |
| --use_hw_mpy[={16\|32\|F5}] | | Replaces all references to the default integer/long multiply routine with the version of the multiply routine that uses the hardware multiplier support. |
| --arg_size | --args | Allocates memory to be used by the loader to pass arguments |
| --fill_value | -f | Sets default fill values for holes within output sections; *fill_value* is a 32-bit constant |
| --ram_model | -cr | Initializes variables at load time |
| --rom_model | -c | Autoinitializes variables at run time |

**Table 2-26. Miscellaneous Options Summary**

| Option | Alias | Description |
|---|---|---|
| --disable_clink | -j | Disables conditional linking of COFF object modules |
| --linker_help | -help | Displays information about syntax and available options |
| --preferred_order | | Prioritizes placement of functions |
| --strict_compatibility | | Performs more conservative and rigorous compatibility checking of input object files |

### 2.3.1 Frequently Used Options

Following are detailed descriptions of options that you will probably use frequently:

**--c_src_interlist**

Invokes the interlist feature, which interweaves original C/C++ source with compiler-generated assembly language. The interlisted C statements may appear to be out of sequence. You can use the interlist feature with the optimizer by combining the --optimizer_interlist and --c_src_interlist options. See Section 3.8. The --c_src_interlist option can have a negative performance and/or code size impact.

**--cmd_file**=*filename*

Appends the contents of a file to the option set. You can use this option to avoid limitations on command line length or C style comments imposed by the host operating system. Use a # or ; at the beginning of a line in the command file to include comments. You can also include comments by delimiting them with /* and */. To specify options, surround hyphens with quotation marks. For example, "--"quiet.

You can use the --cmd_file option multiple times to specify multiple files. For instance, the following indicates that file3 should be compiled as source and file1 and file2 are --cmd_file files:

```
cl430 --cmd_file=file1 --cmd_file=file2 file3
```

**--compile_only**

Suppresses the linker and overrides the --run_linker option, which specifies linking. The --compile_only option's short form is -c. Use this option when you have --run_linker specified in the MSP430_C_OPTION environment variable and you do not want to link. See Section 4.1.3.

**--define**=*name[=def]*

Predefines the constant *name* for the preprocessor. This is equivalent to inserting #define *name def* at the top of each C source file. If the optional[=*def*] is omitted, the *name* is set to 1. The --define option's short form is -D.

If you want to define a quoted string and keep the quotation marks, do one of the following:

- For Windows, use --define=*name*="\"*string def*\"". For example, --define=car="\"sedan\""
- For UNIX, use --define=*name*="'*string def*'". For example, --define=car="'sedan'"
- For Code Composer Studio, enter the definition in a file and include that file with the --cmd_file option.

**--help**

Displays the syntax for invoking the compiler and lists available options. If the --help option is followed by another option or phrase, detailed information about the option or phrase is displayed. For example, to see information about debugging options use --help debug.

**--include_path**=*directory*

Adds *directory* to the list of directories that the compiler searches for #include files. The --include_path option's short form is -I. You can use this option several times to define several directories; be sure to separate the --include_path options with spaces. If you do not specify a directory name, the preprocessor ignores the --include_path option. See Section 2.6.2.1.

**--keep_asm** Retains the assembly language output from the compiler or assembly optimizer. Normally, the compiler deletes the output assembly language file after assembly is complete. The --keep_asm option's short form is -k.

**--quiet** Suppresses banners and progress information from all the tools. Only source filenames and error messages are output. The --quiet option's short form is -q.

**--run_linker** Runs the linker on the specified object files. The --run_linker option and its parameters follow all other options on the command line. All arguments that follow --run_linker are passed to the linker. The --run_linker option's short form is -z. See Section 4.1.

**--skip_assembler** Compiles only. The specified source files are compiled but not assembled or linked. The --skip_assembler option's short form is -n. This option overrides --run_linker. The output is assembly language output from the compiler.

**--src_interlist** Invokes the interlist feature, which interweaves optimizer comments *or* C/C++ source with assembly source. If the optimizer is invoked (--opt_level=*n* option), optimizer comments are interlisted with the assembly language output of the compiler, which may rearrange code significantly. If the optimizer is not invoked, C/C++ source statements are interlisted with the assembly language output of the compiler, which allows you to inspect the code generated for each C/C++ statement. The --src_interlist option implies the --keep_asm option. The --src_interlist option's short form is -s.

**--tool_version** Prints the version number for each tool in the compiler. No compiling occurs.

**--undefine**=*name* Undefines the predefined constant *name*. This option overrides any --define options for the specified constant. The --undefine option's short form is -U.

**--verbose** Displays progress information and toolset version while compiling. Resets the --quiet option.

### 2.3.2 Miscellaneous Useful Options

Following are detailed descriptions of miscellaneous options:

**--check_misra**={all|required|advisory|none|*rulespec*} Displays the specified amount or type of MISRA-C documentation. The *rulespec* parameter is a comma-separated list of specifiers. See Section 5.3 for details.

**--fp_mode**={relaxed|strict} Supports relaxed floating-point mode. In this mode, if the result of a double-precision floating-point expression is assigned to a single-precision floating-point or an integer, the computations in the expression are converted to single-precision computations. Any double-precision constants in the expression are also converted to single-precision if they can be correctly represented as single-precision constants. This behavior does not conform with ISO; but it results in faster code, with some loss in accuracy. In the following example, where N is a number, iN=integer variable, fN=float variable, dN=double variable:

```
il = f1 + f2 * 5.0 -> +, * are float, 5.0 is converted to 5.0f
il = d1 + d2 * d3 -> +, * are float
f1 = f2 + f3 * 1.1; -> +, * are float, 1.1 is converted to 1
```

| | |
|---|---|
| | To enable relaxed floating-point mode use the --fp_mode=relaxed option, which also sets --fp_reassoc=on. To disable relaxed floating-point mode use the --fp_mode=strict option, which also sets --fp_reassoc=off. The default behavior is --fp_mode=strict.<br>If --strict_ansi is specified, --fp_mode=strict is set automatically. You can enable the relaxed floating-point mode with strict ANSI mode by specifying --fp_mode=relaxed after --strict_ansi. |
| **--fp_reassoc**={on\|off} | Enables or disables the reassociation of floating-point arithmetic. If --fp_mode=relaxed is specified, --fp_reassoc=on is set automatically. If --strict_ansi is set, --fp_reassoc=off is set since reassociation of floating-point arithmetic is an ANSI violation. |
| **--keep_unneeded_statics** | Does not delete unreferenced static variables. The parser by default remarks about and then removes any unreferenced static variables. The --keep_unneeded_statics option keeps the parser from deleting unreferenced static variables and any static functions that are referenced by these variable definitions. Unreferenced static functions will still be removed. |
| **--no_const_clink** | Tells the compiler to not generate .clink directives for const global arrays. By default, these arrays are placed in a .const subsection and conditionally linked. |
| **--misra_advisory**={error\|warning\|remark\|suppress} | Sets the diagnostic severity for advisory MISRA-C:2004 rules. |
| **--misra_required**={error\|warning\|remark\|suppress} | Sets the diagnostic severity for required MISRA-C:2004 rules. |
| **--preinclude**=*filename* | Includes the source code of *filename* at the beginning of the compilation. This can be used to establish standard macro definitions. The filename is searched for in the directories on the include search list. The files are processed in the order in which they were specified. |
| **--printf_support**={full\|nofloat\|minimal} | Enables support for smaller, limited versions of the printf and sprintf run-time-support functions. The valid values are:<br>• full: Supports all format specifiers. This is the default.<br>• nofloat: Excludes support for printing floating point values. Supports all format specifiers except %f, %g, %G, %e, and %E.<br>• minimal: Supports the printing of integer, char, or string values without width or precision flags. Specifically, only the %%, %d, %o, %c, %s, and %x format specifiers are supported<br>There is no run-time error checking to detect if a format specifier is used for which support is not included. The --printf_support option precedes the --run_linker option, and must be used when performing the final link. |
| **--sat_reassoc**={on\|off} | Enables or disables the reassociation of saturating arithmetic. |

### 2.3.3 Run-Time Model Options

These options are specific to the MSP430 toolset. Please see the referenced sections for more information.

| | |
|---|---|
| **--code_model**={large\|small} | Specifies the code memory model: small (16-bit function pointers and low 64K memory) or large (20-bit function pointers and 1MB address space). See Section 6.1.1 for details. |
| **--data_model**={restricted\|large\|small} | Specifies the data memory model: small (16-bit data pointers and low 64K memory), restricted (32-bit data pointers, objects restricted to 64K, and 1MB memory), and large (32-bit data pointers and 1MB memory). See Section 6.1.2 for details. |
| **--large_memory_model** | This option is deprecated. Use --data_model=large. |
| **--near_data**={globals\|none} | Specifies when global read/write data must be located in the first 64K of memory. See Section 6.1.3 for details. |
| **--plain_char**={unsigned\|signed} | Specifies how to treat C/C++ plain char variables, default is unsigned. |
| **--silicon_version** | Selects the instruction set version. Using --silicon_version=mspx generates code for MSP430X devices (20-bit code addressing). Using --silicon_version=msp generates code for 16-bit MSP430 devices.<br>Modules assembled/compiled for 16-bit MSP430 devices are not compatible with modules that are assembled/compiled for 20-bit MSPx devices. The linker generates errors if an attempt is made to combine incompatible object files. |
| **--small_enum** | By default, the MSP430 compiler uses 16 bits for every enum. When you use the --small-enum option, the smallest possible byte size for the enumeration type is used. For example, enum example_enum {first = -128, second = 0, third = 127} uses only one byte instead of 16 bits when the --small-enum option is used. Do not link object files compiled with the --small_enum option with object files that have been compiled without it. If you use the --small_enum option, you must use it with all of your C/C++ files; otherwise, you will encounter errors that cannot be detected until run time. |

### 2.3.4 Symbolic Debugging and Profiling Options

The following options are used to select symbolic debugging or profiling:

**--profile:breakpt** Disables optimizations that would cause incorrect behavior when using a breakpoint-based profiler.

**--profile:power** Enables power profiling support by inserting NOPs into the frame code. These NOPs can then be instrumented by the power profiling tooling to track the power usage of functions. If the power profiling tool is not used, this option increases the cycle count of each function because of the NOPs. The --profile:power option also disables optimizations that cannot be handled by the power-profiler.

**--symdebug:coff** Enables symbolic debugging using the alternate STABS debugging format. This may be necessary to allow debugging with older debuggers or custom tools, which do not read the DWARF format.

**--symdebug:dwarf** Generates directives that are used by the C/C++ source-level debugger and enables assembly source debugging in the assembler. The --symdebug:dwarf option's short form is -g. The --symdebug:dwarf option disables many code generator optimizations, because they disrupt the debugger. You can use the --symdebug:dwarf option with the --opt_level (aliased as -O) option to maximize the amount of optimization that is compatible with debugging (see Section 3.9).

For more information on the DWARF debug format, see *The DWARF Debugging Standard*.

**--symdebug:dwarf_subsections**=on|off Changes the way the debug information is represented in the object file. When the option is set to on, the resulting object file supports a rapid form of type merging in the debugging information that is done in the linker. If you have been using the --no_sym_merge linker option to disable type merging of the debugging information in order to reduce link time at the cost of increased .out file size, recompiling with --symdebug:dwarf_subsections=on can realize a reasonable link time without increasing the .out file size. The default behavior is off.

**--symdebug:none** Disables all symbolic debugging output. This option is not recommended; it prevents debugging and most performance analysis capabilities.

**--symdebug:profile_coff** Adds the necessary debug directives to the object file which are needed by the profiler to allow function level profiling with minimal impact on optimization (when used). Using --symdebug:coff may hinder some optimizations to ensure that debug ability is maintained, while this option will not hinder optimization.

You can set breakpoints and profile on function-level boundaries in Code Composer Studio, but you cannot single-step through code as with full debug ability.

**--symdebug:skeletal** Generates as much symbolic debugging information as possible without hindering optimization. Generally, this consists of global-scope information only. This option reflects the default behavior of the compiler.

See Section 2.3.11 for a list of deprecated symbolic debugging options.

### 2.3.5 Specifying Filenames

The input files that you specify on the command line can be C source files, C++ source files, assembly source files, or object files. The compiler uses filename extensions to determine the file type.

| Extension | File Type |
|---|---|
| .asm, .abs, or .s* (extension begins with s) | Assembly source |
| .c | C source |
| .C | Depends on operating system |
| .cpp, .cxx, .cc | C++ source |
| .obj .o* .dll .so | Object |

**NOTE: Case Sensitivity in Filename Extensions**

Case sensitivity in filename extensions is determined by your operating system. If your operating system is not case sensitive, a file with a .C extension is interpreted as a C file. If your operating system is case sensitive, a file with a .C extension is interpreted as a C++ file.

For information about how you can alter the way that the compiler interprets individual filenames, see Section 2.3.6. For information about how you can alter the way that the compiler interprets and names the extensions of assembly source and object files, see Section 2.3.9.

You can use wildcard characters to compile or assemble multiple files. Wildcard specifications vary by system; use the appropriate form listed in your operating system manual. For example, to compile all of the files in a directory with the extension .cpp, enter the following:

```
cl430 *.cpp
```

**NOTE: No Default Extension for Source Files is Assumed**

If you list a filename called example on the command line, the compiler assumes that the entire filename is example not example.c. No default extensions are added onto files that do not contain an extension.

### 2.3.6 Changing How the Compiler Interprets Filenames

You can use options to change how the compiler interprets your filenames. If the extensions that you use are different from those recognized by the compiler, you can use the filename options to specify the type of file. You can insert an optional space between the option and the filename. Select the appropriate option for the type of file you want to specify:

**--asm_file**=*filename* for an assembly language source file
**--c_file**=*filename* for a C source file
**--cpp_file**=*filename* for a C++ source file
**--obj_file**=*filename* for an object file

For example, if you have a C source file called file.s and an assembly language source file called assy, use the --asm_file and --c_file options to force the correct interpretation:

```
cl430 --c_file=file.s --asm_file=assy
```

You cannot use the filename options with wildcard specifications.

### 2.3.7 Changing How the Compiler Processes C Files

The --cpp_default option causes the compiler to process C files as C++ files. By default, the compiler treats files with a .c extension as C files. See Section 2.3.8 for more information about filename extension conventions.

### 2.3.8 Changing How the Compiler Interprets and Names Extensions

You can use options to change how the compiler program interprets filename extensions and names the extensions of the files that it creates. The filename extension options must precede the filenames they apply to on the command line. You can use wildcard specifications with these options. An extension can be up to nine characters in length. Select the appropriate option for the type of extension you want to specify:

| | |
|---|---|
| **--asm_extension**=*new extension* | for an assembly language file |
| **--c_extension**=*new extension* | for a C source file |
| **--cpp_extension**=*new extension* | for a C++ source file |
| **--listing_extension**=*new extension* | sets default extension for listing files |
| **--obj_extension**=*new extension* | for an object file |

The following example assembles the file fit.rrr and creates an object file named fit.o:

```
cl430 --asm_extension=.rrr --obj_extension=.o fit.rrr
```

The period (.) in the extension is optional. You can also write the example above as:

```
cl430 --asm_extension=rrr --obj_extension=o fit.rrr
```

### 2.3.9 Specifying Directories

By default, the compiler program places the object, assembly, and temporary files that it creates into the current directory. If you want the compiler program to place these files in different directories, use the following options:

| | |
|---|---|
| **--abs_directory**=*directory* | Specifies the destination directory for absolute listing files. The default is to use the same directory as the object file directory. For example:<br>`cl430 --abs_directory=d:\abso_list` |
| **--asm_directory**=*directory* | Specifies a directory for assembly files. For example:<br>`cl430 --asm_directory=d:\assembly` |
| **--list_directory**=*directory* | Specifies the destination directory for assembly listing files and cross-reference listing files. The default is to use the same directory as the object file directory. For example:<br>`cl430 --list_directory=d:\listing` |
| **--obj_directory**=*directory* | Specifies a directory for object files. For example:<br>`cl430 --obj_directory=d:\object` |
| **--temp_directory**=*directory* | Specifies a directory for temporary intermediate files. For example:<br>`cl430 --temp_directory=d:\temp` |

### 2.3.10 Assembler Options

Following are assembler options that you can use with the compiler. For more information, see the *MSP430 Assembly Language Tools User's Guide.*

**--absolute_listing** Generates a listing with absolute addresses rather than section-relative offsets.

**--asm_define**=*name*[=*def*] Predefines the constant *name* for the assembler; produces a .set directive for a constant or a .arg directive for a string. If the optional [=*def*] is omitted, the *name* is set to 1. If you want to define a quoted string and keep the quotation marks, do one of the following:

- For Windows, use --asm_define=*name*="\"*string def*\"". For example: `--asm_define=car="\"sedan\""`
- For UNIX, use --asm_define=*name*="'*string def*"'. For example: `--asm_define=car='"sedan"'`
- For Code Composer Studio, enter the definition in a file and include that file with the --cmd_file option.

**--asm_dependency** Performs preprocessing for assembly files, but instead of writing preprocessed output, writes a list of dependency lines suitable for input to a standard make utility. The list is written to a file with the same name as the source file but with a .ppa extension.

**--asm_includes** Performs preprocessing for assembly files, but instead of writing preprocessed output, writes a list of files included with the #include directive. The list is written to a file with the same name as the source file but with a .ppa extension.

**--asm_listing** Produces an assembly listing file.

**--asm_undefine**=*name* Undefines the predefined constant *name*. This option overrides any --asm_define options for the specified name.

**--copy_file**=*filename* Copies the specified file for the assembly module; acts like a .copy directive. The file is inserted before source file statements. The copied file appears in the assembly listing files.

**--cross_reference** Produces a symbolic cross-reference in the listing file.

**--include_file**=*filename* Includes the specified file for the assembly module; acts like a .include directive. The file is included before source file statements. The included file does not appear in the assembly listing files.

**--output_all_syms** Puts labels in the symbol table. Label definitions are written to the COFF symbol table for use with symbolic debugging.

**--syms_ignore_case** Makes letter case insignificant in the assembly language source files. For example, --syms_ignore_case makes the symbols ABC and abc equivalent. *If you do not use this option, case is significant* (this is the default).

### 2.3.11 Deprecated Options

Several compiler options have been deprecated. The compiler continues to accept these options, but they are not recommended for use. Future releases of the tools will not support these options. Table 2-27 lists the deprecated options and the options that have replaced them.

**Table 2-27. Compiler Backwards-Compatibility Options Summary**

| Old Option | Effect | New Option |
|---|---|---|
| -gp | Allows function-level profiling of optimized code | --symdebug:dwarf or -g |
| -gt | Enables symbolic debugging using the alternate STABS debugging format | --symdebug:coff |
| -gw | Enables symbolic debugging using the DWARF debugging format | --symdebug:dwarf or -g |

Additionally, the --symdebug:profile_coff option has been added to enable function-level profiling of optimized code with symbolic debugging using the STABS debugging format (the --symdebug:coff or -gt option).

## 2.4 Controlling the Compiler Through Environment Variables

An environment variable is a system symbol that you define and assign a string to. Setting environment variables is useful when you want to run the compiler repeatedly without re-entering options, input filenames, or pathnames.

**NOTE: C_OPTION and C_DIR**

The C_OPTION and C_DIR environment variables are deprecated. Use the device-specific environment variables instead.

### 2.4.1 Setting Default Compiler Options (MSP430_C_OPTION)

You might find it useful to set the compiler, assembler, and linker default options using the MSP430_C_OPTION environment variable. If you do this, the compiler uses the default options and/or input filenames that you name MSP430_C_OPTION every time you run the compiler.

Setting the default options with these environment variables is useful when you want to run the compiler repeatedly with the same set of options and/or input files. After the compiler reads the command line and the input filenames, it looks for the MSP430_C_OPTION environment variable and processes it.

The table below shows how to set the MSP430_C_OPTION environment variable. Select the command for your operating system:

| Operating System | Enter |
|---|---|
| UNIX (Bourne shell) | **MSP430_C_OPTION="** *option*$_1$ [*option*$_2$ . . .]**"; export MSP430_C_OPTION** |
| Windows | **set MSP430_C_OPTION=** *option*$_1$ [;*option*$_2$ . . .] |

Environment variable options are specified in the same way and have the same meaning as they do on the command line. For example, if you want to always run quietly (the --quiet option), enable C/C++ source interlisting (the --src_interlist option), and link (the --run_linker option) for Windows, set up the MSP430_C_OPTION environment variable as follows:

```
set MSP430_C_OPTION=--quiet --src_interlist --run_linker
```

In the following examples, each time you run the compiler, it runs the linker. Any options following --run_linker on the command line or in MSP430_C_OPTION are passed to the linker. Thus, you can use the MSP430_C_OPTION environment variable to specify default compiler and linker options and then specify additional compiler and linker options on the command line. If you have set --run_linker in the environment variable and want to compile only, use the compiler --compile_only option. These additional examples assume MSP430_C_OPTION is set as shown above:

```
cl430 *c                        ; compiles and links
cl430 --compile_only *.c        ; only compiles
cl430 *.c --run_linker lnk.cmd  ; compiles and links using a command file
cl430 --compile_only *.c --run_linker lnk.cmd
       ; only compiles (--compile_only overrides --run_linker)
```

For details on compiler options, see Section 2.3. For details on linker options, see .

### 2.4.2 Naming an Alternate Directory ( MSP430_C_DIR )

The linker uses the MSP430_C_DIR environment variable to name alternate directories that contain object libraries. The command syntaxes for assigning the environment variable are:

| Operating System | Enter |
|---|---|
| UNIX (Bourne shell) | **MSP430_C_DIR="** *pathname*$_1$ **;** *pathname*$_2$ **;..."; export MSP430_C_DIR** |
| Windows | **set MSP430_C_DIR=** *pathname*$_1$ **;** *pathname*$_2$ **;...** |

The *pathnames* are directories that contain input files. The pathnames must follow these constraints:

- Pathnames must be separated with a semicolon.
- Spaces or tabs at the beginning or end of a path are ignored. For example, the space before and after the semicolon in the following is ignored:

  ```
  set MSP430_C_DIR=c:\path\one\to\tools ; c:\path\two\to\tools
  ```

- Spaces and tabs are allowed within paths to accommodate Windows directories that contain spaces. For example, the pathnames in the following are valid:

  ```
  set MSP430_C_DIR=c:\first path\to\tools;d:\second path\to\tools
  ```

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

| Operating System | Enter |
|---|---|
| UNIX (Bourne shell) | `unset MSP430_C_DIR` |
| Windows | `set MSP430_C_DIR=` |

## 2.5 Precompiled Header Support

Precompiled header files may reduce the compile time for applications whose source files share a common set of headers, or a single file which has a large set of header files. Using precompiled headers, some recompilation is avoided thus saving compilation time.

There are two ways to use precompiled header files. One is the automatic precompiled header file processing and the other is called the manual precompiled header file processing.

### *2.5.1 Automatic Precompiled Header*

The option to turn on automatic precompiled header processing is: --pch. Under this option, the compile step takes a snapshot of all the code prior to the header stop point, and dump it out to a file with suffix .pch. This snapshot does not have to be recompiled in the future compilations of this file or compilations of files with the same header files.

The stop point typically is the first token in the primary source file that does not belong to a preprocessing directive. For example, in the following the stopping point is before int i:

```
#include "x.h"
#include "y.h"
int i;
```

Carefully organizing the include directives across multiple files so that their header files maximize common usage can increase the compile time savings when using precompiled headers.

A precompiled header file is produced only if the header stop point and the code prior to it meet certain requirements.

### *2.5.2 Manual Precompiled Header*

You can manually control the creation and use of precompiled headers by using several command line options. You specify a precompiled header file with a specific filename as follows:

**--create_pch**=*filename*

The --use_pch=*filename* option specifies that the indicated precompiled header file should be used for this compilation. If this precompiled header file is invalid, if its prefix does not match the prefix for the current primary source file for example, a warning is issued and the header file is not used.

If --create_pch=*filename* or --use_pch=*filename* is used with --pch_dir, the indicated filename, which can be a path name, is tacked on to the directory name, unless the filename is an absolute path name.

The --create_pch, --use_pch, and --pch options cannot be used together. If more than one of these options is specified, only the last one is applied. In manual mode, the header stop points are determined in the same way as in automatic mode. The precompiled header file applicability is determined in the same manner.

### *2.5.3 Additional Precompiled Header Options*

The --pch_verbose option displays a message for each precompiled header file that is considered but not used. The --pch_dir=*pathname* option specifies the path where the precompiled header file resides.

## 2.6 Controlling the Preprocessor

This section describes specific features that control the preprocessor, which is part of the parser. A general description of C preprocessing is in section A12 of K&R. The C/C++ compiler includes standard C/C++ preprocessing functions, which are built into the first pass of the compiler. The preprocessor handles:

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

The preprocessor produces self-explanatory error messages. The line number and the filename where the error occurred are printed along with a diagnostic message.

### *2.6.1 Predefined Macro Names*

The compiler maintains and recognizes the predefined macro names listed in Table 2-28.

**Table 2-28. Predefined MSP430 Macro Names**

| Macro Name | Description |
|---|---|
| _ _DATE_ _ (1) | Expands to the compilation date in the form *mmm dd yyyy* |
| _ _FILE_ _ (1) | Expands to the current source filename |
| _ _LARGE_CODE_MODEL_ _ | Defined if --code_model=large is specified |
| _ _LARGE_DATA_MODEL_ _ | Defined if --data_model=large or --data_model=restricted is specified |
| _ _LINE_ _ (1) | Expands to the current line number |
| _ _LONG_PTRDIFF_T_ _ | Defined when --data_model=large is specified. Indicates ptrdiff_t is a long. |
| _ _MSP430_ _ | Always defined |
| _ _MSP430X_ _ | Defined if --silicon_version=mspx is specified |
| _ _MSP430X461X_ _ | Defined if --silicon_version=mspx is specified |
| _ _PTRDIFF_T_TYPE_ _ | Set to the type of ptrdiff_t. Determined by the --data_model option. |
| _ _signed_chars_ _ | Defined if char types are signed by default (--plain_char=signed) |
| _ _SIZE_T_TYPE_ _ | Set to the type of size_t. Determined by the --data_model option. |
| _ _STDC_ _ (1) | Defined to indicate that compiler conforms to ISO C Standard. See Section 5.1 for exceptions to ISO C conformance. |
| _ _STDC_VERSION_ _ | C standard macro |
| _ _TI_COMPILER_VERSION_ _ | Defined to a 7-9 digit integer, depending on if X has 1, 2, or 3 digits. The number does not contain a decimal. For example, version 3.2.1 is represented as 3002001. The leading zeros are dropped to prevent the number being interpreted as an octal. |
| _ _TI_GNU_ATTIBUTE_SUPPORT_ _ | Defined if GCC extensions are enabled (the --gcc option is used); otherwise, it is undefined. |
| _ _TI_STRICT_ANSI_MODE__ | Defined if strict ANSI/ISO mode is enabled (the --strict_ansi option is used); otherwise, it is undefined. |
| _ _TIME_ _ (1) | Expands to the compilation time in the form "*hh:mm:ss*" |
| _ _unsigned_chars_ _ | Defined if char types are unsigned by default (default or -- plain_char=unsigned) |
| _ _UNSIGNED_LONG_SIZE_T_ _ | Defined when --data_model=large is specified. Indicates size_t is an unsigned long. |
| _INLINE | Expands to 1 if optimization is used (--opt_level or -O option); undefined otherwise. Regardless of any optimization, always undefined when --no_inlining is used. |

(1) Specified by the ISO standard

You can use the names listed in Table 2-28 in the same manner as any other defined name. For example,

```
printf ( "%s %s" , __TIME__ , __DATE__);
```

translates to a line such as:

```
printf ("%s %s" , "13:58:17", "Jan 14 1997");
```

### 2.6.2 The Search Path for #include Files

The #include preprocessor directive tells the compiler to read source statements from another file. When specifying the file, you can enclose the filename in double quotes or in angle brackets. The filename can be a complete pathname, partial path information, or a filename with no path information.

- If you enclose the filename in double quotes (" "), the compiler searches for the file in the following directories in this order:
  1. The directory of the file that contains the #include directive and in the directories of any files that contain that file.
  2. Directories named with the --include_path option.
  3. Directories set with the MSP430_C_DIR environment variable.
- If you enclose the filename in angle brackets (< >), the compiler searches for the file in the following directories in this order:
  1. Directories named with the --include_path option.
  2. Directories set with the MSP430_C_DIR environment variable.

See Section 2.6.2.1 for information on using the --include_path option. See Section 2.4.2 for more information on input file directories.

#### 2.6.2.1 Changing the #include File Search Path (--include_path Option)

The --include_path option names an alternate directory that contains #include files. The --include_path option's short form is -I. The format of the --include_path option is:

**--include_path**=*directory1* [**--include_path=** *directory2* ...]

There is no limit to the number of --include_path options per invocation of the compiler; each --include_path option names one *directory*. In C source, you can use the #include directive without specifying any directory information for the file; instead, you can specify the directory information with the --include_path option. For example, assume that a file called source.c is in the current directory. The file source.c contains the following directive statement:

```
#include "alt.h"
```

Assume that the complete pathname for alt.h is:

| | |
|---|---|
| UNIX | /tools/files/alt.h |
| Windows | c:\tools\files\alt.h |

The table below shows how to invoke the compiler. Select the command for your operating system:

| Operating System | Enter |
|---|---|
| UNIX | `cl430 --include_path=tools/files source.c` |
| Windows | `cl430 --include_path=c:\tools\files source.c` |

**NOTE: Specifying Path Information in Angle Brackets**

If you specify the path information in angle brackets, the compiler applies that information relative to the path information specified with --include_path options and the MSP430_C_DIR environment variable.

For example, if you set up MSP430_C_DIR with the following command:

```
MSP430_C_DIR "/usr/include;/usr/ucb"; export MSP430_C_DIR
```

or invoke the compiler with the following command:

```
cl430 --include_path=/usr/include file.c
```

and file.c contains this line:

```
#include <sys/proc.h>
```

the result is that the included file is in the following path:

```
/usr/include/sys/proc.h
```

### 2.6.3 Generating a Preprocessed Listing File (--preproc_only Option)

The --preproc_only option allows you to generate a preprocessed version of your source file with an extension of .pp. The compiler's preprocessing functions perform the following operations on the source file:

- Each source line ending in a backslash (\) is joined with the following line.
- Trigraph sequences are expanded.
- Comments are removed.
- #include files are copied into the file.
- Macro definitions are processed.
- All macros are expanded.
- All other preprocessing directives, including #line directives and conditional compilation, are expanded.

