ARP32 Compiler

Reference Guide

Literature Number: SPRUH24A
January 2012–Revised November 2012

Contents

List of Figures

List of Tables

Read This First

About This Manual

The *ARP32 Optimizing C/C++ Compiler User's Guide* explains how to use these compiler tools:

- Compiler
- Library build utility
- 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/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:

cl-arp32 [*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:

cl-arp32 --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

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

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

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

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

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), 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:

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

SPRUHC9— ARP32 CPU and Instruction Set Reference Guide. Describes the CPU architecture and instruction set of the ARP32 CPU.

SPRUH23 — ARP32 Assembly Language Tools Reference Guide. Describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the ARP32 devices.

Introduction to the Software Development Tools

The ARP32™ is supported by a set of software development tools, which includes an optimizing C/C++ compiler, an assembler, a linker, and assorted utilities.

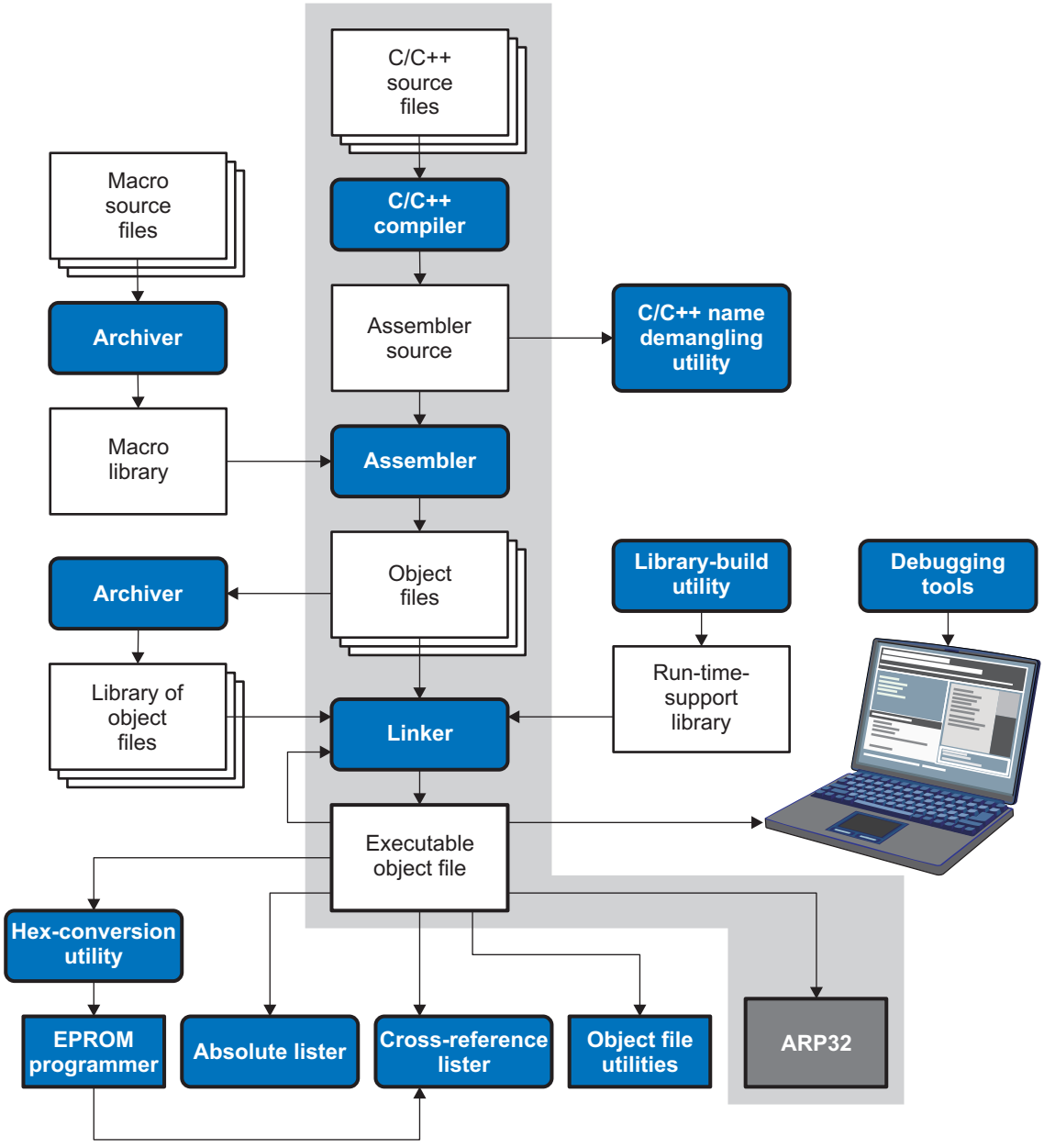
This chapter provides an overview of these tools and introduces the features of the optimizing C/C++ compiler. The assembler and linker are discussed in detail in the *ARP32 Assembly Language Tools User's Guide*.

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. ARP32 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 ARP32 assembly language source code. See Chapter 2.
- The **assembler** translates assembly language source files into machine language relocatable object files. The *ARP32 Assembly Language Tools User's Guide* explains how to use the assembler.
- The **linker** combines relocatable object files into a single absolute executable object file. As it creates the executable file, it performs relocation and resolves external references. The linker accepts relocatable object files and object libraries as input. See Chapter 4. The *ARP32 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 files. The *ARP32 Assembly Language Tools User's Guide* explains how to use the archiver.
- The **run-time-support libraries** contain the standard ISO C and C++ library functions, compiler-utility functions, floating-point arithmetic functions, and C I/O functions that are supported by the compiler. See Chapter 7.

  You can use the **library-build utility** to build your own customized run-time-support library. See Section 7.4. Source code for the standard run-time-support library functions for C and C++ are provided in the self-contained rtssrc.zip file.
- The **hex conversion utility** converts an object file into other object formats. You can download the converted file to an EPROM programmer. The *ARP32 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 *ARP32 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 *ARP32 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** decodes object files to show the assembly instructions that they represent. The *ARP32 Assembly Language Tools User's Guide* explains how to use the disassembler.
- The main product of this development process is an executable object file that can be executed in a **ARP32** device.

1.2 C/C++ Compiler Overview

The following subsections describe the key features of the compiler.

1.2.1 ANSI/ISO Standard

The C and C++ language features in the compiler are implemented in conformance with these ISO standards:

- **ISO-standard C**

  The C/C++ compiler conforms to the C Standard ISO/IEC 9889:1990. The ISO standard supercedes and is the same as the ANSI C standard. There is also a 1999 version of the ISO standard, but the TI compiler conforms to the 1990 standard, not the 1999 standard. The language is also described in the second edition of Kernighan and Ritchie's *The C Programming Language* (K&R).
- **ISO-standard C++**

  The C/C++ compiler conforms to the C++ Standard ISO/IEC 14882:1998. The language is also described in Ellis and Stroustrup's *The Annotated C++ Reference Manual* (ARM), but this is not the standard. 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 types of output files are created by the compiler:

- **ELF object files**

  Executable and linking format (ELF) enables supporting modern language features like early template instantiation and exporting inline functions.

1.2.3 Compiler Interface

These features enable interfacing with the compiler:

- **Compiler program**

  The compiler tools include a compiler program (cl-arp32) 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 are compiler utilities:

- **Library-build utility**

  The library-build utility 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 (dem-arp32) is a debugging aid that translates each mangled name it detects in compiler-generated assembly code, disassembly output, or compiler diagnostic messages 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 ELF files output by the compiler can be converted to EPROM programmer data files by using the hex conversion utility, as described in the *ARP32Assembly Language Tools User's Guide*.

Using the C/C++ Compiler

The compiler translates your source program into machine language object code that the ARP32 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 2.3.9 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.

Invoking the Linker

NOTE: 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 *ARP32 Assembly Language Tools User's Guide*.

2.2 Invoking the C/C++ Compiler

To invoke the compiler, enter:

cl-arp32 [*options*] [*filenames*] [**--run_linker** [*link_options*] *object files*]

