2 COMPILER

Overview

The C/C++ compiler (cc21k) compiles ANSI/ISO standard C and C++ code for ADSP-21xxx DSP systems. Analog Devices extensions to the C and C++ standards in the compiler aid development of DSP applications. The cc21k compiler runs from the VisualDSP++ environment or from an operating system command line shell or interpreter.

The sections of this chapter present the following information on the compiler:

- "Compiler Command-Line Interface" on page 2-3 explains the operation of the compiler as it processes programs, including input and output files and command-line switches.
- "C/C++ Compiler Language Extensions" on page 2-54 contains reference information on ADI extensions to the ANSI/ISO standards for the C and C++ programming languages.
- "Preprocessing a Program" on page 2-102 contains information on the preprocessor, which lets you modify source compilation in a number of ways.
- "C/C++ Run-Time Model" on page 2-121 contains reference information about how C and C++ programs, data, and function calls are implemented on the ADSP-21xxx DSPs. This is important for low-level program analysis and interfacing assembly code with C/C++ programs.

- "C/C++ and Assembly Interface" on page 2-147 describes how to call an assembly language subroutine from within a C or C++ program, and how to call a C/C++ function from within an assembly language program.
- "C/C++ Compiler Glossary" on page 2-177 contains a glossary of compiler-related terms.

The C/C++ compiler (cc21k) processes your C and C++ language source files and produces ADSP-21xxx assembler source files. The assembler source files are assembled by the ADSP-21xxx family assembler (easm21k). The assembler creates Executable and Linkable Format (ELF) object files that can either be linked (using the linker) to create an ADSP-21xxx executable file or included in an archive library (elfar). The way in which the compiler controls the assemble, link, and archive phases of the process depends on the source input files and the compiler options used.

Your source files contain the C/C++ program to be processed by the compiler. The cc21k compiler supports the ANSI/ISO standard definitions of the C and C++ languages. For information on the C language standard, see any of the many reference texts on the C language. Analog Devices recommends the Bjarne Stroustrup text "The C++ Programming Language" from Addison Wesley Longman Publishing Co (ISBN: 0201889544) (1997) as a reference text for the C++ programming language.

The cc21k compiler supports the proposed Embedded C++ standard, which defines a subset of the full ISO/IEC 14882:1998 C++ language standard. The proposal excludes features that can detract from compiler performance in embedded systems, such as exception handling and run-time type identification. In addition to the Embedded C++ standard features, cc21k supports templates and all other features of the full C++ standard with the exception of exception handling and run-time type identifications. The additional supported features provide extra functionality without degrading the compiler performance.

The cc21k compiler supports a set of C/C++ language extensions. These extensions support hardware features of the ADSP-21xxx family DSPs. For information on these extensions, see "C/C++ Compiler Language Extensions" on page 2-54

You set the compiler options from the `Project` menu, `Project Options` command, `Compile` tab of the VisualDSP++ IDDE. The selections on this tab control how the compiler processes your source files, letting you select features that include the language dialect, error reporting, and debugger output. For more information on the VisualDSP++ IDDE, see the *VisualDSP++ 2.0 User's Guide for ADSP-21xxx DSPs* and online Help.

Compiler Command-Line Interface

This section describes the following:

- Method used to invoke the compiler from the command line,
- Types of files used by and generated from the compiler
- The option (switch) set used to tailor the compiler's operation

In the default mode of operation, the compiler runs in C mode. This means that the compiler processes source files written in ANSI/ISO standard C language, supplemented with Analog Devices, Inc. (ADI) extensions. Several options allow you to change the compilation mode and language dialect, thus enforcing certain standards and/or disabling the ADI extensions. Table 2-2 on page 2-9 identifies the options that select the language dialect.

While many options are generic between C and C++ dialects, some of them are valid in C++ mode only. Table 2-3 on page 2-9 provides a summary of the generic C/C++ compiler options. Table 2-4 on page 2-16 provides a summary of the C++-specific compiler options.

For a brief description of each option, see

- "C/C++ Mode Selection Switch Descriptions" on page 2-18
- "C/C++ Compiler Common Switch Descriptions" on page 2-19
- "C++ Mode Compiler Switch Descriptions" on page 2-43.

For information on the corresponding compiler options specified from the VisualDSP++ IDDE, see the *VisualDSP++ 2.0 User's Guide for ADSP-21xxx DSPs* and online Help.

Running the Compiler

Use the following syntax for the cc21k command line:

```
cc21k [-switch [-switch …] sourcefile [sourcefile …]]
```

where:

- *-switch* is the name of the switch to be processed. The compiler supports many switches. These switches select the operations and modes for the compiler and other tools. Command-line switches are case-sensitive, for example, `-O` is not the same as `-o`.
- *`sourcefile`* is the name the file to be preprocessed, compiled, assembled, and/or linked.