### 2.6.4 Continuing Compilation After Preprocessing (--preproc_with_compile Option)

If you are preprocessing, the preprocessor performs preprocessing only; it does not compile your source code. To override this feature and continue to compile after your source code is preprocessed, use the --preproc_with_compile option along with the other preprocessing options. For example, use --preproc_with_compile with --preproc_only to perform preprocessing, write preprocessed output to a file with a .pp extension, and compile your source code.

### 2.6.5 Generating a Preprocessed Listing File With Comments (--preproc_with_comment Option)

The --preproc_with_comment option performs all of the preprocessing functions except removing comments and generates a preprocessed version of your source file with a .pp extension. Use the --preproc_with_comment option instead of the --preproc_only option if you want to keep the comments.

### 2.6.6 Generating a Preprocessed Listing File With Line-Control Information (--preproc_with_line Option)

By default, the preprocessed output file contains no preprocessor directives. To include the #line directives, use the --preproc_with_line option. The --preproc_with_line option performs preprocessing only and writes preprocessed output with line-control information (#line directives) to a file named as the source file but with a .pp extension.

### 2.6.7 Generating Preprocessed Output for a Make Utility (--preproc_dependency Option)

The --preproc_dependency option performs preprocessing only, but instead of writing preprocessed output, writes a list of dependency lines suitable for input to a standard make utility. If you do not supply an optional filename, the list is written to a file with the same name as the source file but with a .pp extension.

### 2.6.8 Generating a List of Files Included With the #include Directive (--preproc_includes Option)

The --preproc_includes option performs preprocessing only, but instead of writing preprocessed output, writes a list of files included with the #include directive. If you do not supply an optional filename, the list is written to a file with the same name as the source file but with a .pp extension.

### 2.6.9 Generating a List of Macros in a File (--preproc_macros Option)

The --preproc_macros option generates a list of all predefined and user-defined macros. If you do not supply an optional filename, the list is written to a file with the same name as the source file but with a .pp extension. Predefined macros are listed first and indicated by the comment /* Predefined */. User-defined macros are listed next and indicated by the source filename.

## 2.7 Understanding Diagnostic Messages

One of the compiler's primary functions is to report diagnostics for the source program. The new linker also reports diagnostics. When the compiler or linker detects a suspect condition, it displays a message in the following format:

"*file.c*=, **line** *n* **:** *diagnostic severity* **:** *diagnostic message*

| | |
|---|---|
| "*file.c*" | The name of the file involved |
| **line** *n* **:** | The line number where the diagnostic applies |
| *diagnostic severity* | The diagnostic message severity (severity category descriptions follow) |
| *diagnostic message* | The text that describes the problem |

Diagnostic messages have an associated severity, as follows:

- A **fatal error** indicates a problem so severe that the compilation cannot continue. Examples of such problems include command-line errors, internal errors, and missing include files. If multiple source files are being compiled, any source files after the current one will not be compiled.
- An **error** indicates a violation of the syntax or semantic rules of the C/C++ language. Compilation continues, but object code is not generated.
- A **warning** indicates something that is valid but questionable. Compilation continues and object code is generated (if no errors are detected).
- A **remark** is less serious than a warning. It indicates something that is valid and probably intended, but may need to be checked. Compilation continues and object code is generated (if no errors are detected). By default, remarks are not issued. Use the --issue_remarks compiler option to enable remarks.

Diagnostics are written to standard error with a form like the following example:

```
"test.c", line 5: error: a break statement may only be used within a loop or switch
    break;
    ^
```

By default, the source line is omitted. Use the --verbose_diagnostics compiler option to enable the display of the source line and the error position. The above example makes use of this option.

The message identifies the file and line involved in the diagnostic, and the source line itself (with the position indicated by the ^ character) follows the message. If several diagnostics apply to one source line, each diagnostic has the form shown; the text of the source line is displayed several times, with an appropriate position indicated each time.

Long messages are wrapped to additional lines, when necessary.

You can use the --display_error_number command-line option to request that the diagnostic's numeric identifier be included in the diagnostic message. When displayed, the diagnostic identifier also indicates whether the diagnostic can have its severity overridden on the command line. If the severity can be overridden, the diagnostic identifier includes the suffix -D (for *discretionary*); otherwise, no suffix is present. For example:

```
"Test_name.c", line 7: error #64-D: declaration does not declare anything
    struct {};
      ^
"Test_name.c", line 9: error #77: this declaration has no storage class or type specifier
    xxxxx;
    ^
```

Because an error is determined to be discretionary based on the error severity associated with a specific context, an error can be discretionary in some cases and not in others. All warnings and remarks are discretionary.

For some messages, a list of entities (functions, local variables, source files, etc.) is useful; the entities are listed following the initial error message:

```
"test.c", line 4: error: more than one instance of overloaded function "f"
            matches the argument list:
            function "f(int)"
            function "f(float)"
            argument types are: (double)
   f(1.5);
   ^
```

In some cases, additional context information is provided. Specifically, the context information is useful when the front end issues a diagnostic while doing a template instantiation or while generating a constructor, destructor, or assignment operator function. For example:

```
"test.c", line 7: error: "A::A()" is inaccessible
   B x;
   ^
          detected during implicit generation of "B::B()" at line 7
```

Without the context information, it is difficult to determine to what the error refers.

### 2.7.1 Controlling Diagnostics

The C/C++ compiler provides diagnostic options to control compiler- and linker-generated diagnostics. The diagnostic options must be specified before the --run_linker option.

| | |
|---|---|
| **--diag_error**=*num* | Categorizes the diagnostic identified by *num* as an error. To determine the numeric identifier of a diagnostic message, use the --display_error_number option first in a separate compile. Then use --diag_error=*num* to recategorize the diagnostic as an error. You can only alter the severity of discretionary diagnostics. |
| **--diag_remark**=*num* | Categorizes the diagnostic identified by *num* as a remark. To determine the numeric identifier of a diagnostic message, use the --display_error_number option first in a separate compile. Then use --diag_remark=*num* to recategorize the diagnostic as a remark. You can only alter the severity of discretionary diagnostics. |
| **--diag_suppress**=*num* | Suppresses the diagnostic identified by *num*. To determine the numeric identifier of a diagnostic message, use the --display_error_number option first in a separate compile. Then use --diag_suppress=*num* to suppress the diagnostic. You can only suppress discretionary diagnostics. |
| **--diag_warning**=*num* | Categorizes the diagnostic identified by *num* as a warning. To determine the numeric identifier of a diagnostic message, use the --display_error_number option first in a separate compile. Then use --diag_warning=*num* to recategorize the diagnostic as a warning. You can only alter the severity of discretionary diagnostics. |

| | |
|---|---|
| **--display_error_number** | Displays a diagnostic's numeric identifier along with its text. Use this option in determining which arguments you need to supply to the diagnostic suppression options (--diag_suppress, --diag_error, --diag_remark, and --diag_warning). This option also indicates whether a diagnostic is discretionary. A discretionary diagnostic is one whose severity can be overridden. A discretionary diagnostic includes the suffix -D; otherwise, no suffix is present. See Section 2.7. |
| **--issue_remarks** | Issues remarks (nonserious warnings), which are suppressed by default. |
| **--no_warnings** | Suppresses warning diagnostics (errors are still issued). |
| **--set_error_limit**=*num* | Sets the error limit to *num*, which can be any decimal value. The compiler abandons compiling after this number of errors. (The default is 100.) |
| **--verbose_diagnostics** | Provides verbose diagnostics that display the original source with line-wrap and indicate the position of the error in the source line |
| **--write_diagnostics_file** | Produces a diagnostics information file with the same source file name with an *.err* extension. (The --write_diagnostics_file option is not supported by the linker.) |

### 2.7.2 How You Can Use Diagnostic Suppression Options

The following example demonstrates how you can control diagnostic messages issued by the compiler. You control the linker diagnostic messages in a similar manner.

```
int one();
int I;
int main()
{
   switch (I){
   case  1;
      return one ();
      break;
   default:
      return 0;
   break;
   }
}
```

If you invoke the compiler with the --quiet option, this is the result:

```
"err.c", line 9: warning: statement is unreachable
"err.c", line 12: warning: statement is unreachable
```

Because it is standard programming practice to include break statements at the end of each case arm to avoid the fall-through condition, these warnings can be ignored. Using the --display_error_number option, you can find out the diagnostic identifier for these warnings. Here is the result:

```
[err.c]
"err.c", line 9: warning #111-D: statement is unreachable
"err.c", line 12: warning #111-D: statement is unreachable
```

Next, you can use the diagnostic identifier of 111 as the argument to the --diag_remark option to treat this warning as a remark. This compilation now produces no diagnostic messages (because remarks are disabled by default).

Although this type of control is useful, it can also be extremely dangerous. The compiler often emits messages that indicate a less than obvious problem. Be careful to analyze all diagnostics emitted before using the suppression options.

## 2.8 Other Messages

Other error messages that are unrelated to the source, such as incorrect command-line syntax or inability to find specified files, are usually fatal. They are identified by the symbol >> preceding the message.

## 2.9 Generating Cross-Reference Listing Information (--gen_acp_xref Option)

The --gen_acp_xref option generates a cross-reference listing file that contains reference information for each identifier in the source file. (The --gen_acp_xref option is separate from --cross_reference, which is an assembler rather than a compiler option.) The cross-reference listing file has the same name as the source file with a *.crl* extension.

The information in the cross-reference listing file is displayed in the following format:

*sym-id name X filename line number column number*

| | | |
|---|---|---|
| *sym-id* | An integer uniquely assigned to each identifier | |
| *name* | The identifier name | |
| *X* | One of the following values: | |
| | D | Definition |
| | d | Declaration (not a definition) |
| | M | Modification |
| | A | Address taken |
| | U | Used |
| | C | Changed (used and modified in a single operation) |
| | R | Any other kind of reference |
| | E | Error; reference is indeterminate |
| *filename* | The source file | |
| *line number* | The line number in the source file | |
| *column number* | The column number in the source file | |

## 2.10 Generating a Raw Listing File (--gen_acp_raw Option)

The --gen_acp_raw option generates a raw listing file that can help you understand how the compiler is preprocessing your source file. Whereas the preprocessed listing file (generated with the --preproc_only, --preproc_with_comment, --preproc_with_line, and --preproc_dependency preprocessor options) shows a preprocessed version of your source file, a raw listing file provides a comparison between the original source line and the preprocessed output. The raw listing file has the same name as the corresponding source file with an *.rl* extension.

The raw listing file contains the following information:

- Each original source line
- Transitions into and out of include files
- Diagnostics
- Preprocessed source line if nontrivial processing was performed (comment removal is considered trivial; other preprocessing is nontrivial)

Each source line in the raw listing file begins with one of the identifiers listed in Table 2-29.

**Table 2-29. Raw Listing File Identifiers**

| Identifier | Definition |
|---|---|
| N | Normal line of source |
| X | Expanded line of source. It appears immediately following the normal line of source if nontrivial preprocessing occurs. |
| S | Skipped source line (false #if clause) |
| L | Change in source position, given in the following format:<br>L *line number filename key*<br>Where *line number* is the line number in the source file. The *key* is present only when the change is due to entry/exit of an include file. Possible values of *key* are: |

**Table 2-29. Raw Listing File Identifiers (continued)**

| Identifier | Definition |
|---|---|
| | 1 = entry into an include file<br>2 = exit from an include file |

The --gen_acp_raw option also includes diagnostic identifiers as defined in Table 2-30.

**Table 2-30. Raw Listing File Diagnostic Identifiers**

| Diagnostic Identifier | Definition |
|---|---|
| E | Error |
| F | Fatal |
| R | Remark |
| W | Warning |

Diagnostic raw listing information is displayed in the following format:

*S filename line number column number diagnostic*

| | |
|---|---|
| *S* | One of the identifiers in Table 2-30 that indicates the severity of the diagnostic |
| *filename* | The source file |
| *line number* | The line number in the source file |
| *column number* | The column number in the source file |
| *diagnostic* | The message text for the diagnostic |

Diagnostics after the end of file are indicated as the last line of the file with a column number of 0. When diagnostic message text requires more than one line, each subsequent line contains the same file, line, and column information but uses a lowercase version of the diagnostic identifier. For more information about diagnostic messages, see Section 2.7.

## 2.11 Using Inline Function Expansion

When an inline function is called, the C/C++ source code for the function is inserted at the point of the call. This is known as inline function expansion. Inline function expansion is advantageous in short functions for the following reasons:

Inline function expansion is performed in one of the following ways:

- Intrinsic operators are inlined by default.
- Code is compiled with definition-controlled inlining.
- When the optimizer is invoked with the --opt_level=3 option (-O3), automatic inline expansion is performed at call sites to small functions. For more information about automatic inline function expansion, see Section 3.7.

**NOTE: Function Inlining Can Greatly Increase Code Size**

Expanding functions inline increases code size, especially inlining a function that is called in a number of places. Function inlining is optimal for functions that are called only from a small number of places and for small functions.

### 2.11.1 Inlining Intrinsic Operators

An operator is intrinsic if it can be implemented very efficiently with the target's instruction set. The compiler automatically inlines the intrinsic operators of the target system by default. Inlining happens whether or not you use the optimizer and whether or not you use any compiler or optimizer options on the command line. These functions are considered the intrinsic operators:

- abs
- labs
- fabs

### 2.11.2 Using the inline Keyword, the --no_inlining Option, and Level 3 Optimization

Definition-controlled inline function expansion is performed when you invoke the compiler with optimization and the compiler encounters the inline keyword in code. Functions with a variable number of arguments are not inlined. In addition, a limit is placed on the depth of inlining for recursive or nonleaf functions. Inlining should be used for small functions or functions that are called in a few places (though the compiler does not enforce this). You can control this type of function inlining with the inline keyword.

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

The semantics of the inline keyword follows that described in the C++ standard. The inline keyword is identically supported in C as a language extension. Because it is a language extension that could conflict with a strictly conforming program, however, the keyword is disabled in strict ANSI C mode (when you use the --strict_ansi compiler option). If you want to use definition-controlled inlining while in strict ANSI C mode, use the alternate keyword _ _inline.

When you want to compile without definition-controlled inlining, use the --no_inlining option.

**NOTE: Using the --no_inlining Option With Level 3 Optimizations**

When you use the --no_inlining option with --opt_level=3 (aliased as -O3) optimizations, automatic inlining is still performed.

## 2.12 Using Interlist

The compiler tools include a feature that interlists C/C++ source statements into the assembly language output of the compiler. The interlist feature enables you to inspect the assembly code generated for each C statement. The interlist behaves differently, depending on whether or not the optimizer is used, and depending on which options you specify.

The easiest way to invoke the interlist feature is to use the --c_src_interlist option. To compile and run the interlist on a program called function.c, enter:

```
cl430 --c_src_interlist function
```

The --c_src_interlist option prevents the compiler from deleting the interlisted assembly language output file. The output assembly file, function.asm, is assembled normally.

When you invoke the interlist feature without the optimizer, the interlist runs as a separate pass between the code generator and the assembler. It reads both the assembly and C/C++ source files, merges them, and writes the C/C++ statements into the assembly file as comments.

Using the --c_src_interlist option can cause performance and/or code size degradation.

Example 2-1 shows a typical interlisted assembly file.

For more information about using the interlist feature with the optimizer, see Section 3.8.

### Example 2-1. An Interlisted Assembly Language File

```
;******************************************************************************
;* MSP430 C/C++ Codegen                                          Unix v0.2.0 *
;* Date/Time created: Tue Jun 29 14:54:28 2004                                *
;******************************************************************************
        .compiler_opts --mem_model:code=flat --mem_model:data=flat --symdebug:none
;       acp430 -@/var/tmp/TI764/AAAv0aGVG
        .sect   ".text"
        .align  2
        .clink
        .global main
;---------------------------------------------------------------------------
;   3 | int main()
;---------------------------------------------------------------------------

;******************************************************************************
;* FUNCTION NAME: main                                                        *
;*                                                                            *
;*   Regs Modified     : SP,SR,r11,r12,r13,r14,r15                            *
;*   Regs Used         : SP,SR,r11,r12,r13,r14,r15                            *
;*   Local Frame Size  : 2 Args + 0 Auto + 0 Save = 2 byte                    *
;******************************************************************************
main:
;* ---------------------------------------------------------------------------*
        SUB.W     #2,SP
;---------------------------------------------------------------------------
;   5 | printf("Hello, world\n");
;---------------------------------------------------------------------------
        MOV.W     #$C$SL1+0,0(SP)       ; |5|
        CALL      #printf               ; |5|
                                        ; |5|
;---------------------------------------------------------------------------
;   7 | return 0;
;---------------------------------------------------------------------------
        MOV.W     #0,r12                ; |7|
        ADD.W     #2,SP                 ; |7|
        RET       ; |7|
        ; |7|
;******************************************************************************
;* STRINGS                                                                    *
;******************************************************************************
        .sect   ".const"
        .align  2
$C$SL1: .string "Hello, world",10,0
;******************************************************************************
;* UNDEFINED EXTERNAL REFERENCES                                              *
;******************************************************************************
        .global printf
```

## 2.13 Enabling Entry Hook and Exit Hook Functions

An entry hook is a routine that is called upon entry to each function in the program. An exit hook is a routine that is called upon exit of each function. Applications for hooks include debugging, trace, profiling, and stack overflow checking.

Entry and exit hooks are enabled using the following options:

**--entry_hook**[=*name*] Enables entry hooks. If specified, the hook function is called *name*. Otherwise, the default entry hook function name is __entry_hook.

**--entry_parm**{=name|address|none} Specify the parameters to the hook function. The name parameter specifies that the name of the calling function is passed to the hook function as an argument. In this case the signature for the hook function is: void hook(const char *name);

The address parameter specifies that the address of the calling function is passed to the hook function. In this case the signature for the hook function is: void hook(void (*addr)());

The none parameter specifies that the hook is called with no parameters. This is the default. In this case the signature for the hook function is: void hook(void);

**--exit_hook**[=*name*] Enables exit hooks. If specified, the hook function is called *name*. Otherwise, the default exit hook function name is __exit_hook.

**--exit_parm**{=name|address|none} Specify the parameters to the hook function. The name parameter specifies that the name of the calling function is passed to the hook function as an argument. In this case the signature for the hook function is: void hook(const char *name);

The address parameter specifies that the address of the calling function is passed to the hook function. In this case the signature for the hook function is: void hook(void (*addr)());

The none parameter specifies that the hook is called with no parameters. This is the default. In this case the signature for the hook function is: void hook(void);

The presence of the hook options creates an implicit declaration of the hook function with the given signature. If a declaration or definition of the hook function appears in the compilation unit compiled with the options, it must agree with the signatures listed above.

In C++, the hooks are declared extern "C". Thus you can define them in C (or assembly) without being concerned with name mangling.

Hooks can be declared inline, in which case the compiler tries to inline them using the same criteria as other inline functions.

Entry hooks and exit hooks are independent. You can enable one but not the other, or both. The same function can be used as both the entry and exit hook.

You must take care to avoid recursive calls to hook functions. The hook function should not call any function which itself has hook calls inserted. To help prevent this, hooks are not generated for inline functions, or for the hook functions themselves.

You can use the --remove_hooks_when_inlining option to remove entry/exit hooks for functions that are auto-inlined by the optimizer.

See for information about the NO_HOOKS pragma.

# Optimizing Your Code

The compiler tools can perform many optimizations to improve the execution speed and reduce the size of C and C++ programs by simplifying loops, software pipelining, rearranging statements and expressions, and allocating variables into registers.

This chapter describes how to invoke different levels of optimization and describes which optimizations are performed at each level. This chapter also describes how you can use the Interlist feature when performing optimization and how you can profile or debug optimized code.

**Topic** **Page**

## 3.1 Invoking Optimization

The C/C++ compiler is able to perform various optimizations. High-level optimizations are performed in the optimizer and low-level, target-specific optimizations occur in the code generator. Use high-level optimizations to achieve optimal code.

The easiest way to invoke optimization is to use the compiler program, specifying the --opt_level=*n* option on the compiler command line. You can use -O*n* to alias the --opt_level option. The *n* denotes the level of optimization (0, 1, 2, and 3), which controls the type and degree of optimization.

- **--opt_level=0** or **-O0**
  - Performs control-flow-graph simplification
  - Allocates variables to registers
  - Performs loop rotation
  - Eliminates unused code
  - Simplifies expressions and statements
  - Expands calls to functions declared inline
- **--opt_level=1** or **-O1**

  Performs all --opt_level=0 (-O0) optimizations, plus:
  - Performs local copy/constant propagation
  - Removes unused assignments
  - Eliminates local common expressions
- **--opt_level=2** or **-O2**

  Performs all --opt_level=1 (-O1) optimizations, plus:
  - Performs loop optimizations
  - Eliminates global common subexpressions
  - Eliminates global unused assignments
  - Performs loop unrolling

  The optimizer uses --opt_level=2 (-O2) as the default if you use --opt_level (-O) without an optimization level.
- **--opt_level=3** or **-O3**

  Performs all --opt_level=2 (-O2) optimizations, plus:
  - Removes all functions that are never called
  - Simplifies functions with return values that are never used
  - Inlines calls to small functions
  - Reorders function declarations; the called functions attributes are known when the caller is optimized
  - Propagates arguments into function bodies when all calls pass the same value in the same argument position
  - Identifies file-level variable characteristics

    If you use --opt_level=3 (-O3), see Section 3.2 and Section 3.3 for more information.
- **--opt_level=4** or **-O4**

  Performs link-time optimization. See Section 3.4 for details.

The levels of optimizations described above are performed by the stand-alone optimization pass. The code generator performs several additional optimizations, particularly processor-specific optimizations. It does so regardless of whether you invoke the optimizer. These optimizations are always enabled, although they are more effective when the optimizer is used.

## 3.2 Performing File-Level Optimization (--opt_level=3 option)

The --opt_level=3 option (aliased as the -O3 option) instructs the compiler to perform file-level optimization. You can use the --opt_level=3 option alone to perform general file-level optimization, or you can combine it with other options to perform more specific optimizations. The options listed in Table 3-1 work with --opt_level=3 to perform the indicated optimization:

**Table 3-1. Options That You Can Use With --opt_level=3**

| If You ... | Use this Option | See |
|---|---|---|
| Have files that redeclare standard library functions | --std_lib_func_defined<br>--std_lib_func_redefined | Section 3.2.1 |
| Want to create an optimization information file | --gen_opt_level=*n* | Section 3.2.2 |
| Want to compile multiple source files | --program_level_compile | Section 3.3 |

### 3.2.1 Controlling File-Level Optimization (--std_lib_func_def Options)

When you invoke the compiler with the --opt_level=3 option, some of the optimizations use known properties of the standard library functions. If your file redeclares any of these standard library functions, these optimizations become ineffective. Use Table 3-2 to select the appropriate file-level optimization option.

**Table 3-2. Selecting a File-Level Optimization Option**

| If Your Source File... | Use this Option |
|---|---|
| Declares a function with the same name as a standard library function | --std_lib_func_redefined |
| Contains but does not alter functions declared in the standard library | --std_lib_func_defined |
| Does not alter standard library functions, but you used the --std_lib_func_redefined or --std_lib_func_defined option in a command file or an environment variable. The --std_lib_func_not_defined option restores the default behavior of the optimizer. | --std_lib_func_not_defined |

### 3.2.2 Creating an Optimization Information File (--gen_opt_info Option)

When you invoke the compiler with the --opt_level=3 option, you can use the --gen_opt_info option to create an optimization information file that you can read. The number following the option denotes the level (0, 1, or 2). The resulting file has an .nfo extension. Use Table 3-3 to select the appropriate level to append to the option.

**Table 3-3. Selecting a Level for the --gen_opt_info Option**

| If you... | Use this option |
|---|---|
| Do not want to produce an information file, but you used the --gen_opt_level=1 or --gen_opt_level=2 option in a command file or an environment variable. The --gen_opt_level=0 option restores the default behavior of the optimizer. | --gen_opt_level=0 |
| Want to produce an optimization information file | --gen_opt_level=1 |
| Want to produce a verbose optimization information file | --gen_opt_level=2 |

## 3.3 Performing Program-Level Optimization (--program_level_compile and --opt_level=3 options)

You can specify program-level optimization by using the --program_level_compile option with the --opt_level=3 option (aliased as -O3). With program-level optimization, all of your source files are compiled into one intermediate file called a *module*. The module moves to the optimization and code generation passes of the compiler. Because the compiler can see the entire program, it performs several optimizations that are rarely applied during file-level optimization:

- If a particular argument in a function always has the same value, the compiler replaces the argument with the value and passes the value instead of the argument.
- If a return value of a function is never used, the compiler deletes the return code in the function.
- If a function is not called directly or indirectly by main(), the compiler removes the function.

To see which program-level optimizations the compiler is applying, use the --gen_opt_level=2 option to generate an information file. See Section 3.2.2 for more information.

In Code Composer Studio, when the --program_level_compile option is used, C and C++ files that have the same options are compiled together. However, if any file has a file-specific option that is not selected as a project-wide option, that file is compiled separately. For example, if every C and C++ file in your project has a different set of file-specific options, each is compiled separately, even though program-level optimization has been specified. To compile all C and C++ files together, make sure the files do not have file-specific options. Be aware that compiling C and C++ files together may not be safe if previously you used a file-specific option.

**NOTE: Compiling Files With the --program_level_compile and --keep_asm Options**

If you compile all files with the --program_level_compile and --keep_asm options, the compiler produces only one .asm file, not one for each corresponding source file.

### 3.3.1 Controlling Program-Level Optimization (--call_assumptions Option)

You can control program-level optimization, which you invoke with --program_level_compile --opt_level=3, by using the --call_assumptions option. Specifically, the --call_assumptions option indicates if functions in other modules can call a module's external functions or modify a module's external variables. The number following --call_assumptions indicates the level you set for the module that you are allowing to be called or modified. The --opt_level=3 option combines this information with its own file-level analysis to decide whether to treat this module's external function and variable declarations as if they had been declared static. Use Table 3-4 to select the appropriate level to append to the --call_assumptions option.

**Table 3-4. Selecting a Level for the --call_assumptions Option**

| If Your Module ... | Use this Option |
|---|---|
| Has functions that are called from other modules and global variables that are modified in other modules | --call_assumptions=0 |
| Does not have functions that are called by other modules but has global variables that are modified in other modules | --call_assumptions=1 |
| Does not have functions that are called by other modules or global variables that are modified in other modules | --call_assumptions=2 |
| Has functions that are called from other modules but does not have global variables that are modified in other modules | --call_assumptions=3 |

In certain circumstances, the compiler reverts to a different --call_assumptions level from the one you specified, or it might disable program-level optimization altogether. Table 3-5 lists the combinations of --call_assumptions levels and conditions that cause the compiler to revert to other --call_assumptions levels.

**Table 3-5. Special Considerations When Using the --call_assumptions Option**

| If Your Option is... | Under these Conditions... | Then the --call_assumptions Level... |
|---|---|---|
| Not specified | The --opt_level=3 optimization level was specified | Defaults to --call_assumptions=2 |
| Not specified | The compiler sees calls to outside functions under the --opt_level=3 optimization level | Reverts to --call_assumptions=0 |
| Not specified | Main is not defined | Reverts to --call_assumptions=0 |
| --call_assumptions=1 or --call_assumptions=2 | No function has main defined as an entry point and functions are not identified by the FUNC_EXT_CALLED pragma | Reverts to --call_assumptions=0 |
| --call_assumptions=1 or --call_assumptions=2 | No interrupt function is defined | Reverts to --call_assumptions=0 |
| --call_assumptions=1 or --call_assumptions=2 | Functions are identified by the FUNC_EXT_CALLED pragma | Remains --call_assumptions=1 or --call_assumptions=2 |
| --call_assumptions=3 | Any condition | Remains --call_assumptions=3 |

In some situations when you use --program_level_compile and --opt_level=3, you *must* use a --call_assumptions option or the FUNC_EXT_CALLED pragma. See Section 3.3.2 for information about these situations.

### 3.3.2 Optimization Considerations When Mixing C/C++ and Assembly

If you have any assembly functions in your program, you need to exercise caution when using the --program_level_compile option. The compiler recognizes only the C/C++ source code and not any assembly code that might be present. Because the compiler does not recognize the assembly code calls and variable modifications to C/C++ functions, the --program_level_compile option optimizes out those C/C++ functions. To keep these functions, place the FUNC_EXT_CALLED pragma (see Section 5.9.8) before any declaration or reference to a function that you want to keep.

Another approach you can take when you use assembly functions in your program is to use the --call_assumptions=*n* option with the --program_level_compile and --opt_level=3 options (see Section 3.3.1).

In general, you achieve the best results through judicious use of the FUNC_EXT_CALLED pragma in combination with --program_level_compile --opt_level=3 and --call_assumptions=1 or --call_assumptions=2.

If any of the following situations apply to your application, use the suggested solution:

**Situation—** Your application consists of C/C++ source code that calls assembly functions. Those assembly functions do not call any C/C++ functions or modify any C/C++ variables.

**Solution —** Compile with --program_level_compile --opt_level=3 --call_assumptions=2 to tell the compiler that outside functions do not call C/C++ functions or modify C/C++ variables. See Section 3.3.1 for information about the --call_assumptions=2 option.

If you compile with the --program_level_compile --opt_level=3 options only, the compiler reverts from the default optimization level (--call_assumptions=2) to --call_assumptions=0. The compiler uses --call_assumptions=0, because it presumes that the calls to the assembly language functions that have a definition in C/C++ may call other C/C++ functions or modify C/C++ variables.

**Situation—** Your application consists of C/C++ source code that calls assembly functions. The assembly language functions do not call C/C++ functions, but they modify C/C++ variables.

**Solution—** Try both of these solutions and choose the one that works best with your code:

- Compile with --program_level_compile --opt_level=3 --call_assumptions=1.
- Add the volatile keyword to those variables that may be modified by the assembly functions and compile with --program_level_compile --opt_level=3 --call_assumptions=2.

See Section 3.3.1 for information about the --call_assumptions=*n* option.