cl-arp32	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 2-6 through Table 2-28.
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:

```
cl-arp32 symtab.c file.c seek.asm --run_linker --library=lnk.cmd
       --library=rtsarp32_v200.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 a help screen summary of the options, enter **cl-arp32** 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 ARP32_C_OPTION environment variable. For a detailed description of the environment variable, see Section 2.4.1.

Table 2-6 through Table 2-28 summarize all options (including link options). Use the references in the tables for more complete descriptions of the options.

Table 2-1. Processor Options

Option	Alias	Effect	Section
−−silicon_version={200|210}	-v	Selects processor version: 200: Base ARP32 or 210: EVE (ARP32 + VCOP)	Section 2.3.4

Table 2-2. 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 Section 3.2
--fp_mode={relaxed|strict}		Enables or disables relaxed floating-point mode	Section 2.3.3

(1) **Note:** Machine-specific options (see Table 2-11) can also affect optimization.

Table 2-3. Advanced Optimization Options[(1)]

Option	Alias	Effect	Section
--auto_inline=[*size*]	-oi	Sets automatic inlining size (--opt_level=3 only). If *size* is not specified, the default is 1.	Section 3.5
--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

(1) **Note:** Machine-specific options (see Table 2-11) can also affect optimization.

Table 2-3. Advanced Optimization Options(1) (continued)

Option	Alias	Effect	Section
--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
--optimizer_interlist	-os	Interlists optimizer comments with assembly statements	Section 3.6
--remove_hooks_when_inlining		Removes entry/exit hooks for auto-inlined functions	Section 2.13
--single_inline		Inlines functions that are only called once	--
--aliased_variables	-ma	Indicates that a specific aliasing technique is used	

Table 2-4. Debug Options

Option	Alias	Effect	Section
--symdebug:dwarf	-g	Enables symbolic debugging	Section 2.3.5
--symdebug:none		Disables all symbolic debugging	Section 2.3.5
--symdebug:skeletal		Enables minimal symbolic debugging that does not hinder optimizations (default behavior)	Section 2.3.5
--optimize_with_debug		Reenables optimizations disabled with --symdebug:dwarf	
--symdebug:dwarf_version=2|3		Specifies the DWARF format version	Section 2.3.5

Table 2-5. 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 2.3.3

Table 2-6. 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 2.3.2
--run_linker	-z	Enables linking	Section 2.3.2
--skip_assembler	-n	Compiles or assembly optimizes only	Section 2.3.2

Table 2-7. Language Options

Option	Alias	Effect	Section
--cpp_default	-fg	Processes all source files with a C extension as C++ source files.	Section 2.3.7
--create_pch=*filename*		Creates a precompiled header file with the name specified	Section 2.5
--embedded_cpp	-pe	Enables embedded C++ mode	Section 5.12.3
--exceptions		Enables C++ exception handling	Section 5.6
--float_operations_allowed={none|all|32|64}		Restricts the types of floating point operations allowed.	Section 2.3.3

Table 2-7. Language Options (continued)

Option	Alias	Effect	Section
--gcc		Enables support for GCC extensions	Section 5.13
--gen_acp_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 2.3.3
--kr_compatible	-pk	Allows K&R compatibility	Section 5.12.1
--multibyte_chars	-pc	Enables support for multibyte character sequences in comments, string literals and character constants.	--
--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.	--
--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
--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 5.12.2
--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 5.12.2
--use_pch=*filename*		Specifies the precompiled header file to use for this compilation	Section 2.5.2

Table 2-8. 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 2-9. Predefined Symbols Options

Option	Alias	Effect	Section
--define=*name*[=*def*]	-D	Predefines *name*	Section 2.3.2
--undefine=*name*	-U	Undefines *name*	Section 2.3.2

Table 2-10. 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	-pden	Displays a diagnostic's identifiers along with its text	Section 2.7.1
--emit_warnings_as_errors	-pdew	Treat warnings as errors	Section 2.7.1
--gen_aux_user_info		Generate user information file (.aux)	--
--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 2-11. Run-Time Model Options

Option	Alias	Effect	Section
--align_structs=*bytecount*		Aligns structures on *bytecount* boundary	Section 2.3.4
--fp_reassoc={on|off}		Enables or disables the reassociation of floating-point arithmetic	Section 2.3.4
--gen_func_subsections={on|off}		Puts each function in a separate subsection in the object file	Section 4.2.2
--mem_model:data={near|far|far_aggregates}		Determines data access model (when not specified, compiler defaults to --mem_model:data=far_aggregates)	Section 6.1.4
--plain_char={signed|unsigned}		Specifies how to treat plain chars, default is signed	Section 2.3.4
--sat_reassoc={on|off}		Enables or disables the reassociation of saturating arithmetic. Default is --sat_reassoc=off.	Section 2.3.3
--saveregs={nested_nmi|nested_std|nested_full}		Facilitates nested interrupts by generating the proper shadow register save and restore operations within an interrupt handler.	Section 2.3.4
--silicon_version={200|210}	-v	Selects processor version: Base ARP32 or EVE (ARP32 + VCOP)	Section 2.3.4
--small_enum		Uses the smallest possible size for the enumeration type	Section 2.3.4
--static_template_instantiation		Instantiates all template entities as needed	Section 2.3.4
--use_dead_funcs_list [=*fname*]		Places each function listed in the file in a separate section	Section 2.3.4

Table 2-12. Entry/Exit Hook Options

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

Table 2-13. Library Function Assumptions Options

Option	Alias	Effect	Section
--printf_support={nofloat|full|minimal}		Enables support for smaller, limited versions of the printf and sprintf run-time-support functions.	Section 2.3.3
--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

Table 2-14. Assembler Options

Option	Alias	Effect	Section
--keep_asm	-k	Keeps the assembly language (.asm) file	Section 2.3.11
--asm_listing	-al	Generates an assembly listing file	Section 2.3.11
--c_src_interlist	-ss	Interlists C source and assembly statements	Section 2.12 Section 3.6
--src_interlist	-s	Interlists optimizer comments (if available) and assembly source statements; otherwise interlists C and assembly source statements	Section 2.3.2
--absolute_listing	-aa	Enables absolute listing	Section 2.3.11
--asm_define=*name*[=*def*]	-ad	Sets the *name* symbol	Section 2.3.11
--asm_dependency	-apd	Performs preprocessing; lists only assembly dependencies	Section 2.3.11
--asm_includes	-api	Performs preprocessing; lists only included #include files	Section 2.3.11
--asm_undefine=*name*	-au	Undefines the predefined constant *name*	Section 2.3.11
--copy_file=*filename*	-ahc	Copies the specified file for the assembly module	Section 2.3.11
--cross_reference	-ax	Generates the cross-reference file	Section 2.3.11

Table 2-14. Assembler Options (continued)

Option	Alias	Effect	Section
--include_file=*filename*	-ahi	Includes the specified file for the assembly module	Section 2.3.11
--no_const_clink		Stops generation of .clink directives for const global arrays.	Section 2.3.3
--output_all_syms	-as	Puts labels in the symbol table	Section 2.3.11
--syms_ignore_case	-ac	Makes case insignificant in assembly source files	Section 2.3.11

Table 2-15. 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 2.3.7
--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 2.3.7
--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 2.3.7
--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 2.3.7

Table 2-16. Directory Specifier Options

Option	Alias	Effect	Section
--abs_directory=*directory*	-fb	Specifies an absolute listing file directory. By default, the compiler uses the .obj directory.	Section 2.3.10
--asm_directory=*directory*	-fs	Specifies an assembly file directory. By default, the compiler uses the current directory.	Section 2.3.10
--list_directory=*directory*	-ff	Specifies an assembly listing file and cross-reference listing file directory By default, the compiler uses the .obj directory.	Section 2.3.10
--obj_directory=*directory*	-fr	Specifies an object file directory. By default, the compiler uses the current directory.	Section 2.3.10
--output_file=*filename*	-fe	Specifies a compilation output file name; can override --obj_directory.	Section 2.3.10
--pp_directory=*dir*		Specifies a preprocessor file directory. By default, the compiler uses the current directory.	Section 2.3.10
--temp_directory=*directory*	-ft	Specifies a temporary file directory. By default, the compiler uses the current directory.	Section 2.3.10

Table 2-17. Default File Extensions Options

Option	Alias	Effect	Section
--asm_extension=[.]*extension*	-ea	Sets a default extension for assembly source files	Section 2.3.9
--c_extension=[.]*extension*	-ec	Sets a default extension for C source files	Section 2.3.9
--cpp_extension=[.]*extension*	-ep	Sets a default extension for C++ source files	Section 2.3.9
--listing_extension=[.]*extension*	-es	Sets a default extension for listing files	Section 2.3.9
--obj_extension=[.]*extension*	-eo	Sets a default extension for object files	Section 2.3.9

Table 2-18. 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 2.3.2

Table 2-19. MISRA-C:2004 Options

Option	Alias	Effect	Section
--check_misra[={all|required|advisory|none|*rulespec*}]		Enables checking of the specified MISRA-C:2004 rules. Default is all.	Section 2.3.3
--misra_advisory={error|warning|remark|suppress}		Sets the diagnostic severity for advisory MISRA-C:2004 rules	Section 2.3.3
--misra_required={error|warning|remark|suppress}		Sets the diagnostic severity for required MISRA-C:2004 rules	Section 2.3.3

2.3.1 Linker Options

The following tables list the linker options. See the *ARP32 Assembly Language Tools User's Guide* for details on these options.

Table 2-20. Linker Basic Options

Option	Alias	Description
--output_file=*file*	-o	Names the executable output file. 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*
--stack_size=*size*	[-]-stack	Sets C system stack size to *size* bytes and defines a global symbol that specifies the stack size. Default = 2K bytes
--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 = 2K bytes

Table 2-21. File Search Path Options

Option	Alias	Description
--library=*file*	-l	Names an archive library or link command *file* as linker input
--search_path=*pathname*	`-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.
--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
--disable_auto_rts		Disables the automatic selection of a run-time-support library

Table 2-22. Command File Preprocessing Options

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

Table 2-23. Diagnostic Options

Option	Alias	Description
--diag_error=*num*		Categorizes the diagnostic identified by *num* as an error
--diag_remark=*num*		Categorizes the diagnostic identified by *num* as a remark
--diag_suppress=*num*		Suppresses the diagnostic identified by *num*
--diag_warning=*num*		Categorizes the diagnostic identified by *num* as a warning
--display_error_number		Displays a diagnostic's identifiers along with its text
--emit_warnings_as_errors	-pdew	Treat warnings as errors
--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=*count*		Sets the error limit to *count*. 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 2-24. Linker Output Options

Option	Alias	Description
--absolute_exe	-a	Produces an absolute, executable object file. 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=*attribute*		Controls the information that appears in the map file.
--relocatable	-r	Produces a nonexecutable, relocatable output object file
--rom		Creates a ROM object
--run_abs	-abs	Produces an absolute listing file
--xml_link_info=*file*		Generates a well-formed XML *file* containing detailed information about the result of a link

Table 2-25. Symbol Management Options

Option	Alias	Description
--entry_point=*symbol*	-e	Defines a global symbol that specifies the primary entry point for the executable object file
--globalize=*pattern*		Changes the symbol linkage to global for symbols that match *pattern*
--hide=*pattern*		Hides symbols that match the specified *pattern*
--localize=*pattern*		Make the symbols that match the specified *pattern* local
--make_global=*symbol*	-g	Makes *symbol* global (overrides -h)
--make_static	-h	Makes all global symbols static
--no_symtable	-s	Strips symbol table information and line number entries from the executable object file
--retain		Retains a list of sections that otherwise would be discarded
--scan_libraries	-scanlibs	Scans all libraries for duplicate symbol definitions
--symbol_map=*refname*=*defname*		Specifies a symbol mapping; references to the *refname* symbol are replaced with references to the *defname* symbol
--undef_sym=*symbol*	-u	Adds *symbol* to the symbol table as an unresolved symbol
--unhide=*pattern*		Excludes symbols that match the specified *pattern* from being hidden

Table 2-26. Run-Time Environment Options

Option	Alias	Description
--arg_size=*size*	--args	Reserve *size* bytes for the argc/argv memory area
--fill_value=*value*	-f	Sets default fill value for holes within output sections
--ram_model	-cr	Initializes variables at load time
--rom_model	-c	Autoinitializes variables at run time
--trampolines[=off|of]		Generates far call trampolines. Default is on.

Table 2-27. Link-Time Optimization Options

Option	Description
--cinit_compression[=compression_kind]	Specifies the type of compression to apply to the c auto initialization data. Default is rle.
--compress_dwarf[=off|on]	Aggressively reduces the size of DWARF information from input object files. Default is on.
--copy_compression[=compression_kind]	Compresses data copied by linker copy tables. Default is rle.
--unused_section_elimination[=off|on]	Eliminates sections that are not needed in the executable module. Default is on.

Table 2-28. Miscellaneous Options

Option	Alias	Description
--linker_help	[-]-help	Displays information about syntax and available options
--minimize_trampolines		Places sections to minimize number of far trampolines required
--preferred_order=*function*		Prioritizes placement of functions
--strict_compatibility[=off|on]		Performs more conservative and rigorous compatibility checking of input object files. Default is on.
--trampoline_min_spacing		When trampoline reservations are spaced more closely than the specified limit, tries to make them adjacent
--zero_init=[off|on]		Controls preinitialization of uninitialized variables. Default is on.

2.3.2 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 3.6. 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:

--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 ARP32_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.

--gen_aux_user_info Generates a user information file and appends the .aux extension.

--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.

2.3.3 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.
--float_operations_allowed={none|all|32|64}	Restricts the type of floating point operations allowed in the application. The default is all. If set to none, 32, or 64, the application is checked for operations that will be performed at runtime. For example, declared variables that are not used will not cause a diagnostic. The checks are performed after relaxed mode optimizations have been performed, so illegal operations that are completely removed result in no diagnostics.
--fp_mode={relaxed|strict}	The default floating-point mode is strict. Relaxed floating-point mode causes double-precision floating-point computations and storage to be converted to single-precision floating-point or integers where possible. This behavior does not conform with ISO; but it results in faster code, with some loss in accuracy. The following specific changes occur in relaxed mode: • If the result of a double-precision floating-point expression is assigned to a single-precision floating-point or an integer or immediately used in a single-precision context, the computations in the expression are converted to single-precision computations. Double-precision constants in the expression are also converted to single-precision if they can be correctly represented as single-precision constants. • Calls to double-precision functions in math.h are converted to their single-precision counterparts if all arguments are single-precision and the result is used in a single-precision context. The math.h header file must be included for this optimization to work. • Division by a constant is converted to inverse multiplication. • Calls to sqrt, sqrtf, and sqrtl are converted directly to the VSQRT instruction. In this case errno will not be set for negative inputs.

In the following examples, iN=integer variable, fN=float variable, and 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.

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 --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:

- full: Supports all format specifiers. This is the default.
- nofloat: Excludes support for printing and scanning floating-point values. Supports all format specifiers except %f, %F, %g, %G, %e, and %E.
- minimal: Supports the printing and scanning of integer, char, or string values without width or precision flags. Specifically, only the %%, %d, %o, %c, %s, and %x format specifiers are supported

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.

2.3.4 Run-Time Model Options

These options are specific to the ARP32 toolset. See the referenced sections for more information. ARP32-specific assembler options are listed in Section 2.3.11.

--align_structs=*bytecount*	Forces alignment of structures to a minimum *bytecount*-byte boundary, where bytecount is a power of 2. To align all structs to a word boundary use --align_structs=4. All structs in the file will contain the bytecount minimum alignment, including nested structs. Only a minimum alignment is set, the data structures are not packed. Your program may break if one file is compiled with --align_structs and another is not, or is a different alignment is used. The offsets of a nested switch could be incorrect if different alignments are used.
--mem_model:data=*type*	Specifies data access model as *type* near, far, or far_aggregates. Default is far_aggregates. See Section 6.1.4.
--plain_char={signed|unsigned}	Specifies how to treat C/C++ plain char variables, default is signed.
--saveregs=*type*	Facilitates nested interrupts by generating the proper shadow register save and restore operations within an interrupt handler. The *type* parameter can be: • nested_nmi: Facilitates single-level nested interrupts by generating save and restore operations for non-maskable interrupt handlers. • nested_std: Facilitates multi-level nested interrupts by generating save and restore operations for non-maskable interrupt handlers and maskable interrupt handlers. • nested_full: Facilitates full multi-level nested interrupts by generating save and restore operations for non-maskable interrupt handlers and maskable interrupt handlers, including the stack pointer register and interrupt enable register.
--silicon_version=*num*	Selects the processor version. Each processor version is based on the standard ARP32 architecture. What differentiates the versions is the accelerator support that is added on. The *num* parameter can be: • 200: Generates code for Base ARP32. • 210: Generates code for EVE, which is ARP32 plus VCOP.
--small_enum	By default, the ARP32 compiler uses 32 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 32 bits when the --small_enum option is used. Similarly, enum a_short_enum {bottom = -32768, middle = 0, top = 32767} fits into two bytes instead of four. 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.
--static_template_instantiation	Instantiates all template entities in the current file as needed though the parser. These instantiations are also given internal (static) linkage.
--use_dead_funcs_list[=*fname*]	Places each function listed in the file in a separate section. The functions are placed in the *fname* section, if specified.

2.3.5 Symbolic Debugging Options

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

--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). For more information on the DWARF debug format, see *The DWARF Debugging Standard*.
--symdebug:dwarf_version={2|3}	Specifies the DWARF debugging format version (2 or 3) to be generated when --symdebug:dwarf or --symdebug:skeletal is specified. By default, the compiler generates DWARF version 3 debug information. For more information on TI extensions to the DWARF language, see *The Impact of DWARF on TI Object Files* (SPRAAB5).
--symdebug:none	Disables all symbolic debugging output. This option is not recommended; it prevents debugging and most performance analysis capabilities.
--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 for a list of deprecated symbolic debugging options.

2.3.6 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
.k	VCOP Kernel-C source file
.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 2.3.7. 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 2.3.10.

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:

```
cl-arp32 *.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.

2.3.7 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:

```
cl-arp32 --c_file=file.s --asm_file=assy
```

You cannot use the filename options with wildcard specifications.

2.3.8 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 2.3.9 for more information about filename extension conventions.

2.3.9 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:

```
cl-arp32 --asm_extension=.rrr --obj_extension=.o fit.rrr
```

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

```
cl-arp32 --asm_extension=rrr --obj_extension=o fit.rrr
```

2.3.10 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: `cl-arp32 --abs_directory=d:\abso_list`
--asm_directory=*directory*	Specifies a directory for assembly files. For example: `cl-arp32 --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: `cl-arp32 --list_directory=d:\listing`
--obj_directory=*directory*	Specifies a directory for object files. For example: `cl-arp32 --obj_directory=d:\object`
--output_file=*filename*	Specifies a compilation output file name; can override --obj_directory . For example: `cl-arp32 --output_file=transfer`
--pp_directory=*directory*	Specifies a preprocessor file directory for object files (default is .). For example: `cl-arp32 --pp_directory=d:\preproc`
--temp_directory=*directory*	Specifies a directory for temporary intermediate files. For example: `cl-arp32 --temp_directory=d:\temp`

2.3.11 Assembler Options

Following are assembler options that you can use with the compiler. For more information, see the *ARP32 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.

--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).

2.4 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 (ARP32_C_OPTION)

You might find it useful to set the compiler, assembler, and linker default options using the ARP32_C_OPTION environment variable. If you do this, the compiler uses the default options and/or input filenames that you name ARP32_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 ARP32_C_OPTION environment variable and processes it.

The table below shows how to set the ARP32_C_OPTION environment variable. Select the command for your operating system:

Operating System	Enter
UNIX (Bourne shell)	**ARP32_C_OPTION="** *option*$_1$ [*option*$_2$ . . .]**"; export ARP32_C_OPTION**
Windows	**set ARP32_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 ARP32_C_OPTION environment variable as follows:

```
set ARP32_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 ARP32_C_OPTION are passed to the linker. Thus, you can use the ARP32_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 ARP32_C_OPTION is set as shown above:

```
cl-arp32 *c                         ; compiles and links
cl-arp32 --compile_only *.c         ; only compiles
cl-arp32 *.c --run_linker lnk.cmd   ; compiles and links using a command file
cl-arp32 --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 the *Linker Description* chapter in the *ARP32 Assembly Language Tools User's Guide*.

2.4.2 Naming an Alternate Directory (ARP32_C_DIR)

The linker uses the ARP32_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)	**ARP32_C_DIR="** *pathname*$_1$ **;** *pathname*$_2$ **;..."; export ARP32_C_DIR**
Windows	**set ARP32_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 ARP32_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 ARP32_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 ARP32_C_DIR`
Windows	`set ARP32_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 2-29.

Table 2-29. Predefined ARP32 Macro Names

Macro Name	Description
_ _ARP32_ _	Always defined
_ _ARP32_V200_ _	Defined if base ARP32 is selected (the --silicon_version=200 or -v200 option is used); otherwise, it is undefined.
_ _DATE_ _ (1)	Expands to the compilation date in the form *mmm dd yyyy*
_ _FILE_ _ (1)	Expands to the current source filename
_ _LINE_ _ (1)	Expands to the current line number
_ _little_endian_ _	Always defined
_ _signed_chars_ _	Defined if char types are signed by default
_ _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.

(1) Specified by the ISO standard

Table 2-29. Predefined ARP32 Macro Names (continued)

Macro Name	Description
_ _TI_EABI_ _	Always defined
_ _TI_GNU_ATTRIBUTE_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.
_ _TI_STRICT_FP_MODE_ _	Defined to 1 if --fp_mode=strict is used (or implied); 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)
_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.

You can use the names listed in Table 2-29 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 ARP32_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 ARP32_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	`cl-arp32 --include_path=/tools/files source.c`
Windows	`cl-arp32 --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 ARP32_C_DIR environment variable.