  A file name can include the drive, directory, file name, and file extension. The compiler supports both Win32- and POSIX-style paths, using either forward or back slashes as the directory delimiter. It also supports Universal Control Network (UNC) path names (starting with two slashes and a network name). If a file name exceeds eight characters in length or contains spaces, enclose it in straight quotes: `"long file name.c"`.

The cc21k compiler uses the file extension to determine what the file contains and what operations to perform upon it. Table 2-1 on page 2-7 lists the allowed extensions. In the default mode of operation, the compiler processes the input file through the listed stages to produce a `.DXE` file.

The following command line, for example:

```
cc21k -O -21062 -Wremarks -o program.dxe source.c
```

runs cc21k with:

`-O`	Specifies optimization for the compiler
`-21062`	Identifies the target processor type in your DSP system
`-Wremarks`	Selects extra diagnostic remarks in addition to warning and error messages
`-o program.dxe`	Selects a name for the compiled, linked output
`source.c`	Specifies the C language source file to be compiled

The following example command line:

```
cc21k -c++ fdot.cpp -T 062.ldf
```

runs cc21k with:

`-c++`	Specifies that all of the source files are written in C++
`fdot.cpp`	Specifies the C++ language source file for your program
`-T 062.ldf`	Specifies the Linker Description File for your DSP system

The normal function of cc21k is to invoke the compiler, assembler, and linker as required to produce an executable object file. The precise operation is determined by the extensions of the input filenames and by various switches.

In normal operation the compiler uses the following files to perform the specified action:

EXTENSION	ACTION
`.c`, `.cpp`, `.cxx`	C/C++ language source file is compiled, assembled, and linked
`.asm`, `.s`	assembly language source file is assembled and linked
`.doj`	object file (from previous assembly) is linked

If multiple files are specified, each is first processed to produce an object file; then all object files are presented to the linker.

This sequence can be stopped at various points by the use of appropriate compiler switches, or by selecting options in the compiler dialogs within the IDDE. These switches are: `-E`, `-P`, `-M`, `-H`, `-S`, `-c`, `-save-temps`.

Because the compiler runs the preprocessor, assembler, and linker as your program is compiled, the compiler's command line can receive input for these programs and direct their operation. Many of the compiler's switches take a file name as an optional parameter. If you do not use the optional output name switch, cc21k names the output for you. Table 2-1 on page 2-7 lists the type of files, names, and extensions cc21k appends to output files.

File searches vary by command line switch and file type. These searches are influenced by the program that is processing the file, any search directories that you select, and any path information that you include in the file name. Table 2-1 on page 2-7 indicates the searches that the preprocessor, compiler, assembler, and linker support. The compiler supports relative and absolute directory names to define file search paths. For information on additional search directories, see the command line switch that controls the specific type of search.

When you provide an input or output file name as an optional parameter, use the following guidelines:

- Use a file name (include the file extension) with either an unambiguous relative path or an absolute path. A file name with an absolute path includes the drive, directory, file name, and file extension. Enclose long file names within straight quotes: `"long file name.c"`. cc21k uses the file extension convention listed in Table 2-1 to determine the input file type.
- Verify that the compiler is using the correct file. If you do not provide the complete file path as part of the parameter or add additional search directories, cc21k looks for input in the current project directory.

Using the verbose output switches for the preprocessor, compiler, assembler, and linker causes each of these tools to echo the name of each file as it is processed.

Table 2-1. Input and Output Files

Input File Extension	File Extension Description
`.c`	C/C++ source file.
`.cpp`, `.cxx`	C++ source file.
`.h`	Header file (referenced by a `#include` directive).
`.pch`	C++ pre-compiled header file.
`.ii`, `.ti`	Template instantiation files — used internally by the compiler when instantiating templates.
`.ipa`, `.opa`	Interprocedural analysis files — used internally by the compiler when performing interprocedural analysis.

Table 2-1. Input and Output Files (Cont'd)

Input File Extension	File Extension Description
.i	Preprocessed C source, created when preprocess only (-E or -P compiler switch) is specified.
.s, .asm	Assembler source file.
.is	Preprocessed assembly source (retained when -save_temps is specified).
.ldf	Linker Description File.
.doj	Object file to be linked.
.dlb	Library of object files to be linked as needed.
.map	DSP system memory map file output.
.sym	DSP system symbol map file output.

C/C++ Compiler Switches

This section describes the command line switches you can use when compiling. It contains a set of tables that provide a brief description of each switch. These tables are organized by type of switch. Following these tables are sections that provide fuller descriptions of each switch.