**Situation—** Your application consists of C/C++ source code and assembly source code. The assembly functions are interrupt service routines that call C/C++ functions; the C/C++ functions that the assembly functions call are never called from C/C++. These C/C++ functions act like main: they function as entry points into C/C++.

**Solution—** Add the volatile keyword to the C/C++ variables that may be modified by the interrupts. Then, you can optimize your code in one of these ways:

- You achieve the best optimization by applying the FUNC_EXT_CALLED pragma to all of the entry-point functions called from the assembly language interrupts, and then compiling with --program_level_compile --opt_level=3 --call_assumptions=2. *Be sure that you use the pragma with all of the entry-point functions.* If you do not, the compiler might remove the entry-point functions that are not preceded by the FUNC_EXT_CALLED pragma.
- Compile with --program_level_compile --opt_level=3 --call_assumptions=3. Because you do not use the FUNC_EXT_CALLED pragma, you must use the --call_assumptions=3 option, which is less aggressive than the --call_assumptions=2 option, and your optimization may not be as effective.

Keep in mind that if you use --program_level_compile --opt_level=3 without additional options, the compiler removes the C functions that the assembly functions call. Use the FUNC_EXT_CALLED pragma to keep these functions.

## 3.4 Link-Time Optimization (--opt_level=4 Option)

Link-time optimization is an optimization mode that allows the compiler to have visibility of the entire program. The optimization occurs at link-time instead of compile-time like other optimization levels.

Link-time optimization is invoked by using the --opt_level=4 option. This option must be used in both the compilation and linking steps. At compile time, the compiler embeds an intermediate representation of the file being compiled into the resulting object file. At link-time this representation is extracted from every object file which contains it, and is used to optimize the entire program.

Link-time optimization provides the same optimization opportunities as program level optimization (Section 3.3), with the following benefits:

- Each source file can be compiled separately. One issue with program-level compilation is that it requires all source files to be passed to the compiler at one time. This often requires significant modification of a customer's build process. With link-time optimization, all files can be compiled separately.
- References to C/C++ symbols from assembly are handled automatically. When doing program-level compilation, the compiler has no knowledge of whether a symbol is referenced externally. When performing link-time optimization during a final link, the linker can determine which symbols are referenced externally and prevent eliminating them during optimization.
- Third party object files can participate in optimization. If a third party vendor provides object files that were compiled with the --opt_level=4 option, those files participate in optimization along with user-generated files. This includes object files supplied as part of the TI run-time support. Object files that were not compiled with –opt_level=4 can still be used in a link that is performing link-time optimization. Those files that were not compiled with –opt_level=4 do not participate in the optimization.
- Source files can be compiled with different option sets. With program-level compilation, all source files must be compiled with the same option set. With link-time optimization files can be compiled with different options. If the compiler determines that two options are incompatible, it issues an error.

### *3.4.1 Option Handling*

When performing link-time optimization, source files can be compiled with different options. When possible, the options that were used during compilation are used during link-time optimization. For options which apply at the program level, --auto_inline for instance, the options used to compile the main function are used. If main is not included in link-time optimization, the option set used for the first object file specified on the command line is used. Some options, --opt_for_speed for instance, can effect a wide range of optimizations. For these options, the program-level behavior is derived from main, and the local optimizations are obtained from the original option set.

Some options are incompatible when performing link-time optimization. These are usually options which conflict on the command line as well, but can also be options that cannot be handled during link-time optimization.

### 3.4.2 Incompatible Types

During a normal link, the linker does not check to make sure that each symbol was declared with the same type in different files. This is not necessary during a normal link. When performing link-time optimization, however, the linker must ensure that all symbols are declared with compatible types in different source files. If a symbol is found which has incompatible types, an error is issued. The rules for compatible types are derived from the C and C++ standards.

## 3.5 Accessing Aliased Variables in Optimized Code

Aliasing occurs when a single object can be accessed in more than one way, such as when two pointers point to the same object or when a pointer points to a named object. Aliasing can disrupt optimization because any indirect reference can refer to another object. The optimizer analyzes the code to determine where aliasing can and cannot occur, then optimizes as much as possible while still preserving the correctness of the program. The optimizer behaves conservatively. If there is a chance that two pointers are pointing to the same object, then the optimizer assumes that the pointers do point to the same object.

The compiler assumes that if the address of a local variable is passed to a function, the function changes the local variable by writing through the pointer. This makes the local variable's address unavailable for use elsewhere after returning. For example, the called function cannot assign the local variable's address to a global variable or return the local variable's address. In cases where this assumption is invalid, use the --aliased_variables compiler option to force the compiler to assume worst-case aliasing. In worst-case aliasing, any indirect reference can refer to such a variable.

## 3.6 Use Caution With asm Statements in Optimized Code

You must be extremely careful when using asm (inline assembly) statements in optimized code. The compiler rearranges code segments, uses registers freely, and can completely remove variables or expressions. Although the compiler never optimizes out an asm statement (except when it is unreachable), the surrounding environment where the assembly code is inserted can differ significantly from the original C/C++ source code.

It is usually safe to use asm statements to manipulate hardware controls such as interrupt masks, but asm statements that attempt to interface with the C/C++ environment or access C/C++ variables can have unexpected results. After compilation, check the assembly output to make sure your asm statements are correct and maintain the integrity of the program.

## 3.7 Automatic Inline Expansion (--auto_inline Option)

When optimizing with the --opt_level=3 option (aliased as -O3) , the compiler automatically inlines small functions. A command-line option, --auto_inline=*size*, specifies the size threshold . Any function larger than the *size* threshold is not automatically inlined. You can use the --auto_inline=*size* option in the following ways:

- If you set the *size* parameter to 0 (--auto_inline=0), automatic inline expansion is disabled.
- If you set the *size* parameter to a nonzero integer, the compiler uses this size threshold as a limit to the size of the functions it automatically inlines. The compiler multiplies the number of times the function is inlined (plus 1 if the function is externally visible and its declaration cannot be safely removed) by the size of the function.

The compiler inlines the function only if the result is less than the size parameter. The compiler measures the size of a function in arbitrary units; however, the optimizer information file (created with the --gen_opt_level=1 or --gen_opt_level=2 option) reports the size of each function in the same units that the --auto_inline option uses.

The --auto_inline=*size* option controls only the inlining of functions that are not explicitly declared as inline. If you do not use the --auto_inline=*size* option, the compiler inlines very small functions.

**Optimization Level 3 and Inlining**

**NOTE:** In order to turn on automatic inlining, you must use the --opt_level=3 option If you desire the --opt_level=3 optimizations, but not automatic inlining, use --auto_inline=0 with the --opt_level=3 option.

**Inlining and Code Size**

**NOTE:** Expanding functions inline increases code size, especially inlining a function that is called in a number of places. Function inlining is optimal for functions that are called only from a small number of places and for small functions. To prevent increases in code size because of inlining, use the --auto_inline=0 and --no_inlining options. These options, used together, cause the compiler to inline intrinsics only.

## 3.8 Using the Interlist Feature With Optimization

You control the output of the interlist feature when compiling with optimization (the --opt_level=*n* or -O*n* option) with the --optimizer_interlist and --c_src_interlist options.

- The --optimizer_interlist option interlists compiler comments with assembly source statements.
- The --c_src_interlist and --optimizer_interlist options together interlist the compiler comments and the original C/C++ source with the assembly code.

When you use the --optimizer_interlist option with optimization, the interlist feature does *not* run as a separate pass. Instead, the compiler inserts comments into the code, indicating how the compiler has rearranged and optimized the code. These comments appear in the assembly language file as comments starting with ;**. The C/C++ source code is not interlisted, unless you use the --c_src_interlist option also.

The interlist feature can affect optimized code because it might prevent some optimization from crossing C/C++ statement boundaries. Optimization makes normal source interlisting impractical, because the compiler extensively rearranges your program. Therefore, when you use the --optimizer_interlist option, the compiler writes reconstructed C/C++ statements.

Example 3-1 shows a function that has been compiled with optimization (--opt_level=2) and the --optimizer_interlist option. The assembly file contains compiler comments interlisted with assembly code.

**NOTE: Impact on Performance and Code Size**

The --c_src_interlist option can have a negative effect on performance and code size.

When you use the --c_src_interlist and --optimizer_interlist options with optimization, the compiler inserts its comments and the interlist feature runs before the assembler, merging the original C/C++ source into the assembly file.

Example 3-2 shows the function from Example 3-1 compiled with the optimization (--opt_level=2) and the --c_src_interlist and --optimizer_interlist options. The assembly file contains compiler comments and C source interlisted with assembly code.

***Example 3-1. The Function From Example 2-1 Compiled With the -O2 and --optimizer_interlist Options***

```
main:
;* --------------------------------------------------------------------------*
        SUB.W      #2,SP
;** 5   ------------------------   printf((const unsigned char *)"Hello, world\n");
        MOV.W      #$C$SL1+0,0(SP)      ; |5|
        CALL       #printf              ; |5|
                                        ; |5|
;** 6   ------------------------   return 0;
        MOV.W      #0,r12               ; |6|
        ADD.W      #2,SP
        RET
```

***Example 3-2. The Function From Example 2-1 Compiled with the --opt_level=2, --optimizer_interlist, and --c_src_interlist Options***

```
main:
;* --------------------------------------------------------------------------*
        SUB.W      #2,SP
;** 5   ------------------------   printf((const unsigned char *)"Hello, world\n");
;----------------------------------------------------------------------------
;   5 | printf ("Hello, world\n");
;----------------------------------------------------------------------------
        MOV.W      #$C$SL1+0,0(SP)      ; |5|
        CALL       #printf              ; |5|
                                        ; |5|
;** 6   ------------------------   return 0;
;----------------------------------------------------------------------------
;   6 | return 0;
;----------------------------------------------------------------------------
        MOV.W      #0,r12               ; |6|
        ADD.W      #2,SP
        RET
```

## 3.9 Debugging Optimized Code

Debugging fully optimized code is not recommended, because the compiler's extensive rearrangement of code and the many-to-many allocation of variables to registers often make it difficult to correlate source code with object code. Profiling code that has been built with the --symdebug:dwarf (aliased as -g) option or the --symdebug:coff option (STABS debug) is not recommended either, because these options can significantly degrade performance. To remedy these problems, you can use the options described in the following sections to optimize your code in such a way that you can still debug or profile the code.

To debug optimized code, use the --opt_level option (aliased as -O) in conjunction with one of the symbolic debugging options (--symdebug:dwarf or --symdebug:coff). The symbolic debugging options generate directives that are used by the C/C++ source-level debugger, but they disable many compiler optimizations. When you use the --opt_level option (which invokes optimization) with the --symdebug:dwarf or --symdebug:coff option, you turn on the maximum amount of optimization that is compatible with debugging.

If you want to use symbolic debugging and still generate fully optimized code, use the --optimize_with_debug option. This option reenables the optimizations disabled by --symdebug:dwarf or --symdebug:coff. However, if you use the --optimize_with_debug option, portions of the debugger's functionality will be unreliable.

**NOTE: Symbolic Debugging Options Affect Performance and Code Size**

Using the --symdebug:dwarf or --symdebug:coff option can cause a significant performance and code size degradation of your code. Use these options for debugging only. Using --symdebug:dwarf or --symdebug:coff when profiling is not recommended.

## 3.10 Controlling Code Size Versus Speed

The latest mechanism for controlling the goal of optimizations in the compiler is represented by the --opt_for_speed=*num* option. The *num* denotes the level of optimization (0-5), which controls the type and degree of code size or code speed optimization:

- --opt_for_speed=0
  Enables optimizations geared towards improving the code size with a *high* risk of worsening or impacting performance.
- --opt_for_speed=1
  Enables optimizations geared towards improving the code size with a *medium* risk of worsening or impacting performance.
- --opt_for_speed=2
  Enables optimizations geared towards improving the code size with a *low* risk of worsening or impacting performance.
- --opt_for_speed=3
  Enables optimizations geared towards improving the code performance/speed with a *low* risk of worsening or impacting code size.
- --opt_for_speed=4
  Enables optimizations geared towards improving the code performance/speed with a *medium* risk of worsening or impacting code size.
- --opt_for_speed=5
  Enables optimizations geared towards improving the code performance/speed with a *high* risk of worsening or impacting code size.

If you specify the option without a parameter, the default setting is --opt_for_speed=4. However, the default behavior of the compiler is as if --opt_for_speed=1 were specified.

## 3.11 What Kind of Optimization Is Being Performed?

The MSP430 C/C++ compiler uses a variety of optimization techniques to improve the execution speed of your C/C++ programs and to reduce their size.

Following are some of the optimizations performed by the compiler:

| Optimization | See |
|---|---|
| Cost-based register allocation | Section 3.11.1 |
| Alias disambiguation | Section 3.11.1 |
| Branch optimizations and control-flow simplification | Section 3.11.3 |
| Data flow optimizations<br>• Copy propagation<br>• Common subexpression elimination<br>• Redundant assignment elimination | Section 3.11.4 |
| Expression simplification | Section 3.11.5 |
| Inline expansion of functions | Section 3.11.6 |
| Induction variable optimizations and strength reduction | Section 3.11.7 |
| Loop-invariant code motion | Section 3.11.8 |
| Loop rotation | Section 3.11.9 |
| Instruction scheduling | Section 3.11.10 |

| MSP430 -Specific Optimization | See |
|---|---|
| Tail merging | Section 3.11.11 |
| Integer division with constant divisor | Section 3.11.12 |
| _never_executed() intrinsic | Section 3.11.13 |

### 3.11.1 Cost-Based Register Allocation

The compiler, when optimization is enabled, allocates registers to user variables and compiler temporary values according to their type, use, and frequency. Variables used within loops are weighted to have priority over others, and those variables whose uses do not overlap can be allocated to the same register.

Induction variable elimination and loop test replacement allow the compiler to recognize the loop as a simple counting loop and software pipeline, unroll, or eliminate the loop. Strength reduction turns the array references into efficient pointer references with autoincrements.

### 3.11.2 Alias Disambiguation

C and C++ programs generally use many pointer variables. Frequently, compilers are unable to determine whether or not two or more I values (lowercase L: symbols, pointer references, or structure references) refer to the same memory location. This aliasing of memory locations often prevents the compiler from retaining values in registers because it cannot be sure that the register and memory continue to hold the same values over time.

Alias disambiguation is a technique that determines when two pointer expressions cannot point to the same location, allowing the compiler to freely optimize such expressions.

### 3.11.3 Branch Optimizations and Control-Flow Simplification

The compiler analyzes the branching behavior of a program and rearranges the linear sequences of operations (basic blocks) to remove branches or redundant conditions. Unreachable code is deleted, branches to branches are bypassed, and conditional branches over unconditional branches are simplified to a single conditional branch.

When the value of a condition is determined at compile time (through copy propagation or other data flow analysis), the compiler can delete a conditional branch. Switch case lists are analyzed in the same way as conditional branches and are sometimes eliminated entirely. Some simple control flow constructs are reduced to conditional instructions, totally eliminating the need for branches.

### 3.11.4 Data Flow Optimizations

Collectively, the following data flow optimizations replace expressions with less costly ones, detect and remove unnecessary assignments, and avoid operations that produce values that are already computed. The compiler with optimization enabled performs these data flow optimizations both locally (within basic blocks) and globally (across entire functions).

- **Copy propagation.** Following an assignment to a variable, the compiler replaces references to the variable with its value. The value can be another variable, a constant, or a common subexpression. This can result in increased opportunities for constant folding, common subexpression elimination, or even total elimination of the variable.
- **Common subexpression elimination.** When two or more expressions produce the same value, the compiler computes the value once, saves it, and reuses it.
- **Redundant assignment elimination.** Often, copy propagation and common subexpression elimination optimizations result in unnecessary assignments to variables (variables with no subsequent reference before another assignment or before the end of the function). The compiler removes these dead assignments.

### 3.11.5 Expression Simplification

For optimal evaluation, the compiler simplifies expressions into equivalent forms, requiring fewer instructions or registers. Operations between constants are folded into single constants. For example, a = (b + 4) - (c + 1) becomes a = b - c + 3.

### 3.11.6 Inline Expansion of Functions

The compiler replaces calls to small functions with inline code, saving the overhead associated with a function call as well as providing increased opportunities to apply other optimizations.

### 3.11.7 Induction Variables and Strength Reduction

Induction variables are variables whose value within a loop is directly related to the number of executions of the loop. Array indices and control variables for loops are often induction variables.

Strength reduction is the process of replacing inefficient expressions involving induction variables with more efficient expressions. For example, code that indexes into a sequence of array elements is replaced with code that increments a pointer through the array.

Induction variable analysis and strength reduction together often remove all references to your loop-control variable, allowing its elimination.

### 3.11.8 Loop-Invariant Code Motion

This optimization identifies expressions within loops that always compute to the same value. The computation is moved in front of the loop, and each occurrence of the expression in the loop is replaced by a reference to the precomputed value.

### 3.11.9 Loop Rotation

The compiler evaluates loop conditionals at the bottom of loops, saving an extra branch out of the loop. In many cases, the initial entry conditional check and the branch are optimized out.

### 3.11.10 Instruction Scheduling

The compiler performs instruction scheduling, which is the rearranging of machine instructions in such a way that improves performance while maintaining the semantics of the original order. Instruction scheduling is used to improve instruction parallelism and hide pipeline latencies. It can also be used to reduce code size.

### 3.11.11 Tail Merging

If you are optimizing for code size, tail merging can be very effective for some functions. Tail merging finds basic blocks that end in an identical sequence of instructions and have a common destination. If such a set of blocks is found, the sequence of identical instructions is made into its own block. These instructions are then removed from the set of blocks and replaced with branches to the newly created block. Thus, there is only one copy of the sequence of instructions, rather than one for each block in the set.

### 3.11.12 Integer Division With Constant Divisor

The optimizer attempts to rewrite integer divide operations with constant divisors. The integer divides are rewritten as a multiply with the reciprocal of the divisor. This occurs at optimization level 2 (--opt_level=2 or -O2) and higher. You must also compile with the --opt_for_speed option, which selects compile for speed.

### 3.11.13 _never_executed Intrinsic

The _never_executed( )intrinsic can be used to assert to the compiler that a switch expression can only take on values represented by the case labels within a switch block. This assertion enables the compiler to avoid generating test code for handling values not specified by the switch case labels. This assertion is specifically suited for handling values that characterize a vector generator. See Section 6.7.3 for details on the _never_executed( ) intrinsic.

# Linking C/C++ Code

The C/C++ compiler and assembly language tools provide two methods for linking your programs:

- You can compile individual modules and link them together. This method is especially useful when you have multiple source files.
- You can compile and link in one step. This method is useful when you have a single source module.

This chapter describes how to invoke the linker with each method. It also discusses special requirements of linking C/C++ code, including the run-time-support libraries, specifying the type of initialization, and allocating the program into memory. For a complete description of the linker, see the *MSP430 Assembly Language Tools User's Guide*.

**Topic** **Page**

## 4.1 Invoking the Linker Through the Compiler (-z Option)

This section explains how to invoke the linker after you have compiled and assembled your programs: as a separate step or as part of the compile step.

### 4.1.1 Invoking the Linker Separately

This is the general syntax for linking C/C++ programs as a separate step:

**cl430 --run_linker** {**--rom_model | --ram_model**} *filenames*
[*options*] [**--output_file=** *name.out*] **--library=** *library* [*lnk.cmd*]

| | |
|---|---|
| **cl430 --run_linker** | The command that invokes the linker. |
| **--rom_model** \| **--ram_model** | Options that tell the linker to use special conventions defined by the C/C++ environment. When you use cl430 --run_linker, you must use **--rom_model** or **--ram_model**. The --rom_model option uses automatic variable initialization at run time; the --ram_model option uses variable initialization at load time. |
| *filenames* | Names of object files, linker command files, or archive libraries. The default extension for all input files is *.obj*; any other extension must be explicitly specified. The linker can determine whether the input file is an object or ASCII file that contains linker commands. The default output filename is *a.out*, unless you use the --output_file option to name the output file. |
| *options* | Options affect how the linker handles your object files. Linker options can only appear after the **--run_linker** option on the command line, but otherwise may be in any order. (Options are discussed in detail in the *MSP430 Assembly Language Tools User's Guide*.) |
| **--output_file=** *name.out* | Names the output file. |
| **--library=** *library* | Identifies the appropriate archive library containing C/C++ run-time-support and floating-point math functions, or linker command files. If you are linking C/C++ code, you must use a run-time-support library. You can use the libraries included with the compiler, or you can create your own run-time-support library. If you have specified a run-time-support library in a linker command file, you do not need this parameter. The --library option's short form is -l. |
| *lnk.cmd* | Contains options, filenames, directives, or commands for the linker. |

When you specify a library as linker input, the linker includes and links only those library members that resolve undefined references. The linker uses a default allocation algorithm to allocate your program into memory. You can use the MEMORY and SECTIONS directives in the linker command file to customize the allocation process. For information, see the *MSP430 Assembly Language Tools User's Guide*.

You can link a C/C++ program consisting of modules prog1.obj, prog2.obj, and prog3.obj, with an executable filename of prog.out with the command:

```
cl430 --run_linker --rom_model prog1 prog2 prog3 --output_file=prog.out
     --library=rts430.lib
```

### 4.1.2 Invoking the Linker as Part of the Compile Step

This is the general syntax for linking C/C++ programs as part of the compile step:

**cl430***filenames* [*options*] **--run_linker** {**--rom_model | --ram_model**} *filenames*
[*options*] [**--output_file=** *name.out*] **--library=** *library* [*lnk.cmd*]

The **--run_linker** option divides the command line into the compiler options (the options before --run_linker) and the linker options (the options following --run_linker). The --run_linker option must follow all source files and compiler options on the command line.

All arguments that follow --run_linker on the command line are passed to the linker. These arguments can be linker command files, additional object files, linker options, or libraries. These arguments are the same as described in Section 4.1.1.

All arguments that precede --run_linker on the command line are compiler arguments. These arguments can be C/C++ source files, assembly files, or compiler options. These arguments are described in Section 2.2.

You can compile and link a C/C++ program consisting of modules prog1.c, prog2.c, and prog3.c, with an executable filename of prog.out with the command:

```
cl430 prog1.c prog2.c prog3.c --run_linker --rom_model --output_file=prog.out --library=rts430.lib
```

**NOTE: Order of Processing Arguments in the Linker**

The order in which the linker processes arguments is important. The compiler passes arguments to the linker in the following order:

1. Object filenames from the command line
2. Arguments following the --run_linker option on the command line
3. Arguments following the --run_linker option from the MSP430_C_OPTION environment variable

### 4.1.3 Disabling the Linker (--compile_only Compiler Option)

You can override the --run_linker option by using the --compile_only compiler option. The -run_linker option's short form is -z and the --compile_only option's short form is -c.

The --compile_only option is especially helpful if you specify the --run_linker option in the MSP430_C_OPTION environment variable and want to selectively disable linking with the --compile_only option on the command line.

## 4.2 Linker Code Optimizations

These options are used to further optimize your code.

### 4.2.1 Generate List of Dead Functions (--generate_dead_funcs_list Option)

In order to facilitate the removal of unused code, the linker generates a feedback file containing a list of functions that are never referenced. The feedback file must be used the next time you compile the source files. The syntax for the --generate_dead_funcs_list option is:

**--generate_dead_funcs_list=** *filename*

If *filename* is not specified, a default filename of dead_funcs.txt is used.

Proper creation and use of the feedback file entails the following steps:

1. Compile all source files using the --gen_func_subsections compiler option. For example:
   ```
   cl430 file1.c file2.c --gen_func_subsections
   ```
2. During the linker, use the --generate_dead_funcs_list option to generate the feedback file based on the generated object files. For example:
   ```
   cl430 --run_linker file1.obj file2.obj --generate_dead_funcs_list=feedback.txt
   ```
   Alternatively, you can combine steps 1 and 2 into one step. When you do this, you are not required to specify --gen_func_subsections when compiling the source files as this is done for you automatically. For example:
   ```
   cl430 file1.c file2.c --run_linker --generate_dead_funcs_list=feedback.txt
   ```
3. Once you have the feedback file, rebuild the source. Give the feedback file to the compiler using the --use_dead_funcs_list option. This option forces each dead function listed in the file into its own subsection. For example:
   ```
   cl430 file1.c file2.c --use_dead_funcs_list=feedback.txt
   ```
4. Invoke the linker with the newly built object files. The linker removes the subsections. For example:
   ```
   cl430 --run_linker file1.obj file2.obj
   ```
   Alternatively, you can combine steps 3 and 4 into one step. For example:
   ```
   cl430 file1.c file2.c --use_dead_funcs_list=feedback.txt --run_linker
   ```

**NOTE: Dead Functions Feedback**

The feedback file generated with the -gen_dead_funcs_list option is version controlled. It must be generated by the linker in order to be processed correctly by the compiler.

### 4.2.2 Generating Function Subsections (--gen_func_subsections Compiler Option)

When the linker places code into an executable file, it allocates all the functions in a single source file as a group. This means that if any function in a file needs to be linked into an executable, then all the functions in the file are linked in. This can be undesirable if a file contains many functions and only a few are required for an executable.

This situation may exist in libraries where a single file contains multiple functions, but the application only needs a subset of those functions. An example is a library .obj file that contains a signed divide routine and an unsigned divide routine. If the application requires only signed division, then only the signed divide routine is required for linking. By default, both the signed and unsigned routines are linked in since they exist in the same .obj file.

The --gen_func_subsections compiler option remedies this problem by placing each function in a file in its own subsection. Thus, only the functions that are referenced in the application are linked into the final executable. This can result in an overall code size reduction.

## 4.3 Controlling the Linking Process

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

- Include the compiler's run-time-support library
- Specify the type of boot-time initialization
- Determine how you want to allocate your program into memory

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

For more information about how to operate the linker, see the linker description in the *MSP430 Assembly Language Tools User's Guide*

### *4.3.1 Including the Run-Time-Support Library*

You must include a run-time-support library in the linker process. The following sections describe two methods for including the run-time-support library.

#### 4.3.1.1 Manual Run-Time-Support Library Selection

You must link all C/C++ programs with a run-time-support library. The library contains standard C/C++ functions as well as functions used by the compiler to manage the C/C++ environment. You must use the --library linker option to specify which MSP430 run-time-support library to use. The --library option also tells the linker to look at the --search_path options and then the MSP430_C_DIR environment variable to find an archive path or object file. To use the --library linker option, type on the command line:

**cl430 --run_linker** {**--rom_model** | **--ram_model**} *filenames* **--library=** *libraryname*

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

By default, if a library introduces an unresolved reference and multiple libraries have a definition for it, then the definition from the same library that introduced the unresolved reference is used. Use the --priority option if you want the linker to use the definition from the first library on the command line that contains the definition.

#### 4.3.1.2 Automatic Run-Time-Support Library Selection

If the --rom_model or --ram_model option is specified during the linker and the entry point for the program (normally c_int00) is not resolved by any specified object file or library, the linker attempts to automatically include the best compatible run-time-support library for your program. The chosen run-time-support library is linked in as if it was specified with the --library option last on the command line. Alternatively, you can always force the linker to choose an appropriate run-time-support library by specifying “libc.a” as an argument to the --library option, or when specifying the run-time-support library name explicitly in a linker command file.

The automatic selection of a run-time-support library can be disabled with the --disable_auto_rts option.

If the --issue_remarks option is specified before the --run_linker option during the linker, a remark is generated indicating which run-time support library was linked in. If a different run-time-support library is desired, you must specify the name of the desired run-time-support library using the --library option and in your linker command files when necessary.

For example:

```
cl430 --issue_remarks main.c --run_linker --rom_model

<Linking>

remark: linking in "libc.a"

remark: linking in "rts430.lib" in place of "libc.a"
```

### 4.3.2 Run-Time Initialization

You must link all C/C++ programs with code to initialize and execute the program called a bootstrap routine. The bootstrap routine is responsible for the following tasks:

- Set up the stack
- Process the .cinit run-time initialization table to autoinitialize global variables (when using the --rom_model option)
- Call all global constructors (.pinit) for C++
- Call main.com
- Call exit when main returns

A sample bootstrap routine is _c_int00, provided in boot.obj in the run-time support object libraries. The entry point is usually set to the starting address of the bootstrap routine.

**NOTE: The _c_int00 Symbol**

If you use the --ram_model or --rom_model link option, _c_int00 is automatically defined as the entry point for the program.

### 4.3.3 Initialization by the Interrupt Vector

If your program begins running from load time, you must set up the reset vector to branch to _c_int00. This causes boot.obj to be loaded from the library and your program is initialized correctly. The boot.obj places the address of _c_int00 into a section named .reset. This section can then be allocated at the reset vector location using the linker.

### 4.3.4 Initialization of the FRAM Memory Protection Unit

The linker supports initialization of the FRAM memory protection unit (MPU). The linker uses a boot routine that performs MPU initialization based on the definition of certain symbols. The TI provided linker command files that are used by default for different devices define the necessary symbols so MPU initialization happens automatically. Code and data sections are automatically given the correct access permissions. If you want to manually adjust how the MPU is initialized you can modify the __mpuseg and __mpusam definitions in the linker command file. The MPU-specific boot routine is used when these two symbols are defined and it sets the value of the MPUSEG and MPUSAM registers based on these values. If you do not want the MPU initialized you can remove these definitions from the linker command file.

### 4.3.5 Global Object Constructors

Global C++ variables that have constructors and destructors require their constructors to be called during program initialization and their destructors to be called during program termination. The C++ compiler produces a table of constructors to be called at startup.

Constructors for global objects from a single module are invoked in the order declared in the source code, but the relative order of objects from different object files is unspecified.

Global constructors are called after initialization of other global variables and before main( ) is called. Global destructors are invoked during exit( ), similar to functions registered through atexit( ).

Section 6.8.7 discusses the format of the global constructor table.

### 4.3.6 Specifying the Type of Global Variable Initialization

The C/C++ compiler produces data tables for initializing global variables. Section 6.8.4 discusses the format of these initialization tables. The initialization tables are used in one of the following ways:

- Global variables are initialized at *run time*. Use the --rom_model linker option (see Section 6.8.5).
- Global variables are initialized at *load time*. Use the --ram_model linker option (see Section 6.8.6).

When you link a C/C++ program, you must use either the --rom_model or --ram_model option. These options tell the linker to select initialization at run time or load time.