For example, if you set up ARP32_C_DIR with the following command:

```
ARP32_C_DIR "/usr/include;/usr/ucb"; export ARP32_C_DIR
```

or invoke the compiler with the following command:

```
cl-arp32 --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.
--emit_warnings_as_errors	Treats all warnings as errors. This option cannot be used with the --no_warnings option. The --diag_remark option takes precedence over this option. This option takes precedence over the --diag_warning option.
--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 2-30.

Table 2-30. 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: L *line number filename key* 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: 1 = entry into an include file 2 = exit from an include file

The --gen_acp_raw option also includes diagnostic identifiers as defined in Table 2-31.

Table 2-31. 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 2-31 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 3.5.

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 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.

2.11.2 Automatic Inlining

When optimizing with the --opt_level=3 or --opt_level=2 option (aliased as -O3 or -O2), the compiler automatically inlines certain functions. For more information, see Section 3.5.

2.11.3 Inlining Restrictions

There are several restrictions on what functions can be inlined for both automatic inlining and definition-controlled inlining. Functions with local static variables or a variable number of arguments are not inlined, with the exception of functions declared as static inline. In functions declared as static inline, expansion occurs despite the presence of local static variables. In addition, a limit is placed on the depth of inlining for recursive or nonleaf functions. Furthermore, inlining should be used for small functions or functions that are called in a few places (though the compiler does not enforce this).

At a given call site, a function may be disqualified from inlining if it:

- Is not defined in the current compilation unit
- Never returns
- Is recursive
- Has a FUNC_CANNOT_INLINE pragma
- Has a variable length argument list
- Has a different number of arguments than the call site
- Has an argument whose type is incompatible with the corresponding call site argument
- Has a class, structe or union parameter
- Contains a volatile local variable or argument
- Is not declared inline and contains an asm() statement that is not a comment
- Is not declared inline and it is main()
- Is not declared inline and it is an interrupt function
- Is not declared inline and returns void but its return value is needed.
- Is not declared inline and will require too much stack space for local array or structure variables.

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:

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 3.6.

Example 2-1. An Interlisted Assembly Language File

```
main:
;------------------------------------------------------------------------
;   6 | printf("Hello, world\n");
;------------------------------------------------------------------------
        MVK       $C$SL1+0, R0
        SUB       0x4, SP
        MVKH      $C$SL1+0, R0
        STW       R0, *+SP(4)
        CALL      printf
        NOP       ; [DP_32_ALU]
        ; CALL OCCURS {printf}
;* ---------------------------------------------------------------------------*
        .align  4
        ADD       0x4, SP
        RET
;------------------------------------------------------------------------
;   8 | return 0;
;------------------------------------------------------------------------
        MVKS      0, R0
        ; RETURN OCCURS
```

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 Section 5.8.10 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, 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 optimization levels, such as --opt_level=2 and --opt_level=3, 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.

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 --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_info=0
Want to produce an optimization information file	--gen_opt_info=1
Want to produce a verbose optimization information file	--gen_opt_info=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.

Compiling Files With the --program_level_compile and --keep_asm Options

NOTE: 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.

3.4 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.

3.5 Automatic Inline Expansion (--auto_inline Option)

When optimizing with the --opt_level=3 option or --opt_level=2 option (aliased as -O3 and -O2, respectively), the compiler automatically inlines small functions. A command-line option, --auto_inline=*size*, specifies the size threshold for automatic inlining. This option controls only the inlining of functions that are not explicitly declared as inline.

When the --auto_inline option is not used, the compiler sets the size limit based on the optimization level and the optimization goal (performance versus code size). If the -auto_inline size parameter is set to 0, automatic inline expansion is disabled. If the --auto_inline size parameter is set to a non-zero integer, the compiler automatically inlines any function smaller than *size*. (This is a change from previous releases, which inlined functions for which the product of the function size and the number of calls to it was less than *size*. The new scheme is simpler, but will usually lead to more inlining for a given value of *size*.)

The compiler measures the size of a function in arbitrary units; however the optimizer information file (created with the --gen_opt_info=1 or --gen_opt_info=2 option) reports the size of each function in the same units that the --auto_inline option uses. When --auto_inline is used, the compiler does not attempt to prevent inlining that causes excessive growth in compile time or size; use with care.

When --auto_inline option is not used, the decision to inline a function at a particular call-site is based on an algorithm that attempts to optimize benefit and cost. The compiler inlines eligible functions at call-sites until a limit on size or compilation time is reached.

When deciding what to inline, the compiler collects all eligible call-sites in the module being compiled and sorts them by the estimated benefit over cost. Functions declared static inline are ordered first, then leaf functions, then all others eligible. Functions that are too big are not included.

Inlining behavior varies, depending on which compile-time options are specified:

- The code size limit is smaller when compiling for code size rather than performance. The --auto_inline option overrides this size limit.
- At --opt_level=3, the compiler auto-inlines aggressively if compiling for performance.
- At --opt_level=2, the compiler only automatically inlines small functions.

Some Functions Cannot Be Inlined

NOTE: For a call-site to be considered for inlining, it must be legal to inline the function and inlining must not be disabled in some way. See the inlining restrictions in Section 2.11.3.

Optimization Level 3 or 2 and Inlining

NOTE: In order to turn on automatic inlining, you must use the --opt_level=3 option or --opt_level=2 option. At --opt_level=2, only small functions are auto-inlined. If you desire the --opt_level=3 or 2 optimizations, but not automatic inlining, use --auto_inline=0 with the --opt_level=3 or 2 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.

3.6 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.

Impact on Performance and Code Size

NOTE: 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.

3.7 Debugging Optimized Code (--symdebug:dwarf, --symdebug:coff, and --opt_level Options)

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 is not recommended either, because this option 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 (aliased as -O) option in conjunction with the symbolic debugging option (--symdebug:dwarf). The symbolic debugging option generates directives that are used by the C/C++ source-level debugger, but it disables many compiler optimizations. When you use the --opt_level option (which invokes optimization) with the --symdebug:dwarf 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. However, if you use the --optimize_with_debug option, portions of the debugger's functionality will be unreliable.

Symbolic Debugging Options Affect Performance and Code Size

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

3.8 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.

3.9 What Kind of Optimization Is Being Performed?

The ARP32 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 3.9.1
Alias disambiguation	Section 3.9.1
Branch optimizations and control-flow simplification	Section 3.9.3
Data flow optimizations • Copy propagation • Common subexpression elimination • Redundant assignment elimination	Section 3.9.4
Expression simplification	Section 3.9.5
Inline expansion of functions	Section 3.9.6
Function Symbol Aliasing	Section 3.9.7
Induction variable optimizations and strength reduction	Section 3.9.8
Loop-invariant code motion	Section 3.9.9
Loop rotation	Section 3.9.10
Instruction scheduling	Section 3.9.11

ARP32-Specific Optimization	See
Tail merging	Section 3.9.12
Autoincrement addressing	Section 3.9.13
Epilog inlining	Section 3.9.14
Removing comparisons to zero	Section 3.9.15
Integer division with constant divisor	Section 3.9.16
Hardware Loop Acceleration (HLA)	Section 3.9.17

3.9.1 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 unroll or eliminate the loop. Strength reduction turns the array references into efficient pointer references with autoincrements.

3.9.2 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.

3.9.3 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.

3.9.4 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.

3.9.5 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.

3.9.6 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.

3.9.7 Function Symbol Aliasing

The compiler recognizes a function whose definition contains only a call to another function. If the two functions have the same signature (same return value and same number of parameters with the same type, in the same order), then the compiler can make the calling function an alias of the called function.

For example, consider the following:

```
int bbb(int arg1, char *arg2);

int aaa(int n, char *str)
{
   return bbb(n, str);
}
```

For this example, the compiler makes aaa an alias of bbb, so that at link time all calls to function aaa should be redirected to bbb. If the linker can successfully redirect all references to aaa, then the body of function aaa can be removed and the symbol aaa is defined at the same address as bbb.

3.9.8 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.

3.9.9 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.

3.9.10 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.

3.9.11 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 latencies. It can also be used to reduce code size.

3.9.12 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.

3.9.13 Autoincrement Addressing

For pointer expressions of the form *p++, the compiler uses efficient ARP32 autoincrement addressing modes. In many cases, where code steps through an array in a loop such as below, the loop optimizations convert the array references to indirect references through autoincremented register variable pointers.

```
for (I = 0; I <N; ++I) a(I)...
```

3.9.14 Epilog Inlining

If the epilog of a function is a single instruction, that instruction replaces all branches to the epilog. This increases execution speed by removing the branch.

3.9.15 Removing Comparisons to Zero

Because most of the 32-bit instructions and some of the 16-bit instructions can modify the status register when the result of their operation is 0, explicit comparisons to 0 may be unnecessary. The ARP32 C/C++ compiler removes comparisons to 0 if a previous instruction can be modified to set the status register appropriately.

3.9.16 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.

3.9.17 Hardware Loop Accelerator

The ARP32 architecture has hardware support for loop execution through the Hardware Loop Accelerator (HLA). This support is enabled whenever an advanced optimization level is given (-o2, -o3) during compilation. HLA results in a cycle reduction of critical inner loops; these cycles are normally spent in loop control operations such as loop index increment, decrement, compare, and branch operations. HLA converts these loops to a zero-overhead rewind to the top of the loop for up to two levels of nested loops. The structure and operation of HLA is comformable for easy mapping of all loop constructs available in C/C++.

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 *ARP32 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:

cl-arp32 --run_linker {**--rom_model | --ram_model**} *filenames*
[*options*] [**--output_file=** *name.out*] **--library=** *library* [*lnk.cmd*]

cl-arp32 --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 cl-arp32 --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 *ARP32 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 *ARP32 Assembly Language Tools User's Guide*.

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

```
cl-arp32 --run_linker --rom_model prog1 prog2 prog3 --output_file=prog.out
     --library=rtsarp32_v200.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:

cl-arp32 *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 object files prog1.c, prog2.c, and prog3.c, with an executable object file filename of prog.out with the command:

```
cl-arp32 prog1.c prog2.c prog3.c --run_linker --rom_model --output_file=prog.out --
library=rtsarp32_v200.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 ARP32_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 ARP32_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:
   ```
   cl-arp32 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:
   ```
   cl-arp32 --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:
   ```
   cl-arp32 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:
   ```
   cl-arp32 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:
   ```
   cl-arp32 --run_linker file1.obj file2.obj
   ```
   Alternatively, you can combine steps 3 and 4 into one step. For example:
   ```
   cl-arp32 file1.c file2.c --use_dead_funcs_list=feedback.txt --run_linker
   ```

NOTE: Dead Functions Feedback

The format of the feedback file generated with --gen_dead_funcs_list is tightly controlled. It must be generated by the linker in order to be processed correctly by the compiler. The format of this file may change over time, so the file contains a version format number to allow backward compatibility.

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 *ARP32 Assembly Language Tools User's Guide*

4.3.1 Including the Run-Time-Support Library

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. The following sections describe two methods for including the run-time-support library.

4.3.1.1 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 after any other libraries specified with the --library option on the command line. Alternatively, you can 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.

Example 4-1. Using the --issue_remarks Option

```
cl-arp32 --silicon_version=200 --issue_remarks main.c --run_linker --rom_model

<Linking>

remark: linking in "libc.a"

remark: linking in "rtsarp32_v200.lib" in place of "libc.a"
```

4.3.1.2 Manual Run-Time-Support Library Selection

You should use the --library linker option to specify which ARP32 run-time-support library to use. The --library option also tells the linker to look at the --search_path options and then the ARP32_C_DIR environment variable to find an archive path or object file. To use the --library linker option, type on the command line:

cl-arp32 --run_linker {**--rom_model** | **--ram_model**} *filenames* **--library=** *libraryname*

4.3.1.3 Library Order for Searching for Symbols

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.

4.3.2 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:

1. Set up status and configuration registers
2. Set up the stack and secondary system stack
3. Process the .cinit run-time initialization table to autoinitialize global variables (when using the --rom_model option)
4. Call all global object constructors (.init_array)
5. Call the function main
6. 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 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 the function main is called. Global destructors are invoked during the function exit, similar to functions registered through atexit.

4.3.4 Specifying the Type of Global Variable Initialization

The C/C++ compiler produces data tables for initializing global variables. Section 6.7.2.4 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 6.7.2.3).
- Global variables are initialized at *load time*. Use the --ram_model linker option (see Section 6.7.2.5).

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).

NOTE: Boot Loader

A loader is not included as part of the C/C++ compiler tools. You can use the ARP32 simulator with the source debugger as a loader.

4.3.5 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. See Section 6.1.1 for a complete description of how the compiler uses these sections.

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 string literals. String literals are placed in the .const:.string subsection to enable greater link-time placement control.
.data	Global and static non-const variables that are explicitly initialized.
.fardata	Far non-const global and static variables that are explicitly initialized.
.init_array	Table of constructors to be called at startup.
.rodata	Global and static variables that have near and const qualifiers.
.text	Executable code and constants.

Table 4-2. Uninitialized Sections Created by the Compiler

Name	Contents
.bss	Global and static variables
.far	Global and static variables declared far
.stack	Stack
.sysmem	Memory for malloc functions (heap)

The .data, .bss and .rodata sections are considered near sections. That is, they are reachable using the Global Data Page (GDP) register. The .far, .fardata and .const are far sections. That is, they are not reachable using the GDP register.

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.

The linker provides MEMORY and SECTIONS directives for allocating sections. For more information about allocating sections into memory, see the *ARP32 Assembly Language Tools User's Guide*.

4.3.6 A Sample Linker Command File

Example 4-2 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 0x6000 bytes
--heap_size	Tells the linker to set the heap size to 0x50000 bytes

To link the program, use the following syntax:

cl-arp32 --run_linker *object_file*(s) **--output_file** *outfile* **--map_file** *mapfile* **lnk.cmd**

Example 4-2. Linker Command File

```
--rom_model
--stack_size 0x6000
--heap_size  0x50000

/* SPECIFY THE SYSTEM MEMORY MAP */

MEMORY
{
        VECS      :   origin = 00000000h   length = 00000020h
        ISAINIT   :   origin = 00000020h   length = 000000e0h
        EXITSECT  :   origin = 00000100h   length = 00000200h
        STACKSECT :   origin = 00000300h   length = 00006000h
        ON_CHIPA  :   origin = 00006300h   length = 00220000h
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */

SECTIONS
{
    .vecs       >       VECS
    .isainit    >       ISAINIT

    .text       >       ON_CHIPA
    .stack      >       STACKSECT

    GROUP
    {
        .bss            /* This section ordering groups all GDP reachable */
        .data           /* sections within a single group.                */
        .rodata
    }>ON_CHIPA

    .cinit      >       ON_CHIPA
    .cio        >       ON_CHIPA
    .const      >       ON_CHIPA
    .switch     >       ON_CHIPA
    .sysmem     >       ON_CHIPA
    .far        >       ON_CHIPA
    .fardata    >       ON_CHIPA
}
```

Example 4-3. Linker Command File with Multiple Memory Pages

```
--rom_model
--stack_size 0x6000
--heap_size  0x50000

/* SPECIFY THE SYSTEM MEMORY MAP */

MEMORY
{
    PAGE 0:
        INTVECS   :   origin = 00000000h   length = 00000030h
        EXITSECT  :   origin = 00000100h   length = 00000100h
        RESET     :   origin = 00000200h   length = 000002c0h
        ON_CHIP   :   origin = 00000500h   length = 0003FAFFh