C/C++ Compiler Switch Summaries

This section contains a set of tables that summarize generic and specific switches (options), as follows:

- Table 2-2, "C or C++ Mode Selection Switches," on page 2-9
- Table 2-3, "C/C++ Compiler Common Switches," on page 2-9
- Table 2-4, "C++ Mode Compiler Switches," on page 2-16.

A brief description of each switch follows the tables, beginning on page 2-18.

Table 2-2. C or C++ Mode Selection Switches

Switch Name	Description
`-analog` (page 2-18)	Supports ANSI/ISO standard C with Analog Devices extensions. Default mode.
`-c++` (page 2-18)	Supports ANSI/ISO standard C++ with Analog Devices extensions.
`-traditional` (page 2-19)	Supports pre-ANSI K&R C.

Table 2-3. C/C++ Compiler Common Switches

Switch Name	Description
`sourcefile` (page 2-19)	Specifies file to be compiled.
`-@ filename` (page 2-19)	Reads command-line input from the file.
`-21020` (page 2-20)	Generates code for ADSP-21020 DSPs.
`-21060` (page 2-20)	Generates code for ADSP-21060 DSPs.
`-21061` (page 2-20)	Generates code for ADSP-21061 DSPs.
`-21062` (page 2-20)	Generates code for ADSP-21062 DSPs.
`-21065L` (page 2-21)	Generates code for ADSP-21065L DSPs.

Table 2-3. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
`-21160` (page 2-21)	Generate code for ADSP-21160 DSPs.
`-21161` (page 2-21)	Generate code for ADSP-21161 DSPs.
`-A` *`name[tokens]`* (page 2-21)	Asserts the specified name as a predicate.
`-aligned-stack` (page 2-22)	Aligns the program stack on a double-word boundary.
`-alttok` (page 2-22)	Allows alternative keywords and sequences in sources.
`-auto-inline` *`factor`* (page 2-23)	Controls how much the compiler automatically inlines functions.
`-build-lib` (page 2-23)	Directs the librarian to build a library file.
`-C` (page 2-23)	Retains preprocessor comments in the output file; must run with the `-E` or `-P` switch).
`-c` (page 2-23)	Compiles and/or assembles only, but do not link.
`-const-read-write` (page 2-24)	Specifies that data accessed via a pointer to const data may be modified elsewhere.
`-D`*`macro`*`[=def]` (page 2-24)	Defines *`macro`*.
`-default-linkage` (page 2-24)	Sets the default linkage type (`C`, `C++`, `asm`).
`-double-size [-32|-64]` (page 2-24)	Selects 32- or 64-bit IEEE format for double.

Table 2-3. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
`-dry` (page 2-25)	Displays, but does not perform, main driver actions (verbose dry-run).
`-dryrun` (page 2-25)	Displays, but does not perform, top-level driver actions (terse dry-run).
`-E` (page 2-26)	Preprocesses, but does not compile, the source file.
`-EE` (page 2-26)	Preprocesses and compiles the source file.
`-extra-keywords` (page 2-26)	Recognizes ADI extensions to ANSI/ISO standards for C and C++. Default mode.
`-flags-`*`tool`* (page 2-27)	Passes command-line switches through the compiler to other build tools.
`-full-version` (page 2-27)	Displays the version number of the driver and any processes invoked by the driver.
`-g` (page 2-27)	Generates DWARF-2 debug information.
`-H` (page 2-28)	Outputs a list of included header files, but does not compile.
`-HH` (page 2-28)	Outputs a list of included header files and compiles.
`-h[elp]` (page 2-28)	Outputs a list of command-line switches.
`-ipa` (page 2-29)	Specifies that inter-procedural analysis should be performed for optimization between translation units.
`-I`*`directory`* (page 2-28)	Appends *`directory`* to the standard search path.

Table 2-3. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
`-include filename` (page 2-29)	Includes the header file for preprocessing.
`-L directory` (page 2-29)	Appends `directory` to the standard library search path.
`-l library` (page 2-29)	Searches `library` for functions when linking.
`-M` (page 2-30)	Generates make rules only, but does not compile.
`-MM` (page 2-30)	Generates make rules and compiles.
`-map filename` (page 2-30)	Directs the linker to generate a memory map of all symbols.
`-mem` (page 2-30)	Enables memory initialization.
`-no-aligned-stack` (page 2-30)	Does not double-word align the program stack.
`-no-alttok` (page 2-30)	Does not allow alternative keywords and sequences in sources.
`-no-builtin` (page 2-31)	Recognizes only built-in functions that begin with two underscores(__).
`-no-defs` (page 2-31)	Disables preprocessor definitions: macros, include directories, library directories, run-time headers, or keyword extensions.
`-no-extra-keywords` (page 2-31)	Does not accept ADI keyword extensions that might affect ISO/ANSI standards for C and C++.
`-no-inline` (page 2-31)	Ignores the `inline` keyword.