When you compile and link programs, the --rom_model option is the default. If used, the --rom_model option must follow the --run_linker option (see Section 4.1). The following list outlines the linking conventions used with --rom_model or --ram_model:

- The symbol _c_int00 is defined as the program entry point; it identifies the beginning of the C/C++ boot routine in boot.obj. When you use --rom_model or --ram_model, _c_int00 is automatically referenced, ensuring that boot.obj is automatically linked in from the run-time-support library.
- The initialization output section is padded with a termination record so that the loader (load-time initialization) or the boot routine (run-time initialization) knows when to stop reading the initialization tables.
- The global constructor output section is padded with a termination record.
- When initializing at load time (the --ram_model option), the following occur:
  - The linker sets the initialization table symbol to -1. This indicates that the initialization tables are not in memory, so no initialization is performed at run time.
  - The STYP_COPY flag is set in the initialization table section header. STYP_COPY is the special attribute that tells the loader to perform autoinitialization directly and not to load the initialization table into memory. The linker does not allocate space in memory for the initialization table.
- When autoinitializing at run time (--rom_model option), the linker defines the initialization table symbol as the starting address of the initialization table. The boot routine uses this symbol as the starting point for autoinitialization.
- The linker defines the starting address of the global constructor table. The boot routine uses this symbol as the beginning of the table of global constructors.

**NOTE: Boot Loader**

A loader is not included as part of the C/C++ compiler tools.

### 4.3.7 Specifying Where to Allocate Sections in Memory

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

The compiler creates two basic kinds of sections: initialized and uninitialized. Table 4-1 summarizes the initialized sections. Table 4-2 summarizes the uninitialized sections.

**Table 4-1. Initialized Sections Created by the Compiler**

| Name | Contents |
|---|---|
| .cinit | Tables for explicitly initialized global and static variables |
| .const | Global and static const variables that are explicitly initialized and contain string literals |
| .econst | |
| .pinit | Table of constructors to be called at startup |
| .text | Executable code and constants |

**Table 4-2. Uninitialized Sections Created by the Compiler**

| Name | Contents |
|---|---|
| .args | Linker-created section used to pass arguments from the command line of the loader to the program |
| .bss | Global and static variables |
| .stack | Stack |
| .sysmem | Memory for malloc functions (heap) |

When you link your program, you must specify where to allocate the sections in memory. In general, initialized sections are linked into ROM or RAM; uninitialized sections are linked into RAM. With the exception of .text, the initialized and uninitialized sections created by the compiler cannot be allocated into internal program memory. See Section 6.1.4 for a complete description of how the compiler uses these sections.

The linker provides MEMORY and SECTIONS directives for allocating sections. For more information about allocating sections into memory, see the *MSP430 Assembly Language Tools User's Guide*.

### 4.3.8 A Sample Linker Command File

Example 4-1 shows a typical linker command file that links a 32-bit C program. The command file in this example is named lnk32.cmd and lists several link options:

| | |
|---|---|
| **−−rom_model** | Tells the linker to use autoinitialization at run time |
| **--stack_size** | Tells the linker to set the C stack size at 0x140 bytes |
| **--heap_size** | Tells the linker to set the heap size to 0x120 bytes |
| **--library** | Tells the linker to use an archive library file, rts430.lib |

To link the program, enter:

**cl430 --run_linker** *object_file(s)* **--output_file=** *file* **--map_file=** *file* **lnk.cmd**

### Example 4-1. Linker Command File

```
--rom_model
--stack_size=0x0140
--heap_size=0x120
--library=rts430.lib

/*****************************************************************************/
/* SPECIFY THE SYSTEM MEMORY MAP                                             */
/*****************************************************************************/

MEMORY
{
    SFR(R) : origin = 0x0000, length = 0x0010
    PERIPHERALS_8BIT : origin = 0x0010, length = 0x00F0
    PERIPHERALS_16BIT: origin = 0x0100, length = 0x0100
    RAM(RW) : origin = 0x0200, length = 0x0800
    INFOA : origin = 0x1080, length = 0x0080
    INFOB : origin = 0x1000, length = 0x0080
    FLASH : origin = 0x1100, length = 0xEEE0
    VECTORS(R) : origin = 0xFFE0, length = 0x001E
    RESET : origin = 0xFFFE, length = 0x0002
}
/*****************************************************************************/
/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY                               */
/*****************************************************************************/

SECTIONS
{
    .bss     : {} > RAM               /* GLOBAL & STATIC VARS              */
    .sysmem  : {} > RAM               /* DYNAMIC MEMORY ALLOCATION AREA    */
    .stack   : {} > RAM               /* SOFTWARE SYSTEM STACK             */
    .text    : {} > FLASH             /* CODE                              */
    .cinit   : {} > FLASH             /* INITIALIZATION TABLES             */
    .const   : {} > FLASH             /* CONSTANT DATA                     */
    .cio     : {} > RAM               /* C I/O BUFFER                      */
    .pinit   : {} > RAM               /* C++ CONSTRUCTOR TABLES            */
    .intvecs : {} > VECTORS           /* MSP430 INTERRUPT VECTORS          */
    .reset   : {} > RESET             /* MSP430 RESET VECTOR               */
```

# MSP430 C/C++ Language Implementation

The C/C++ compiler supports the C/C++ language standard that was developed by a committee of the American National Standards Institute (ANSI) and subsequently adopted by the International Standards Organization (IS0).

The C++ language supported by the MSP430 is defined by the ANSI/ISO/IEC 14882:1998 standard with certain exceptions.

**Topic** **Page**

## 5.1 Characteristics of MSP430 C

The compiler supports the C language as defined by ISO/IEC 9899:1990, which is equivalent to American National Standard for Information Systems-Programming Language C X3.159-1989 standard, commonly referred to as C89, published by the American National Standards Institute. The compiler can also accept many of the language extensions found in the GNU C compiler (see Section 5.14). The compiler does not support C99.

The ANSI/ISO standard identifies some features of the C language that are affected by characteristics of the target processor, run-time environment, or host environment. For reasons of efficiency or practicality, this set of features can differ among standard compilers.

Unsupported features of the C library are:

- The run-time library has minimal support for wide and multi-byte characters. The type wchar_t is implemented as int. The wide character set is equivalent to the set of values of type char. The library includes the header files <wchar.h> and <wctype.h>, but does not include all the functions specified in the standard. So-called multi-byte characters are limited to single characters. There are no shift states. The mapping between multi-byte characters and wide characters is simple equivalence; that is, each wide character maps to and from exactly a single multi-byte character having the same value.
- The run-time library includes the header file <locale.h>, but with a minimal implementation. The only supported locale is the C locale. That is, library behavior that is specified to vary by locale is hard-coded to the behavior of the C locale, and attempting to install a different locale by way of a call to setlocale() will return NULL.

## 5.2 Characteristics of MSP430 C++

The MSP430 compiler supports C++ as defined in the ANSI/ISO/IEC 14882:1998 standard, including these features:

- Complete C++ standard library support, with exceptions noted below.
- Templates
- Exceptions, which are enabled with the --exceptions option; see Section 5.6.
- Run-time type information (RTTI), which can be enabled with the --rtti compiler option.

The *exceptions* to the standard are as follows:

- The <complex> header and its functions are not included in the library.
- The library supports wide chars, in that template functions and classes that are defined for char are also available for wide char. For example, wide char stream classes wios, wiostream, wstreambuf and so on (corresponding to char classes ios, iostream, streambuf) are implemented. However, there is no low-level file I/O for wide chars. Also, the C library interface to wide char support (through the C++ headers <cwchar> and <cwctype>) is limited as described above in the C library.
- If the definition of an inline function contains a static variable, and it appears in multiple compilation units (usually because it's a member function of a class defined in a header file), the compiler generates multiple copies of the static variable rather than resolving them to a single definition. The compiler emits a warning (#1369) in such cases.
- The reinterpret_cast type does not allow casting a pointer-to-member of one class to a pointer-to-member of another class if the classes are unrelated.
- Two-phase name binding in templates, as described in [tesp.res] and [temp.dep] of the standard, is not implemented.
- The export keyword for templates is not implemented.
- A typedef of a function type cannot include member function cv-qualifiers.
- A partial specialization of a class member template cannot be added outside of the class definition.

## 5.3 Using MISRA-C:2004

You can alter your code to work with the MISRA-C:2004 rules. The following enable/disable the rules:

- The --check_misra option enables checking of the specified MISRA-C:2004 rules.
- The CHECK_MISRA pragma enables/disables MISRA-C:2004 rules at the source level. This pragma is equivalent to using the --check_misra option. SeeSection 5.9.2.
- RESET_MISRA pragma resets the specified MISRA-C:2004 rules to the state they were before any CHECK_MISRA pragmas were processed. See Section 5.9.15.

The syntax of the option and pragmas are:

**--check_misra=**{all|required|advisory|none|*rulespec*}
**#pragma CHECK_MISRA ("**{all|required|advisory|none|*rulespec*}**");**
**#pragma RESET_MISRA ("**{all|required|advisory|*rulespec*}**");**

The *rulespec* parameter is a comma-separated list of these specifiers:

| | |
|---|---|
| -[-]X | Enable (or disable) all rules in topic X. |
| -[-]X-Z | Enable (or disable) all rules in topics X through Z. |
| -[-]X.A | Enable (or disable) rule A in topic X. |
| -[-]X.A-C | Enable (or disable) rules A through C in topic X. |

Example: --check_misra=1-5,-1.1,7.2-4

- Checks topics 1 through 5
- Disables rule 1.1 (all other rules from topic 1 remain enabled)
- Checks rules 2 through 4 in topic 7

Two options control the severity of certain MISRA-C:2004 rules:

- The --misra_required option sets the diagnostic severity for required MISRA-C:2004 rules.
- The --misra_advisory option sets the diagnostic severity for advisory MISRA-C:2004 rules.

The syntax for these options is:

**--misra_advisory=**{error|warning|remark|suppress}
**--misra_required=**{error|warning|remark|suppress}

## 5.4 Data Types

Table 5-1 lists the size, representation, and range of each scalar data type for the MSP430 compiler. Many of the range values are available as standard macros in the header file limits.h.

**Table 5-1. MSP430 C/C++ Data Types**

| | | | Range | |
|---|---|---|---|---|
| **Type** | **Size** | **Representation** | **Minimum** | **Maximum** |
| char, signed char | 8 bits | ASCII | -128 | -127 |
| unsigned char, bool | 8 bits | ASCII | 0 | 255 |
| short, signed short | 16 bits | 2s complement | -32 768 | 32 767 |
| unsigned short, wchar_t | 16 bits | Binary | 0 | 65 535 |
| int, signed int | 16 bits | 2s complement | -32 768 | 32 767 |
| unsigned int | 16 bits | Binary | 0 | 65 535 |
| long, signed long | 32 bits | 2s complement | -2 147 483 648 | 2 147 483 647 |
| unsigned long | 32 bits | Binary | 0 | 4 294 967 295 |
| enum | 16 bits | 2s complement | -32 768 | 32 767 |
| float | 32 bits | IEEE 32-bit | 1.175 495e-38(1) | 3.40 282 35e+38 |
| double | 32 bits | IEEE 32-bit | 1.175 495e-38(1) | 3.40 282 35e+38 |
| long double | 32 bits | IEEE 32-bit | 1.175 495e-308(1) | 3.40 282 35e+38 |
| pointers, references, pointer to data members | 16 bits | Binary | 0 | 0xFFFF |
| MSP430X large-data model pointers, references, pointer to data members (2) | 20 bits | Binary | 0 | 0xFFFFF |
| MSP430 function pointers | 16 bits | Binary | 0 | 0xFFFF |
| MSP430X function pointers(3) | 20 bits | Binary | 0 | 0xFFFFF |

(1) Figures are minimum precision.
(2) MSP430X large-data model is specified by --silicon_version=mspx --data_model=large
(3) MSP430X devices are specified by --silicon_version=mspx

## 5.5 Keywords

The MSP430 C/C++ compiler supports the standard const, restrict, and volatile keywords. In addition, the C/C++ compiler extends the C/C++ language through the support of the interrupt keyword.

### 5.5.1 The const Keyword

The C/C++ compiler supports the ANSI/ISO standard keyword *const*. This keyword gives you greater optimization and control over allocation of storage for certain data objects. You can apply the const qualifier to the definition of any variable or array to ensure that its value is not altered.

If you define an object as const, the .const section allocates storage for the object. The const data storage allocation rule has two exceptions:

- If the keyword volatile is also specified in the definition of an object (for example, volatile const int x). Volatile keywords are assumed to be allocated to RAM. (The program does not modify a const volatile object, but something external to the program might.)
- If the object has automatic storage (function scope).

In both cases, the storage for the object is the same as if the const keyword were not used.

The placement of the const keyword within a definition is important. For example, the first statement below defines a constant pointer p to a variable int. The second statement defines a variable pointer q to a constant int:

```
int * const p = &x;
const int * q = &x;
```

Using the const keyword, you can define large constant tables and allocate them into system ROM. For example, to allocate a ROM table, you could use the following definition:

```
const int digits[] = {0,1,2,3,4,5,6,7,8,9};
```

### 5.5.2 The interrupt Keyword

The compiler extends the C/C++ language by adding the interrupt keyword, which specifies that a function is treated as an interrupt function.

Functions that handle interrupts follow special register-saving rules and a special return sequence. The implementation stresses safety. The interrupt routine does not assume that the C run-time conventions for the various CPU register and status bits are in effect; instead, it re-establishes any values assumed by the run-time environment. When C/C++ code is interrupted, the interrupt routine must preserve the contents of all machine registers that are used by the routine or by any function called by the routine. When you use the interrupt keyword with the definition of the function, the compiler generates register saves based on the rules for interrupt functions and the special return sequence for interrupts.

You can only use the interrupt keyword with a function that is defined to return void and that has no parameters. The body of the interrupt function can have local variables and is free to use the stack or global variables. For example:

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

The name c_int00 is the C/C++ entry point. This name is reserved for the system reset interrupt. This special interrupt routine initializes the system and calls the function main. Because it has no caller, c_int00 does not save any registers.

Use the alternate keyword, __interrupt, if you are writing code for strict ANSI/ISO mode (using the --strict_ansi compiler option).

**HWI Objects and the interrupt Keyword**

**NOTE:** The interrupt keyword must not be used when BIOS HWI objects are used in conjunction with C functions. The HWI_enter/HWI_exit macros and the HWI dispatcher contain this functionality, and the use of the C modifier can cause negative results.

### 5.5.3 The restrict Keyword

To help the compiler determine memory dependencies, you can qualify a pointer, reference, or array with the restrict keyword. The restrict keyword is a type qualifier that can be applied to pointers, references, and arrays. Its use represents a guarantee by you, the programmer, that within the scope of the pointer declaration the object pointed to can be accessed only by that pointer. Any violation of this guarantee renders the program undefined. This practice helps the compiler optimize certain sections of code because aliasing information can be more easily determined.

In Example 5-1, the restrict keyword is used to tell the compiler that the function func1 is never called with the pointers a and b pointing to objects that overlap in memory. You are promising that accesses through a and b will never conflict; therefore, a write through one pointer cannot affect a read from any other pointers. The precise semantics of the restrict keyword are described in the 1999 version of the ANSI/ISO C Standard.

***Example 5-1. Use of the restrict Type Qualifier With Pointers***

```
void func1(int * restrict a, int * restrict b)
{
  /* func1's code here */
}
```

Example 5-2 illustrates using the restrict keyword when passing arrays to a function. Here, the arrays c and d should not overlap, nor should c and d point to the same array.

***Example 5-2. Use of the restrict Type Qualifier With Arrays***

```
void func2(int c[restrict], int d[restrict])
{
  int i;

  for(i = 0; i < 64; i++)
  {
    c[i] += d[i];
    d[i] += 1;
  }
}
```

### 5.5.4 The volatile Keyword

The compiler analyzes data flow to avoid memory accesses whenever possible. If you have code that depends on memory accesses exactly as written in the C/C++ code, you must use the volatile keyword to identify these accesses. A variable qualified with a volatile keyword is allocated to an uninitialized section (as opposed to a register). The compiler does not optimize out any references to volatile variables.

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

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

However, in this example, *ctrl is a loop-invariant expression, so the loop is optimized down to a single-memory read. To get the desired result, define *ctrl as:

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

Here the *ctrl pointer is intended to reference a hardware location, such as an interrupt flag.

## 5.6 C++ Exception Handling

The compiler supports all the C++ exception handling features as defined by the ANSI/ISO 14882 C++ Standard. More details are discussed in *The C++ Programming Language, Third Edition* by Bjarne Stroustrup.

The compiler --exceptions option enables exception handling. The compiler's default is no exception handling support.

For exceptions to work correctly, all C++ files in the application must be compiled with the --exceptions option, regardless of whether exceptions occur in a particular file. Mixing exception-enabled object files and libraries with object files and libraries that do not have exceptions enabled can lead to undefined behavior. Also, when using --exceptions, you need to link with run-time-support libraries whose name contains _eh. These libraries contain functions that implement exception handling.

Using --exceptions causes

Using --exceptions causes the compiler to insert exception handling code. This code will increase the code size of the program.

See Section 7.1 for details on the run-time libraries.

## 5.7 Register Variables and Parameters

The C/C++ compiler treats register variables (variables defined with the register keyword) differently, depending on whether you use the --opt_level (-O) option.

- **Compiling with optimization**

  The compiler ignores any register definitions and allocates registers to variables and temporary values by using an algorithm that makes the most efficient use of registers.
- **Compiling without optimization**

  If you use the register keyword, you can suggest variables as candidates for allocation into registers. The compiler uses the same set of registers for allocating temporary expression results as it uses for allocating register variables.

The compiler attempts to honor all register definitions. If the compiler runs out of appropriate registers, it frees a register by moving its contents to memory. If you define too many objects as register variables, you limit the number of registers the compiler has for temporary expression results. This limit causes excessive movement of register contents to memory.

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

The register storage class is meaningful for parameters as well as local variables. Normally, in a function, some of the parameters are copied to a location on the stack where they are referenced during the function body. The compiler copies a register parameter to a register instead of the stack, which speeds access to the parameter within the function.

For more information about register conventions, see Section 6.3.

## 5.8 The asm Statement

The C/C++ compiler can embed assembly language instructions or directives directly into the assembly language output of the compiler. This capability is an extension to the C/C++ language—the *asm* statement. The asm (or __asm) statement provides access to hardware features that C/C++ cannot provide. The asm statement is syntactically like a call to a function named asm, with one string constant argument:

**asm("** *assembler text* **");**

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

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

The inserted code must be a legal assembly language statement. Like all assembly language statements, the line of code inside the quotes must begin with a label, a blank, a tab, or a comment (asterisk or semicolon). The compiler performs no checking on the string; if there is an error, the assembler detects it. For more information about the assembly language statements, see the *MSP430 Assembly Language Tools User's Guide*.

The asm statements do not follow the syntactic restrictions of normal C/C++ statements. Each can appear as a statement or a declaration, even outside of blocks. This is useful for inserting directives at the very beginning of a compiled module.

Use the alternate statement __asm("assembler text") if you are writing code for strict ANSI/ISO C mode (using the --strict_ansi option).

**NOTE: Avoid Disrupting the C/C++ Environment With asm Statements**

Be careful not to disrupt the C/C++ environment with asm statements. The compiler does not check the inserted instructions. Inserting jumps and labels into C/C++ code can cause unpredictable results in variables manipulated in or around the inserted code. Directives that change sections or otherwise affect the assembly environment can also be troublesome.

Be especially careful when you use optimization with asm statements. Although the compiler cannot remove asm statements, it can significantly rearrange the code order near them and cause undesired results.

## 5.9 Pragma Directives

Pragma directives tell the compiler how to treat a certain function, object, or section of code. The MSP430 C/C++ compiler supports the following pragmas:

- BIS_IE1_INTERRUPT
- CHECK_MISRA
- CODE_SECTION
- DATA_ALIGN
- DATA_SECTION
- DIAG_SUPPRESS, DIAG_REMARK, DIAG_WARNING, DIAG_ERROR, and DIAG_DEFAULT
- FUNC_CANNOT_INLINE
- FUNC_EXT_CALLED
- FUNC_IS_PURE
- FUNC_NEVER_RETURNS
- FUNC_NO_GLOBAL_ASG
- FUNC_NO_IND_ASG
- FUNCTION_OPTIONS
- INTERRUPT
- NO_HOOKS
- RESET_MISRA

The arguments *func* and *symbol* cannot be defined or declared inside the body of a function. You must specify the pragma outside the body of a function; and the pragma specification must occur before any declaration, definition, or reference to the func or symbol argument. If you do not follow these rules, the compiler issues a warning and may ignore the pragma.

For the pragmas that apply to functions or symbols, the syntax for the pragmas differs between C and C++. In C, you must supply the name of the object or function to which you are applying the pragma as the first argument. In C++, the name is omitted; the pragma applies to the declaration of the object or function that follows it.

### *5.9.1 The BIS_IE1_INTERRUPT*

The BIS_IE1_INTERRUPT pragma treats the named function as an interrupt routine. Additionally, the compiler generates a BIS operation on the IE1 special function register upon function exit. The mask value, which must be an 8-bit constant literal, is logically ORed with the IE1 SFR, just before the RETI instruction. The compiler assumes the IE1 SFR is mapped to address 0x0000.

The syntax of the pragma in C is:

**#pragma BIS_IE1_INTERRUPT (** *func* **,** *mask* **);**

The syntax of the pragma in C++ is:

**#pragma BIS_IE1_INTERRUPT (** *mask* **);**

In C, the argument func is the name of the function that is an interrupt. In C++, the pragma applies to the next function declared.

### 5.9.2 The CHECK_MISRA Pragma

The CHECK_MISRA pragma enables/disables MISRA-C:2004 rules at the source level. This pragma is equivalent to using the --check_misra option.

The syntax of the pragma in C is:

**#pragma CHECK_MISRA ("**{all|required|advisory|none|*rulespec*}**");**

The *rulespec* parameter is a comma-separated list of specifiers. See Section 5.3 for details.

The RESET_MISRA pragma can be used to reset any CHECK_MISRA pragmas; see Section 5.9.15.

### 5.9.3 The CODE_SECTION Pragma

The CODE_SECTION pragma allocates space for the *symbol* in C, or the next symbol declared in C++, in a section named *section name*.

The syntax of the pragma in C is:

**#pragma CODE_SECTION (** *symbol* **, "** *section name* **");**

The syntax of the pragma in C++ is:

**#pragma CODE_SECTION ("** *section name* **");**

The CODE_SECTION pragma is useful if you have code objects that you want to link into an area separate from the .text section.

The following examples demonstrate the use of the CODE_SECTION pragma.

***Example 5-3. Using the CODE_SECTION Pragma C Source File***

```
#pragma CODE_SECTION(funcA,"codeA")
int funcA(int a)

{
    int i;
    return (i = a);
}
```

***Example 5-4. Generated Assembly Code From Example 5-3***

```
    .sect "codeA"
    .align 2
    .clink
    .global funcA
;******************************************************************************
;* FUNCTION NAME: funcA                                                       *
;*                                                                            *
;* Regs Modified       : SP,SR,r12                                            *
;* Regs Used           : SP,SR,r12                                            *
;* Local Frame Size    : 0 Args + 4 Auto + 0 Save = 4 byte                    *
;******************************************************************************
funcA:
;* ---------------------------------------------------------------------------*
      SUB.W     #4,SP
      MOV.W     r12,0(SP)               ; |4|
      MOV.W     0(SP),2(SP)             ; |6|
      MOV.W     2(SP),r12               ; |6|
      ADD.W     #4,SP
      RET
```

### Example 5-5. Using the CODE_SECTION Pragma C++ Source File

```
#pragma CODE_SECTION("codeB")
int i_arg(int x) { return 1; }
int f_arg(float x) { return 2; }
```

### Example 5-6. Generated Assembly Code From Example 5-5

```
      .sect "codeB"
      .align   2
      .clink
      .global  i_arg__Fi
;******************************************************************************
;* FUNCTION NAME: i_arg(int)                                                  *
;*                                                                            *
;*   Regs Modified     : SP,SR,r12                                            *
;*   Regs Used         : SP,SR,r12                                            *
;*   Local Frame Size  : 0 Args + 2 Auto + 0 Save = 2 byte                    *
;******************************************************************************
i_arg__Fi:
;* ---------------------------------------------------------------------------*
        SUB.W     #2,SP
        MOV.W     r12,0(SP)             ; |2|
        MOV.W     #1,r12                ; |2|
        ADD.W     #2,SP
        RET
      .sect ".text"
      .align      2
      .clink
      .global     f_arg__Ff

;******************************************************************************
;* FUNCTION NAME: f_arg(float)                                                *
;*                                                                            *
;*   Regs Modified     : SP,SR,r12                                            *
;*   Regs Used         : SP,SR,r12,r13                                        *
;*   Local Frame Size  : 0 Args + 4 Auto + 0 Save = 4 byte                    *
;******************************************************************************
f_arg__Ff:
;* ---------------------------------------------------------------------------*
        SUB.W     #4,SP
        MOV.W     r12,0(SP)             ; |3|
        MOV.W     r13,2(SP)             ; |3|
        MOV.W     #2,r12                ; |3|
        ADD.W     #4,SP
        RET
```

### 5.9.4 The DATA_ALIGN Pragma

The DATA_ALIGN pragma aligns the *symbol* in C, or the next symbol declared in C++, to an alignment boundary. The alignment boundary is the maximum of the symbol's default alignment value or the value of the *constant* in bytes. The constant must be a power of 2.

The syntax of the pragma in C is:

**#pragma DATA_ALIGN (** *symbol* **,** *constant* **);**

The syntax of the pragma in C++ is:

**#pragma DATA_ALIGN (** *constant* **);**

### 5.9.5 The DATA_SECTION Pragma

The DATA_SECTION pragma allocates space for the *symbol* in C, or the next symbol declared in C++, in a section named *section name*.

The syntax of the pragma in C is:

**#pragma DATA_SECTION (** *symbol* **, "** *section name* **");**

The syntax of the pragma in C++ is:

**#pragma DATA_SECTION ("** *section name* **");**

The DATA_SECTION pragma is useful if you have data objects that you want to link into an area separate from the .bss section. If you allocate a global variable using a DATA_SECTION pragma and you want to reference the variable in C code, you must declare the variable as extern far.

Example 5-7 through Example 5-9 demonstrate the use of the DATA_SECTION pragma.

***Example 5-7. Using the DATA_SECTION Pragma C Source File***

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

***Example 5-8. Using the DATA_SECTION Pragma C++ Source File***

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

***Example 5-9. Using the DATA_SECTION Pragma Assembly Source File***

```
        .global  bufferA
        .bss     bufferA,512,2
        .global  bufferB
bufferB: .usect "my_sect",512,2
```

### 5.9.6 The Diagnostic Message Pragmas

The following pragmas can be used to control diagnostic messages in the same ways as the corresponding command line options:

| Pragma | Option | Description |
|---|---|---|
| DIAG_SUPPRESS *num* | -pds=*num*[, $num_2$, $num_3$...] | Suppress diagnostic *num* |
| DIAG_REMARK *num* | -pdsr=*num*[, $num_2$, $num_3$...] | Treat diagnostic *num* as a remark |
| DIAG_WARNING *num* | -pdsw=*num*[, $num_2$, $num_3$...] | Treat diagnostic *num* as a warning |
| DIAG_ERROR *num* | -pdse=*num*[, $num_2$, $num_3$...] | Treat diagnostic *num* as an error |
| DIAG_DEFAULT *num* | n/a | Use default severity of the diagnostic |

The syntax of the pragmas in C is:

**#pragma DIAG**_XXX [=]*num*[, $num_2$, $num_3$...]

The diagnostic affected (*num*) is specified using either an error number or an error tag name. The equal sign (=) is optional. Any diagnostic can be overridden to be an error, but only diagnostics with a severity of discretionary error or below can have their severity reduced to a warning or below, or be suppressed. The diag_default pragma is used to return the severity of a diagnostic to the one that was in effect before any pragmas were issued (i.e., the normal severity of the message as modified by any command-line options).

The diagnostic identifier number is output along with the message when the -pden command line option is specified.

### 5.9.7 The FUNC_CANNOT_INLINE Pragma

The FUNC_CANNOT_INLINE pragma instructs the compiler that the named function cannot be expanded inline. Any function named with this pragma overrides any inlining you designate in any other way, such as using the inline keyword. Automatic inlining is also overridden with this pragma; see Section 2.11.

The pragma must appear before any declaration or reference to the function that you want to keep. In C, the argument *func* is the name of the function that cannot be inlined. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

**#pragma FUNC_CANNOT_INLINE (** *func* **);**

The syntax of the pragma in C++ is:

**#pragma FUNC_CANNOT_INLINE;**

### 5.9.8 The FUNC_EXT_CALLED Pragma

When you use the --program_level_compile option, the compiler uses program-level optimization. When you use this type of optimization, the compiler removes any function that is not called, directly or indirectly, by main. You might have C/C++ functions that are called by hand-coded assembly instead of main.

The FUNC_EXT_CALLED pragma specifies to the optimizer to keep these C functions or any other functions that these C/C++ functions call. These functions act as entry points into C/C++.

The pragma must appear before any declaration or reference to the function that you want to keep. In C, the argument *func* is the name of the function that you do not want removed. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

```
#pragma FUNC_EXT_CALLED ( func );
```

The syntax of the pragma in C++ is:

```
#pragma FUNC_EXT_CALLED;
```

Except for _c_int00, which is the name reserved for the system reset interrupt for C/C++programs, the name of the interrupt (the *func* argument) does not need to conform to a naming convention.

When you use program-level optimization, you may need to use the FUNC_EXT_CALLED pragma with certain options. See Section 3.3.2.

### 5.9.9 The FUNC_IS_PURE Pragma

The FUNC_IS_PURE pragma specifies to the compiler that the named function has no side effects. This allows the compiler to do the following:

- Delete the call to the function if the function's value is not needed
- Delete duplicate functions

The pragma must appear before any declaration or reference to the function. In C, the argument *func* is the name of a function. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

```
#pragma FUNC_IS_PURE ( func );
```

The syntax of the pragma in C++ is:

```
#pragma FUNC_IS_PURE;
```

### 5.9.10 The FUNC_NEVER_RETURNS Pragma

The FUNC_NEVER_RETURNS pragma specifies to the compiler that the function never returns to its caller.