    PAGE 1:
        DATASECT  :   origin = 00000004h   length = 0001FFFFh
        STACKSECT :   origin = 00020000h   length = 0003FFFFh
        SL2       :   origin = 00400000h   length = 00040000h
}

/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */

SECTIONS
{
    .intvecs(NOINIT) >  INTVECS PAGE 0
    .inthandler      >  RESET PAGE 0

    .init_array >       DATASECT PAGE 1
    .text       >       ON_CHIP PAGE 0

    GROUP
    {
        .bss            /* This section ordering groups all GDP reachable */
        .data           /* sections within a single group.                */
        .rodata
    }>DATASECT PAGE 1

    .stack      >       STACKSECT PAGE 1
    .cinit      >       DATASECT PAGE 1
    .cio        >       ON_CHIP PAGE 0
    .const      >       DATASECT PAGE 1
    .switch     >       SL2 PAGE 1
    .sysmem     >       SL2 PAGE 1
    .far        >       SL2 PAGE 1
    .fardata    >       DATASECT PAGE 1
}
```

ARP32 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 ARP32 is defined by the ANSI/ISO/IEC 14882:1998 standard with certain exceptions.

Topic **Page**

5.1 Characteristics of ARP32 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 5.13). 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 ARP32 C++

The ARP32 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 compiler does not support embedded C++ run-time-support libraries.
- The library supports wide chars (wchar_t), in that template functions and classes that are defined for char are also available for wchar_t. 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.
- 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. See Section 5.8.1.
- RESET_MISRA pragma resets the specified MISRA-C:2004 rules to the state they were before any CHECK_MISRA pragmas were processed. See Section 5.8.11.

The syntax of the option and pragmas is:

--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 ARP32 compiler. Many of the range values are available as standard macros in the header file limits.h.

Table 5-1. ARP32 C/C++ Data Types

			Range	
Type	**Size**	**Representation**	**Minimum**	**Maximum**
signed char	8 bits	ASCII	-128	127
char, 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	32 bits	2s complement	-2 147 483 648	2 147 483 647
unsigned int	32 bits	Binary	0	4 294 967 295
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
long long, signed long long	64 bits	2s complement	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
unsigned long long	64 bits	Binary	0	18 446 744 073 709 551 615
float	32 bits	IEEE 32-bit	1.175 494e-38(1)	3.40 282 346e+38
double	64 bits	IEEE 64-bit	2.22 507 385e-308	1.79 769 313e+308
long double	64 bits	IEEE 64-bit	2.22 507 385e-308(1)	1.79 769 313e+308
pointers, references, pointer to data members	32 bits	Binary	0	0xFFFFFFFF

(1) Figures are minimum precision.

The type of the storage container for an enumerated type is the smallest integer type that contains all the enumerated values. The container types for enumerators are shown in Table 5-2.

Table 5-2. EABI Enumerator Types

Lower Bound Range	Upper Bound Range	Enumerator Type
0 to 255	0 to 255	unsigned char
-128 to 1	-128 to 127	signed char
0 to 65 535	256 to 65 535	unsigned short
-128 to 1	128 to 32 767	short, signed short
-32 768 to -129	-32 768 to 32 767	
0 to 4 294 967 295	2 147 483 648 to 4 294 967 295	unsigned int
-32 768 to -1	32 767 to 2 147 483 647	int, signed int
-2 147 483 648 to -32 769	-2 147 483 648 to 2 147 483 647	
0 to 2 147 483 647	65 536 to 2 147 483 647	

The compiler determines the type based on the range of the lowest and highest elements of the enumerator.

For example, the following code results in an enumerator type of int:

```
enum COLORS
{  green  = -200,
   blue   = 1,
   yellow = 2,
   red    = 60000  }
```

For example, the following code results in an enumerator type of short:

```
enum COLORS
{  green  = -200,
   blue   = 1,
   yellow = 2,
```

```
        red    = 3  }
```

5.5 Keywords

The ARP32 C/C++ compiler supports the standard const and volatile keywords. In addition, the C/C++ compiler extends the C/C++ language through the support of the cregister, 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 cregister Keyword

The compiler extends the C/C++ language by adding the cregister keyword to allow high level language access to control registers.

When you use the cregister keyword on an object, the compiler compares the name of the object to a list of standard control registers for ARP32 (see Table 5-3). If the name matches, the compiler generates the code to reference the control register. If the name does not match, the compiler issues an error.

Table 5-3. Valid Control Registers

Register	Description
ICR	Interrupt Clear Register
IER	Interrupt Enable Register
IFR	Interrupt Flag Register
IRP	Interrupt Return Pointer
ISR	Interrupt Set Pointer
LR	Link Register
NRP	NMI Return Pointer
TSCH	Time Stamp Counter High Register
TSCL	Time Stamp Counter Low Register

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

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

To use the control registers in Table 5-3, you must declare each register through this syntax:

extern cregister volatile unsigned int *register* **;**

Once you have declared the register, you can use the register name directly. See the *ARP32 CPU and Instruction Set Architecture Reference Guide* for detailed information on the control registers.

See Example 5-1 for an example that declares and uses control registers.

Example 5-1. Define and Use Control Registers

```
extern cregister volatile unsigned int IFR;
extern cregister volatile unsigned int ISR;
extern cregister volatile unsigned int ICR;
extern cregister volatile unsigned int IER;
extern cregister volatile unsigned int TSCH;
extern cregister volatile unsigned int TSCL;
extern cregister volatile unsigned int NRP;
main()
{
   printf("IFR = %x\n", IFR);
}
```

5.5.3 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. This keyword is an IRQ interrupt.

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.

5.5.4 The near and far Keywords

The ARP32 C/C++ compiler extends the C/C++ language with the near and far keywords to specify how global and static variables are accessed.

Syntactically, the near and far keywords are treated as storage class modifiers. They can appear before, after, or in between the storage class specifiers and types. With the exception of near and far, two storage class modifiers cannot be used together in a single declaration. The following examples are legal combinations of near and far with other storage class modifiers:

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

5.5.4.1 near and far Data Objects

Global and static data objects can be accessed in the following two ways:

near keyword The compiler assumes that the data item can be accessed relative to the data page pointer. For example:

```
LDW   *+GDP(_address), R0
```

far keyword The compiler cannot access the data item via the DP. This can be required if the total amount of program data is larger than the offset allowed (32K) from the DP. For example:

```
MVK   _address, R1
 MVKH  _address, R1
 LDW   *+R1(0), R0
```

Once a variable has been defined to be far, all external references to this variable in other C files or headers must also contain the far keyword. This is also true of the near keyword. However, you will get compiler or linker errors when the far keyword is not used everywhere. Not using the near keyword everywhere only leads to slower data access times.

If you use the DATA_SECTION pragma, the object is indicated as a far variable, and this cannot be overridden. If you reference this object in another file, then you need to use *extern far* when declaring this object in the other source file. This ensures access to the variable, since the variable might not be in the .bss section. For details, see Section 5.8.4.

NOTE: Defining Global Variables in Assembly Code

If you also define a global variable in assembly code with the .usect directive (where the variable is not assigned in the .bss section) or you allocate a variable into separate section using a #pragma DATA_SECTION directive; and you want to reference that variable in C code, you must declare the variable as extern far. This ensures the compiler does not try to generate an illegal access of the variable by way of the data page pointer.

When data objects do not have the near or far keyword specified, the compiler will use far accesses to aggregate data and near accesses to non-aggregate data. For more information on the data memory model and ways to control accesses to data, see Section 6.1.4.

5.5.5 The volatile Keyword

The compiler eliminates redundant memory accesses whenever possible, using data flow analysis to figure out when it is legal. However, some memory accesses may be special in some way that the compiler cannot see, and in such cases you must use the volatile keyword to prevent the compiler from optimizing away something important. The compiler does not optimize out any accesses to variables declared volatile. The number and order of accesses of a volatile variable are exactly as they appear in the C/C++ code, no more and no less.

There are different ways to understand how volatile works, but fundamentally it is a hint to the compiler that something it cannot understand is going on, and so the compiler should not try to be over-clever.

Any variable which might be modified by something external to the obvious control flow of the program (such as an interrupt service routine) must be declared volatile. This tells the compiler that an interrupt function might modify the value at any time, so the compiler should not perform optimizations which will change the number or order of accesses of that variable. This is the primary purpose of the volatile keyword. 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.

Volatile must also be used when accessing memory locations that represent memory-mapped peripheral devices. Such memory locations might change value in ways that the compiler cannot predict. These locations might change if accessed, or when some other memory location is accessed, or when some signal occurs.

Volatile must also be used for local variables in a function which calls setjmp, if the value of the local variables needs to remain valid if a longjmp occurs.

Example 5-2. Volatile for Local Variables With setjmp

```
#include <stdlib.h>
jmp_buf context;
void function()
{
    volatile int x = 3;
    switch(setjmp(context))
    {
        case 0: setup(); break;
        default:
        {
           printf("x == %d\n", x); /* We can only reach here if longjmp has occurred; because x's
                                      lifetime begins before the setjmp and lasts through the longjmp,
                                      the C standard requires x be declared "volatile" */
           break;
        }
    }
}
```

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.

Exception handling requires support in the run-time-support library, which come in exception-enabled and exception-disabled forms; you must link with the correct form. When using automatic library selection (the default), the linker automatically selects the correct library Section 4.3.1.1. If you select the library manually, you must use run-time-support libraries whose name contains _eh if you enable exceptions.

Using --exceptions causes the compiler to insert exception handling code. This code will increase the code size of the programand has a minimal execution time cost if exceptions are never thrown, but will slightly increase the data size for the exception-handling tables.

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.

ARP32 has two global registers, __SP and __GDP, which are registers treated in a special manner than other control registers.

For more information about register conventions, see Section 6.3.

5.8 Pragma Directives

Pragma directives tell the compiler how to treat a certain function, object, or section of code. The ARP32 C/C++ compiler supports the following pragmas:

- CHECK_MISRA (See Section 5.8.1)
- CODE_SECTION (See Section 5.8.2)
- DATA_ALIGN (See Section 5.8.3)
- DATA_SECTION (See Section 5.8.4)
- DIAG_SUPPRESS, DIAG_REMARK, DIAG_WARNING, DIAG_ERROR, and DIAG_DEFAULT (See Section 5.8.5)
- FUNC_EXT_CALLED (See Section 5.8.6)
- FUNCTION_OPTIONS (See Section 5.8.7)
- INTERRUPT (See Section 5.8.8)
- LOCATION (EABI only; see Section 5.8.9)
- NOINIT (EABI only; see)
- NO_HOOKS (See Section 5.8.10)
- PERSISTENT (EABI only; see)
- RESET_MISRA (See Section 5.8.11)
- RETAIN (See Section 5.8.12)
- SET_CODE_SECTION (See Section 5.8.13)
- SET_DATA_SECTION (See Section 5.8.13)
- WEAK (See Section 5.8.14)

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 (except CLINK and RETAIN), 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.

5.8.1 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 5.8.11.

5.8.2 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(fn, "my_sect")

int fn(int x)
{
    return x;
}
```

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

```
        .sect   "my_sect"
        .align  2
        .clink
        .global fn

;******************************************************************************
;* FUNCTION NAME: fn                                                          *
;*                                                                            *
;*   Regs Modified     : R0                                                   *
;*   Regs Used         : R0,R2                                                *
;*   Local Frame Size  : 0 Args + 0 Auto + 0 Save = 0 byte                    *
;******************************************************************************
fn:
;* ---------------------------------------------------------------------------*
        RET
        MV        R2, R0
        ; RETURN OCCURS
```

5.8.3 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* **);**

5.8.4 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 5-5 through Example 5-7 demonstrate the use of the DATA_SECTION pragma.

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

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

Example 5-6. Using the DATA_SECTION Pragma C++ Source File

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

Example 5-7. Using the DATA_SECTION Pragma Assembly Source File

```
          .global _bufferA
          .bss    _bufferA,512,4
          .global _bufferB
_bufferB: .usect  "my_sect",512,4
```

5.8.5 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.

5.8.6 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 .

5.8.7 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*" **);**

5.8.8 The INTERRUPT Pragma

The INTERRUPT pragma enables you to handle interrupts directly with C code. The pragma specifies that the function is an interrupt. The type of interrupt is specified by the pragma; the IRQ (interrupt request) interrupt type is assumed if none is given.

The syntax of the pragma in C is:

#pragma INTERRUPT (*func*[, *interrupt_type*] **);**

The syntax of the pragma in C++ is:

#pragma INTERRUPT [**(** *interrupt_type* **)**] **;**

In C, the argument *func* is the name of a function. In C++, the pragma applies to the next function declared. The optional argument *interrupt_type* specifies an interrupt type. The registers that are saved and the return sequence depend upon the interrupt type. If the interrupt type is omitted from the interrupt pragma, the interrupt type IRQ is assumed. These are the valid interrupt types:

Interrupt Type	Description
DABT	Data abort
FIQ	Fast interrupt request
IRQ	Interrupt request
PABT	Prefetch abort
RESET	System reset
SWI	Software interrupt
UDEF	Undefined instruction

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.

5.8.9 The LOCATION Pragma

The compiler supports the ability to specify the run-time address of a variable at the source level. This can be accomplished with the LOCATION pragma or attribute. Location support is only available in EABI.

The syntax of the pragma in C is:

```
#pragma LOCATION( x , address );
int x;
```

The syntax of the pragmas in C++ is:

```
#pragma LOCATION(address );
int x;
```

The syntax of the GCC attribute is:

```
int x __attribute__((location(address )));
```

5.8.10 The NO_HOOKS Pragma

The NO_HOOKS pragma prevents entry and exit hook calls from being generated for a function.

The syntax of the pragma in C is:

```
#pragma NO_HOOKS ( func );
```

The syntax of the pragma in C++ is:

```
#pragma NO_HOOKS;
```

See Section 2.13 for details on entry and exit hooks.

5.8.11 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 5.8.1) 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.

5.8.12 The RETAIN Pragma

The RETAIN pragma can be applied to a code or data symbol. It causes a .retain directive to be generated into the section that contains the definition of the symbol. The .retain directive indicates to the linker that the section is ineligible for removal during conditional linking. Therefore, regardless whether or not the section is referenced by another section in the application that is being compiled and linked, it will be included in the output file result of the link.

The syntax of the pragma in C/C++ is:

```
#pragma RETAIN ( symbol )
```

5.8.13 The SET_CODE_SECTION and SET_DATA_SECTION Pragmas

These pragmas can be used to set the section for all declarations below the pragma.

The syntax of the pragmas in C/C++ is:

```
#pragma SET_CODE_SECTION ("section name")
```

```
#pragma SET_DATA_SECTION ("section name")
```

In Example 5-8 x and y are put in the section mydata. To reset the current section to the default used by the compiler, a blank paramater should be passed to the pragma. An easy way to think of the pragma is that it is like applying the CODE_SECTION or DATA_SECTION pragma to all symbols below it.