Table 2-3. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
`-no-mem` (page 2-31)	Disables memory initialization.
`-no-restrict` (page 2-32)	Disables the `restrict` keyword.
`-no-std-def` (page 2-32)	Disables preprocessor definitions and ADI keyword extensions that do not have leading underscores(__).
`-no-std-inc` (page 2-32)	Searches for preprocessor include header files only in the current directory and in directories specified with the `-I` switch.
`-no-std-lib` (page 2-32)	Searches for only those library files specified with the `-l` switch.
`-O` (page 2-33)	Enables code optimizations.
`-o` *`filename`* (page 2-33)	Specifies the output file.
`-Os` (page 2-33)	Disables optimizations that increase code size.
`-P` (page 2-33)	Omits line numbers in the preprocessor output.
`-PP` (page 2-33)	Preprocesses and compiles the source file. Output does not contain `#inline` directives.
`-path-`*`tool directory`* (page 2-34)	Uses the specified directory as the location of the specified compilation tool (assembler, compiler, driver, librarian, or linker).
`-path-install` *`directory`* (page 2-34)	Uses the specified directory as the location of all compilation tools.
`-path-output` *`directory`* (page 2-34)	Specifies the location of non-temporary files.

Table 2-3. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
`-path-temp` *`directory`* (page 2-34)	Specifies the location of temporary files.
`-pch` (page 2-34)	Generates and uses precompiled header files (*.pch)
`-pchdir directory` (page 2-35)	Specifies the location of PCHRepository.
`-pedantic` (page 2-35)	Issues compiler warnings for any constructs that are not ISO/ANSI standard C/C++-compliant.
`-pedantic-errors` (page 2-35)	Issues compiler errors for any constructs that are not ISO/ANSI standard C/C++-compliant.
`-pplist` *`filename`* (page 2-35)	Outputs a raw preprocessed listing to the specified file.
`-proc` *`identifier`* (page 2-36)	Specifies that the compiler should produce code suitable for the specified DSP.
`-R` *`directory`* (page 2-36)	Appends *`directory`* to the standard search path for source files.
`-reserve` *`register`* (page 2-37)	Reserves *`register`*.
`-restrict` (page 2-37)	Enables the `restrict` keyword.
`-S` (page 2-37)	Stops compilation before running the assembler.
`-s` (page 2-37)	Removes debug info from the output executable file.
`-save-temps` (page 2-38)	Saves intermediate files.

Table 2-3. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
`-show` (page 2-38)	Displays the driver command-line information.
`-signed-char` (page 2-38)	Makes the `char` data type signed.
`-syntax-only` (page 2-38)	Checks the source code for compiler syntax errors, but does not write any output.
`-T` *`filename`* (page 2-38)	Specifies the Linker Description File.
`-threads` (page 2-39)	Specifies that support for multi-threaded applications is to be enabled.
`-time` (page 2-39)	Displays the elapsed time as part of the output information on each part of the compilation process.
`-traditional` (page 2-19)	Applies traditional C compiler rules (consistent with pre-ANSI K&R C compilers).
`-U`*`macro`* (page 2-39)	Undefines *`macro`*.
`-unsigned-char` (page 2-39)	Makes the `char` data type unsigned.
`-v` (page 2-39)	Displays both the version and command-line information.
`-verbose` (page 2-40)	Displays command-line information.
`-version` (page 2-40)	Displays version information.
`-warn-protos` (page 2-40)	Produces a warnings when a function is called without a prototype.

Table 2-3. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
`-Werror` *number* (page 2-40)	Overrides the default severity of the specified error message.
`-Wdriver-limit` *number* (page 2-40)	Halts the driver after reaching the specified number of errors.
`-Werror-limit` *number* (page 2-41)	Stops compiling after reaching the specified number of errors.
`-Wremarks` (page 2-40)	Issues compiler remarks.
`-Wterse` (page 2-40)	Issues only the briefest form of compiler warning, errors, and remarks.
`-w` (page 2-40)	Disables all warnings.
`-write-files` (page 2-41)	Enables compiler I/O redirection.
`-xref` *filename* (page 2-42)	Outputs cross-reference information to the specified file.

Table 2-4. C++ Mode Compiler Switches

Switch Name	Description
`-explicit` (page 2-43)	Supports the explicit specifier on constructor declarations. This is the default mode.
`-instant[all|used]` (page 2-43)	Instantiates all or used members of a class.
`-namespace` (page 2-43)	Supports namespaces. This is the default mode.