The pragma must appear before any declaration or reference to the function that you want to keep. In C, the argument *func* is the name of the function that does not return. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

```
#pragma FUNC_NEVER_RETURNS ( func );
```

The syntax of the pragma in C++ is:

```
#pragma FUNC_NEVER_RETURNS;
```

### 5.9.11 The FUNC_NO_GLOBAL_ASG Pragma

The FUNC_NO_GLOBAL_ASG pragma specifies to the compiler that the function makes no assignments to named global variables and contains no asm statements.

The pragma must appear before any declaration or reference to the function that you want to keep. In C, the argument *func* is the name of the function that makes no assignments. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

```
#pragma FUNC_NO_GLOBAL_ASG ( func );
```

The syntax of the pragma in C++ is:

```
#pragma FUNC_NO_GLOBAL_ASG;
```

### 5.9.12 The FUNC_NO_IND_ASG Pragma

The FUNC_NO_IND_ASG pragma specifies to the compiler that the function makes no assignments through pointers and contains no asm statements.

The pragma must appear before any declaration or reference to the function that you want to keep. In C, the argument *func* is the name of the function that makes no assignments. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

```
#pragma FUNC_NO_IND_ASG ( func );
```

The syntax of the pragma in C++ is:

```
#pragma FUNC_NO_IND_ASG;
```

### 5.9.13 The FUNCTION_OPTIONS Pragma

The FUNCTION_OPTIONS pragma allows you to compile a specific function in a C or C++ file with additional command-line compiler options. The affected function will be compiled as if the specified list of options appeared on the command line after all other compiler options. In C, the pragma is applied to the function specified. In C++, the pragma is applied to the next function.

The syntax of the pragma in C is:

```
#pragma FUNCTION_OPTIONS (func, "additional options");
```

The syntax of the pragma in C++ is:

```
#pragma FUNCTION_OPTIONS("additional options");
```

### 5.9.14 The INTERRUPT Pragma

The INTERRUPT pragma enables you to handle interrupts directly with C code. In C, the argument *func* is the name of a function. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

```
#pragma INTERRUPT ( func );
```

The syntax of the pragma in C++ is:

```
#pragma INTERRUPT ;
```

The code for the function will return via the IRP (interrupt return pointer).

Except for _c_int00, which is the name reserved for the system reset interrupt for C programs, the name of the interrupt (the *func* argument) does not need to conform to a naming convention.

**HWI Objects and the INTERRUPT Pragma**

**NOTE:** The INTERRUPT pragma must not be used when BIOS HWI objects are used in conjunction with C functions. The HWI_enter/HWI_exit macros and the HWI dispatcher contain this functionality, and the use of the C modifier can cause negative results.

### 5.9.15 The RESET_MISRA Pragma

The RESET_MISRA pragma resets the specified MISRA-C:2004 rules to the state they were before any CHECK_MISRA pragmas (see Section 5.9.2) were processed. For instance, if a rule was enabled on the command line but disabled in the source, the RESET_MISRA pragma resets it to enabled. This pragma accepts the same format as the --check_misra option, except for the "none" keyword.

The syntax of the pragma in C is:

```
#pragma RESET_MISRA ("{all|required|advisory|rulespec}");
```

The *rulespec* parameter is a comma-separated list of specifiers. See Section 5.3 for details.

### 5.9.16 The vector Pragma

The vector pragma indicates that the function that follows is to be used as the interrupt vector routine for the listed vectors. The syntax of the pragma is:

**#pragma vector =** *vec1*[**,** *vec2* **,** *vec3*, ...]

The vector pragma requires linker command file support. The command file must specify output sections for each interrupt vector of the form .int*xx* where *xx* is the number of the interrupt vector. The output sections mut map to the physical memory location of the appropriate interrupt vector. The standard linker command files are set up to handle the vector pragma.

The __even_in_range intrinsic provides a hint to the compiler when generating switch statements for interrupt vector routines. The intrinsic is usually used as follows:

```
switch (__even_in_range( x , NUM ))
{
   ...
}
```

The __even_in_range intrinsic returns the value *x* to control the switch statement, but also tells the compiler that *x* must be an even value in the range of 0 to *NUM*, inclusive.

## 5.10 The _Pragma Operator

The MSP430 C/C++ compiler supports the C99 preprocessor _Pragma() operator. This preprocessor operator is similar to #pragma directives. However, _Pragma can be used in preprocessing macros (#defines).

The syntax of the operator is:

**_Pragma ("** *string_literal* **");**

The argument *string_literal* is interpreted in the same way the tokens following a #pragma directive are processed. The string_literal must be enclosed in quotes. A quotation mark that is part of the string_literal must be preceded by a backward slash.

You can use the _Pragma operator to express #pragma directives in macros. For example, the DATA_SECTION syntax:

**#pragma DATA_SECTION(** *func* **,"** *section* **");**

Is represented by the _Pragma() operator syntax:

**_Pragma ("DATA_SECTION(** *func* **,\"** *section* **\")")**

The following code illustrates using _Pragma to specify the DATA_SECTION pragma in a macro:

```
...

#define EMIT_PRAGMA(x) _Pragma(#x)
#define COLLECT_DATA(var) EMIT_PRAGMA(DATA_SECTION(var,"mysection"))

COLLECT_DATA(x)
int x;

...
```

The EMIT_PRAGMA macro is needed to properly expand the quotes that are required to surround the section argument to the DATA_SECTION pragma.

## 5.11 Object File Symbol Naming Conventions (Linknames)

Each externally visible identifier is assigned a unique symbol name to be used in the object file, a so-called *linkname*. This name is assigned by the compiler according to an algorithm which depends on the name, type, and source language of the symbol. This algorithm may add a prefix to the identifier (typically an underscore), and it may *mangle* the name.

The linkname for all objects and functions is the same as the name in the C source with an added underscore prefix. This prevents any C identifier from colliding with any identifier in the assembly code namespace, such as an assembler keyword.

Name mangling encodes the types of the parameters of a function in the linkname for a function. Name mangling only occurs for C++ functions which are not declared 'extern "C"'. Mangling allows function overloading, operator overloading, and type-safe linking. Be aware that the return value of the function is not encoded in the mangled name, as C++ functions cannot be overloaded based on the return value.

The mangling algorithm used closely follows that described in The Annotated Reference Manual (ARM).

For example, the general form of a C++ linkname for a function named func is:

**_func__F** *parmcodes*

Where parmcodes is a sequence of letters that encodes the parameter types of func.

For this simple C++ source file:

```
int foo(int i){ }   //global C++ function
```

This is the resulting assembly code:

```
_foo__Fi
```

The linkname of foo is _foo__Fi, indicating that foo is a function that takes a single argument of type int. To aid inspection and debugging, a name demangling utility is provided that demangles names into those found in the original C++ source. See Chapter 8 for more information.

## 5.12 Initializing Static and Global Variables

The ANSI/ISO C standard specifies that global (extern) and static variables without explicit initializations must be initialized to 0 before the program begins running. This task is typically done when the program is loaded. Because the loading process is heavily dependent on the specific environment of the target application system, the compiler itself makes no provision for initializing to 0 otherwise uninitialized static storage class variables at run time. It is up to your application to fulfill this requirement.

**Initialize Global Objects**

**NOTE:** You should explicitly initialize all global objects which you expected the compiler would set to zero by default.

### 5.12.1 Initializing Static and Global Variables With the Linker

If your loader does not preinitialize variables, you can use the linker to preinitialize the variables to 0 in the object file. For example, in the linker command file, use a fill value of 0 in the .bss section:

```
SECTIONS
   {
      ...

      .bss: {} = 0x00;
      ...
   }
```

Because the linker writes a complete load image of the zeroed .bss section into the output COFF file, this method can have the unwanted effect of significantly increasing the size of the output file (but not the program).

If you burn your application into ROM, you should explicitly initialize variables that require initialization. The preceding method initializes .bss to 0 only at load time, not at system reset or power up. To make these variables 0 at run time, explicitly define them in your code.

For more information about linker command files and the SECTIONS directive, see the linker description information in the *MSP430 Assembly Language Tools User's Guide.*

### 5.12.2 Initializing Static and Global Variables With the const Type Qualifier

Static and global variables of type *const* without explicit initializations are similar to other static and global variables because they might not be preinitialized to 0 (for the same reasons discussed in Section 5.12). For example:

```
const int zero;      /*  may not be initialized to 0  */
```

However, the initialization of const global and static variables is different because these variables are declared and initialized in a section called .const. For example:

```
const int zero = 0   /*  guaranteed to be 0  */
```

This corresponds to an entry in the .const section:

```
   .sect   .const
_zero
   .word   0
```

This feature is particularly useful for declaring a large table of constants, because neither time nor space is wasted at system startup to initialize the table. Additionally, the linker can be used to place the .const section in ROM.

You can use the DATA_SECTION pragma to put the variable in a section other than .const. For example, the following C code:

```
#pragma DATA_SECTION (var, ".mysect");
   const int zero=0;
```

is compiled into this assembly code:

```
   .sect   .mysect
_zero
   .word   0
```

## 5.13 Changing the ANSI/ISO C Language Mode

The --kr_compatible, --relaxed_ansi, and --strict_ansi options let you specify how the C/C++ compiler interprets your source code. You can compile your source code in the following modes:

- Normal ANSI/ISO mode
- K&R C mode
- Relaxed ANSI/ISO mode
- Strict ANSI/ISO mode

The default is normal ANSI/ISO mode. Under normal ANSI/ISO mode, most ANSI/ISO violations are emitted as errors. Strict ANSI/ISO violations (those idioms and allowances commonly accepted by C/C++ compilers, although violations with a strict interpretation of ANSI/ISO), however, are emitted as warnings. Language extensions, even those that conflict with ANSI/ISO C, are enabled.

K&R C mode does not apply to C++ code.

### *5.13.1 Compatibility With K&R C (--kr_compatible Option)*

The ANSI/ISO C/C++ language is a superset of the de facto C standard defined in Kernighan and Ritchie's *The C Programming Language*. Most programs written for other non-ANSI/ISO compilers correctly compile and run without modification.

There are subtle changes, however, in the language that can affect existing code. Appendix C in *The C Programming Language* (second edition, referred to in this manual as K&R) summarizes the differences between ANSI/ISO C and the first edition's C standard (the first edition is referred to in this manual as K&R C).

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

The specific differences between the ANSI/ISO version of C and the K&R version of C are as follows:

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

```
unsigned short u;
int i;
if (u < i)         /* SIGNED comparison, unless --kr_compatible used */
```

- ANSI/ISO prohibits combining two pointers to different types in an operation. In most K&R compilers, this situation produces only a warning. Such cases are still diagnosed when --kr_compatible is used, but with less severity:

```
int *p;
char *q = p;       /* error without --kr_compatible, warning with --kr_compatible */
```

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

```
a;                 /* illegal unless --kr_compatible used */
```

- ANSI/ISO interprets file scope definitions that have no initializers as *tentative definitions.* In a single module, multiple definitions of this form are fused together into a single definition. Under K&R, each definition is treated as a separate definition, resulting in multiple definitions of the same object and usually an error. For example:

```
int a;
int a;             /* illegal if --kr_compatible used, OK if not */
```

  Under ANSI/ISO, the result of these two definitions is a single definition for the object a. For most K&R compilers, this sequence is illegal, because int a is defined twice.

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

```
extern int a;
static int a;      /* illegal unless --kr_compatible used */
```

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

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

- ANSI/ISO specifies that bit fields must be of type int or unsigned. With --kr_compatible, bit fields can be legally defined with any integral type. For example:

```
struct s
{
   short f : 2;    /* illegal unless --kr_compatible used */
};
```

- K&R syntax allows a trailing comma in enumerator lists:

```
enum { a, b, c, }; /* illegal unless --kr_compatible used */
```

- K&R syntax allows trailing tokens on preprocessor directives:

```
#endif NAME        /* illegal unless --kr_compatible used */
```

### 5.13.2 Enabling Strict ANSI/ISO Mode and Relaxed ANSI/ISO Mode (--strict_ansi and --relaxed_ansi Options)

Use the --strict_ansi option when you want to compile under strict ANSI/ISO mode. In this mode, error messages are provided when non-ANSI/ISO features are used, and language extensions that could invalidate a strictly conforming program are disabled. Examples of such extensions are the inline and asm keywords.

Use the --relaxed_ansi option when you want the compiler to ignore strict ANSI/ISO violations rather than emit a warning (as occurs in normal ANSI/ISO mode) or an error message (as occurs in strict ANSI/ISO mode). In relaxed ANSI/ISO mode, the compiler accepts extensions to the ANSI/ISO C standard, even when they conflict with ANSI/ISO C.

### 5.13.3 Enabling Embedded C++ Mode (--embedded_cpp Option)

The compiler supports the compilation of embedded C++. In this mode, some features of C++ are removed that are of less value or too expensive to support in an embedded system. When compiling for embedded C++, the compiler generates diagnostics for the use of omitted features.

Embedded C++ is enabled by compiling with the --embedded_cpp option.

Embedded C++ omits these C++ features:

- Templates
- Exception handling
- Run-time type information
- The new cast syntax
- The keyword mutable
- Multiple inheritance
- Virtual inheritance

Under the standard definition of embedded C++, namespaces and using-declarations are not supported. The MSP430 compiler nevertheless allows these features under embedded C++ because the C++ run-time-support library makes use of them. Furthermore, these features impose no run-time penalty.

## 5.14 GNU C Compiler Extensions

The GNU compiler, GCC, provides a number of language features not found in the ANSI standard C. When the --gcc option is used many of the features defined for GCC 3.4 are enabled. The definition and official examples of these extensions can be found at http://gcc.gnu.org/onlinedocs/gcc-3.4.6/gcc/C-Extensions.html.

The GCC extensions are supported only for C source code, they are not available for C++ source code. The extensions that the TI C compiler supports are listed in Table 5-2.

**Table 5-2. GCC Language Extensions**

| Extensions | Descriptions |
|---|---|
| Statement expressions | Putting statements and declarations inside expressions (useful for creating smart 'safe' macros) |
| Local labels | Labels local to a statement expression |
| Labels as values(1) | Pointers to labels and computed gotos |
| Nested functions(1) | As in Algol and Pascal, lexical scoping of functions |
| Constructing calls(1) | Dispatching a call to another function |
| Naming types(2) | Giving a name to the type of an expression |
| typeof operator | typeof referring to the type of an expression |
| Generalized lvalues | Using question mark (?) and comma (,) and casts in lvalues |
| Conditionals | Omitting the middle operand of a ?: expression |
| Hex floats | Hexadecimal floating-point constants |
| Complex(1) | Data types for complex numbers |
| Zero length | Zero-length arrays |
| Variadic macros | Macros with a variable number of arguments |
| Variable length(1) | Arrays whose length is computed at run time |
| Empty structures | Structures with no members |
| Subscripting | Any array can be subscripted, even if it is not an lvalue. |
| Escaped newlines(1) | Slightly looser rules for escaped newlines |
| Multi-line strings(2) | String literals with embedded newlines |
| Pointer arithmetic | Arithmetic on void pointers and function pointers |
| Initializers | Non-constant initializers |
| Compound literals | Compound literals give structures, unions, or arrays as values |
| Designated initializers(1) | Labeling elements of initializers |
| Cast to union | Casting to union type from any member of the union |
| Case ranges | 'Case 1 ... 9' and such |
| Mixed declarations | Mixing declarations and code |
| Function attributes | Declaring that functions have no side effects, or that they can never return |
| Attribute syntax | Formal syntax for attributes |
| Function prototypes | Prototype declarations and old-style definitions |
| C++ comments | C++ comments are recognized. |
| Dollar signs | A dollar sign is allowed in identifiers. |
| Character escapes | The character ESC is represented as \e |
| Variable attributes | Specifying the attributes of variables |
| Type attributes | Specifying the attributes of types |
| Alignment | Inquiring about the alignment of a type or variable |
| Inline | Defining inline functions (as fast as macros) |
| Assembly labels | Specifying the assembler name to use for a C symbol |
| Extended asm(1) | Assembler instructions with C operands |
| Constraints(1) | Constraints for asm operands |

(1) Not supported
(2) Feature defined for GCC 3.0; definition and examples at http://gcc.gnu.org/onlinedocs/gcc-3.0.4/gcc/C-Extensions.html

**Table 5-2. GCC Language Extensions (continued)**

| Extensions | Descriptions |
|---|---|
| Alternate keywords | Header files can use __const__, __asm__, etc |
| Explicit reg vars[1] | Defining variables residing in specified registers |
| Incomplete enum types | Define an enum tag without specifying its possible values |
| Function names | Printable strings which are the name of the current function |
| Return address | Getting the return or frame address of a function<br>__builtin_return_address is recognized but always returns zero<br>__builtin_frame_address is recognized but always returns zero |
| Other built-ins | Other built-in functions include:<br>__builtin_constant_p<br>__builtin_expect |
| Vector extensions[1] | Using vector instructions through built-in functions |
| Target built-ins[1] | Built-in functions specific to particular targets |
| Pragmas[1] | Pragmas accepted by GCC |
| Unnamed fields | Unnamed struct/union fields within structs/unions |
| Thread-local[3] | Per-thread variables |

(3) Not supported

### 5.14.1 Function and Variable Attributes

The TI compiler implements only three attributes for variables and functions. All others are simply ignored. Table 5-3 lists the attributes that are supported.

**Table 5-3. TI-Supported GCC Function and Variable Attributes**

| Attributes | Description |
|---|---|
| deprecated | This function or variable exists but the compiler generates a warning if it is used. |
| section | Place this function or variable in the specified section. |
| unused | This function or variable is allowed to appear as unused. Do not issue a warning if it is unused. |

### 5.14.2 Type Attributes

The TI compiler implements only two attributes for types as listed in Table 5-4. All others are simply ignored.

The packed attribute is implemented only for enumerated types; other uses of the packed attribute are rejected.

**Table 5-4. TI-Supported GCC Type Attributes**

| Attributes | Description |
|---|---|
| packed | enum type: represent using the smallest sized integer type that fits |
| unused | Variables of this type are allowed to appear to be unused. Do not issue a warning if such a variable is unused. |

### 5.14.3 Built-In Functions

TI provides support for only the four built-in functions in Table 5-5.

**Table 5-5. TI-Supported GCC Built-In Functions**

| Function | Description |
|---|---|
| __builtin_constant_p(*expr*) | Returns true only if *expr* is a constant at compile time. |
| __builtin_expect(*expr*, CONST) | Returns *expr*. The compiler uses this function to optimize along paths determined by conditional statements such as if-else. While this function can be used anywhere in your code, it only conveys useful information to the compiler if it is the entire predicate of an if statement and CONST is 0 or 1. For example, the following indicates that you expect the predicate "a == 3" to be true most of the time:<br>`if (__builtin_expect(a == 3, 1))` |
| __builtin_return_address(int *level*) | Returns 0. |
| __builtin_frame_address(int *level*) | Returns 0. |

## 5.15 Compiler Limits

Due to the variety of host systems supported by the C/C++ compiler and the limitations of some of these systems, the compiler may not be able to successfully compile source files that are excessively large or complex. In general, exceeding such a system limit prevents continued compilation, so the compiler aborts immediately after printing the error message. Simplify the program to avoid exceeding a system limit.

Some systems do not allow filenames longer than 500 characters. Make sure your filenames are shorter than 500.

The compiler has no arbitrary limits but is limited by the amount of memory available on the host system. On smaller host systems such as PCs, the optimizer may run out of memory. If this occurs, the optimizer terminates and the shell continues compiling the file with the code generator. This results in a file compiled with no optimization. The optimizer compiles one function at a time, so the most likely cause of this is a large or extremely complex function in your source module. To correct the problem, your options are:

- Don't optimize the module in question.
- Identify the function that caused the problem and break it down into smaller functions.
- Extract the function from the module and place it in a separate module that can be compiled without optimization so that the remaining functions can be optimized.

# Run-Time Environment

This chapter describes the MSP430 C/C++ run-time environment. To ensure successful execution of C/C++ programs, it is critical that all run-time code maintain this environment. It is also important to follow the guidelines in this chapter if you write assembly language functions that interface with C/C++ code.

**Topic** **Page**

## 6.1 Memory Model

The MSP430 compiler treats memory as a single linear block that s partitioned into subblocks of code and data. Each subblock of code or data generated by a C program is placed in its own continuous memory space. The compiler assumes that a full 16-bit address space is available in target memory.

### 6.1.1 Code Memory Models

The MSP430 compiler supports two different code memory models, small and large, which are controlled by the --code_model option. The small code model uses 16-bit function pointers and requires all code to be placed in the low 64K of memory. This is the only valid code model for 16-bit MSP430 devices. The large code model provides a 1MB address space for code and uses 20-bit function pointers. It is the default for MSP430X devices. Interrupt service routines must still be placed in the low 64K of memory (see Section 6.6.5).

The small code model is slightly more efficient in terms of run-time performance and memory usage when compared to the large code model. Therefore, it is beneficial to use the small code model when all code will fit in the low 64K of memory. Modules assembled/compiled using the small-code model are not compatible with modules that are assembled/compiled using large-code model. The linker generates an error if any attempt is made to combine object files that use different code memory models. An appropriate run-time library must be used as well.

### 6.1.2 Data Memory Models

The MSP430 compiler supports three different data memory models: small, restricted and large. The data model used is controlled by the --data_model option. The 16-bit MSP430 devices always use the small data memory model. The 20-bit MSP430X devices can use any data memory model and use the restricted data model by default.

The small data model requires that all data be located in the low 64K of memory. Data pointers are 16-bits in size. This is the most efficient data model in terms of performance and application size.

The restricted data model allows data to be located throughout the entire 1MB address space available on MSP430X devices with only a minimal efficiency penalty over the small data model. It is restricted because individual objects (structures, arrays, etc.) cannot be larger than 64K in size. Data pointers are 32-bits in size.

The large data model also allows data to be located throughout the entire 1MB address space and also places no restriction on the maximum size of an individual object. Permitting individual objects to be greater than 64K in size causes code generated for the large data model to be less efficient than code generated for the restricted data model.

The maximum size of an object (size_t) and the maximum difference between two pointers (ptrdiff_t) are increased from 16-bits to 32-bits in the large data model. Applications that rely on size_t or ptrdiff_t to be a specific size may need to be updated.

Object files built with different data models are not compatible. All files in an application must be built with the same data model and a corresponding run-time library must be used as well.

### 6.1.3 Support for Near Data

All current MSP430X devices do not have any writeable memory above the 64K boundary. For these devices, even when the restricted or large data models are used, only constant data will be placed above 64K. The compiler can take advantage of this knowledge to produce more efficient code. This is controlled by the --near_data option.

When --near_data=globals is specified it tells the compiler that all global read/write data must be located in the first 64K of memory. This is the default behavior. If --near_data=none is specified it tells the compiler that it cannot rely on this assumption to generate more efficient code.

**NOTE: The Linker Defines the Memory Map**

The linker, not the compiler, defines the memory map and allocates code and data into target memory. The compiler assumes nothing about the types of memory available, about any locations not available for code or data (holes), or about any locations reserved for I/O or control purposes. The compiler produces relocatable code that allows the linker to allocate code and data into the appropriate memory spaces.

For example, you can use the linker to allocate global variables into on-chip RAM or to allocate executable code into external ROM. You can allocate each block of code or data individually into memory, but this is not a general practice (an exception to this is memory-mapped I/O, although you can access physical memory locations with C/C++ pointer types).

### 6.1.4 Sections

The compiler produces relocatable blocks of code and data called *sections*. The sections are allocated into memory in a variety of ways to conform to a variety of system configurations. For more information about sections and allocating them, see the introductory object module information in the *MSP430 Assembly Language Tools User's Guide*.

There are two basic types of sections:

- **Initialized sections** contain data or executable code. The C/C++ compiler creates the following initialized sections:
  - The **.cinit section** and the **.pinit section** contain tables for initializing variables and constants.
  - The **.const section** contains string constants, switch tables, and data defined with the C/C++ qualifier *const* (provided the constant is not also defined as *volatile*).
  - The **.text section** contains all the executable code as well as string literals and compiler-generated constants.
- **Uninitialized sections** reserve space in memory (usually RAM). A program can use this space at run time to create and store variables. The compiler creates the following uninitialized sections:
  - The **.bss section** reserves space for global and static variables. At boot or load time, the C/C++ boot routine or the loader copies data out of the .cinit section (which can be in ROM) and stores it in the .bss section.
  - The **.stack section** reserves memory for the C/C++ software stack.
  - The **.sysmem section** reserves space for dynamic memory allocation. The reserved space is used by dynamic memory allocation routines, such as malloc, calloc, realloc, or new. If a C/C++ program does not use these functions, the compiler does not create the .sysmem section.

The assembler creates the default sections .text, .bss, and .data. The C/C++ compiler, however, does not use the .data section. You can instruct the compiler to create additional sections by using the CODE_SECTION and DATA_SECTION pragmas (see Section 5.9.3 and Section 5.9.5).

The linker takes the individual sections from different modules and combines sections that have the same name. The resulting output sections and the appropriate placement in memory for each section are listed in Table 6-1. You can place these output sections anywhere in the address space as needed to meet system requirements.

**Table 6-1. Summary of Sections and Memory Placement**

| Section | Type of Memory | Section | Type of Memory |
|---|---|---|---|
| .bss | RAM | .pinit | ROM or RAM |
| .cinit | ROM or RAM | .stack | RAM |
| .const | ROM or RAM | .sysmem | RAM |
| .data | ROM or RAM | .text | ROM or RAM |

You can use the SECTIONS directive in the linker command file to customize the section-allocation process. For more information about allocating sections into memory, see the linker description chapter in the *MSP430 Assembly Language Tools User's Guide*.

### 6.1.5 C/C++ Software Stack

The C/C++ compiler uses a stack to:

- Allocate local variables
- Pass arguments to functions
- Save register contents

The run-time stack grows from the high addresses to the low addresses. The compiler uses the R13 register to manage this stack. R13 is the *stack pointer* (SP), which points to the next unused location on the stack.

The linker sets the stack size, creates a global symbol, __STACK_SIZE, and assigns it a value equal to the stack size in bytes. The default stack size is 2048 bytes. You can change the stack size at link time by using the --stack_size option with the linker command. For more information on the --stack_size option, see .

**Save-On-Entry Registers and C/C+ Stack Size**

**NOTE:** Since register sizes increase for MSP430X devices (specified with --silicon_version=mspx), saving and restoring save-on-entry registers requires 32-bits of stack space for each register saved on the stack. When you are porting code originally written for 16-bit MSP430 devices, you may need to increase the C stack size from previous values.

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

The C/C++ environment automatically decrements SP at the entry to a function to reserve all the space necessary for the execution of that function. The stack pointer is incremented at the exit of the function to restore the stack to the state before the function was entered. If you interface assembly language routines to C/C++ programs, be sure to restore the stack pointer to the same state it was in before the function was entered.

For more information about using the stack pointer, see Section 6.3; for more information about the stack, see Section 6.4.

**NOTE: Stack Overflow**

The compiler provides no means to check for stack overflow during compilation or at run time. A stack overflow disrupts the run-time environment, causing your program to fail. Be sure to allow enough space for the stack to grow. You can use the --entry_hook option to add code to the beginning of each function to check for stack overflow; see Section 2.13.

### 6.1.6 Dynamic Memory Allocation

The run-time-support library supplied with the MSP430 compiler contains several functions (such as malloc, calloc, and realloc) that allow you to allocate memory dynamically for variables at run time.

Memory is allocated from a global pool, or heap, that is defined in the .sysmem section. You can set the size of the .sysmem section by using the --heap_size=*size* option with the linker command. The linker also creates a global symbol, __SYSMEM_SIZE, and assigns it a value equal to the size of the heap in bytes. The default size is 128 bytes. For more information on the --heap_size option, see .

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

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

use a pointer and call the malloc function:

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

### 6.1.7 Initialization of Variables

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

In situations where a program is loaded directly from an object file into memory and run, you can avoid having the .cinit section occupy space in memory. A loader can read the initialization tables directly from the object file (instead of from ROM) and perform the initialization directly at load time instead of at run time. You can specify this to the linker by using the --ram_model link option. For more information, see Section 6.8.

## 6.2 Object Representation

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

### 6.2.1 Data Type Storage

Table 6-2 lists register and memory storage for various data types:

**Table 6-2. Data Representation in Registers and Memory**

| Data Type | Register Storage | Memory Storage |
|---|---|---|
| char, signed char | Bits 0-7 of register(1) | 8 bits aligned to 8-bit boundary |
| unsigned char, bool | Bits 0-7 of register | 8 bits aligned to 8-bit boundary |
| short, signed short | Bits 0-15 of register(1) | 16 bits aligned to 16-bit (word) boundary |
| unsigned short, wchar_t | Bits 0-15 of register | 16 bits aligned to 16-bit (word) boundary |
| int, signed int | Bits 0-15 of register | 16 bits aligned to 16-bit (word) boundary |
| unsigned int | Bits 0-15 of register | 16 bits aligned to 16-bit (word) boundary |
| enum | Bits 0-15 of register | 16 bits aligned to 16-bit (word) boundary |
| long. signed long | Register pair | 32 bits aligned to 16-bit (word) boundary |
| unsigned long | Register pair | 32 bits aligned to 16-bit (word) boundary |
| float | Register pair | 32 bits aligned to 16-bit (word) boundary |
| double | Register pair | 32 bits aligned to 16-bit (word) boundary |
| long double | Register pair | 32 bits aligned to 16-bit (word) boundary |
| struct | Members are stored as their individual types require. | Members are stored as their individual types require; aligned according to the member with the most restrictive alignment requirement. |

(1) Negative values are sign-extended to bit 15.

**Table 6-2. Data Representation in Registers and Memory (continued)**

| Data Type | Register Storage | Memory Storage |
|---|---|---|
| array | Members are stored as their individual types require. | Members are stored as their individual types require; aligned to 16-bit (word) boundary. All arrays inside a structure are aligned according to the type of each element in the array. |
| pointer to data member | Bits 0-15 of register | 16 bits aligned to 16-bit (word) boundary |
| MSP430X large-data model pointer to data member[2] | Bits 0-20 of register | 32 bits aligned to 16-bit (word) boundary |
| MSP430 pointer to function | Bits 0-15 of register | 16 bits aligned to 16-bit (word) boundary |
| MSP430X[3] pointer to function | Bits 0-20 of register | 32 bits aligned to 16-bit (word) boundary |

(2) MSP430X large-data model is specified by --silicon_version=mspx --data_model=large
(3) MSP430X is specified with the --silicon_version=mspx option.