Example 5-8. Setting Section With SET_DATA_SECTION Pragma

```
#pragma SET_DATA_SECTION("mydata")
int x;
int y;
#pragma SET_DATA_SECTION()
```

The pragmas apply to both declarations and definitions. If applied to a declaration and not the definition, the pragma that is active at the declaration is used to set the section for that symbol. Here is an example:

Example 5-9. Setting a Section With SET_CODE_SECTION Pragma

```
#pragma SET_CODE_SECTION("func1")
extern void func1();
#pragma SET_CODE_SECTION()
...
void func1() { ... }
```

In Example 5-9 func1 is placed in section func1. If conflicting sections are specified at the declaration and definition, a diagnostic is issued.

The current CODE_SECTION and DATA_SECTION pragmas and GCC attributes can be used to override the SET_CODE_SECTION and SET_DATA_SECTION pragmas. For example:

Example 5-10. Overriding SET_DATA_SECTION Setting

```
#pragma DATA_SECTION(x, "x_data")
#pragma SET_DATA_SECTION("mydata")
int x;
int y;
#pragma SET_DATA_SECTION()
```

In Example 5-10 x is placed in x_data and y is placed in mydata. No diagnostic is issued for this case.

The pragmas work for both C and C++. In C++, the pragmas are ignored for templates and for implictly created objects, such as implicit constructors and virtual function tables.

5.8.14 The WEAK Pragma

The WEAK pragma gives weak binding to a symbol.

The syntax of the pragma in C is:

#pragma WEAK (*symbol* **);**

The syntax of the pragma in C++ is:

#pragma WEAK;

The WEAK pragma makes *symbol* a weak reference if it is a reference, or a weak definition, if it is a definition. The symbol can be a data or function variable.

In effect, unresolved weak *references* do not cause linker errors and do not have any effect at run time. The following apply for weak references:

- Libraries are not searched to resolve weak references. It is not an error for a weak reference to remain unresolved.

- During linking, the value of an undefined weak reference is:
  - Zero if the relocation type is absolute
  - The address of the place if the relocation type is PC-relative
  - The address of the nominal base address if the relocation type is base-relative.

A weak *definition* does not change the rules by which object files are selected from libraries. However, if a link set contains both a weak definition and a non-weak definition, the non-weak definition is always used.

5.9 The _Pragma Operator

The ARP32 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.

5.10 EABI Application Binary Interface

An ABI should define how functions that are written separately, and compiled or assembled separately can work together. This involves standardizing the data type representation, register conventions, and function structure and calling conventions. It should define linkname generation from C symbol names. It should define the object module format and the debug format. It should document how the system is initialized. In the case of C++ it should define C++ name mangling and exception handling support.

The __TI_EABI_ASSEMBLER predefined symbol is always set to 1.

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.

If a C identifier would collide with an assembler keyword, the compiler will escape the identifier with double parallel bars, which instructs the assembler not to treat the identifier as a keyword. You are responsible for making sure that C identifiers do not collide with user-defined assembly code identifiers.

C++ functions have the same initial character. Additionally, the function name is mangled further. 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.

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:

```
_Z3fooi
```

The linkname of foo is _Z3fooi, 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.

The mangling algorithm follows that described in the Itanium C++ ABI (http://www.codesourcery.com/cxx-abi/abi.html).

`int foo(int i) { }` would be mangled "_Z3fooi"

EABI Mode C++ Demangling

NOTE: Please refer to http://www.arm.com/products/DevTools/ABI.html for details.

5.12 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.

5.12.1 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 */
```

5.12.2 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. The GCC language extensions described in Section 5.13 are available in relaxed ANSI/ISO mode.

5.12.3 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 ARP32 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.

The compiler does not support embedded C++ run-time-support libraries.

5.13 GNU Language Extensions

The GNU compiler collection (GCC) defines a number of language features not found in the ANSI/ISO C and C++ standards. The definition and examples of these extensions (for GCC version 3.4) can be found at the GNU web site, http://gcc.gnu.org/onlinedocs/gcc-3.4.6/gcc/C-Extensions.html.

Most of these extensions are also available for C++ source code.

5.13.1 Extensions

Most of the GCC language extensions are available in the TI compiler when compiling in relaxed ANSI mode (--relaxed_ansi) or if the --gcc option is used.

The extensions that the TI compiler supports are listed in Table 5-4, which is based on the list of extensions found at the GNU web site. The shaded rows describe extensions that are not supported.

Table 5-4. 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	Pointers to labels and computed gotos
Nested functions	As in Algol and Pascal, lexical scoping of functions
Constructing calls	Dispatching a call to another function
Naming types(1)	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
long long	Double long word integers and long long int type
Hex floats	Hexadecimal floating-point constants
Complex	Data types for complex numbers
Zero length	Zero-length arrays
Variadic macros	Macros with a variable number of arguments
Variable length	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	Slightly looser rules for escaped newlines
Multi-line strings(1)	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	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

(1) Feature defined for GCC 3.0; definition and examples at http://gcc.gnu.org/onlinedocs/gcc-3.0.4/gcc/C-Extensions.html

Table 5-4. GCC Language Extensions (continued)

Extensions	Descriptions
Inline	Defining inline functions (as fast as macros)
Assembly labels	Specifying the assembler name to use for a C symbol
Extended asm	Assembler instructions with C operands
Constraints	Constraints for asm operands
Alternate keywords	Header files can use __const__, __asm__, etc
Explicit reg vars	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 (limited support)
Other built-ins	Other built-in functions (see)
Vector extensions	Using vector instructions through built-in functions
Target built-ins	Built-in functions specific to particular targets
Pragmas	Pragmas accepted by GCC
Unnamed fields	Unnamed struct/union fields within structs/unions
Thread-local	Per-thread variables

5.13.2 Function Attributes

The following GCC function attributes are supported: always_inline, const, constructor, deprecated, format, format_arg, malloc, noinline, noreturn, pure, section, unused, used, visibility, and warn_unused_result.

The format attribute is applied to the declarations of printf, fprintf, sprintf, snprintf, vprintf, vfprintf, vsprintf, vsnprintf, scanf, fscanf, vfscanf, vscanf, vsscanf, and sscanf in stdio.h. Thus when GCC extensions are enabled, the data arguments of these functions are type checked against the format specifiers in the format string argument and warnings are issued when there is a mismatch. These warnings can be suppressed in the usual ways if they are not desired.

The malloc attribute is applied to the declarations of malloc, calloc, realloc and memalign in stdlib.h.

5.13.3 Variable Attributes

The following variable attributes are supported: aligned, deprecated, mode, packed, section, transparent_union, unused, used, and weak.

The used attribute is defined in GCC 4.2 (see http://gcc.gnu.org/onlinedocs/gcc-4.2.4/gcc/Variable-Attributes.html#Variable-Attributes).

The packed attribute for structure and union types is available only when there is hardware support for unaligned accesses. This means when --unaligned_access=on, which it is by default for the Cortex devices (A8, R4, M3, M4).

5.13.4 Type Attributes

The following type attributes are supported: aligned, deprecated, packed, transparent_union, unused, and visibility.

Members of a packed structure are stored as closely to each other as possible, omitting additional bytes of padding usually added to preserve word-alignment. For example, assuming a word-size of 4 bytes ordinarily has 3 bytes of padding between members c1 and i, and another 3 bytes of trailing padding after member c2, leading to a total size of 12 bytes:

```
struct unpacked_struct { char c1; int i; char c2;};
```

However, the members of a packed struct are byte-aligned. Thus the following does not have any bytes of padding between or after members and totals 6 bytes:

```
struct __attribute__((__packed__)) packed_struct { char c1; int i; char c2; };
```

Subsequently, packed structures in an array are packed together without trailing padding between array elements.

Bit fields of a packed structure are bit-aligned. The byte alignment of adjacent struct members that are not bit fields does not change. However, there are no bits of padding between adjacent bit fields.

The packed attribute can only be applied to the original definition of a structure or union type. It cannot be applied with a typedef to a non-packed structure that has already been defined, nor can it be applied to the declaration of a struct or union object. Therefore, any given structure or union type can only be packed or non-packed, and all objects of that type will inherit its packed or non-packed attribute.

The packed attribute is not applied recursively to structure types that are contained within a packed structure. Thus, in the following example the member s retains the same internal layout as in the first example above. There is no padding between c and s, so s falls on an unaligned boundary:

```
struct __attribute__((__packed__)) outer_packed_struct { char c; struct unpacked_struct s; };
```

It is illegal to implicitly or explicitly cast the address of a packed struct member as a pointer to any non-packed type except an unsigned char. In the following example, p1, p2, and the call to foo are all illegal.

```
void foo(int *param);
struct packed_struct ps;

int *p1 = &ps.i;
int *p2 = (int *)&ps.i;
foo(&ps.i);
```

However, it is legal to explicitly cast the address of a packed struct member as a pointer to an unsigned char:

```
unsigned char *pc = (unsigned char *)&ps.i;
```

Packed can also be applied to enumerated types. On an enum, packed indicates that the smallest integral type should be used.

The TI compiler also supports an unpacked attribute for an enumeration type to allow you to indicate that the representation is to be an integer type that is no smaller than int; in other words, it is not *packed*.

5.14 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 ARP32 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 ARP32 compiler treats memory as multiple linear block that is 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 32-bit address space is available in target memory.

The Linker Defines the Memory Map

NOTE: 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).

6.1.1 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 file information in the *ARP32 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** contains tables for initializing variables and constants.
  - The **.init_array section** contains global constructor tables.
  - The **.data section** contains initialized global and static variables that are within reach of the GDP register offset.
  - The **.fardata** section contains non-const, initialized global and static variables that are not within reach of the GDP register offset.
  - The **.const section** contains string constants, string literals, switch tables, and data defined with the C/C++ qualifiers *far* and *const* (provided the constant is not also defined as *volatile*). String literals are placed in the .const:.string subsection to enable greater link-time placement control.
  - The **.rodata section** reserves space for const near global and static variables that are within reach of the GDP register offset.
  - The **.text section** contains all the executable code 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 uninitialized global and static variables.
  - The **.far section** reserves space for uninitialized global and static variables that are not within reach of the GDP register offset..
  - 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 .bss, .data and .rodata sections must be adjacent within memory. The GDP is set to correspond to the address of the beginning of this group.

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 5.8.2 and Section 5.8.4).

The linker takes the individual sections from different object files 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	.init_array	ROM or RAM
.cinit	ROM or RAM	.pinit	ROM or RAM
.const	ROM or RAM	.rodata	ROM or RAM
.data	ROM or RAM	.stack	RAM
.far	RAM	sysmem	RAM
.fardata	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 *ARP32 Assembly Language Tools User's Guide*.

6.1.2 C/C++ System 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 *stack pointer* (SP) to manage this stack. SP points to the next unused location on the stack.

The linker sets the stack size, creates a global symbol, __TI_STACK_SIZE, and assigns it a value equal to the stack size in bytes. The default stack size is 24K 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 the linker description chapter in the *ARP32 Assembly Language Tools User's Guide*.

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.

Stack Overflow

NOTE: 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.

6.1.3 Dynamic Memory Allocation

The run-time-support library supplied with the ARP32 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, __TI_SYSMEM_SIZE, and assigns it a value equal to the size of the heap in bytes. The default size is 24K bytes. For more information on the --heap_size option, see the linker description chapter in the *ARP32 Assembly Language Tools User's Guide*.

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));
```

6.1.4 Data Address Model

If a near or far keyword is not specified for an object, the compiler generates far accesses to aggregate data and near accesses to all other data. This means that structures, unions, C++ classes, and arrays are not accessed through the global data pointer (GDP).

Non-aggregate data, by default, is placed in the .bss section and is accessed using relative-offset addressing from the global data pointer (GDP). GDP points to the beginning of the .bss section. Accessing data via the data page pointer is generally faster and uses fewer instructions than the mechanism used for far data accesses.

If you want to use near accesses to aggregate data, you must specify the --mem_model:data=near option, or declare your data with the near keyword.

If you have too much static and extern data to fit within a 19-bit scaled offset from the beginning of the .bss section, you cannot use --mem_model:data=near. The linker will issue an error message if there is a DP-relative data access that will not reach.

The --mem_model:data=*type* option controls how data is accessed:

--mem_model:data=near	Data accesses default to near
--mem_model:data=far	Data accesses default to far
--mem_model:data=far_aggregates	Data accesses to aggregate data default to far, data accesses to non-aggregate data default to near. This is the default behavior.

The --mem_model:data options do not affect the access to objects explicitly declared with the near of far keyword.

By default, all run-time-support data is defined as near.

For more information on near and far accesses to data, see Section 5.5.4.

6.1.5 Trampoline Generation for Function Calls

The ARP32 compiler generates trampolines by default. Trampolines are a method for modifying function calls at link time to reach destinations that would normally be too far away. When a function call is out of range of its destination, the linker will generate an indirect branch (or trampoline) to that destination, and will redirect the function call to point to the trampoline. The end result is that these function calls branch to the trampoline, and then the trampoline branches to the final destination.

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 (halfword) boundary
unsigned short, wchar_t	Bits 0-15 of register	16 bits aligned to 16-bit (halfword) boundary
int, signed int	Bits 0-31 of register	32 bits aligned to 32-bit (word) boundary
unsigned int	Bits 0-31 of register	32 bits aligned to 32-bit (word) boundary
enum	Smallest integer type that contains all the enumerated values. See Table 5-2 for details.	See Table 5-2.
long. signed long	Bits 0-31 of register	32 bits aligned to 32-bit (word) boundary
unsigned long	Bits 0-31 of register	32 bits aligned to 32-bit (word) boundary
long long	Even/odd register pair	64 bits aligned to 32-bit (word) boundary
unsigned long long	Even/odd register pair	64 bits aligned to 32-bit (word) boundary
float	Bits 0-31 of register	32 bits aligned to 32-bit (word) boundary
double	Register pair	64 bits aligned to 32-bit (word) boundary
long double	Register pair	64 bits aligned to 32-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.
array	Members are stored as their individual types require.	Members are stored as their individual types require; aligned to 32-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 0-31 of register	32 bits aligned to 32-bit (word) boundary
pointer to member function	Components stored as their individual types require	64 bits aligned to 32-bit (word) boundary

(1) Negative values are sign-extended to bit 31.

6.2.1.1 char and short Data Types (signed and unsigned)

The char and unsigned char data types are stored in memory as a single byte and are loaded to and stored from bits 0-7 of a register (see Figure 6-1). Objects defined as short or unsigned short are stored in memory as two bytes at a halfword (2 byte) aligned address and they are loaded to and stored from bits 0-15 of a register (see Figure 6-1).

Figure 6-1. Char and Short Data Storage Format

Signed 8-bit char

																								MS						LS
S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	I	I	I	I	I	I	I
31																								7						0

Unsigned 8-bit char

																								MS						LS
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	U	U	U	U	U	U	U
31																								7						0

Signed 16-bit short

																MS														LS
S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	I	I	I	I	I	I	I	I	I	I	I	I	I	I
31																								15						0

Unsigned 16-bit short

																MS														LS
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U
31																								15						0

LEGEND: S = sign, I = signed integer, U = unsigned integer, MS = most significant, LS = least significant

6.2.1.2 float, int, and long Data Types (signed and unsigned)

The int, unsigned int, float, long and unsigned long data types are stored in memory as 32-bit objects at word (4 byte) aligned addresses. Objects of these types are loaded to and stored from bits 0-31 of a register, as shown in Figure 6-2. In big-endian mode, 4-byte objects are loaded to registers by moving the first byte (that is, the lower address) of memory to bits 24-31 of the register, moving the second byte of memory to bits 16-23, moving the third byte to bits 8-15, and moving the fourth byte to bits 0-7. In little-endian mode, 4-byte objects are loaded to registers by moving the first byte (that is, the lower address) of memory to bits 0-7 of the register, moving the second byte to bits 8-15, moving the third byte to bits 16-23, and moving the fourth byte to bits 24-31.