Table 2-4. C++ Mode Compiler Switches (Cont'd)

Switch Name	Description
`-newforinit` (page 2-43)	Limits the scope of any symbol declared within a "for" statement.
`-newvec` (page 2-44)	Allows the overloading of `new[]` and `delete[]`.
`-no-demangle` (page 2-44)	Prevents filtering of any linker errors through the demangler.
`-no-explicit` (page 2-44)	Does not support the explicit specifier on constructor declarations.
`-no-namespace` (page 2-44)	Does not support namespaces.
`-no-newvec` (page 2-44)	Does not allow the overloading of `new[]` and `delete[]`.
`-no-std` (page 2-44)	Disables the implicit use of the `std` namespace.
`-no-wchar` (page 2-45)	Disables `new wchar_t`.
`-std` (page 2-45)	Enables the implicit use of the `std` namespace.
`-strict` (page 2-45)	Generates error messages for non-ANSI constructs.
`-strictwarn` (page 2-45)	Generates warning messages for non-ANSI constructs.
`-tpautooff` (page 2-46)	Disables automatic instantiation of templates.
`-trdforinit` (page 2-46)	Limits the scope of any symbol declared within a "for" statement.

Table 2-4. C++ Mode Compiler Switches (Cont'd)

Switch Name	Description
`-typename` (page 2-46)	Recognizes the `typename` keyword. This is the default mode.
`-wchar` (page 2-46)	Enables `new wchar_t`.

C/C++ Mode Selection Switch Descriptions

-analog

The `-analog` (Analog C compilation) switch directs the compiler to support Analog Devices extensions to ANSI/ISO standard C. This is the default mode. For more information about these extensions, see "C/C++ Compiler Language Extensions" on page 2-54.

-c++

The `-c++` (C++ mode) switch directs the compiler to compile the source file(s) written in ANSI/ISO standard C++ with Analog Devices language extensions. When using this switch, source files with an extension of .c will be compiled and linked in C++ mode.

-traditional

The `-traditional` (traditional compilation) switch directs the compiler to apply the following rules (consistent with pre-ANSI K&R C compilers) to compilation:

- All `extern` declarations (including implicit declarations of functions) take effect globally
- The keywords `inline`, `signed`, `const`, and `volatile` are not recognized
- Analog Devices C/C++ language extensions are disabled except for the forms of the extra keywords that begin with a double underscore (__).
- Pointer/integer comparisons are always allowed

C/C++ Compiler Common Switch Descriptions

sourcefile

The *`sourcefile`* parameter (or parameters) specifies the name of the file (or files) to be preprocessed, compiled, assembled, and/or linked. A file name can include the drive, directory, file name, and file extension. cc21k uses the file extension to determine the operations to perform. Table 2-1 on page 2-7 lists the permitted extensions and matching compiler operations.

-@ filename

The `@` *`filename`* (command file) switch directs the compiler to read command-line input from *`filename`*.

-21020

The `-21020` (compile for ADSP-21020) switch directs the compiler to generate code suitable for the ADSP-21020. When compiling for ADSP-21020, the C/C++ preprocessor defines the `__2102x__`, `__ADSP21000__`, and `__ADSP21020__` macros. The `ADSP21000` and `ADSP21020` macros are also defined unless the `-pedantic-errors` switch is used.

-21060

The `-21060` (compile for ADSP-21060) switch directs the compiler to generate code suitable for the ADSP-21060. When compiling for ADSP-21060, the C/C++ preprocessor defines the `__2106x__`, `__ADSP21000__`, and `__ADSP21060__` macros. The `ADSP21000` and `ADSP21060` macros are also defined unless the `-pedantic-errors` switch is used.

-21061

The `-21061` (compile for ADSP-21061) switch directs the compiler to generate code suitable for the ADSP-21061. When compiling for the ADSP-21061, the C/C++ preprocessor defines the `__2106x__`, `__ADSP21000__`, and `__ADSP21061__` macros. The `ADSP21000` and `ADSP21061` macros are also defined unless the `-pedantic-errors` switch is used.

-21062

The `-21062` (compile for ADSP-21062) switch directs the compiler to generate code suitable for the ADSP-21062. When compiling for the ADSP-21062, the C/C++ preprocessor defines the `__2106x__`, `__ADSP21000__`, and `__ADSP21062__` macros. The `ADSP21000` and `ADSP21062` macros are also defined unless the `-pedantic-errors` switch is used.

-21065L

The `-21065L` (compile for ADSP-21065L) switch directs the compiler to generate code suitable for the ADSP-21065L. When compiling for the ADSP-21065L, the C/C++ preprocessor defines the `__2106x__`, `__ADSP21000__`, and `__ADSP21065L__` macros. The `ADSP21000` and `ADSP21065L` macros are also defined unless the `-pedantic-errors` switch is used.