#### 6.2.1.1 Pointer to Member Function Types

Pointer to member function objects are stored as a structure with three members, and the layout is equivalent to:

```
struct {
   short int d;
   short int i;
   union {
      void (f) ();
      long 0; }
   };
```

The parameter d is the offset to be added to the beginning of the class object for this pointer. The parameter I is the index into the virtual function table, offset by 1. The index enables the NULL pointer to be represented. Its value is -1 if the function is nonvirtual. The parameter f is the pointer to the member function if it is nonvirtual, when I is 0. The 0 is the offset to the virtual function pointer within the class object.

#### 6.2.1.2 Structure and Array Alignment

Structures are aligned according to the member with the most restrictive alignment requirement. Structures do not contain padding after the last member. Arrays are always word aligned. Elements of arrays are stored in the same manner as if they were individual objects.

#### 6.2.1.3 Field/Structure Alignment

When the compiler allocates space for a structure, it allocates as many words as are needed to hold all of the structure's members and to comply with alignment constraints for each member.

When a structure contains a 32-bit (long) member, the long is aligned to a 1-word (16-bit) boundary. This may require padding before, inside, or at the end of the structure to ensure that the long is aligned accordingly and that the sizeof value for the structure is an even value.

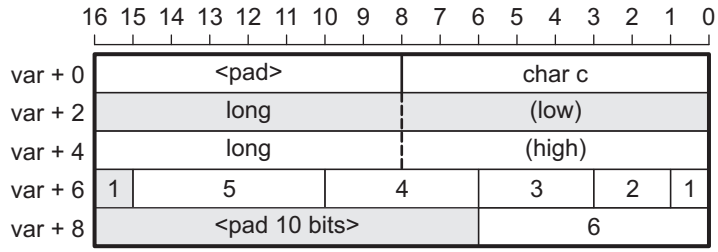
All non-field types are aligned on word or byte boundaries. Fields are allocated as many bits as requested. Adjacent fields are packed into adjacent bits of a word, but they do not overlap words. If a field would overlap into the next word, the entire field is placed into the next word.

Fields are packed as they are encountered; the least significant bits of the structure word are filled first. Example 6-1 shows the C code definition of var while Figure 6-1 shows the memory layout of var.

***Example 6-1. C Code Definition of var***

```
struct example { char c; long l; int bf1:1; int bf2:2; int bf3:3; int bf4:4; int bf5:5; int bf6:6; };
```

**Figure 6-1. Memory Layout of var**




### 6.2.2 Character String Constants

In C, a character string constant is used in one of the following ways:

- To initialize an array of characters. For example:

  ```
  char s[] = "abc";
  ```

  When a string is used as an initializer, it is simply treated as an initialized array; each character is a separate initializer. For more information about initialization, see Section 6.8.

- In an expression. For example:

  ```
  strcpy (s, "abc");
  ```

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

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

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

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

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

  ```
  const char   *a = "abc"
  a[1] = 'x';             /* Incorrect! */
  ```

## 6.3 Register Conventions

Strict conventions associate specific registers with specific operations in the C/C++ environment. If you plan to interface an assembly language routine to a C/C++ program, you must understand and follow these register conventions.

The register conventions dictate how the compiler uses registers and how values are preserved across function calls. Table 6-3 shows the types of registers affected by these conventions.Table 6-4 summarizes how the compiler uses registers and whether their values are preserved across calls. For information about how values are preserved across calls, see Section 6.4.

**Table 6-3. How Register Types Are Affected by the Conventions**

| Register Type | Description |
|---|---|
| Argument register | Passes arguments during a function call |
| Return register | Holds the return value from a function call |
| Expression register | Holds a value |
| Argument pointer | Used as a base value from which a function's parameters (incoming arguments) are accessed |
| Stack pointer | Holds the address of the top of the software stack |
| Program counter | Contains the current address of code being executed |

**Table 6-4. Register Usage and Preservation Conventions**

| Register | Alias | Usage | Preserved by Function (1) |
|---|---|---|---|
| R0 | PC | Program counter | N/A |
| R1 | SP | Stack pointer | N/A(2) |
| R2 | SR | Status register | N/A |
| R3 | | Constant generator | N/A |
| R4-R10 | | Expression register | Child |
| R11 | | Expression register | Parent |
| R12 | | Expression register, argument pointer, return register | Parent |
| R13 | | Expression register, argument pointer, return register | Parent |
| R14 | | Expression register, argument pointer | Parent |
| R15 | | Expression register, argument pointer | Parent |

(1) The parent function refers to the function making the function call. The child function refers to the function being called.
(2) The SP is preserved by the convention that everything pushed on the stack is popped off before returning.

## 6.4 Function Structure and Calling Conventions

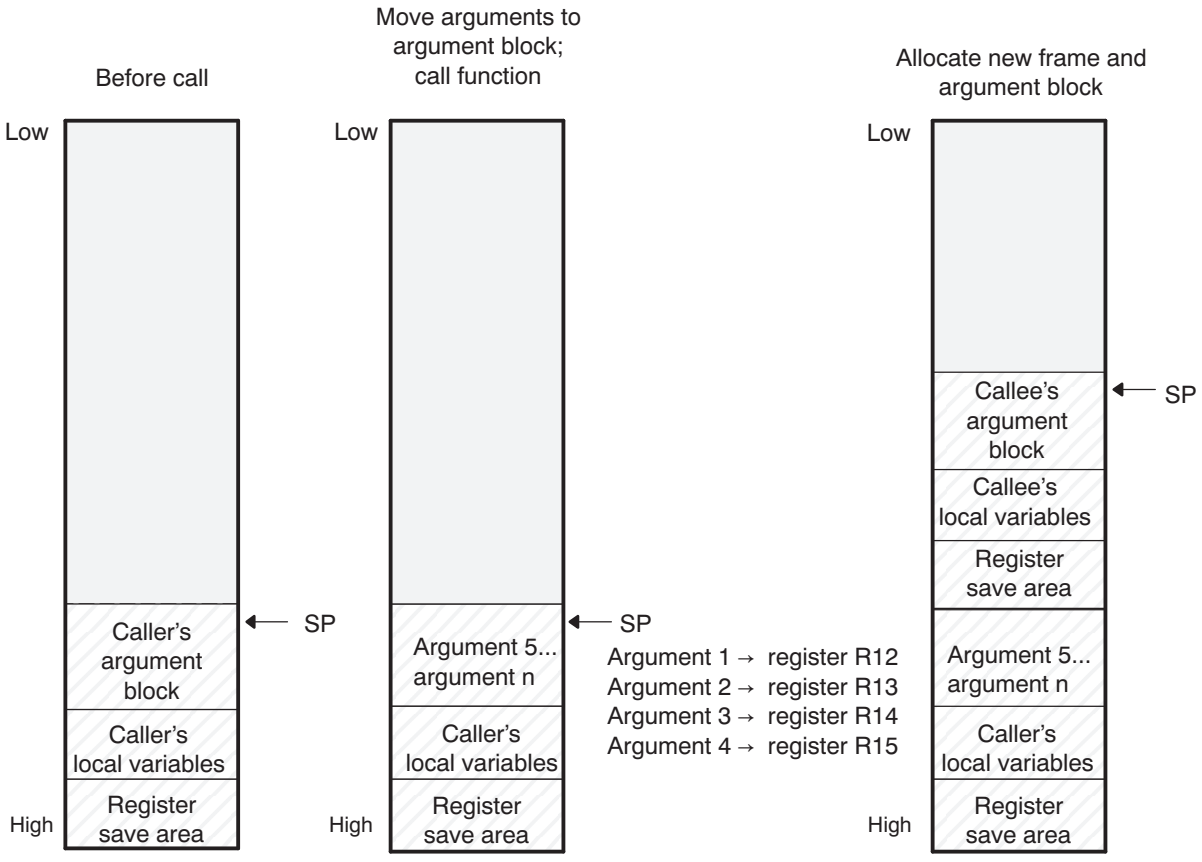
The C/C++ compiler imposes a strict set of rules on function calls. Except for special run-time support functions, any function that calls or is called by a C/C++ function must follow these rules. Failure to adhere to these rules can disrupt the C/C++ environment and cause a program to fail.

The following sections use this terminology to describe the function-calling conventions of the C/C++ compiler:

- **Argument block**. The part of the local frame used to pass arguments to other functions. Arguments are passed to a function by moving them into the argument block rather than pushing them on the stack. The local frame and argument block are allocated at the same time.
- **Register save area**. The part of the local frame that is used to save the registers when the program calls the function and restore them when the program exits the function.
- **Save-on-call registers**. Registers R11-R15. The called function does not preserve the values in these registers; therefore, the calling function must save them if their values need to be preserved.
- **Save-on-entry registers**. Registers R4-R10. It is the called function's responsibility to preserve the values in these registers. If the called function modifies these registers, it saves them when it gains control and preserves them when it returns control to the calling function.

Figure 6-2 illustrates a typical function call. In this example, arguments are passed to the function, and the function uses local variables and calls another function. The first four arguments are passed to registers R12-R15. This example also shows allocation of a local frame and argument block for the called function. Functions that have no local variables and do not require an argument block do not allocate a local frame.

**Figure 6-2. Use of the Stack During a Function Call**




**Legend:** SP: stack pointer

### 6.4.1 How a Function Makes a Call

A function (parent function) performs the following tasks when it calls another function (child function).

1. The calling function (parent) is responsible for preserving any save-on-call registers across the call that are live across the call. (The save-on-call registers are R11-R15.)
2. If the called function (child) returns a structure, the caller allocates space for the structure and passes the address of that space to the called function as the first argument.
3. The caller places the first arguments in registers R12-R15, in that order. The caller moves the remaining arguments to the argument block in reverse order, placing the leftmost remaining argument at the lowest address. Thus, the leftmost remaining argument is placed at the top of the stack.
4. The caller calls the function.

### 6.4.2 How a Called Function Responds

A called function (child function) must perform the following tasks:

1. If the function is declared with an ellipsis, it can be called with a variable number of arguments. The called function pushes these arguments on the stack if they meet both of these criteria:
   - The argument includes or follows the last explicitly declared argument.
   - The argument is passed in a register.
2. The called function pushes register values of all the registers that are modified by the function and that must be preserved upon exit of the function onto the stack. Normally, these registers are the save-on-entry registers (R4-R10) if the function contains calls. If the function is an interrupt, additional registers may need to be preserved. For more information, see Section 6.6.
3. The called function allocates memory for the local variables and argument block by subtracting a constant from the SP. This constant is computed with the following formula:

   *size of all local variables + max = constant*

   The *max* argument specifies the size of all parameters placed in the argument block for each call.
4. The called function executes the code for the function.
5. If the called function returns a value, it places the value in R12 (or R12 and R13 values).
6. If the called function returns a structure, it copies the structure to the memory block that the first argument, R12, points to. If the caller does not use the return value, R12 is set to 0. This directs the called function not to copy the return structure.

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

   You must be careful to properly declare functions that return structures, both at the point where they are called (so the caller properly sets up the first argument) and at the point where they are declared (so the function knows to copy the result).
7. The called function deallocates the frame and argument block by adding the constant computed in .
8. The called function restores all registers saved in .
9. The called function ( _f) returns.

The following example is typical of how a called function responds to a call:

```
func:                   ; Called function entry point
        PUSH.W    r10
        PUSH.W    r9    ; Save SOE registers
        SUB.W     #2,SP ; Allocate the frame
        :
        :               ; Body of function
        :
        ADD.W     #2,SP ; Deallocate the frame
        POP       r9    ; Restore SOE registers
        POP       r10
        RET             ; Return
```

### 6.4.3 Accessing Arguments and Local Variables

A function accesses its local nonregister variables indirectly through the stack pointer (SP or R1) and its stack arguments. The SP always points to the top of the stack (points to the most recently pushed value).

Since the stack grows toward smaller addresses, the local data on the stack for the C/C++ function is accessed with a positive offset from the SP register.

## 6.5 Interfacing C and C++ With Assembly Language

The following are ways to use assembly language with C/C++ code:

- Use separate modules of assembled code and link them with compiled C/C++ modules (see Section 6.5.1).
- Use assembly language variables and constants in C/C++ source (see Section 6.5.2).
- Use inline assembly language embedded directly in the C/C++ source (see Section 6.5.4).

### 6.5.1 Using Assembly Language Modules With C/C++ Code

Interfacing C/C++ with assembly language functions is straightforward if you follow the calling conventions defined in Section 6.4, and the register conventions defined in Section 6.3. C/C++ code can access variables and call functions defined in assembly language, and assembly code can access C/C++ variables and call C/C++ functions.

Follow these guidelines to interface assembly language and C:

- You must preserve any dedicated registers modified by a function. Dedicated registers include:
  - Save-on-entry registers (R4-R10)
  - Stack pointer (SP or R1)

  If the SP is used normally, it does not need to be explicitly preserved. In other words, the assembly function is free to use the stack as long as anything that is pushed onto the stack is popped back off before the function returns (thus preserving SP).

  Any register that is not dedicated can be used freely without first being saved.
- Interrupt routines must save *all* the registers they use. For more information, see Section 6.6.
- When you call a C/C++ function from assembly language, load the designated registers with arguments and push the remaining arguments onto the stack as described in Section 6.4.1.

  Remember that a function can alter any register not designated as being preserved without having to restore it. If the contents of any of these registers must be preserved across the call, you must explicitly save them.
- Functions must return values correctly according to their C/C++ declarations. Double values are returned in R12 and R13, and structures are returned as described in of Section 6.4.1. Any other values are returned in R12.
- No assembly module should use the .cinit section for any purpose other than autoinitialization of global variables. The C/C++ startup routine assumes that the .cinit section consists *entirely* of initialization tables. Disrupting the tables by putting other information in .cinit can cause unpredictable results.
- The compiler assigns linknames to all external objects. Thus, when you are writing assembly language code, you must use the same linknames as those assigned by the compiler. See Section 5.11 for more information.
- Any object or function declared in assembly language that is accessed or called from C/C++ must be declared with the .def or .global directive in the assembly language modifier. This declares the symbol as external and allows the linker to resolve references to it.

  Likewise, to access a C/C++ function or object from assembly language, declare the C/C++ object with the .ref or .global directive in the assembly language module. This creates an undeclared external reference that the linker resolves.

- Any assembly routines that interface with MSP430x C programs are required to conform to the large-code model:
  - Use CALLA/RETA instead of CALL/RET
  - Use PUSHM.A/POPM.A to save and restore any used save-on-entry registers. The entire 20-bit register must be saved/restored.
  - Manipulation of function pointers requires 20-bit operations (OP.A)
  - If interfacing with C code compiled for the large-data model, data pointer manipulation must be performed using 20-bit operations (OP.A).

Example 6-2 illustrates a C++ function called main, which calls an assembly language function called asmfunc, Example 6-3. The asmfunc function takes its single argument, adds it to the C++ global variable called gvar, and returns the result.

***Example 6-2. Calling an Assembly Language Function From a C/C++ Program***

```
extern "C" {
extern int asmfunc(int a); /* declare external asm function */
int gvar = 0;              /* define global variable        */
}

void main()
{
   int I = 5;

   I = asmfunc(I);         /* call function normally       */
```

***Example 6-3. Assembly Language Program Called by Example 6-2***

```
      .global asmfunc
      .global gvar
asmfunc:
      MOV   &gvar,R11
      ADD   R11,R12
      RET
```

In the C++ program in Example 6-2, the extern "C" declaration tells the compiler to use C naming conventions (i.e., no name mangling). When the linker resolves the .global asmfunc reference, the corresponding definition in the assembly file will match.

The parameter I is passed in R12, and the result is returned in R12.

### 6.5.2 Accessing Assembly Language Variables From C/C++

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

#### 6.5.2.1 Accessing Assembly Language Global Variables

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

1. Use the .bss or .usect directive to define the variable.
2. Use the .def or .global directive to make the definition external.
3. Use the appropriate linkname in assembly language.
4. In C/C++, declare the variable as *extern* and access it normally.

Example 6-5 and Example 6-4 show how you can access a variable defined in .bss.

***Example 6-4. Assembly Language Variable Program***

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

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

***Example 6-5. C Program to Access Assembly Language From Example 6-4***

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

#### 6.5.2.2 Accessing Assembly Language Constants

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

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

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

You can use casts and #defines to ease the use of these symbols in your program, as in Example 6-6 and Example 6-7.

***Example 6-6. Accessing an Assembly Language Constant From C***

```
extern int table_size;        /*external ref */
#define TABLE_SIZE ((int) (&table_size))
      .                       /* use cast to hide address-of */
      .
      .
for (I=0; i<TABLE_SIZE; ++I)  /* use like normal symbol */
```

***Example 6-7. Assembly Language Program for Example 6-6***

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

Because you are referencing only the symbol's value as stored in the symbol table, the symbol's declared type is unimportant. In Example 6-6, int is used. You can reference linker-defined symbols in a similar manner.

### *6.5.3 Sharing C/C++ Header Files With Assembly Source*

You can use the .cdecls assembler directive to share C headers containing declarations and prototypes between C and assembly code. Any legal C/C++ can be used in a .cdecls block and the C/C++ declarations will cause suitable assembly to be generated automatically, allowing you to reference the C/C++ constructs in assembly code. For more information, see the C/C++ header files chapter in the *MSP430 Assembly Language Tools User's Guide*.

### 6.5.4 Using Inline Assembly Language

Within a C/C++ program, you can use the asm statement to insert a single line of assembly language into the assembly language file created by the compiler. A series of asm statements places sequential lines of assembly language into the compiler output with no intervening code. For more information, see Section 5.8.

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

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

**NOTE: Using the asm Statement**

Keep the following in mind when using the asm statement:

- Be extremely careful not to disrupt the C/C++ environment. The compiler does not check or analyze the inserted instructions.
- Avoid inserting jumps or labels into C/C++ code because they can produce unpredictable results by confusing the register-tracking algorithms that the code generator uses.
- Do not change the value of a C/C++ variable when using an asm statement. This is because the compiler does not verify such statements. They are inserted as is into the assembly code, and potentially can cause problems if you are not sure of their effect.
- Do not use the asm statement to insert assembler directives that change the assembly environment.
- Avoid creating assembly macros in C code and compiling with the --symdebug:dwarf (or -g) option. The C environment's debug information and the assembly macro expansion are not compatible.

## 6.6 Interrupt Handling

As long as you follow the guidelines in this section, you can interrupt and return to C/C++ code without disrupting the C/C++ environment. When the C/C++ environment is initialized, the startup routine does not enable or disable interrupts. If the system is initialized by way of a hardware reset, interrupts are disabled. If your system uses interrupts, you must handle any required enabling or masking of interrupts. Such operations have no effect on the C/C++ environment and are easily incorporated with asm statements or calling an assembly language function.

### 6.6.1 Saving Registers During Interrupts

When C/C++ code is interrupted, the interrupt routine must preserve the contents of all machine registers that are used by the routine or by any functions called by the routine. Register preservation must be explicitly handled by the interrupt routine.

### 6.6.2 Using C/C++ Interrupt Routines

A C/C++ interrupt routine is like any other C/C++ function in that it can have local variables and register variables. Except for software interrupt routines, an interrupt routine must be declared with no arguments and must return void. For example:

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

If a C/C++ interrupt routine does not call any other functions, only those registers that the interrupt handler uses are saved and restored. However, if a C/C++ interrupt routine does call other functions, these functions can modify unknown registers that the interrupt handler does not use. For this reason, the routine saves all the save-on-call registers if any other functions are called. (This excludes banked registers.) Do not call interrupt handling functions directly.

Interrupts can be handled directly with C/C++ functions by using the interrupt pragma or the interrupt keyword. For information, see Section 5.9.14 and Section 5.5.2, respectively.

### 6.6.3 Using Assembly Language Interrupt Routines

You can handle interrupts with assembly language code as long as you follow the same register conventions the compiler does. Like all assembly functions, interrupt routines can use the stack, access global C/C++ variables, and call C/C++ functions normally. When calling C/C++ functions, be sure that any save-on-call registers are preserved before the call because the C/C++ function can modify any of these registers. You do not need to save save-on-entry registers because they are preserved by the called C/C++ function.

### 6.6.4 Interrupt Vectors

The interrupt vectors for the MSP430 and MSP430X devices are 16 bits. Therefore, interrupt service routines (ISRs) must be placed into the low 64K of memory. Convenience macros are provided in the MSP430X device headers file to declare interrupts to ensure 16-bit placement when linking.

Alternatively, use the CODE_SECTIONS pragma to place the code for ISRs into sections separate from the default .text sections. Use the linker command file and the SECTIONS directive to ensure the code sections associated with ISRs are placed into low memory.

### 6.6.5 Other Interrupt Information

An interrupt routine can perform any task performed by any other function, including accessing global variables, allocating local variables, and calling other functions.

When you write interrupt routines, keep the following points in mind:

- It is your responsibility to handle any special masking of interrupts.
- A C/C++ interrupt routine cannot be called explicitly.
- In a system reset interrupt, such as c_int00, you cannot assume that the run-time environment is set up; therefore, you *cannot allocate local variables*, and you *cannot save any information on the run-time stack*.
- In assembly language, remember to precede the name of a C/C++ interrupt with the appropriate linkname. For example, refer to c_int00 as _c_int00.

## 6.7 Using Intrinsics to Access Assembly Language Statements

The compiler recognizes a number of intrinsic operators. Intrinsics are used like functions and produce assembly language statements that would otherwise be inexpressible in C/C++. You can use C/C++ variables with these intrinsics, just as you would with any normal function. The intrinsics are specified with a leading underscore, and are accessed by calling them as you do a function. For example:

```
short state;
   :
state = _get_SR_register();
```

No declaration of the intrinsic functions is necessary.

### 6.7.1 MSP430 Intrinsics

Table 6-5 lists all of the intrinsic operators in the MSP430 C/C++ compiler. A function-like prototype is presented for each intrinsic that shows the expected type for each parameter. If the argument type does not match the parameter, type conversions are performed on the argument.

For more information on the resulting assembly language mnemonics, see the *MSP430x1xx Family User's Guide*, the *MSP430x3xx Family User's Guide*, and the *MSP430x4xx Family User's Guide*.

**Table 6-5. MSP430 Intrinsics**

| Intrinsic | | Generated Assembly |
|---|---|---|
| unsigned short | _bcd_add_short(unsigned short *op1*, unsigned short *op2*); | MOV *op1*, *dst*<br>CLRC<br>DADD *op2*, *dst* |
| unsigned long | _bcd_add_long(unsigned long *op1*, unsigned long *op2*); | MOV *op1_low*, *dst_low*<br>MOV *op1_hi*, *dst_hi*<br>CLRC<br>DADD *op2_low*, *dst_low*<br>DADD *op2_hi*, *dst_hi* |
| unsigned short | _bic_SR_register(unsigned short *mask*); | BIC *mask*, SR |
| unsigned short | _bic_SR_register_on_exit(unsigned short *mask*); | BIC *mask*, saved_SR |
| unsigned short | _bis_SR_register(unsigned short *mask*); | BIS *mask*, SR |
| unsigned short | _bis_SR_register_on_exit(unsigned short *mask*); | BIS *mask*, saved_SR |
| unsigned long | _data16_read_addr(unsigned short *addr*); | MOV.W addr, Rx<br>MOVA 0(Rx), dst |
| void | _data16_write_addr (unsigned short *addr*, unsigned long *src*); | MOV.W addr, Rx<br>MOVA src, 0(Rx) |
| unsigned char | _data20_read_char(unsigned long *addr*);(1) | MOVA *addr*, Rx<br>MOVX.B 0(Rx), *dst* |
| unsigned long | _data20_read_long(unsigned long *addr*);(1) | MOVA *addr*, Rx<br>MOVX.W 0(Rx), *dst.lo*<br>MOVX.W 2(Rx), *dst.hi* |
| unsigned short | _data20_read_short(unsigned long *addr*);(1) | MOVA *addr*, Rx<br>MOVX.W 0(Rx), *dst* |
| void | _data20_write_char(unsigned long *addr*, unsigned char *src*);(1) | MOVA *addr*, Rx<br>MOVX.B *src*, 0(Rx) |
| void | _data20_write_long(unsigned long *addr*, unsigned long *src*);(1) | MOVA *addr*, Rx<br>MOVX.W *src.lo*, 0(Rx)<br>MOVX.W *src.hi*, 2(Rx) |
| void | _data20_write_short(unsigned long *addr*, unsigned short *src*);(1) | MOVA *addr*, Rx<br>MOVX.W *src*, 0(Rx) |
| void | _delay_cycles(unsigned long); | See Section 6.7.2. |
| void | _disable_interrupt(void);<br>OR<br>_disable_interrupts(void); | DINT |
| void | _enable_interrupt(void);<br>OR<br>_enable_interrupts(void); | EINT |

(1) Intrinsic encodes multiple instructions depending on the code. The most common instructions produced are presented here.

**Table 6-5. MSP430 Intrinsics (continued)**

| Intrinsic | | Generated Assembly |
|---|---|---|
| unsigned int | _even_in_range(unsigned int, unsigned int); | See Section 5.9.16. |
| unsigned short | _get_interrupt_state(void); | MOV SR, *dst* |
| unsigned short | _get_R4_register(void); | MOV.W R4, dst |
| unsigned short | _get_R5_register(void); | MOV.W R5, dst |
| unsigned short | _get_SP_register(void); | MOV SP, *dst* |
| unsigned short | _get_SR_register(void); | MOV SR, *dst* |
| unsigned short | _get_SR_register_on_exit(void); | MOV saved_SR, *dst* |
| void | _low_power_mode_0(void); | BIS.W #0x18, SR |
| void | _low_power_mode_1(void); | BIS.W #0x58, SR |
| void | _low_power_mode_2(void); | BIS.W #0x98, SR |
| void | _low_power_mode_3(void); | BIS.W #0xD8, SR |
| void | _low_power_mode_4(void); | BIS.W #0xF8, SR |
| void | _low_power_mode_off_on_exit(void); | BIC.W #0xF0, saved_SR |
| void | _never_executed(void); | See Section 6.7.3. |
| void | _no_operation(void); | NOP |
| void | _op_code(unsigned short); | Encodes whatever instruction corresponds to the argument. |
| void | _set_interrupt_state(unsigned short *src*); | MOV *src*, SR |
| void | _set_R4_register(unsigned short *src*); | MOV.W *src*, R4 |
| void | _set_R5_register(unsigned short *src*); | MOV.W *src*, R5 |
| void | _set_SP_register(unsigned short *src*); | MOV *src*, SP |
| unsigned short | _swap_bytes(unsigned short *src*); | MOV *src*, *dst*<br>SWPB *dst* |

### 6.7.2 The __delay_cycle Intrinsic

The __delay_cycles intrinsic inserts code to consume precisely the number of specified cycles with no side effects. The number of cycles delayed must be a compile-time constant.

### 6.7.3 The _never_executed Intrinsic

The MSP430 C/C++ Compiler supports a _never_executed( ) intrinsic that can be used to assert that a default label in a switch block is never executed. If you assert that a default label is never executed the compiler can generate more efficient code based on the values specified in the case labels within a switch block.

#### 6.7.3.1 Using _never_executed With a Vector Generator

The _never_executed( ) intrinsic is specifically useful for testing the values of an MSP430 interrupt vector generator such as the vector generator for Timer A (TAIV). MSP430 vector generator values are mapped to an interrupt source and are characterized in that they fall within a specific range and can only take on even values. A common way to handle a particular interrupt source represented in a vector generator is to use a switch statement. However, a compiler is constrained by the C language in that it can make no assumptions about what values a switch expression may have. The compiler will have to generate code to handle every possible value, which leads to what would appear to be inefficient code.

The _never_executed( ) intrinsic can be used to assert to the compiler that a switch expression can only take on values represented by the case labels within a switch block. Having this assertion, the compiler can avoid generating test code for handling values not specified by the switch case labels. Having this assertion is specifically suited for handling values that characterize a vector generator.

Example 6-8 illustrates a switch block that handles the values of the Timer B (TBIV) vector generator.

***Example 6-8. TBIV Vector Generator***

```
__interrupt void Timer_B1 (void)
{
 switch( TBIV )
 {
   case  0: break; /* Do nothing */
   case  2: TBCCR1 += 255;
            state +=1;
            break;
   case  4: TBCCR0 = 254;
            TBCCR1 = 159;
            state =200;
            break;
   case  6: break;
   case  8: break;
   case 10: break;
   case 12: break;
   case 14: break;
       default:  _never_executed();
 }
}
```

In Example 6-8 using the _never_executed( ) intrinsic asserts that the value of TBIV can only take on the values specified by the case labels, namely the even values from 0 to 14. Normally, the compiler would have to generate code to handle any value which would result in extra range checks. Instead, for this example, the compiler will generate a switch table where the value of TBIV is simply added to the PC to jump to the appropriate code block handling each value represented by the case labels.

### 6.7.3.2 Using _never_executed With General Switch Expressions

Using the _never_executed( ) intrinsic at the default label can also improve the generated switch code for more general switch expressions that do not involve vector generator type values.

***Example 6-9. General Switch Statement***

```
switch( val)
{
  case 0:
  case 5: action(a); break;

  case 14: action(b); break;

  default: _never_executed();
}
```

Normally, for the switch expression values 0 and 5, the compiler generates code to test for both 0 and 5 since the compiler must handle the possible values 1−4. The _never_executed( ) intrinsic in Example 6-9 asserts that val cannot take on the values 1−4 and therefore the compiler only needs to generate a single test (val < 6) to handle both case labels.

Additionally, using the _never_executed( ) intrinsic results in the assertion that if val is not 0 or 5 then it has to be 14 and the compiler has no need to generate code to test for val == 14.

The _never_executed( ) intrinsic is only defined when specified as the single statement following a default case label. The compiler ignores the use of the intrinsic in any other context.

## 6.8 System Initialization

Before you can run a C/C++ program, you must create the C/C++ run-time environment. The C/C++ boot routine performs this task using a function called c_int00 (or _c_int00). The run-time-support source library, rts.src, contains the source to this routine in a module named boot.c (or boot.asm).