Figure 6-2. 32-Bit Data Storage Format

Single-precision floating char

									MS																				LS
S	E	E	E	E	E	E	E	E	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
31								23																					0

Signed 32-bit integer, enum, or long char

MS																														LS
S	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I
31																														0

Unsigned 32-bit integer or long

MS																														LS
U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U
31																														0

LEGEND: S = sign, M = Mantissa, U = unsigned integer, E = exponent, I = signed integer, MS = most significant, LS = least significant

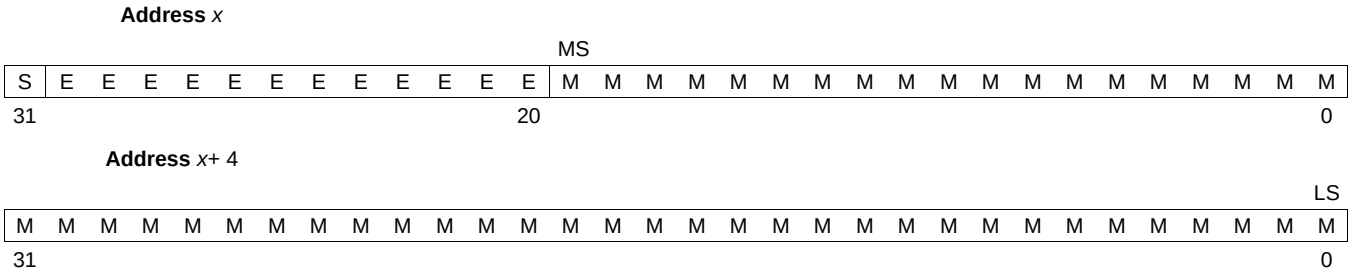
6.2.1.3 double, long double, and long long Data Types (signed and unsigned)

Double, long double, long long and unsigned long long data types are stored in memory in a pair of registers and are always referenced as a pair. These types are stored as 64-bit objects at word (4 byte) aligned addresses. The word at the lowest address contains the sign bit, the exponent, and the most significant part of the mantissa. The word at the higher address contains the least significant part of the mantissa. This is true regardless of the endianness of the target.

Objects of this type are loaded into and stored in register pairs, as shown in [(1)]. The most significant memory word contains the sign bit, exponent, and the most significant part of the mantissa. The least significant memory word contains the least significant part of the mantissa.

(1) The SP is preserved by the convention that everything pushed on the stack is popped off before returning.

Figure 6-3. Double-Precision Floating-Point Data Storage Format



LEGEND: S = sign, M = mantissa, E = exponent, MS = most significant, LS = least significant

6.2.1.4 Pointer to Data Member Types

Pointer to data member objects are stored in memory like an unsigned int (32 bit) integral type. Its value is the byte offset to the data member in the class, plus 1. The zero value is reserved to represent the NULL pointer to the data member.

6.2.1.5 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.

6.2.1.6 Structure and Array Alignment

Structures are aligned according to the member with the most restrictive alignment requirement. Structures are padded so that the size of the structure is a multiple of its alignment. Arrays are always word aligned. Elements of arrays are stored in the same manner as if they were individual objects.

6.2.2 Bit Fields

Bit fields are the only objects that are packed within a byte. That is, two bit fields can be stored in the same byte. Bit fields can range in size from 1 to 64 bits, but they never span a 4-byte boundary. See Figure 6-3 for long long data type storage.

Bit fields are packed into registers from the LSB to the MSB in the order in which they are defined, and packed in memory from LSbyte to MSbyte.

Plain int bit fields are unsigned. Use a signed int if you need a signed bit field.

Bit fields are treated as the declared type. The size and alignment of the struct containing the bit field depends on the declared type of the bit field. For example, this struct uses up 4 bytes and is aligned at 4 bytes:

```
struct st {int a:4};
```

Unnamed bit fields affect the alignment of the struct or union. For example, this struct uses 4 bytes and is aligned at a 4-byte boundary:

```
struct st {char a:4; int :22;};
```

Bit fields declared volatile are accessed according to the bit field's declared type. A volatile bit field reference generates exactly one reference to its storage; multiple volatile bit field accesses are not merged.

Figure 6-4 illustrates bit-field packing, using the following bit field definitions:

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

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

Figure 6-4. Bit-Field Packing in Little-Endian Format

Little-endian register

MS LS

X	E	E	E	E	E	E	E	E	E	D	D	C	C	C	B	B	B	B	B	B	B	B	B	A	A	A	A	A	A	A	A
X	8	7	6	5	4	3	2	1	0	1	0	2	1	0	9	8	7	6	5	4	3	2	1	0	6	5	4	3	2	1	0

31 0

Little-endian memory

Byte 0								Byte 1								Byte 2								Byte 3							
B	A	A	A	A	A	A	A	B	B	B	B	B	B	B	B	E	E	D	D	C	C	C	B	X	E	E	E	E	E	E	E
0	6	5	4	3	2	1	0	8	7	6	5	4	3	2	1	1	0	1	0	2	1	0	9	X	8	7	6	5	4	3	2

LEGEND: X = not used, MS = most significant, LS = least significant

6.2.3 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 6.7.

- 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 CSL1 points to the string):

  ```
          .sect   ".const:.string"
  $C$SL1: .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
Link register	Contains the return address of a function call
Program counter	Contains the current address of code being executed

Table 6-4. Register Usage

Register	Usage	Preserved by Function [1]
R0-R1	Return register, expression register	Parent
R2-R4	Argument register, expression register	Parent
R5-R7	Expression register	Child
CSR	Program status register	N/A
LR	Link register	Child
PC	Program counter register	N/A
SP	Stack pointer register	Child

[1] The parent function refers to the function making the function call. The child function refers to the function being called.

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.

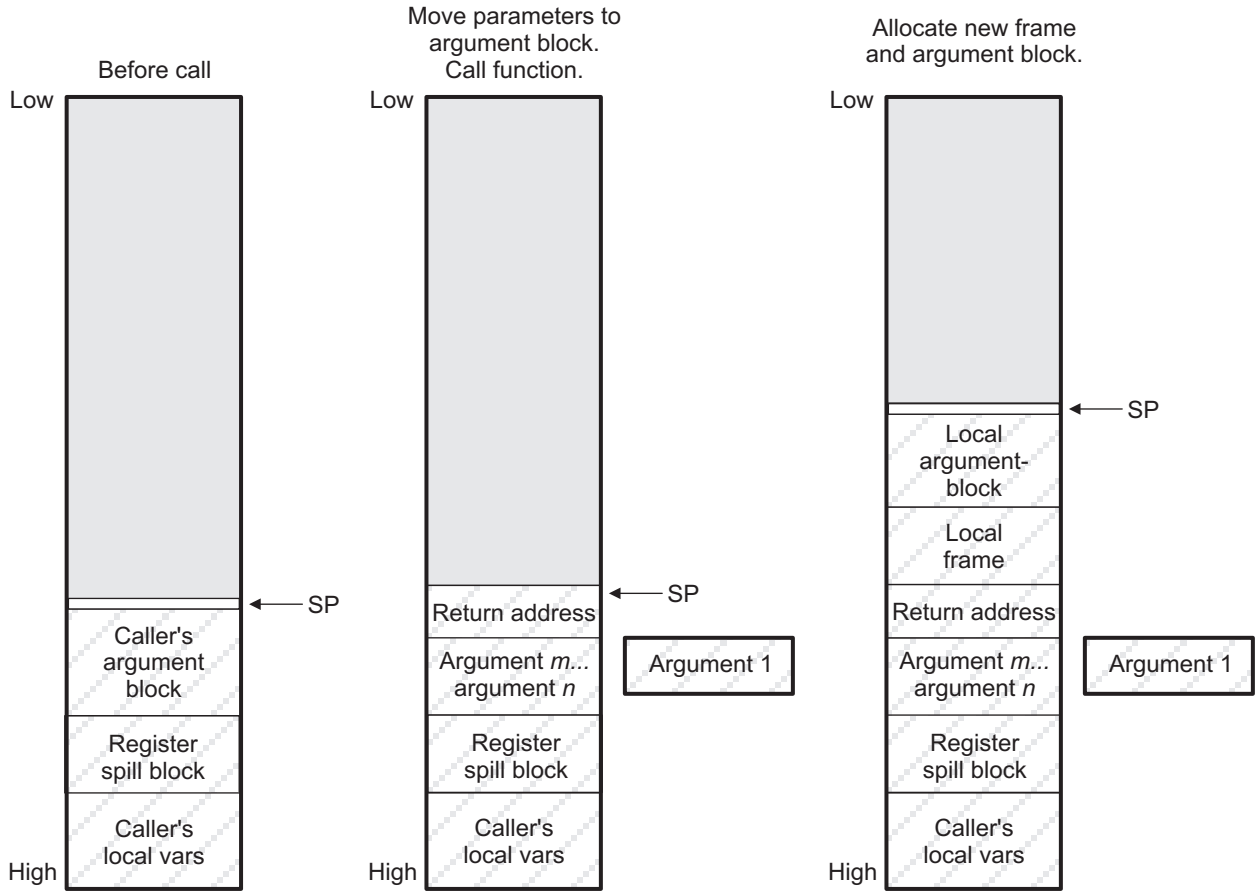
The following sections use this terminology to describe the function-calling conventions of the C/C++ compiler:

- **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 R0-R4. 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 R5-R7. 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 6-5 illustrates a typical function call. In this example, parameters that cannot be placed in registers are passed to the function on the stack. The function then allocates local variables and calls another function. This example shows the allocated 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.

The term *argument block* refers to the part of the local frame used to pass arguments to other functions. Parameters 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.

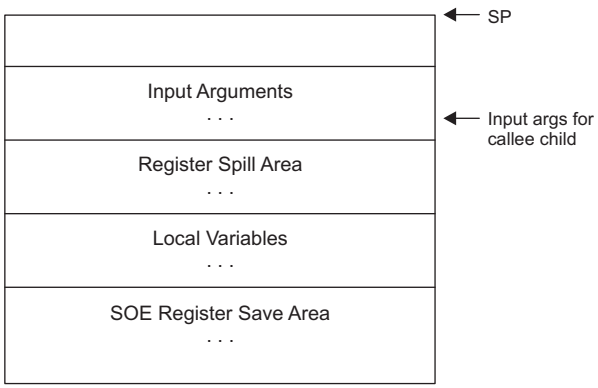
Figure 6-5. Use of the Stack During a Function Call



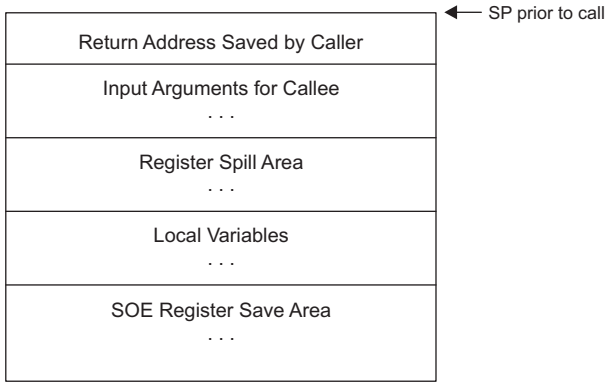
6.4.1 How a Function Makes a Call

A function (parent function) performs the following tasks when it calls another function (child function).

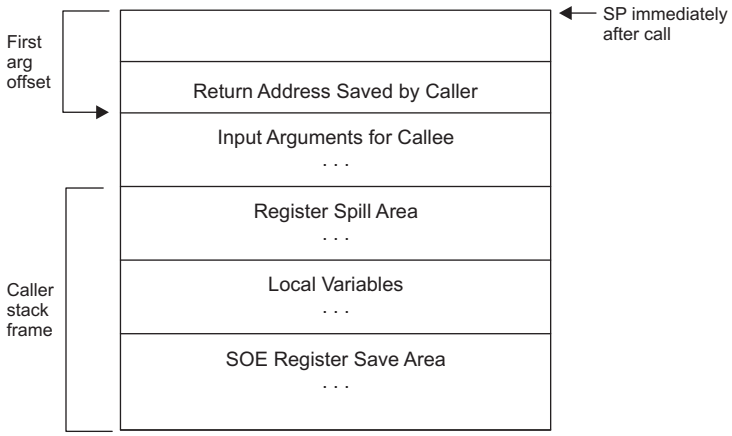
1. The calling function (parent) saves save-on-call registers that are live across a call as well as input arguments for the callee function. The save-on-call registers are R0-R4.
2. If the called function (child) returns a structure, the caller allocates space for the structure and passes the address of that space to the called function as the first argument.
3. The caller places the first arguments in registers R2-R4, 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.



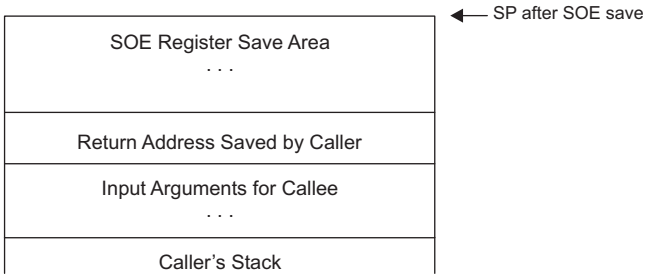
4. The CALL instruction is executed. The return address is saved to the LR register. The previous return address is saved to the 32-bit return address block.



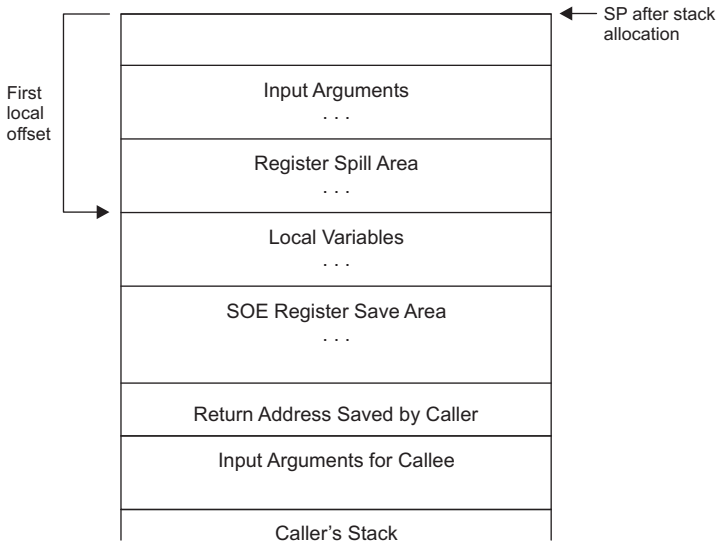
5. The CALL instruction decrements the SP by 32 bits and jumps to location.



6. The CALLEE saves save-on-entry registers.



7. The CALLEE allocates its stack frame.



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 (R5-R7). If the function is an interrupt, no registers are saved here because the ARP32 hardware will automatically save all registers into corresponding shadow registers (SR0-SR7). The only exception to this rule for interrupts is if the programmer asks that the compiler generate context save and restore code to facilitate nested interrupts, in which case the shadow registers (SR0-SR7) are saved onto the stack and then later restored. See --saveregs={nested_nmi, nested_std, nested_full} for more information.For more information, see the --saveregs discussion in Section 2.3.4.
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 + number of spilled registers + 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 R0 (or R0 and R1 for 64-bit values).
6. If the called function returns a structure, it copies the structure to the memory block that the return-by-reference register, R1, points to. If the caller does not use the return value, R1 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 to the SP.
8. The called function restores all save-on-entry registers.

9. The called function invokes the RET instruction, which restores the PC value with the return address from the LR register. The RET instruction then increments the SP by 32 bits.

The following example is typical of how a called function responds to a call:

```
            ; called function entry point
STRF R7,R5  ; save R5, R6, R7
SUB 0xc, SP ; allocate frame
...         ; body of the function
ADD 0xc, SP ; deallocate frame
LDRF R5, R7 ; restore R5, R6, R7
RET         ; execute RET, causing a return
```

6.4.3 Accessing Arguments and Local Variables

A function accesses its local nonregister variables indirectly through the stack pointer (SP) and its stack arguments. The SP always points to the top of the stack (points to the most recently pushed value) . For example:

```
LDW       *+SP(4), R0  ; load local var from stack
```

Since the stack grows toward smaller addresses, the local and argument data on the stack for the C/C++ function is accessed with a positive offset from the SP register.

6.5 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.

6.5.1 Saving Registers During Interrupts

When C/C++ code is interrupted, the interrupt routine does not need to preserve the contents of machine registers since that is handled automatically by the hardware. The hardware saves all machine registers into corresponding shadow registers. If you intend to enable nested interrupts, the interrupt routine must store the shadow registers to the stack on entry and restore them on exit. This can be done using the --savereg option in the compiler. There are three modes available with --savereg:

- --savereg=nested_nmi facilitates single-level nested interrupts by generating save and restore operations for non-maskable interrupt handlers.
- --savereg=nested_std facilitates multi-level nested interrupts by generating save and restore operations for non-maskable interrupt handlers and maskable interrupt handlers.
- --savereg=nested_full facilitates full multi-level nested interrupts by generating save and restore operations for non-maskable interrupt handlers and maskable interrupt handlers, including the stack pointer register and interrupt enable register.

6.5.2 Using C/C++ Interrupt Routines

When C/C++ code is interrupted, the interrupt routine does not need to preserve the contents of machine registers since that is handled automatically by the hardware. The hardware saves all machine registers into corresponding shadow registers. For example:

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

If you intend to enable nested interrupts, the interrupt routine must store the shadow registers to the stack on entry and restore them on exit. This can be done using the --savereg option in the compiler. See Section 6.5.1 for the three modes available.

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 5.8.8 and Section 5.5.3, respectively.

6.5.3 How to Map Interrupt Routines to Interrupt Vectors

It is possible to use C to map C interrupt routines to interrupt vectors for ARP32. You can do this in any C file or in a specific intvecs.c file. Follow these steps:

1. Create intvecs.c and include your interrupt routines. For each routine:
   (a) Use the interrupt keyword or INTERRUPT pragma to designate the interrupt routine. (Use NMI_INTERRUPT pragma to designate NMI interrupt routines). The compiler uses the interrupt keyword or pragma to ensure that all interrupt handlers are properly compiled and that they are not removed during the link step.
   (b) Remember that an interrupt routine must have a void return type and accept no arguments.

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

2. Create a vec_table array to identify references to each routine. Using the DATA_SECTION pragma, place the array in a unique section for placement (for example, .intvecs):

```
#pragma DATA_SECTION(vec_table,".intvecs");
void (*vec_table[10])(void) =
{
    &_reset_handler,
    &_nmi_handler,
    &_swi_handler,
    &_dummy_handler,
    &_int_4_handler,
    &_int_5_handler,
    &_int_6_handler,
    &_int_7_handler,
    &_dummy_handler,
    &_dummy_handler
};
```

3. Within a linker command file, place the vector section (.intvecs) at the ARP32 interrupt vector table location (0x00000000) and flag it as NOINIT to prevent it from being automatically initialized at boot time:

```
MEMORY
{
    INTVECS : origin = 0000000h  length = 00000030h
    ...
}

SECTIONS
{
    ...
    .intvecs(NOINIT)    >       INTVECS PAGE 0
    ...
}
```

6.5.4 Using Software Interrupts

A software interrupt is a synchronous exception generated by the execution of a particular instruction. Applications use software interrupts to request services from a protected system, such as an operating system.

6.5.5 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 directly from C/C+= code.
- 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*.
- Interrupt routines are not reentrant. If an interrupt routine enables interrupts of its type, it must save a copy of the return address and SPSR (the saved program status register) before doing so.

6.6 Built-In Functions

Built-in functions are predefined by the compiler. They can be called like a regular function as long as the arp32.h header file is included in the source file. This header file supplies the proper prototype, though their definition is supplied by the compiler.

The ARP32 compiler supports the following built-in functions. For more information on additional built-in functions, see arp32.h, which is included in the ARP32 installation package.

Table 6-5. Built-In ARP32 Functions

Description	Syntax
Standard Built-In Functions	
Absolute value	int _abs (int);
Signed Minimum of two integers	int _min (int, int);
Signed Maximum of two integers	int _max (int, int);
Unsigned Minimum of two integers	unsigned int _minu (unsigned int, unsigned int);
Unsigned Maximum of two integers	unsigned int _maxu (unsigned int, unsigned int);
Left-most bit detect	unsigned int _lmbd (unsigned int, unsigned int);
Convert Unsigned int to long long	long long _itoll(unsigned int, unsigned int);
Extract unsigned int 'hi' from long long	unsigned int _hill (long long);
Extract unsigned int 'lo' from long long	unsigned int _loll (long long);
Interrupt Control Built-In Functions	
Extract Interrupt ID number	unsigned int _get_inum();
Enable all interrupts	unsigned int _enable_interrupts();
Disable all interrupts	unsigned int _disable_interrupts();
Restore Prior Interrupt State	void _restore_interrupts(unsigned int);
Time Stamp Counter Control Built-In Functions	
Start Time Stamp Counter	void _tsc_start();
Get Time Stamp Counter value	unsigned long long _tsc_gettime();

6.7 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 files. 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 file to the symbol c_int00.

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

1. Reserved space for the run-time stack and sets up the initial value of the stack pointer (SP) and global data pointer (GDP).
2. Calls the function _ _TI_auto_init to perform the C/C++ autoinitialization.

   The _ _TI_auto_init function does the following tasks:

   - Processes the binit copy table, if present.
   - Performs C autoinitialization of global/static variables. For more information, Section 6.7.2.
   - Calls C++ initialization routines for file scope construction from the global constructor table. For more information, see Section 6.7.2.6.
3. 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.

6.7.1 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 stack is aligned at a 32-bit boundary at function entry so that local 64-bit variables are allocated in the stack with correct alignment. 64-bit values are maintained as two 32-bit values.

6.7.2 EABI Automatic Initialization of Variables

Any global variables declared as preinitialized 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.

6.7.2.1 EABI Zero Initializing Variables

In ANSI C, global and static variables that are not explicitly initialized, must be set to 0 before program execution. The C/C++ EABI compiler supports preinitialization of uninitialized variables by default. This can be turned off by specifying the linker option --zero_init=off.

6.7.2.2 EABI Direct Initialization

The EABI compiler uses direct initialization to initialize global variables. For example, consider the following C code:

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

The compiler allocates the variables 'i' and 'a[] to .data section and the initial values are placed directly.

```
        .global i
        .data
        .align  4
i:
        .field          23,32                   ; i @ 0

        .global a
        .data
        .align  4
a:
        .field          1,32                    ; a[0] @ 0
        .field          2,32                    ; a[1] @ 32
        .field          3,32                    ; a[2] @ 64
        .field          4,32                    ; a[3] @ 96
        .field          5,32                    ; a[4] @ 128
```

Each compiled module that defines static or global variables contains these .data sections. The linker treats the .data section like any other initialized section and creates an output section. In the load-time initialization model, the sections are loaded into memory and used by the program. See Section 6.7.2.5.

In the run-time initialization model, the linker uses the data in these sections to create initialization data and an additional initialization table. The boot routine processes the initialization table to copy data from load addresses to run addresses. See Section 6.7.2.3.

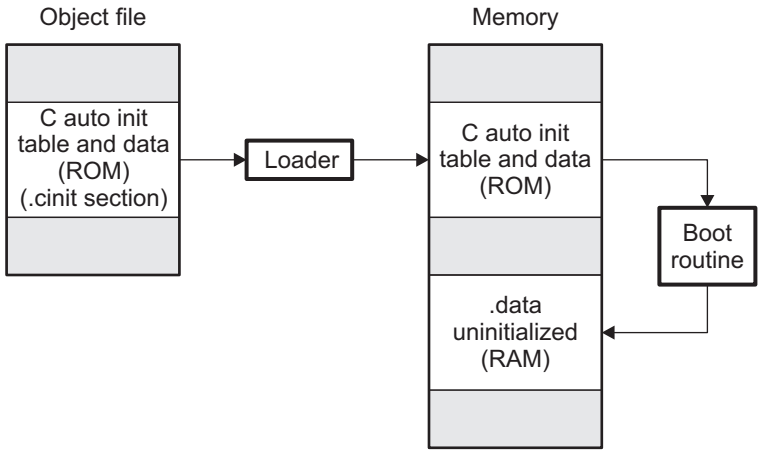
6.7.2.3 Autoinitialization of Variables at Run Time in EABI Mode

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 linker creates an initialization table and initialization data from the direct initialized sections in the compiled module. The table and data are used by the C/C++ boot routine to initialize variables in RAM using the table and data in ROM.

Figure 6-6 illustrates autoinitialization at run time in EABI Mode. Use this method in any system where your application runs from code burned into ROM.

Figure 6-6. Autoinitialization at Run Time in EABI Mode



6.7.2.4 Autoinitialization Tables in EABI Mode

In EABI mode, the compiled object files do not have initialization tables. The variables are initialized directly . The linker, when the --rom_model option is specified, creates C auto initialization table and the initialization data. The linker creates both the table and the initialization data in an output section named .cinit.

The autoinitialization table has the following format:

_TI_CINIT_Base:

32-bit load address	32-bit run address
⋮	⋮
32-bit load address	32-bit run address

_TI_CINT_Limit:

The linker defined symbols __TI_CINIT_Base and __TI_CINIT_Limit point to the start and end of the table, respectively. Each entry in this table corresponds to one output section that needs to be initialized. The initialization data for each output section could be encoded using different encoding.

The load address in the C auto initialization record points to initialization data with the following format:

8-bit index	Encoded data

The first 8-bits of the initialization data is the handler index. It indexes into a handler table to get the address of a handler function that knows how to decode the following data.

The handler table is a list of 32-bit function pointers.

_TI_Handler_Table_Base:

32-bit handler 1 address
⋮
32-bit handler n address

_TI_Handler_Table_Limit:

The *encoded data* that follows the 8-bit index can be in one of the following format types. For clarity the 8-bit index is also depicted for each format.

6.7.2.4.1 Length Followed by Data Format in EABI Mode

8-bit index	24-bit padding	32-bit length (N)	N byte initialization data (not compressed)

The compiler uses 24-bit padding to align the length field to a 32-bit boundary. The 32-bit length field encodes the length of the initialization data in bytes (N). N byte initialization data is not compressed and is copied to the run address as is.

The run-time support library has a function __TI_zero_init() to process this type of initialization data. The first argument to this function is the address pointing to the byte after the 8-bit index. The second argument is the run address from the C auto initialization record.

6.7.2.4.2 Zero Initialization Format in EABI Mode

8-bit index	24-bit padding	32-bit length (N)

The compiler uses 24-bit padding to align the length field to a 32-bit boundary. The 32-bit length field encodes the number of bytes to be zero initialized.

The run-time support library has a function __TI_zero_init() to process the zero initialization. The first argument to this function is the address pointing to the byte after the 8-bit index. The second argument is the run address from the C auto initialization record.

6.7.2.4.3 Run Length Encoded (RLE) Format in EABI Mode

8-bit index	Initialization data compressed using run length encoding

The data following the 8-bit index is compressed using Run Length Encoded (RLE) format. uses a simple run length encoding that can be decompressed using the following algorithm:

1. Read the first byte, Delimiter (D).
2. Read the next byte (B).
3. If B != D, copy B to the output buffer and go to step 2.
4. Read the next byte (L).
   (a) If L == 0, then length is either a 16-bit, a 24-bit value, or we've reached the end of the data, read next byte (L).
      (i) If L == 0, length is a 24-bit value or the end of the data is reached, read next byte (L).
         (i) If L == 0, the end of the data is reached, go to step 7.
         (ii) Else L <<= 16, read next two bytes into lower 16 bits of L to complete 24-bit value for L.
      (ii) Else L <<= 8, read next byte into lower 8 bits of L to complete 16-bit value for L.
   (b) Else if L > 0 and L < 4, copy D to the output buffer L times. Go to step 2.
   (c) Else, length is 8-bit value (L).
5. Read the next byte (C); C is the repeat character.
6. Write C to the output buffer L times; go to step 2.
7. End of processing.

The run-time support library has a routine __TI_decompress_rle24() to decompress data compressed using RLE. The first argument to this function is the address pointing to the byte after the 8-bit index. The second argument is the run address from the C auto initialization record.

RLE Decompression Routine

NOTE: The previous decompression routine, __TI_decompress_rle(), is included in the run-time-support library for decompressing RLE encodings that are generated by older versions of the linker.

6.7.2.4.4 Lempel-Ziv-Storer-Szymanski Compression (LZSS) Format

8-bit index	Initialization data compressed using LZSS

The data following the 8-bit index is compressed using LZSS compression. The run-time support library has the routine __TI_decompress_lzss() to decompress the data compressed using LZSS. The first argument to this function is the address pointing to the byte after the 8-bit index. The second argument is the run address from the C auto initialization record.

6.7.2.5 Initialization of Variables at Load Time in EABI Mode

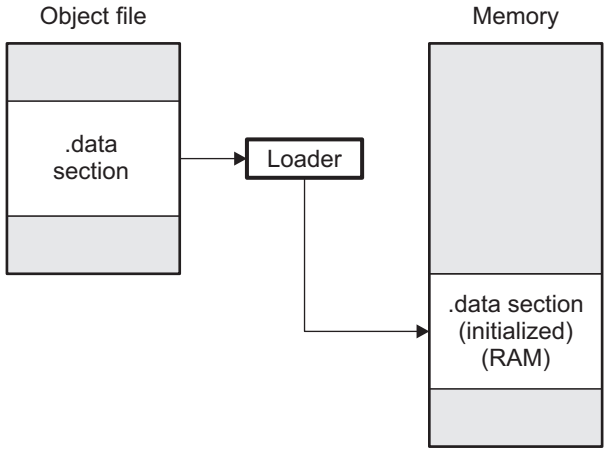
Initialization of variables at load time enhances performance by reducing boot time and by saving the memory used by the initialization tables. To use this method, invoke the linker with the --ram_model option.

When you use the --ram_model link option, the linker does not generate C autoinitialization tables and data. The direct initialized sections (.data) in the compiled object files are combined according to the linker command file to generate initialized output sections. The loader loads the initialized output sections into memory. After the load, the variables are assigned their initial values.

Since the linker does not generate the C autoinitialization tables, no boot time initialization is performed.

Figure 6-7 illustrates the initialization of variables at load time in EABI mode.

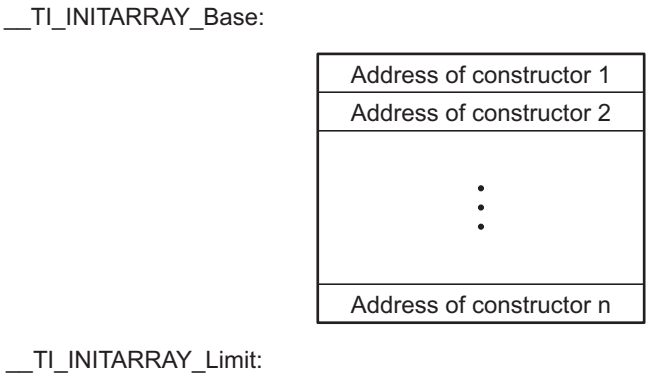
Figure 6-7. Initialization at Load Time in EABI Mode



6.7.2.6 Global Constructors in EABI Mode

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 .init_array. The linker combines the .init_array section form each input file to form a single table in the .init_array section. The boot routine uses this table to execute the constructors. The linker defines two symbols to identify the combined .init_array table as shown below. This table is not null terminated by the linker.

Figure 6-8. Constructor Table for EABI Mode



Using Run-Time-Support Functions and Building Libraries

Some of the features of C/C++ (such as I/O, dynamic memory allocation, string operations, and trigonometric functions) are provided as an ANSI/ISO C/C++ standard library, rather than as part of the compiler itself. The TI implementation of this library is the run-time-support library (RTS). The C/C++ compiler implements the complete ISO standard library except for those facilities that handle exception conditions, signal 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 utility 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

ARP32 compiler releases include pre-built run-time libraries that provide all the standard capabilities. Separate libraries are provided for 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
- Fundamental 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.

When a library is linked, the linker includes only those library members required to resolve undefined references. For more information about linking, see the *ARP32 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

You must use the header files provided with the compiler run-time support when using functions from C/C++ standard library. Set the ARP32_C_DIR environment variable to the include directory where the tools are installed.

7.1.3 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 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.

7.2 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.

The formatting rules for long long data types require ll (lowercase LL) in the format string. For example:

```
printf("%lld", 0x0011223344556677);
printf("llx", 0x0011223344556677);
```

Debugger Required for Default HOST

NOTE: For the default HOST device to work, there must be a debugger to handle the C I/O requests; the default HOST device cannot work by itself in an embedded system. To work in an embedded system, you will need to provide an appropriate driver for your system.

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 (refer to the *Linker Description* chapter in the*ARP32 Assembly Language Tools User's Guide*).

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:

```
cl32 main.c --run_linker --heap_size=400 --library=rtsarp32_v200.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 can 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 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() as in Example 7-1. 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 ARP32 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 infeasible to include all possible run-time-support library variants, compiler releases pre-build only a small number of very commonly-used libraries such as rtsarp32.lib.

To provide maximum flexibility, the run-time-support source code is provided as part of each compiler release. You can build the missing libraries as desired. The linker can also automatically build missing libraries. This is accomplished with a new library build process, the core of which is the executable mklib, which is available beginning with CCS 5.1

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 are required:

- sh (Bourne shell)
- unzip (InfoZIP unzip 5.51 or later, or equivalent)

  You can download the software from http://www.info-zip.org.
- gmake (GNU make 3.81 or later)

  More information is available from GNU at http://www.gnu.org/software/make. GNU make (gmake) is also available in earlier versions of Code Composer Studio. 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 platformts should expliclty report "This program build for Windows32" when the following is executed from the Command Prompt window:

  ```
  gmake -h
  ```

All three of these programs are provided as a non-optional feature of CCS 5.1. They are also available as part of the optional XDC Tools feature if you are using an earlier version of CCS.

The mklib program looks for these executables in the following order:

1. in your PATH
2. in the directory getenv("CCS_UTILS_DIR")/cygwin
3. in the directory getenv("CCS_UTILS_DIR")/bin
4. in the directory getenv("XDCROOT")
5. in the directory getenv("XDCROOT")/bin

If you are invoking mklib from the command line, and these executables are not in your path, you must set the environment variable CCS_UTILS_DIR such that getenv("CCS_UTILS_DIR")/bin contains the correct programs.

7.4.2 Using the Library-Build Process

You should normally let the linker automatically rebuild libraries as needed. If necessary, you can run mklib directly to populate libraries. See Section 7.4.2.2 for situations when you might want to do this.

7.4.2.1 Automatic Standard Library Rebuilding by the Linker

The linker looks for run-time-support libraries primarily through the ARP32_C_DIR environment variable. Typically, one of the pathnames in ARP32_C_DIR is *your install directory*/lib, which contains all of the pre-built libraries, as well as the index library libc.a. The linker looks in ARP32_C_DIR to find a library that is the best match for the build attributes of the application. The build attributes are set indirectly according to the command-line options used to build the application. Build attributes include things like CPU revision. If the library is explicitly named (e.g. rtsarp32.lib), run-time support looks for that library exactly; otherwise, it uses the index library libc.a to pick an appropriate library.

The index library describes a set of libraries with different build attributes. The linker will compare the build attributes for each potential library with the build attributes of the application and will pick the best fit. For details on the index library, see the archiver chapter in the *ARP32 Assembly Language Tools User's Guide*.

Now that the linker has decided which library to use, it checks whether the run-time-support library is present in ARP32_C_DIR . The library must be in exactly the same directory as the index library libc.a. If the library is not present, the linker will will invoke mklib to build it. This happens when the library is missing, regardless of whether the user specified the name of the library directly or allowed the linker to pick the best library from the index library.

The mklib program builds the requested library and places it in 'lib' directory part of ARP32_C_DIR in the same directory as the index library, so it is available for subsequent compilations.

Things to watch out for:

- The linker invokes **mklib** and waits for it to finish before finishing the link, so you will experience a one-time delay when an uncommonly-used library is built for the first time. Build times of 1-5 minutes have been observed. This depends on the power of the host (number of CPUs, etc).
- In a shared installation, where an installation of the compiler is shared among more than one user, it is possible that two users might cause the linker to rebuild the same library at the same time. The **mklib** program tries to minimize the race condition, but it is possible one build will corrupt the other. In a shared environment, all libraries which might be needed should be built at install time; see Section 7.4.2.2 for instructions on invoking **mklib** directly to avoid this problem.
- The index library must exist, or the linker is unable to rebuild libraries automatically.
- The index library must be in a user-writable directory, or the library is not built. If the compiler installation must be installed read-only (a good practice for shared installation), any missing libraries must be built at installation time by invoking **mklib** directly.
- The **mklib** program is specific to a certain version of a certain library; you cannot use one compiler version's run-time support's **mklib** to build a different compiler version's run-time support library.

7.4.2.2 Invoking mklib Manually

You may need to invoke **mklib** directly in special circumstances:

- The compiler installation directory is read-only or shared.
- You want to build a variant of the run-time-support library that is not pre-configured in the index library **libc.a** or known to mklib. (e.g. a variant with source-level debugging turned on.)

7.4.2.2.1 Building Standard Libraries

You can invoke mklib directly to build any or all of the libraries indexed in the index library **libc.a**. The libraries are built with the standard options for that library; the library names and the appropriate standard option sets are known to mklib.

This is most easily done by changing the working directory to be the compiler run-time-support library directory 'lib' and invoking the **mklib** executable there:

```
mklib --pattern=rtsarp32.lib
```

7.4.2.2.2 Shared or Read-Only Library Directory

If the compiler tools are to be installed in shared or read-only directory, mklib cannot build the standard libraries at link time; the libraries must be built before the library directory is made shared or read-only.

At installation time, the installing user must build all of the libraries which will be used by any user. To build all possible libraries, change the working directory to be the compiler RTS library directory 'lib' and invoke the mklib executable there:

```
mklib --all
```

Some targets have many libraries, so this step can take a long time. To build a subset of the libraries, invoke mklib individually for each desired library.

7.4.2.2.3 Building Libraries With Custom Options

You can build a library with any extra custom options desired. This is useful for building a debugging version of the library, or with silicon exception workarounds enabled. The generated library is not a standard library, and must not be placed in the 'lib' directory. It should be placed in a directory local to the project which needs it. To build a debugging version of the library rtsarp32.lib, change the working directory to the 'lib' directory and run the command:

```
mklib --pattern=rtsarp32.lib --name=rtsarp32_debug.lib --install_to=$Project/Debug --
extra_options="-g"
```

7.4.2.2.4 The mklib Program Option Summary

Run the following command to see the full list of options. These are described in Table 7-1.

```
mklib --help
```

Table 7-1. The mklib Program Options

Option	Effect
--index=*filename*	The index library (libc.a) for this release. Used to find a template library for custom builds, and to find the source files (rtssrc.zip). REQUIRED.
--pattern=*filename*	Pattern for building a library. If neither --extra_options nor --options are specified, the library will be the standard library with the standard options for that library. If either --extra_options or --options are specified, the library is a custom library with custom options. REQUIRED unless --all is used.
--all	Build all standard libraries at once.
--install_to=*directory*	The directory into which to write the library. For a standard library, this defaults to the same directory as the index library (libc.a). For a custom library, this option is REQUIRED.
--compiler_bin_dir=*directory*	The directory where the compiler executables are. When invoking mklib directly, the executables should be in the path, but if they are not, this option must be used to tell mklib where they are. This option is primarily for use when mklib is invoked by the linker.
--name=*filename*	File name for the library with no directory part. Only useful for custom libraries.
--options='*str***'**	Options to use when building the library. The default options (see below) are *replaced* by this string. If this option is used, the library will be a custom library.
--extra_options='*str***'**	Options to use when building the library. The default options (see below) are also used. If this option is used, the library will be a custom library.
--list_libraries	List the libraries this script is capable of building and exit. ordinary system-specific directory.
--log=*filename*	Save the build log as *filename*.
--tmpdir=*directory*	Use *directory* for scratch space instead of the ordinary system-specific directory.
--gmake=*filename*	Gmake-compatible program to invoke instead of "gmake"
--parallel=*N*	Compile *N* files at once ("gmake -j N").
--query=*filename*	Does this script know how to build FILENAME?
--help or **--h**	Display this help.
--quiet or **--q**	Operate silently.
--verbose or **--v**	Extra information to debug this executable.

Examples:

To build all standard libraries and place them in the compiler's library directory:

```
mklib --all --index=$C_DIR/lib
```

To build one standard library and place it in the compiler's library directory:

```
mklib --pattern=rtsarp32.lib --index=$C_DIR/lib
```

To build a custom library that is just like rtsarp32.lib, but has symbolic debugging support enabled:

```
mklib --pattern=rtsarp32.lib --extra_options="-g" --index=$C_DIR/lib --install_to=$Project/Debug -
-name=rtsarp32_debug.lib
```

7.4.3 Extending mklib

The **mklib** API is a uniform interface that allows Code Composer Studio to build libraries without needing to know exactly what underlying mechanism is used to build it. Each library vendor (e.g. the TI compiler) provides a library-specific copy of 'mklib' in the library directory that can be invoked, which understands a standardized set of options, and understands how to build the library. This allows the linker to automatically build application-compatible versions of any vendor's library without needing to register the library in advance, as long as the vendor supports mklib.

7.4.3.1 Underlying Mechanism

The underlying mechanism can be anything the vendor desires. For the compiler run-time-support libraries, mklib is just a wrapper which knows how to unpack Makefile from rtssrc.zip and invoke gmake with the appropriate options to build each library. If necessary, mklib can be bypassed and Makefile used directly, but this mode of operation is not supported by TI, and the you are responsible for any changes to Makefile. The format of the Makefile and the interface between mklib and the Makefile is subject to change without notice. The mklib program is the forward-compatible path.

7.4.3.2 Libraries From Other Vendors

Any vendor who wishes to distribute a library that can be rebuilt automatically by the linker must provide:

- An index library (like 'libc.a', but with a different name)
- A copy of mklib specific to that library
- A copy of the library source code (in whatever format is convenient)

These things must be placed together in one directory that is part of the linker's library search path (specified either in ARP32_C_DIR or with the linker --search_path option).

If mklib needs extra information that is not possible to pass as command-line options to the compiler, the vendor will need to provide some other means of discovering the information (such as a configuration file written by a wizard run from inside CCS).

The vendor-supplied mklib must at least accept all of the options listed in Table 7-1 without error, even if they do not do anything.

C++ Name Demangler

The C++ compiler implements function overloading, operator overloading, and type-safe linking by encoding a function's prototype and namespace in its link-level name. The process of encoding the prototype into the linkname is often referred to as name mangling. When you inspect mangled names, such as in assembly files, disassembler output, or compiler or linker diagnostics, 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:

dem-arp32 [*options*] [*filenames*]

dem-arp32	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, the disassembly file, and the linker map file. If no filenames are specified on the command line, dem-arp32 uses standard input.

By default, the C++ name demangler outputs to standard output. 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:

--abi=eabi	Demangles EABI identifiers
-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

```
_Z20calories_in_a_bananav:
;* --------------------------------------*
        STRF      R5, R5
        SUB       0x4, SP
        ADD       0x4, SP, R2
        CALL      _ZN6bananaC1Ev
        NOP

        ADD       0x4, SP, R2
        CALL      _ZN6banana8caloriesEv
        NOP
        ADD       0x4, SP, R2
        CALL      _ZN6bananaD1Ev
        MV        R0, R5

        ADD       0x4, SP
        MV        R5, R0
        LDRF      R5, R5
        RET
        NOP
```

Executing the C++ name demangler demangles all names that it believes to be mangled. Enter:

```
dem-arp32 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():
;* ---------------------------------*
        STRF      R5, R5
        SUB       0x4, SP
        ADD       0x4, SP, R2
        CALL      banana::banana()
        NOP

        ADD       0x4, SP, R2
        CALL      banana::calories()
        NOP
        ADD       0x4, SP, R2
        CALL      banana::~banana()
        MV        R0, R5

        ADD       0x4, SP
        MV        R5, R0
        LDRF      R5, R5
        RET
        NOP
```

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*

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.

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 Linkable Format; a system of object files configured according to the System V Application Binary Interface specification.

emulator— A hardware development system that duplicates the ARP32 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 object file— A linked, executable object file that is downloaded and executed on 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 object file.

input section— A section from an object file that will be linked into an executable object file.

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 executable object file 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 object file 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.

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 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.

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 ARP32 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 ^.

trip count— The number of times that a loop executes before it terminates.

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.

word— A 32-bit addressable location in target memory