-21160

The `-21160` (compile for ADSP-21160) switch directs the compiler to generate code suitable for the ADSP-21160. When compiling for the ADSP-21160, the C/C++ preprocessor defines `__2116x__`, `__ADSP21000__`, and `__ADSP21160__` macros. The `ADSP21000` and `ADSP21160` macros are also defined unless the `-pedantic-errors` switch is used.

-21161

The `-21161` (compile for ADSP-21161) switch directs the compiler to generate code suitable for the ADSP-21161. When compiling for the ADSP-21161, the C/C++ preprocessor defines the `__2116x__`, `__ADSP21000__`, and `__ADSP21161__` macros. The `ADSP21000` and `ADSP21161` macros are also defined unless the `-pedantic-errors` switch is used.

-A *name*[*tokens*]

The `-A` (assert) switch directs the compiler to assert *`name`* as a predicate with the specified *`tokens`*. This has the same effect as the `#assert` preprocessor directive.The following assertions are predefined:

```
system()   machine()  cpu()   compiler()

embedded   adsp21xxx
```

-aligned-stack

The `-aligned-stack` (align stack) switch directs the compiler to align the program stack on a double-word boundary.

-alttok

The `-alttok` (alternative tokens) switch directs the compiler to allow alternative operator keywords and digraph sequences in source files. This is the default mode.

ANSI C trigraphs sequences are always expanded (even with the `-no-alttok` option), and only digraph sequences are expanded in C source files.

The following operator keywords are enabled by default:

Keyword	Equivalent
`and`	`&&`
`and_eq`	`&=`
`bitand`	`&`
`bitor`	`|`
`compl`	`~`
`or`	`||`
`or_eq`	`|=`
`not`	`!`
`not_eq`	`!=`
`xor`	`^`
`xor_eq`	`^=`

-auto-inline *factor*

The `-auto-inline` (auto inline) switch directs the optimizer to automatically inline functions where the reduction in execution time justifies the increase in code size. The amount of inlining that is effected is specified by *factor,* which is a floating point number that determines how aggressively functions are inlined. The amount of inlining effected is shown by the following examples:

0.0	Reject all inlining.
1.0	Inline if code size increase does not exceed speed improvement.
10.0	Inline allowing a reasonable amount of code size increase.
1000.0	Inline practically everything

-build-lib

The `-build-lib` (build library) switch directs the compiler to use the librarian to produce a library file (`.dlb`) as the output instead of using the linker to produce an executable file (`.dxe`). The `-o` option must be used to specify the name of the resulting library.

-C

The `-C` (comments) switch, which may only be run in combination with the `-E` or `-P` switches, directs the C/C++ preprocessor to retain comments in its output file.

-c

The `-c` (compile only) switch directs the compiler to compile and/or assemble the source files, but stop before linking. The output is an object file (`.doj`) for each source file.

-const-read-write

The cc21k compiler's default behavior assumes that data referenced through const pointers will never change. The `-const-read-write` switch changes the cc21k's behavior to match the ANSI C assumption, which is that other non-const pointers may be used to change the data at some point.

-D*macro*[=***definition***]

The `-D` (define macro) switch directs the compiler to define a macro. If you do not include the optional definition string, the compiler defines the macro as the string '1'. Note that the compiler processes all `-D` switches on the command line before any `-U` (undefine macro) switches.

-default-linkage[**-asm | -c | -c++**]

The `-default-linkage-asm` (assembler linkage), `-default-linkage-c` (C linkage), and `-default-linkage-c++` (C++ linkage) directs the compiler to set the default linkage type. C linkage is the default type in C mode, and C++ linkage is the default type in C++ mode.

This switch can be invoked with the `Definitions:` dialog field located in the IDDE's `Compile` dialog, `Preprocessor` selection.

-double-size[**-32|-64**]

The `-double-size-32` (double is 32 bits) and the `-double-size-64` (double is 64 bits) switches select the storage format that the compiler uses for type `double`. The default mode is `-double-size-32`.

The C/C++ type `double` poses a special problem for the compiler. The C and C++ languages default to `double` for floating-point constants and many floating-point calculations. If `double` has the customary size of 64 bits, many programs will inadvertently use slow speed emulated 64-bit floating-point arithmetic, even when variables are declared consistently as `float`.

To avoid this problem, cc21k provides a mode in which `double` is the same size as `float`. This mode is enabled with the `-double-size-32` switch and is the default mode.

Representing `double` using 32 bits gives good performance and provides enough precision for most DSP applications. This, however, does not fully conform to the C and C++ standards. The standard requires that `double` maintains 10 digits of precision, which requires 64 bits of storage. The `-double-size-64` switch sets the size of `double` to 64 bits for full standard conformance.