To begin running the system, the c_int00 function can be called by reset hardware. You must link the c_int00 function with the other object modules. This occurs automatically when you use the --rom_model or --ram_model link option and include a standard run-time-support library as one of the linker input files.

When C/C++ programs are linked, the linker sets the entry point value in the executable output module to the symbol c_int00.

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

1. Reserves space for the user mode run-time stack, and sets up the initial value of the stack pointer (SP)
2. It initializes global variables by copying the data from the initialization tables to the storage allocated for the variables in the .bss section. If you are initializing variables at load time (--ram_model option), a loader performs this step before the program runs (it is not performed by the boot routine). For more information, see Section 6.8.3.
3. Executes the global constructors found in the global constructors table. For more information, see Section 6.8.7.
4. Calls the function main to run the C/C++ program

You can replace or modify the boot routine to meet your system requirements. However, the boot routine *must* perform the operations listed above to correctly initialize the C/C++ environment.

### *6.8.1 System Pre-Initialization*

The _c_int00( ) initialization routine also provides a mechanism for an application to perform the MSP430 setup (set I/O registers, enable/disable timers, etc.) before the C/C++ environment is initialized.

Before calling the routine that initializes C/C++ global data and calls any C++ constructors, the boot routine makes a call to the function _system_pre_init( ). A developer can implement a customized version of _system_pre_init( ) to perform any application-specific initialization before proceeding with C/C++ environment setup. In addition, the default C/C++ data initialization can be bypassed if _system_pre_init( ) returns a 0. By default, _system_pre_init( ) should return a non-zero value.

In order to perform application-specific initializations, you can create a customized version of _system_pre_init( ) and add it to the application project. The customized version will replace the default definition included in the run-time library if it is linked in before the run-time library.

The default stubbed version of _system_pre_init( ) is included with the run-time library. It is located in the file pre_init.c and is included in the run-time source library (rts.src). The archiver utility (ar430) can be used to extract pre_init.c from the source library.

### *6.8.2 Run-Time Stack*

The run-time stack is allocated in a single continuous block of memory and grows down from high addresses to lower addresses. The SP points to the top of the stack.

The code does not check to see if the run-time stack overflows. Stack overflow occurs when the stack grows beyond the limits of the memory space that was allocated for it. Be sure to allocate adequate memory for the stack.

The stack size can be changed at link time by using the --stack_size link option on the linker command line and specifying the stack size as a constant directly after the option.

The C/C++ boot routine shipped with the compiler sets up the user/thread mode run-time stack. If your program uses a run-time stack when it is in other operating modes, you must also allocate space and set up the run-time stack corresponding to those modes.

### 6.8.3 Automatic Initialization of Variables

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

The compiler builds tables in a special section called .cinit that contains data for initializing global and static variables. Each compiled module contains these initialization tables. The linker combines them into a single table (a single .cinit section). The boot routine or a loader uses this table to initialize all the system variables.

**NOTE: Initializing Variables**

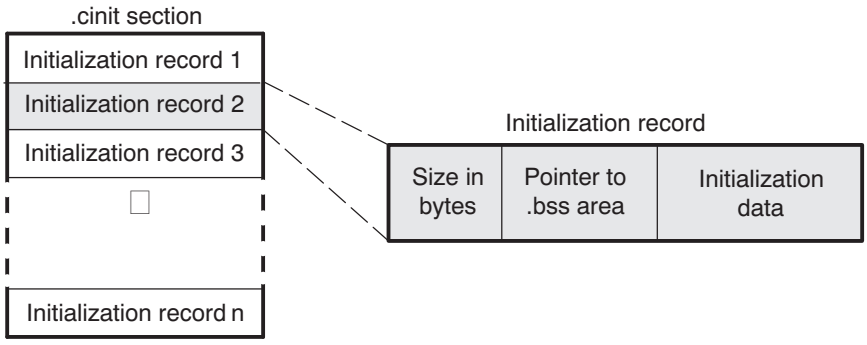
In ANSI/ISO C, global and static variables that are not explicitly initialized must be set to 0 before program execution. The C/C++ compiler does not perform any preinitialization of uninitialized variables. Explicitly initialize any variable that must have an initial value of 0.

Global variables are either autoinitialized at run time or at load time; see Section 6.8.5 and Section 6.8.6. Also see Section 5.12.

### 6.8.4 Initialization Tables

The tables in the .cinit section consist of variable-size initialization records. Each variable that must be autoinitialized has a record in the .cinit section. Figure 6-3 shows the format of the .cinit section and the initialization records.

**Figure 6-3. Format of Initialization Records in the .cinit Section**




The fields of an initialization record contain the following information:

- The first field of an initialization record contains the size (in bytes) of the initialization data. The width of this field is one word (16-bit).
- The second field contains the starting address of the area within the .bss section where the initialization data must be copied. The width of this field is one word.
- The third field contains the data that is copied into the .bss section to initialize the variable. The width of this field is variable.

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

Example 6-10 shows initialized global variables defined in C. Example 6-11 shows the corresponding initialization table.

***Example 6-10. Initialized Variables Defined in C***

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

***Example 6-11. Initialized Information for Variables Defined in Example 6-10***

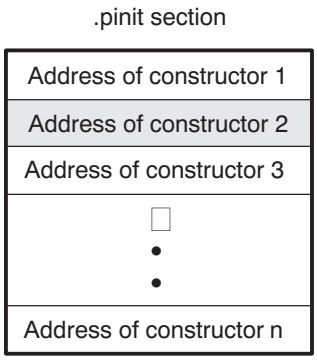
```
   .sect    ".cinit"
   .align   2
   .field   2,16
   .field   i+0,16
   .field   23,16          ; i @ 0

   .sect    ".cinit"
   .align   2
   .field   $C$IR_1,16
   .field   a+0,16
   .field   1,16           ; a[0] @ 0
   .field   2,16           ; a[1] @ 16
   .field   3,16           ; a[2] @ 32
   .field   4,16           ; a[3] @ 48
   .field   5,16           ; a[4] @ 64
$C$IR_1:    .set 10
   .global  i
   .bss     i,2,2
   .global  a
   .bss     a,10,2
```

The .cinit section must contain only initialization tables in this format. When interfacing assembly language modules, do not use the .cinit section for any other purpose.

The table in the .pinit section simply consists of a list of addresses of constructors to be called (see Figure 6-4). The constructors appear in the table after the .cinit initialization.

**Figure 6-4. Format of Initialization Records in the .pinit Section**




When you use the --rom_model or --ram_model option, the linker combines the .cinit sections from all the C modules and appends a null word to the end of the composite .cinit section. This terminating record appears as a record with a size field of 0 and marks the end of the initialization tables.

Likewise, the --rom_model or --ram_model link option causes the linker to combine all of the .pinit sections from all C/C++ modules and append a null word to the end of the composite .pinit section. The boot routine knows the end of the global constructor table when it encounters a null constructor address.

The const-qualified variables are initialized differently; see Section 5.5.1.

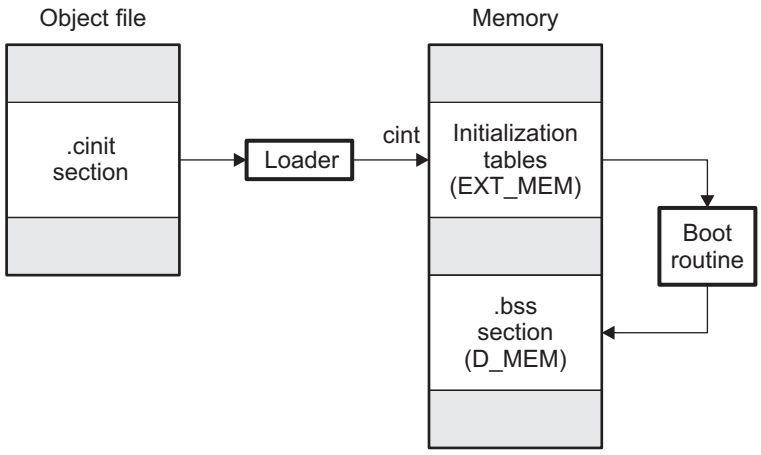
### 6.8.5 Autoinitialization of Variables at Run Time

Autoinitializing variables at run time is the default method of autoinitialization. To use this method, invoke the linker with the --rom_model option.

Using this method, the .cinit section is loaded into memory along with all the other initialized sections, and global variables are initialized at run time. The linker defines a special symbol called cinit that points to the beginning of the initialization tables in memory. When the program begins running, the C/C++ boot routine copies data from the tables (pointed to by .cinit) into the specified variables in the .bss section. This allows initialization data to be stored in ROM and copied to RAM each time the program starts.

Figure 6-5 illustrates autoinitialization at run time. Use this method in any system where your application runs from code burned into ROM.

**Figure 6-5. Autoinitialization at Run Time**




### 6.8.6 Initialization of Variables at Load Time

Initialization of variables at load time enhances performance by reducing boot time and by saving the memory used by the initialization tables. To use this method, invoke the linker with the --ram_model option.

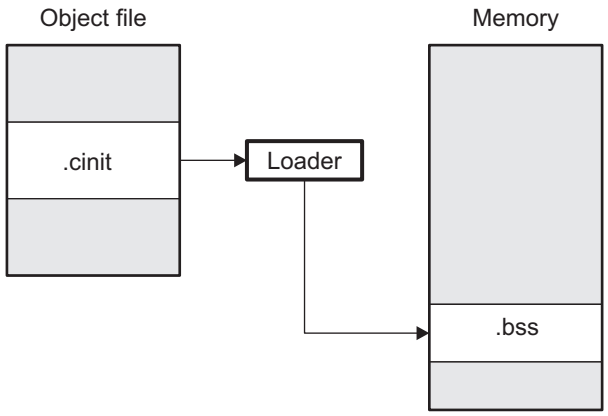
When you use the --ram_model link option, the linker sets the STYP_COPY bit in the .cinit section's header. This tells the loader not to load the .cinit section into memory. (The .cinit section occupies no space in the memory map.) The linker also sets the cinit symbol to -1 (normally, cinit points to the beginning of the initialization tables). This indicates to the boot routine that the initialization tables are not present in memory; accordingly, no run-time initialization is performed at boot time.

A loader (which is not part of the compiler package) must be able to perform the following tasks to use initialization at load time:

- Detect the presence of the .cinit section in the object file
- Determine that STYP_COPY is set in the .cinit section header, so that it knows not to copy the .cinit section into memory
- Understand the format of the initialization tables

Figure 6-6 illustrates the initialization of variables at load time.

**Figure 6-6. Initialization at Load Time**




Regardless of the use of the --rom_model or --ram_model options, the .pinit section is always loaded and processed at run time.

### 6.8.7 Global Constructors

All global C++ variables that have constructors must have their constructor called before main (). The compiler builds a table of global constructor addresses that must be called, in order, before main () in a section called .pinit. The linker combines the .pinit section form each input file to form a single table in the .pinit section. The boot routine uses this table to execute the constructors.

## 6.9 Compiling for 20-Bit MSP430X Devices

The MSP430 tools support compiling and linking code for MSP430 and MSP430X (MSP430X) devices. See the following for more information on options and topics that apply to compiling for the MSP430X devices:

- Use the --silicon_version=mspx option to compile for MSP430X devices. See Section 2.3.3.
- Function pointers are 20-bits. See Table 5-1 and .
- The compiler supports a large-code memory model while generating code for MSP430X devices. See Section 6.1.1.
- The compiler supports a large-data memory model while generating code for MSP430X devices. See Section 6.1.2.
- Any assembly routines that interface with MSP430X C programs must fit the large code model. See Section 6.5.1.
- Interrupt service routines must be placed into low memory. See Section 6.6.4.
- Link with the rts430x.lib or rts430x_eh.lib run-time-support library.

# Using Run-Time-Support Functions and Building Libraries

Some of the tasks that a C/C++ program performs (such as I/O, dynamic memory allocation, string operations, and trigonometric functions) are not part of the C/C++ language itself. However, the ANSI/ISO C standard defines a set of run-time-support functions that perform these tasks. The C/C++ compiler implements the complete ISO standard library except for those facilities that handle exception conditions and locale issues (properties that depend on local language, nationality, or culture). Using the ANSI/ISO standard library ensures a consistent set of functions that provide for greater portability.

In addition to the ANSI/ISO-specified functions, the run-time-support library includes routines that give you processor-specific commands and direct C language I/O requests. These are detailed inSection 7.1 and Section 7.2.

A library-build process is provided with the code generation tools that lets you create customized run-time-support libraries. This process is described in Section 7.4 .

**Topic** **Page**

## 7.1 C and C++ Run-Time Support Libraries

MSP430 compiler releases include pre-built run-time libraries that provide all the standard capabilities. Separate libraries are provided for and C++ exception support. See Section 7.4 for information on the library-naming conventions.

The run-time-support library contains the following:

- ANSI/ISO C/C++ standard library
- C I/O library
- Low-level support functions that provide I/O to the host operating system
- Intrinsic arithmetic routines
- System startup routine, _c_int00
- Functions and macros that allow C/C++ to access specific instructions

The run-time-support libraries do not contain functions involving signals and locale issues.

The C++ library supports wide chars, in that template functions and classes that are defined for char are also available for wide char. For example, wide char stream classes wios, wiostream, wstreambuf and so on (corresponding to char classes ios, iostream, streambuf) are implemented. However, there is no low-level file I/O for wide chars. Also, the C library interface to wide char support (through the C++ headers <cwchar> and <cwctype>) is limited as described in Section 5.1.

The C++ library included with the compiler is licensed from Dinkumware, Ltd. The Dinkumware C++ library is a fully conforming, industry-leading implementation of the standard C++ library.

TI does not provide documentation that covers the functionality of the C++ library. TI suggests referring to one of the following sources:

- *The Standard C++ Library: A Tutorial and Reference*,Nicolai M. Josuttis, Addison-Wesley, ISBN 0-201-37926-0
- *The C++ Programming Language* (Third or Special Editions), Bjarne Stroustrup, Addison-Wesley, ISBN 0-201-88954-4 or 0-201-70073-5
- Dinkumware's online reference at http://dinkumware.com/manuals

### 7.1.1 Linking Code With the Object Library

When you link your program, you must specify the object library as one of the linker input files so that references to the I/O and run-time-support functions can be resolved. You can either specify the library or allow the compiler to select one for you. See Section 4.3.1 for further information.

You should specify libraries *last* on the linker command line because the linker searches a library for unresolved references when it encounters the library on the command line. You can also use the --reread_libs linker option to force repeated searches of each library until the linker can resolve no more references.

When a library is linked, the linker includes only those library members required to resolve undefined references. For more information about linking, see the *MSP430 Assembly Language Tools User's Guide*.

C, C++, and mixed C and C++ programs can use the same run-time-support library. Run-time-support functions and variables that can be called and referenced from both C and C++ will have the same linkage.

### 7.1.2 Header Files

To include the correct set of header files depending on which library you are using, you can set the MSP_C_DIR environment variable to the specific include directory: "include\lib". The source for the libraries is included in the rtssrc.zip file. See Section 7.4 for details on rebuilding.

### 7.1.3 Modifying a Library Function

You can inspect or modify library functions by unzipping the source file (rtssrc.zip), changing the specific function file, and rebuilding the library. When extracted (with any standard unzip tool on windows, linux, or unix), this zip file will recreate the run-time source tree for the run-time library.

You can also build a new library this way, rather than rebuilding into rts430.lib. See Section 7.4.

### 7.1.4 Changes to the Run-Time-Support Libraries

The following changes and additions apply to the run-time-support libraries in the /lib subdirectory of the release package.

#### 7.1.4.1 Minimal Support for Internationalization

The library now includes the header files <locale.h>, <wchar.h>, and <wctype.h>, which provide APIs to support non-ASCII character sets and conventions. Our implementation of these APIs is limited in the following ways:

- The library has minimal support for wide and multi-byte characters. The type wchar_t is implemented as int. The wide character set is equivalent to the set of values of type char. The library includes the header files <wchar.h> and <wctype.h> but does not include all the functions specified in the standard. So-called multi-byte characters are limited to single characters. There are no shift states. The mapping between multi-byte characters and wide characters is simple equivalence; that is, each wide character maps to and from exactly a single multi-byte character having the same value.
- The C library includes the header file <locale.h> but with a minimal implementation. The only supported locale is the C locale. That is, library behavior that is specified to vary by locale is hard-coded to the behavior of the C locale, and attempting to install a different locale via a call to setlocale() will return NULL.

#### 7.1.4.2 Allowable Number of Open Files

In the <stdio.h> header file, the value for the macro FOPEN_MAX has been changed from 12 to the value of the macro _NFILE, which is set to 10. The impact is that you can only have 10 files simultaneously open at one time (including the pre-defined streams - stdin, stdout, stderr).

The C standard requires that the minimum value for the FOPEN_MAX macro is 8. The macro determines the maximum number of files that can be opened at one time. The macro is defined in the stdio.h header file and can be modified by changing the value of the _NFILE macro.

### 7.1.5 Nonstandard Header Files in rtssrc.zip

The rtssrc.zip self-processing zip file contains these non-ANSI include files that are used to build the library:

- The *values.h* file contains the definitions necessary for recompiling the trigonometric and transcendental math functions. If necessary, you can customize the functions in values.h.
- The *file.h* file includes macros and definitions used for low-level I/O functions.
- The *format.h* file includes structures and macros used in printf and scanf.
- The *430cio.h* file includes low-level, target-specific C I/O macro definitions. If necessary, you can customize 430cio.h.
- The *rtti.h* file includes internal function prototypes necessary to implement run-time type identification.
- The *vtbl.h* file contains the definition of a class's virtual function table format.

### 7.1.6 Library Naming Conventions

The run-time support libraries now have the following naming scheme:

**rts430**[x[l]][_eh].lib

| | |
|---|---|
| **rts430** | Indicates an MSP430 library. |
| x | Optional x indicates an MSP430X library. |
| l | Optional l after x indicates a large-data model MSP430X library. |
| *_eh* | Indicates the library has exception handling support |

## 7.2 The C I/O Functions

The C I/O functions make it possible to access the host's operating system to perform I/O. The capability to perform I/O on the host gives you more options when debugging and testing code.

The I/O functions are logically divided into layers: high level, low level, and device-driver level.

With properly written device drivers, the C-standard high-level I/O functions can be used to perform I/O on custom user-defined devices. This provides an easy way to use the sophisticated buffering of the high-level I/O functions on an arbitrary device.

**NOTE: C I/O Mysteriously Fails**

If there is not enough space on the heap for a C I/O buffer, operations on the file will silently fail. If a call to printf() mysteriously fails, this may be the reason. The heap needs to be at least large enough to allocate a block of size BUFSIZ (defined in stdio.h) for every file on which I/O is performed, including stdout, stdin, and stderr, plus allocations performed by the user's code, plus allocation bookkeeping overhead. Alternately, declare a char array of size BUFSIZ and pass it to setvbuf to avoid dynamic allocation. To set the heap size, use the --heap_size option when linking (see ).

**NOTE: Open Mysteriously Fails**

The run-time support limits the total number of open files to a small number relative to general-purpose processors. If you attempt to open more files than the maximum, you may find that the open will mysteriously fail. You can increase the number of open files by extracting the source code from rts.src and editing the constants controlling the size of some of the C I/O data structures. The macro _NFILE controls how many FILE (fopen) objects can be open at one time (stdin, stdout, and stderr count against this total). (See also FOPEN_MAX.) The macro _NSTREAM controls how many low-level file descriptors can be open at one time (the low-level files underlying stdin, stdout, and stderr count against this todal). The macro _NDEVICE controls how many device drivers are installed at one time (the HOST device counts against this total).

### 7.2.1 High-Level I/O Functions

The high-level functions are the standard C library of stream I/O routines (printf, scanf, fopen, getchar, and so on). These functions call one or more low-level I/O functions to carry out the high-level I/O request. The high-level I/O routines operate on FILE pointers, also called *streams*.

Portable applications should use only the high-level I/O functions.

To use the high-level I/O functions, include the header file stdio.h, or cstdio for C++ code, for each module that references a C I/O function.

For example, given the following C program in a file named main.c:

```
#include <stdio.h>;

void main()
{
   FILE *fid;

   fid = fopen("myfile","w");
   fprintf(fid,"Hello, world\n");
   fclose(fid);

   printf("Hello again, world\n");
}
```

Issuing the following compiler command compiles, links, and creates the file main.out from the run-time-support library:

```
cl430 main.c --run_linker --heap_size=400 --library=rts430.lib --output_file=main.out
```

Executing main.out results in

```
Hello, world
```

being output to a file and

```
Hello again, world
```

being output to your host's stdout window.

### 7.2.2 Overview of Low-Level I/O Implementation

The low-level functions are comprised of seven basic I/O functions: open, read, write, close, lseek, rename, and unlink. These low-level routines provide the interface between the high-level functions and the device-level drivers that actually perform the I/O command on the specified device.

The low-level functions are designed to be appropriate for all I/O methods, even those which are not actually disk files. Abstractly, all I/O channels may be treated as files, although some operations (such as lseek) may not be appropriate. See Section 7.2.3 for more details.

The low-level functions are inspired by, but not identical to, the POSIX functions of the same names. The low-level functions are designed to be appropriate for all I/O methods, even those which are not actually disk files. Abstractly, all I/O channels may be treated as files, although some operations (such as lseek) may not be appropriate. See the device-driver section for more details.

The low-level functions operate on file descriptors. A file descriptor is an integer returned by open, representing an opened file. Multiple file descriptors may be associated with a file; each has its own independent file position indicator.

## open — *Open File for I/O*

**Syntax**

#include <file.h>

**int open (const char *** *path* **, unsigned** *flags* **, int** *file_descriptor* **);**

**Description**

The open function opens the file specified by *path* and prepares it for I/O.

- The *path* is the filename of the file to be opened, including an optional directory path and an optional device specifier (see Section 7.2.5).
- The *flags* are attributes that specify how the file is manipulated. The flags are specified using the following symbols:

```
O_RDONLY   (0x0000)   /* open for reading */
O_WRONLY   (0x0001)   /* open for writing */
O_RDWR     (0x0002)   /* open for read & write */
O_APPEND   (0x0008)   /* append on each write */
O_CREAT    (0x0200)   /* open with file create */
O_TRUNC    (0x0400)   /* open with truncation */
O_BINARY   (0x8000)   /* open in binary mode */
```

  Low-level I/O routines allow or disallow some operations depending on the flags used when the file was opened. Some flags may not be meaningful for some devices, depending on how the device implements files.
- The *file_descriptor* is assigned by open to an opened file.

  The next available file descriptor is assigned to each new file opened.

**Return Value**

The function returns one of the following values:

| | |
|---|---|
| non-negative file descriptor | if successful |
| -1 | on failure |

## close — *Close File for I/O*

**Syntax**

#include <file.h>

**int close (int** *file_descriptor* **);**

**Description**

The close function closes the file associated with *file_descriptor*.

The *file_descriptor* is the number assigned by open to an opened file.

**Return Value**

The return value is one of the following:

| | |
|---|---|
| 0 | if successful |
| -1 | on failure |

## read — *Read Characters from a File*

**Syntax**

#include <file.h>

**int read (int** *file_descriptor* **, char *** *buffer* **, unsigned** *count* **);**

**Description**

The read function reads *count* characters into the *buffer* from the file associated with *file_descriptor*.

- The *file_descriptor* is the number assigned by open to an opened file.
- The *buffer* is where the read characters are placed.
- The *count* is the number of characters to read from the file.

**Return Value**

The function returns one of the following values:

| | |
|---|---|
| 0 | if EOF was encountered before any characters were read |
| # | number of characters read (may be less than *count*) |
| -1 | on failure |

## write — *Write Characters to a File*

**Syntax**

#include <file.h>

**int write (int** *file_descriptor* **, const char *** *buffer* **, unsigned** *count* **);**

**Description**

The write function writes the number of characters specified by *count* from the *buffer* to the file associated with *file_descriptor*.

- The *file_descriptor* is the number assigned by open to an opened file.
- The *buffer* is where the characters to be written are located.
- The *count* is the number of characters to write to the file.

**Return Value**

The function returns one of the following values:

| | |
|---|---|
| # | number of characters written if successful (may be less than *count*) |
| -1 | on failure |

## lseek *Set File Position Indicator*

**Syntax for C**

#include <file.h>

**off_t lseek (int** *file_descriptor* **, off_t** *offset* **, int** *origin* **);**

**Description**

The lseek function sets the file position indicator for the given file to a location relative to the specified origin. The file position indicator measures the position in characters from the beginning of the file.

- The *file_descriptor* is the number assigned by open to an opened file.
- The *offset* indicates the relative offset from the *origin* in characters.
- The *origin* is used to indicate which of the base locations the *offset* is measured from. The *origin* must be one of the following macros:

  **SEEK_SET** (0x0000) Beginning of file

  **SEEK_CUR** (0x0001) Current value of the file position indicator

  **SEEK_END** (0x0002) End of file

**Return Value**

The return value is one of the following:

| | |
|---|---|
| # | new value of the file position indicator if successful |
| (off_t)-1 | on failure |

## unlink *Delete File*

**Syntax**

#include <file.h>

**int unlink (const char *** *path* **);**

**Description**

The unlink function deletes the file specified by *path*. Depending on the device, a deleted file may still remain until all file descriptors which have been opened for that file have been closed. See Section 7.2.3.

The *path* is the filename of the file, including path information and optional device prefix. (See Section 7.2.5.)

**Return Value**

The function returns one of the following values:

| | |
|---|---|
| 0 | if successful |
| -1 | on failure |

**rename** ***Rename File***

**Syntax for C**

#include {<stdio.h> | <file.h>}

**int rename (const char *** *old_name* **, const char *** *new_name* **);**

**Syntax for C++**

#include {<cstdio> | <file.h>}

**int std::rename (const char *** *old_name* **, const char *** *new_name* **);**

**Description**

The rename function changes the name of a file.

- The *old_name* is the current name of the file.
- The *new_name* is the new name for the file.

**NOTE:** The optional device specified in the new name must match the device of the old name. If they do not match, a file copy would be required to perform the rename, and rename is not capable of this action.

**Return Value**

The function returns one of the following values:

| | |
|---|---|
| 0 | if successful |
| -1 | on failure |

**NOTE:** Although rename is a low-level function, it is defined by the C standard and can be used by portable applications.

### *7.2.3 Device-Driver Level I/O Functions*

At the next level are the device-level drivers. They map directly to the low-level I/O functions. The default device driver is the HOST device driver, which uses the debugger to perform file operations. The HOST device driver is automatically used for the default C streams stdin, stdout, and stderr.

The HOST device driver shares a special protocol with the debugger running on a host system so that the host can perform the C I/O requested by the program. Instructions for C I/O operations that the program wants to perform are encoded in a special buffer named _CIOBUF_ in the .cio section. The debugger halts the program at a special breakpoint (C$$IO$$), reads and decodes the target memory, and performs the requested operation. The result is encoded into _CIOBUF_, the program is resumed, and the target decodes the result.

The HOST device is implemented with seven functions, HOSTopen, HOSTclose, HOSTread, HOSTwrite, HOSTlseek, HOSTunlink, and HOSTrename, which perform the encoding. Each function is called from the low-level I/O function with a similar name.

A device driver is composed of seven required functions. Not all function need to be meaningful for all devices, but all seven must be defined. Here we show the names of all seven functions as starting with DEV, but you may chose any name except for HOST.

## DEV_open

***Open File for I/O***

**Syntax**

**int DEV_open (const char *** *path* **, unsigned** *flags* **, int** *llv_fd* **);**

**Description**

This function finds a file matching *path* and opens it for I/O as requested by *flags*.

- The *path* is the filename of the file to be opened. If the name of a file passed to open has a device prefix, the device prefix will be stripped by open, so DEV_open will not see it. (See Section 7.2.5 for details on the device prefix.)
- The *flags* are attributes that specify how the file is manipulated. The flags are specified using the following symbols:

```
O_RDONLY   (0x0000)   /* open for reading */
O_WRONLY   (0x0001)   /* open for writing */
O_RDWR     (0x0002)   /* open for read & write */
O_APPEND   (0x0008)   /* append on each write */
O_CREAT    (0x0200)   /* open with file create */
O_TRUNC    (0x0400)   /* open with truncation */
O_BINARY   (0x8000)   /* open in binary mode */
```

  See POSIX for further explanation of the flags.
- The *llv_fd* is treated as a suggested low-level file descriptor. This is a historical artifact; newly-defined device drivers should ignore this argument. This differs from the low-level I/O open function.

This function must arrange for information to be saved for each file descriptor, typically including a file position indicator and any significant flags. For the HOST version, all the bookkeeping is handled by the debugger running on the host machine. If the device uses an internal buffer, the buffer can be created when a file is opened, or the buffer can be created during a read or write.

**Return Value**

This function must return -1 to indicate an error if for some reason the file could not be opened; such as the file does not exist, could not be created, or there are too many files open. The value of errno may optionally be set to indicate the exact error (the HOST device does not set errno). Some devices might have special failure conditions; for instance, if a device is read-only, a file cannot be opened O_WRONLY.

On success, this function must return a non-negative file descriptor unique among all open files handled by the specific device. It need not be unique across devices. Only the low-level I/O functions will see this device file descriptor; the low-level function open will assign its own unique file descriptor.

## DEV_close — *Close File for I/O*

**Syntax**

**int DEV_close (int** *dev_fd* **);**

**Description**

This function closes a valid open file descriptor.

On some devices, DEV_close may need to be responsible for checking if this is the last file descriptor pointing to a file that was unlinked. If so, it is responsible for ensuring that the file is actually removed from the device and the resources reclaimed, if appropriate.

**Return Value**

This function should return -1 to indicate an error if the file descriptor is invalid in some way, such as being out of range or already closed, but this is not required. The user should not call close() with an invalid file descriptor.

## DEV_read — *Read Characters from a File*

**Syntax**

**int DEV_read (int** *dev_fd* **, char *** *bu* **, unsigned** *count* **);**

**Description**

The read function reads *count* bytes from the input file associated with *dev_fd*.

- The *dev_fd* is the number assigned by open to an opened file.
- The *buf* is where the read characters are placed.
- The *count* is the number of characters to read from the file.