With `-double-size-32`, a `double` is stored in 32-bit IEEE single-precision format and is operated on using fast hardware floating-point instructions. Standard math functions such as `sin` also operate on 32-bit values. This mode is the default and is recommended for most programs. Calculations that need higher precision can be done with the `long double` type, which is always 64 bits.

With `-double-size-64`, a `double` is stored in 64-bit IEEE single precision format and is operated on using slow floating-point emulation software. Standard math functions such as `sin` also operate on 64-bit values and are similarly slow. This mode is recommended only for porting code that requires that `double` have more than 32 bits of precision.

The `-double-size-32` switch defines the `__DOUBLES_ARE_FLOATS__` macro, while the `-double-size-64` switch undefines the `__DOUBLES_ARE_FLOATS__` macro.

-dry

The `-dry` (verbose dry-run) switch directs the compiler to display main driver actions, but not to perform them.

-dryrun

The `-dryrun` (terse dry-run) switch directs the compiler to display top-level driver actions, but not to perform them.

-E

The `-E` (stop after preprocessing) switch directs the compiler to stop after the C/C++ preprocessor runs (without compiling). The output, preprocessed source code, prints to the standard output stream unless the output file is specified with `-o`. Note that the `-C` switch can only be run in combination with the `-E` switch.

This switch can be invoked with the `Stop after: Preprocessor` check box located in the IDDE's `Compile` dialog box, `General` selection.

-EE

The `-EE` (run after preprocessing) switch is similar to the `-E` switch, but it does not halt compilation after preprocessing.

-extra-keywords

The `-extra-keywords` (enable short-form keywords) switch directs the compiler to recognize the Analog Devices keyword extensions to ANSI/ISO standard C and C++, such as `pm` and `dm`, without leading underscores, which can affect conforming ANSI/ISO C and C++ programs. This is the default mode.

-flags[**-asm | -compiler | -gco | -lib | -link | -mem | -prelink**]*--switch* [, *argument* [, ...]]

The `-flags` (command-line input) switch directs the compiler to pass command-line switches to the other build tools.

Table 2-5. Build Tools `-flags` Options

Option	Tool
-flags-asm	Assembler
-flags-compiler	Compiler
-flags-lib	Library builder
-flags-prelink	Prelinker
-flags-link	Linker
-flags-mem	Memory initializer

-full-version

The `-version` (display version) switch directs the compiler to display version information for all the compilation tools as they process each file.

-g

The `-g` (generate debug information) switch directs the compiler to output symbols and other information used by the debugger. When `-g` is used without `-O`, then the `-no-inline` option is also implied. If the `-g` switch is used in conjunction with the enable optimization (`-O`) switch, the compiler performs standard optimizations. It also outputs symbols and other information to provide limited source level debugging through the VisualDSP++ debugger. The debugging capability enabled by this combination of options is line debugging and global variable debugging.

When `-g` and `-O` are specified, no debug information is available for local variables and the standard optimizations can sometimes re-arrange program code in a way that inaccurate line number information may be produced. For full debugging capabilities, use the `-g` switch without the `-O` switch.

This switch can be invoked with the `Generate debug information` check box located in the IDDE's `Compile` dialog, `General` selection.

-H

The `-H` (list headers) switch directs the compiler to output only a list of the files included by the preprocessor via the `#include` directive, without compiling.

-HH

The `-HH` (list headers and compile) switch directs the compiler to output to the standard output stream a list of the files included by the preprocessor via the `#include` directive. After preprocessing, compilation proceeds normally.

-h[elp]

The `-help` (command-line help) switch directs the compiler to output a list of command-line switches with a brief syntax description.

-I ***directory***[{;|,}***directory…***]

The `-I` (include search directory) switch directs the C/C++ preprocessor to append the directory to the search path for include files. This option may be specified more than once; all specified directories are added to the search path.

-include ***filename***

The `-include` (include file) switch directs the preprocessor to process the specified file before processing the regular input file. Any `-D` and `-U` options on the command line are processed before an `-include` switch.

-ipa

The `-ipa` (interprocedural analysis) switch turns on Interprocedural Analysis (IPA) in the compiler. This option enables optimization across the entire program, including between source files that were compiled separately. The `-ipa` option should be applied to all C anc C++ files in the program. For more information, see "Interprocedural Analysis" on page 2-51. Specifying `-ipa` also implies `-O`.

-L ***directory***[{;|,}***directory…***]

The `-L` (library search directory) switch directs the linker to append the directory to the search path for library files.

-l ***library***

The `-l` (link library) switch directs the linker to search the library for functions and global variables when linking. The library name is the portion of the file name between the `lib` prefix and `.dlb` extension.