**Return Value**

This function must return -1 to indicate an error if for some reason no bytes could be read from the file. This could be because of an attempt to read from a O_WRONLY file, or for device-specific reasons.

If count is 0, no bytes are read and this function returns 0.

This function returns the number of bytes read, from 0 to count. 0 indicates that EOF was reached before any bytes were read. It is not an error to read less than count bytes; this is common if the are not enough bytes left in the file or the request was larger than an internal device buffer size.

## DEV_write — *Write Characters to a File*

**Syntax**

**int DEV_write (int** *dev_fd* **, const char *** *buf* **, unsigned** *count* **);**

**Description**

This function writes *count* bytes to the output file.

- The *dev_fd* is the number assigned by open to an opened file.
- The *buffer* is where the write characters are placed.
- The *count* is the number of characters to write to the file.

**Return Value**

This function must return -1 to indicate an error if for some reason no bytes could be written to the file. This could be because of an attempt to read from a O_RDONLY file, or for device-specific reasons.

## DEV_lseek *Set File Position Indicator*

**Syntax**

**off_t lseek (int** *dev_fd* **, off_t** *offset* **, int** *origin* **);**

**Description**

This function sets the file's position indicator for this file descriptor as lseek.

If lseek is supported, it should not allow a seek to before the beginning of the file, but it should support seeking past the end of the file. Such seeks do not change the size of the file, but if it is followed by a write, the file size will increase.

**Return Value**

If successful, this function returns the new value of the file position indicator.

This function must return -1 to indicate an error if for some reason no bytes could be written to the file. For many devices, the lseek operation is nonsensical (e.g. a computer monitor).

## DEV_unlink *Delete File*

**Syntax**

**int DEV_unlink (const char *** *path* **);**

**Description**

Remove the association of the pathname with the file. This means that the file may no longer by opened using this name, but the file may not actually be immediately removed.

Depending on the device, the file may be immediately removed, but for a device which allows open file descriptors to point to unlinked files, the file will not actually be deleted until the last file descriptor is closed. See Section 7.2.3.

**Return Value**

This function must return -1 to indicate an error if for some reason the file could not be unlinked (delayed removal does not count as a failure to unlink.)

If successful, this function returns 0.

## DEV_rename *Rename File*

**Syntax**

**int DEV_rename (const char *** *old_name* **, const char *** *new_name* **);**

**Description**

This function changes the name associated with the file.

- The *old_name* is the current name of the file.
- The *new_name* is the new name for the file.

**Return Value**

This function must return -1 to indicate an error if for some reason the file could not be renamed, such as the file doesn't exist, or the new name already exists.

**NOTE:** It is inadvisable to allow renaming a file so that it is on a different device. In general this would require a whole file copy, which may be more expensive than you expect.

If successful, this function returns 0.

### 7.2.4 Adding a User-Defined Device Driver for C I/O

The function add_device allows you to add and use a device. When a device is registered with add_device, the high-level I/O routines can be used for I/O on that device.

You can use a different protocol to communicate with any desired device and install that protocol using add_device; however, the HOST functions should not be modified. The default streams stdin, stdout, and stderr can be remapped to a file on a user-defined device instead of HOST by using freopen(). Example (see email). If the default streams are reopened in this way, the buffering mode will change to _IOFBF (fully buffered). To restore the default buffering behavior, call setvbuf on each reopened file with the appropriate value (_IOLBF for stdin and stdout, _IONBF for stderr).

The default streams stdin, stdout, and stderr can be mapped to a file on a user-defined device instead of HOST by using freopen() as shown in Example 7-1. Each function must set up and maintain its own data structures as needed. Some function definitions perform no action and should just return.

***Example 7-1. Mapping Default Streams to Device***

```
#include <stdio.h>
#include <file.h>
#include "mydevice.h"

void main()
{
    add_device("mydevice", _MSA,
               MYDEVICE_open, MYDEVICE_close,
               MYDEVICE_read, MYDEVICE_write,
               MYDEVICE_lseek, MYDEVICE_unlink, MYDEVICE_rename);

    /*--------------------------------------------------------------------*/
    /* Re-open stderr as a MYDEVICE file                                  */
    /*--------------------------------------------------------------------*/
    if (!freopen("mydevice:stderrfile", "w", stderr))
    {
        puts("Failed to freopen stderr");
        exit(EXIT_FAILURE);
    }

    /*--------------------------------------------------------------------*/
    /* stderr should not be fully buffered; we want errors to be seen as  */
    /* soon as possible.  Normally stderr is line-buffered, but this example */
    /* doesn't buffer stderr at all.  This means that there will be one call */
    /* to write() for each character in the message.                      */
    /*--------------------------------------------------------------------*/
    if (setvbuf(stderr, NULL, _IONBF, 0))
    {
        puts("Failed to setvbuf stderr");
        exit(EXIT_FAILURE);
    }

    /*--------------------------------------------------------------------*/
    /* Try it out!                                                        */
    /*--------------------------------------------------------------------*/
    printf("This goes to stdout\n");
    fprintf(stderr, "This goes to stderr\n"); }
```

**NOTE: Use Unique Function Names**

The function names open, read, write, close, lseek, rename, and unlink are used by the low-level routines. Use other names for the device-level functions that you write.

Use the low-level function add_device() to add your device to the device_table. The device table is a statically defined array that supports *n* devices, where *n* is defined by the macro _NDEVICE found in stdio.h/cstdio.

The first entry in the device table is predefined to be the host device on which the debugger is running. The low-level routine add_device() finds the first empty position in the device table and initializes the device fields with the passed-in arguments. For a complete description, see the add_device function.

### 7.2.5 The device Prefix

A file can be opened to a user-defined device driver by using a device prefix in the pathname. The device prefix is the device name used in the call to add_device followed by a colon. For example:

```
FILE *fptr = fopen("mydevice:file1", "r");
int fd = open("mydevice:file2, O_RDONLY, 0);
```

If no device prefix is used, the HOST device will be used to open the file.

## add_device *Add Device to Device Table*

**Syntax for C**

#include <file.h>

**int add_device(char *** *name*,
**unsigned** *flags* **,**
**int (*** *dopen* **)(const char *** *path*, **unsigned** *flags*, **int llv_fd),**
**int (*** *dclose* **)( int dev_fd),**
**int (*** *dread* **)(int** *dev_fd*, **char *** *buf*, **unsigned** *count*),
**int (*** *dwrite* **)(int** *dev_fd*, **const char *** *buf*, **unsigned** *count*),
**off_t (*** *dlseek* **)(int dev_fd, off_t** *ioffset*, **int** *origin*),
**int (*** *dunlink* **)(const char *** *path*),
**int (*** *drename* **)(const char *** *old_name*, **const char *** *new_name*));

**Defined in**

lowlev.c in rtssrc.zip

**Description**

The add_device function adds a device record to the device table allowing that device to be used for I/O from C. The first entry in the device table is predefined to be the HOST device on which the debugger is running. The function add_device() finds the first empty position in the device table and initializes the fields of the structure that represent a device.

To open a stream on a newly added device use fopen( ) with a string of the format *devicename* **:** *filename* as the first argument.

- The *name* is a character string denoting the device name. The name is limited to 8 characters.
- The *flags* are device characteristics. The flags are as follows:

  _**SSA** Denotes that the device supports only one open stream at a time

  _**MSA** Denotes that the device supports multiple open streams

  More flags can be added by defining them in file.h.
- The *dopen*, *dclose*, *dread*, *dwrite*, *dlseek*, *dunlink*, and *drename* specifiers are function pointers to the functions in the device driver that are called by the low-level functions to perform I/O on the specified device. You must declare these functions with the interface specified in Section 7.2.2. The device driver for the HOST that the MSP430 debugger is run on are included in the C I/O library.

**Return Value**

The function returns one of the following values:

| | |
|---|---|
| 0 | if successful |
| -1 | on failure |

**Example**

Example 7-2 does the following:

- Adds the device *mydevice* to the device table
- Opens a file named *test* on that device and associates it with the FILE pointer *fid*
- Writes the string *Hello, world* into the file
- Closes the file

Example 7-2 illustrates adding and using a device for C I/O:

***Example 7-2. Program for C I/O Device***

```
#include <file.h>
#include <stdio.h>
/*****************************************************************************/
/* Declarations of the user-defined device drivers                           */
/*****************************************************************************/
extern int   MYDEVICE_open(const char *path, unsigned flags, int fno);
extern int   MYDEVICE_close(int fno);
extern int   MYDEVICE_read(int fno, char *buffer, unsigned count);
extern int   MYDEVICE_write(int fno, const char *buffer, unsigned count);
extern off_t MYDEVICE_lseek(int fno, off_t offset, int origin);
extern int   MYDEVICE_unlink(const char *path);
extern int   MYDEVICE_rename(const char *old_name, char *new_name);
main()
{
   FILE *fid;
   add_device("mydevice", _MSA, MYDEVICE_open, MYDEVICE_close, MYDEVICE_read,
              MYDEVICE_write, MYDEVICE_lseek, MYDEVICE_unlink, MYDEVICE_rename);
   fid = fopen("mydevice:test","w");
   fprintf(fid,"Hello, world\n");

   fclose(fid);
}
```

## 7.3 Handling Reentrancy (_register_lock() and _register_unlock() Functions)

The C standard assumes only one thread of execution, with the only exception being extremely narrow support for signal handlers. The issue of reentrancy is avoided by not allowing you to do much of anything in a signal handler. However, BIOS applications have multiple threads which need to modify the same global program state, such as the CIO buffer, so reentrancy is a concern.

Part of the problem of reentrancy remains your responsibility, but the run-time-support environment does provide rudimentary support for multi-threaded reentrancy by providing support for critical sections. This implementation does not protect you from reentrancy issues such as calling run-time-support functions from inside interrupts; this remains your responsibility.

The run-time-support environment provides hooks to install critical section primitives. By default, a single-threaded model is assumed, and the critical section primitives are not employed. In a multi-threaded system such as BIOS, the kernel arranges to install semaphore lock primitive functions in these hooks, which are then called when the run-time-support enters code that needs to be protected by a critical section.

Throughout the run-time-support environment where a global state is accessed, and thus needs to be protected with a critical section, there are calls to the function _lock(). This calls the provided primitive, if installed, and acquires the semaphore before proceeding. Once the critical section is finished, _unlock() is called to release the semaphore.

Usually BIOS is responsible for creating and installing the primitives, so you do not need to take any action. However, this mechanism can be used in multi-threaded applications which do not use the BIOS LCK mechanism.

You should not define the functions _lock() and _unlock() functions directly; instead, the installation functions are called to instruct the run-time-support environment to use these new primitives:

```
void _register_lock  (void (  *lock)());

void _register_unlock(void (*unlock)());
```

The arguments to _register_lock() and _register_unlock() should be functions which take no arguments and return no values, and which implement some sort of global semaphore locking:

```
extern volatile sig_atomic_t *sema = SHARED_SEMAPHORE_LOCATION;
static int sema_depth = 0;
static void my_lock(void)
{
    while (ATOMIC_TEST_AND_SET(sema, MY_UNIQUE_ID) != MY_UNIQUE_ID);
    sema_depth++;
}
static void my_unlock(void)
{
    if (!--sema_depth) ATOMIC_CLEAR(sema);
}
```

The run-time-support nests calls to _lock(), so the primitives must keep track of the nesting level.

## 7.4 Library-Build Process

When using the C/C++ compiler, you can compile your code under a number of different configurations and options that are not necessarily compatible with one another. Because it would be cumbersome to include all possible combinations in individual run-time-support libraries, this package includes a basic run-time-support library, rts430.lib. Also included are library versions that support various MSP430 devices and versions that support C++ exception handling.

You can also build your own run-time-support libraries using the self-contained run-time-support build process, which is found in rtssrc.zip. This process is described in this chapter and the archiver described in the *MSP430 Assembly Language Tools User's Guide*.

### 7.4.1 Required Non-Texas Instruments Software

To use the self-contained run-time-support build process to rebuild a library with custom options, the following support items are required:

- Perl version 5.6 or later available as perl

  Perl is a high-level programming language designed for process, file, and text manipulation. It is:

  – Generally available from http://www.perl.org/get.htm
  – Available from ActiveState.com as ActivePerl for the PC
  – Available as part of the Cygwin package for the PC

  It must be installed and added to PATH so it is available at the command-line prompt as perl. To ensure perl is available, open a Command Prompt window and execute:

  ```
  perl -v
  ```

  No special or additional Perl modules are required beyond the standard perl module distribution.
- GNU-compatible command-line make tool, such as gmake

  More information is available from GNU at http://www.gnu.org/software/ make. This file requires a host C compiler to build. GNU make (gmake) is shipped as part of Code Composer Studio on Windows. GNU make is also included in some Unix support packages for Windows, such as the MKS Toolkit, Cygwin, and Interix. The GNU make used on Windows platforms should explicitly report This program built for Windows32 when the following is executed from the Command Prompt window:

  ```
  gmake -h
  ```

### 7.4.2 Using the Library-Build Process

Once the perl and gmake tools are available, unzip the rtssrc.zip into a new, empty directory. See the Makefile for additional information on how to customize a library build by modifying the LIBLIST and/or the OPT_XXX macros

Once the desired changes have been made, simply use the following syntax from the command-line while in the rtssrc.zip top level directory to rebuild the selected rtsname library.

**gmake** *rtsname*

To use custom options to rebuild a library, simply change the list of options for the appropriate base listed in Section 7.1.6 and then rebuild the library. See the tables in Section 2.3 for a summary of available generic and MSP430-specific options.

To build an library with a completely different set of options, define a new OPT_XXX base, choose the type of library per Section 7.1.6, and then rebuild the library. Not all library types are supported by all targets. You may need to make changes to targets_rts_cfg.pm to ensure the proper files are included in your custom library.

# C++ Name Demangler

The C++ compiler implements function overloading, operator overloading, and type-safe linking by encoding a function's signature in its link-level name. The process of encoding the signature into the linkname is often referred to as name mangling. When you inspect mangled names, such as in assembly files or linker output, it can be difficult to associate a mangled name with its corresponding name in the C++ source code. The C++ name demangler is a debugging aid that translates each mangled name it detects to its original name found in the C++ source code.

These topics tell you how to invoke and use the C++ name demangler. The C++ name demangler reads in input, looking for mangled names. All unmangled text is copied to output unaltered. All mangled names are demangled before being copied to output.

**Topic** **Page**

## 8.1 Invoking the C++ Name Demangler

The syntax for invoking the C++ name demangler is:

**dem430** [*options* ] [*filenames*]

| | |
|---|---|
| **dem430** | Command that invokes the C++ name demangler. |
| *options* | Options affect how the name demangler behaves. Options can appear anywhere on the command line. (Options are discussed in Section 8.2.) |
| *filenames* | Text input files, such as the assembly file output by the compiler, the assembler listing file, and the linker map file. If no filenames are specified on the command line, dem430 uses standard in. |

By default, the C++ name demangler outputs to standard out. You can use the -o file option if you want to output to a file.

## 8.2 C++ Name Demangler Options

The following options apply only to the C++ name demangler:

| | |
|---|---|
| **-h** | Prints a help screen that provides an online summary of the C++ name demangler options |
| **-o** *file* | Outputs to the given file rather than to standard out |
| **-u** | Specifies that external names do not have a C++ prefix |
| **-v** | Enables verbose mode (outputs a banner) |

## 8.3 Sample Usage of the C++ Name Demangler

The examples in this section illustrate the demangling process. Example 8-1 shows a sample C++ program. Example 8-2 shows the resulting assembly that is output by the compiler. In this example, the linknames of all the functions are mangled; that is, their signature information is encoded into their names.

***Example 8-1. C++ Code for calories_in_a_banana***

```
class banana {
public:
   int calories(void);
   banana();
   ~banana();
};

int calories_in_a_banana(void)
{
   banana x;
   return x.calories();
}
```

***Example 8-2. Resulting Assembly for calories_in_a_banana***

```
calories_in_a_banana__Fv:
;* ---------------------------------------------------------------------------*
        SUB.W     #4,SP
        MOV.W     SP,r12                ; |10|
        ADD.W     #2,r12                ; |10|
        CALL      #__ct__6bananaFv      ; |10|
                                        ; |10|
        MOV.W     SP,r12                ; |11|
        ADD.W     #2,r12                ; |11|
        CALL      #calories__6bananaFv  ; |11|
                                        ; |11|
        MOV.W     r12,0(SP)             ; |11|
        MOV.W     SP,r12                ; |11|
        ADD.W     #2,r12                ; |11|
        MOV.W     #2,r13                ; |11|
        CALL      #__dt__6bananaFv      ; |11|
                                        ; |11|
        MOV.W     0(SP),r12             ; |11|
        ADD.W     #4,SP
        RET
```

Executing the C++ name demangler demangles all names that it believes to be mangled. Enter:

```
dem430 calories_in_a_banana.asm
```

The result is shown in Example 8-3. The linknames in Example 8-2 ___ct__6bananaFv, _calories__6bananaFv, and ___dt__6bananaFv are demangled.

***Example 8-3. Result After Running the C++ Name Demangler***

```
calories_in_a_banana():
;* ---------------------------------------------------------------------------*
        SUB.W     #4,SP
        MOV.W     SP,r12                ; |10|
        ADD.W     #2,r12                ; |10|
        CALL      #banana::banana()      ; |10|
                                          ; |10|
        MOV.W     SP,r12                ; |11|
        ADD.W     #2,r12                ; |11|
        CALL      #banana::calories()  ; |11|
                                          ; |11|
        MOV.W     r12,0(SP)             ; |11|
        MOV.W     SP,r12                ; |11|
        ADD.W     #2,r12                ; |11|
        MOV.W     #2,r13                ; |11|
        CALL      #banana::~banana()      ; |11|
                                          ; |11|
        MOV.W     0(SP),r12             ; |11|
        ADD.W     #4,SP
        RET
```

# Glossary

**absolute lister—** A debugging tool that allows you to create assembler listings that contain absolute addresses.

**assignment statement—** A statement that initializes a variable with a value.

**autoinitialization—** The process of initializing global C variables (contained in the .cinit section) before program execution begins.

**autoinitialization at run time—** An autoinitialization method used by the linker when linking C code. The linker uses this method when you invoke it with the --rom_model link option. The linker loads the .cinit section of data tables into memory, and variables are initialized at run time.

**alias disambiguation—** A technique that determines when two pointer expressions cannot point to the same location, allowing the compiler to freely optimize such expressions.

**aliasing—** The ability for a single object to be accessed in more than one way, such as when two pointers point to a single object. It can disrupt optimization, because any indirect reference could refer to any other object.

**allocation—** A process in which the linker calculates the final memory addresses of output sections.

**ANSI—** American National Standards Institute; an organization that establishes standards voluntarily followed by industries.

**archive library—** A collection of individual files grouped into a single file by the archiver.

**archiver—** A software program that collects several individual files into a single file called an archive library. With the archiver, you can add, delete, extract, or replace members of the archive library.

**assembler—** A software program that creates a machine-language program from a source file that contains assembly language instructions, directives, and macro definitions. The assembler substitutes absolute operation codes for symbolic operation codes and absolute or relocatable addresses for symbolic addresses.

**assignment statement—** A statement that initializes a variable with a value.

**autoinitialization—** The process of initializing global C variables (contained in the .cinit section) before program execution begins.

**autoinitialization at run time—** An autoinitialization method used by the linker when linking C code. The linker uses this method when you invoke it with the --rom_model link option. The linker loads the .cinit section of data tables into memory, and variables are initialized at run time.

**big endian—** An addressing protocol in which bytes are numbered from left to right within a word. More significant bytes in a word have lower numbered addresses. Endian ordering is hardware-specific and is determined at reset. See also *little endian*

**BIS—** Bit instruction set.

**block—** A set of statements that are grouped together within braces and treated as an entity.

**.bss section—** One of the default object file sections. You use the assembler .bss directive to reserve a specified amount of space in the memory map that you can use later for storing data. The .bss section is uninitialized.

**byte—** Per ANSI/ISO C, the smallest addressable unit that can hold a character.

**C/C++ compiler—** A software program that translates C source statements into assembly language source statements.

**code generator—** A compiler tool that takes the file produced by the parser or the optimizer and produces an assembly language source file.

**COFF—** Common object file format; a system of object files configured according to a standard developed by AT&T. These files are relocatable in memory space.

**command file—** A file that contains options, filenames, directives, or commands for the linker or hex conversion utility.

**comment—** A source statement (or portion of a source statement) that documents or improves readability of a source file. Comments are not compiled, assembled, or linked; they have no effect on the object file.

**compiler program—** A utility that lets you compile, assemble, and optionally link in one step. The compiler runs one or more source modules through the compiler (including the parser, optimizer, and code generator), the assembler, and the linker.

**configured memory—** Memory that the linker has specified for allocation.

**constant—** A type whose value cannot change.

**cross-reference listing—** An output file created by the assembler that lists the symbols that were defined, what line they were defined on, which lines referenced them, and their final values.

**.data section—** One of the default object file sections. The .data section is an initialized section that contains initialized data. You can use the .data directive to assemble code into the .data section.

**direct call—** A function call where one function calls another using the function's name.

**directives—** Special-purpose commands that control the actions and functions of a software tool (as opposed to assembly language instructions, which control the actions of a device).

**disambiguation—** See *alias disambiguation*

**dynamic memory allocation—** A technique used by several functions (such as malloc, calloc, and realloc) to dynamically allocate memory for variables at run time. This is accomplished by defining a large memory pool (heap) and using the functions to allocate memory from the heap.

**ELF—** Executable and linking format; a system of object files configured according to the System V Application Binary Interface specification.

**emulator—** A hardware development system that duplicates the MSP430 operation.

**entry point—** A point in target memory where execution starts.

**environment variable—** A system symbol that you define and assign to a string. Environmental variables are often included in Windows batch files or UNIX shell scripts such as .cshrc or .profile.

**epilog—** The portion of code in a function that restores the stack and returns.

**executable module—** A linked object file that can be executed in a target system.

**expression—** A constant, a symbol, or a series of constants and symbols separated by arithmetic operators.

**external symbol—** A symbol that is used in the current program module but defined or declared in a different program module.

**file-level optimization—** A level of optimization where the compiler uses the information that it has about the entire file to optimize your code (as opposed to program-level optimization, where the compiler uses information that it has about the entire program to optimize your code).

**function inlining—** The process of inserting code for a function at the point of call. This saves the overhead of a function call and allows the optimizer to optimize the function in the context of the surrounding code.

**global symbol—** A symbol that is either defined in the current module and accessed in another, or accessed in the current module but defined in another.

**high-level language debugging—** The ability of a compiler to retain symbolic and high-level language information (such as type and function definitions) so that a debugging tool can use this information.

**indirect call—** A function call where one function calls another function by giving the address of the called function.

**initialization at load time—** An autoinitialization method used by the linker when linking C/C++ code. The linker uses this method when you invoke it with the --ram_model link option. This method initializes variables at load time instead of run time.

**initialized section—** A section from an object file that will be linked into an executable module.

**input section—** A section from an object file that will be linked into an executable module.

**integrated preprocessor—** A C/C++ preprocessor that is merged with the parser, allowing for faster compilation. Stand-alone preprocessing or preprocessed listing is also available.

**interlist feature—** A feature that inserts as comments your original C/C++ source statements into the assembly language output from the assembler. The C/C++ statements are inserted next to the equivalent assembly instructions.

**intrinsics—** Operators that are used like functions and produce assembly language code that would otherwise be inexpressible in C, or would take greater time and effort to code.

**ISO—** International Organization for Standardization; a worldwide federation of national standards bodies, which establishes international standards voluntarily followed by industries.

**K&R C—** Kernighan and Ritchie C, the de facto standard as defined in the first edition of *The C Programming Language* (K&R). Most K&R C programs written for earlier, non-ISO C compilers should correctly compile and run without modification.

**label—** A symbol that begins in column 1 of an assembler source statement and corresponds to the address of that statement. A label is the only assembler statement that can begin in column 1.

**linker—** A software program that combines object files to form an object module that can be allocated into system memory and executed by the device.

**listing file—** An output file, created by the assembler, that lists source statements, their line numbers, and their effects on the section program counter (SPC).

**little endian—** An addressing protocol in which bytes are numbered from right to left within a word. More significant bytes in a word have higher numbered addresses. Endian ordering is hardware-specific and is determined at reset. See also *big endian*

**loader—** A device that places an executable module into system memory.

**loop unrolling—** An optimization that expands small loops so that each iteration of the loop appears in your code. Although loop unrolling increases code size, it can improve the performance of your code.

**macro—** A user-defined routine that can be used as an instruction.

**macro call—** The process of invoking a macro.

**macro definition—** A block of source statements that define the name and the code that make up a macro.

**macro expansion—** The process of inserting source statements into your code in place of a macro call.

**map file—** An output file, created by the linker, that shows the memory configuration, section composition, section allocation, symbol definitions and the addresses at which the symbols were defined for your program.

**memory map—** A map of target system memory space that is partitioned into functional blocks.

**name mangling—** A compiler-specific feature that encodes a function name with information regarding the function's arguments return types.

**object file—** An assembled or linked file that contains machine-language object code.

**object library—** An archive library made up of individual object files.

**object module—** A linked, executable object file that can be downloaded and executed on a target system.

**operand—** An argument of an assembly language instruction, assembler directive, or macro directive that supplies information to the operation performed by the instruction or directive.

**optimizer—** A software tool that improves the execution speed and reduces the size of C programs.

**options—** Command-line parameters that allow you to request additional or specific functions when you invoke a software tool.

**output module—** A linked, executable object file that is downloaded and executed on a target system.

**output section—** A final, allocated section in a linked, executable module.

**parser—** A software tool that reads the source file, performs preprocessing functions, checks the syntax, and produces an intermediate file used as input for the optimizer or code generator.

**partitioning—** The process of assigning a data path to each instruction.

**pipelining—** A technique where a second instruction begins executing before the first instruction has been completed. You can have several instructions in the pipeline, each at a different processing stage.

**pop—** An operation that retrieves a data object from a stack.

**pragma—** A preprocessor directive that provides directions to the compiler about how to treat a particular statement.

**preprocessor—** A software tool that interprets macro definitions, expands macros, interprets header files, interprets conditional compilation, and acts upon preprocessor directives.

**program-level optimization—** An aggressive level of optimization where all of the source files are compiled into one intermediate file. Because the compiler can see the entire program, several optimizations are performed with program-level optimization that are rarely applied during file-level optimization.

**prolog—** The portion of code in a function that sets up the stack.

**push—** An operation that places a data object on a stack for temporary storage.

**quiet run—** An option that suppresses the normal banner and the progress information.

**raw data—** Executable code or initialized data in an output section.

**relocation—** A process in which the linker adjusts all the references to a symbol when the symbol's address changes.

**run-time environment—** The run time parameters in which your program must function. These parameters are defined by the memory and register conventions, stack organization, function call conventions, and system initialization.

**run-time-support functions—** Standard ISO functions that perform tasks that are not part of the C language (such as memory allocation, string conversion, and string searches).

**run-time-support library—** A library file, rts.src, that contains the source for the run time-support functions.

**section—** A relocatable block of code or data that ultimately will be contiguous with other sections in the memory map.

**sign extend—** A process that fills the unused MSBs of a value with the value's sign bit.

**simulator—** A software development system that simulates MSP430 operation.

**source file—** A file that contains C/C++ code or assembly language code that is compiled or assembled to form an object file.

**stand-alone preprocessor—** A software tool that expands macros, #include files, and conditional compilation as an independent program. It also performs integrated preprocessing, which includes parsing of instructions.

**static variable—** A variable whose scope is confined to a function or a program. The values of static variables are not discarded when the function or program is exited; their previous value is resumed when the function or program is reentered.

**storage class—** An entry in the symbol table that indicates how to access a symbol.

**string table—** A table that stores symbol names that are longer than eight characters (symbol names of eight characters or longer cannot be stored in the symbol table; instead they are stored in the string table). The name portion of the symbol's entry points to the location of the string in the string table.

**structure—** A collection of one or more variables grouped together under a single name.

**subsection—** A relocatable block of code or data that ultimately will occupy continuous space in the memory map. Subsections are smaller sections within larger sections. Subsections give you tighter control of the memory map.

**symbol—** A string of alphanumeric characters that represents an address or a value.

**symbolic debugging—** The ability of a software tool to retain symbolic information that can be used by a debugging tool such as a simulator or an emulator.

**target system—** The system on which the object code you have developed is executed.

**.text section—** One of the default object file sections. The .text section is initialized and contains executable code. You can use the .text directive to assemble code into the .text section.

**trigraph sequence—** A 3-character sequence that has a meaning (as defined by the ISO 646-1983 Invariant Code Set). These characters cannot be represented in the C character set and are expanded to one character. For example, the trigraph ??' is expanded to ^.

**unconfigured memory—** Memory that is not defined as part of the memory map and cannot be loaded with code or data.

**uninitialized section—** A object file section that reserves space in the memory map but that has no actual contents. These sections are built with the .bss and .usect directives.

**unsigned value—** A value that is treated as a nonnegative number, regardless of its actual sign.

**variable—** A symbol representing a quantity that can assume any of a set of values.

**veneer—** A sequence of instructions that serves as an alternate entry point into a routine if a state change is required.

**word—** A 16-bit addressable location in target memory

## IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

| **Products** | | **Applications** | |
|---|---|---|---|
| Amplifiers | amplifier.ti.com | Audio | www.ti.com/audio |
| Data Converters | dataconverter.ti.com | Automotive | www.ti.com/automotive |
| DLP® Products | www.dlp.com | Communications and Telecom | www.ti.com/communications |
| DSP | dsp.ti.com | Computers and Peripherals | www.ti.com/computers |
| Clocks and Timers | www.ti.com/clocks | Consumer Electronics | www.ti.com/consumer-apps |
| Interface | interface.ti.com | Energy | www.ti.com/energy |
| Logic | logic.ti.com | Industrial | www.ti.com/industrial |
| Power Mgmt | power.ti.com | Medical | www.ti.com/medical |
| Microcontrollers | microcontroller.ti.com | Security | www.ti.com/security |
| RFID | www.ti-rfid.com | Space, Avionics & Defense | www.ti.com/space-avionics-defense |
| RF/IF and ZigBee® Solutions | www.ti.com/lprf | Video and Imaging | www.ti.com/video |
| | | Wireless | www.ti.com/wireless-apps |

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265