For example, the `-lc` compiler switch directs the linker to search in the library named `c`. This library resides in a file named `libc.dlb`.

Normally, you should list all object files on the command line before using the `-l` switch; this ensures that functions referred by object files are loaded from the library in the given order. This option may be specified more than once; libraries are searched as encountered during the left-to-right processing of the command line.

-M

The `-M` (generate make rules only) switch directs the compiler not to compile the source file, but to output a rule suitable for the make utility, describing the dependencies of the main program file. The format of the make rule output by the preprocessor is:

```
object-file: include-file …
```

-MM

The `-MM` (generate make rules and compile) switch directs the preprocessor to print to standard out a rule describing the dependencies of the main program file. After preprocessing, compilation proceeds normally.

-map *filename*

The `-map` (generate a memory map) switch directs the linker to output a memory map of all symbols. The map file name corresponds to the *`filename`* argument. For example, if the argument is `test`, the map file name is `test.map`. The `.map` extension is added where necessary.

-mem

The `-mem` (enable memory initialization) switch directs the compiler to run the `mem21k` initializer.

-no-aligned-stack

The `no-aligned-stack` (disable stack alignment) switch directs the compiler to not align the program stack on a double-word boundary.

-no-alttok

The `-no-alttok` (disable alternative tokens) switch directs the compiler not to accept alternative operator keywords and digraph sequences in the source files. For more information, see "-alttok" on page 2-22.

-no-builtin

The `-no-builtin` (no built-in functions) switch directs the compiler to ignore built-in functions that begin with two underscores (__). Note that this switch influences many functions. For more information on built-in functions, see "Linking Library Functions" on page 3-4.

-no-defs

The `-no-defs` (disable defaults) switch directs the preprocessor not to define any default preprocessor macros, include directories, library directories, libraries, and run-time headers. It also disables the Analog Devices cc21k C/C++ keyword extensions.

-no-extra-keywords

The `-no-extra-keywords` (disable short-form keywords) switch directs the compiler not to recognize the Analog Devices keyword extensions that might affect conformance to ISO/ANSI standards for C and C++ languages. These include keywords such as `pm` and `dm`, which may be used as identifiers in standard conforming programs. Alternate keywords, which are prefixed with two leading underscores such as `__pm` and `__dm`, continue to work.

-no-inline

The `-no-inline` (disable inline keyword) switch directs the compiler not to perform any high-level optimizations associated with function inlining.

-no-mem

The `-no-mem` (disable memory initialization) switch directs the compiler not to run the `mem21k` initializer. Note that if you use `-no-mem`, the compiler does not initialize globals and statics.

-no-restrict

The `-no-restrict` (disable restrict) switch directs the compiler to disable recognition of the `restrict` keyword as a type qualifier for pointers and array parameters to functions.

-no-std-def

The `-no-std-def` (disable standard macro definitions) switch prevents the compiler from defining default preprocessor macro definitions. Note that this switch also disables the Analog Devices keyword extensions that have no leading underscores, such as `pm` and `dm`.

-no-std-inc

The `-no-std-inc` (disable standard include search) switch directs the C/C++ preprocessor to search for header files in the current directory and directories specified with the `-I` switch.

This switch can be invoked with the `Ignore standard include paths` check box located in the IDDE's `Compile` dialog box, `Preprocessor` selection.

-no-std-lib

The `-no-std-lib` (disable standard library search) switch directs the linker to search for libraries in only the current project directory and directories specified with the `-L` switch.

-nothreads

The `-nothreads` (disable thread-safe build) switch specifies that all compiled code and libraries used in the build need not be thread-safe. This is the default setting.

-O

The `-O` (enable optimizations) switch directs the compiler to produce code that is optimized for performance. Optimizations are not enabled by default for the cc21k compiler.

This switch can be invoked with the `Enable optimization` check box located in the IDDE's `Compile` dialog, `General` selection.

-Os

The `-Os` (optimize for size) switch directs the compiler to produce code that is optimized for size. This is achieved by performing all optimizations except those that increase code size. When this option is used, the compiler disables loop unrolling, delay slot filling, and jump avoidance.

-o *filename*

The `-o` (output file) switch directs the compiler to use *`filename`* for the name of the final output file.

-P

The `-P` (omit line numbers) switch directs the compiler to stop after the C/C++ preprocessor runs (without compiling) and to omit the `#line` preprocessor command with line number information from the preprocessor output. The `-C` switch can be used in conjunction with `-P` to retain comments.

-PP

The `-PP` (omit line numbers and compile) switch is similar to the `-P` switch; however, it does not halt compilation after preprocessing.

-path [**-asm | -compiler | -lib | -link | - mem | -prelink**] ***directory***

The `-path` (tool location) switch directs the compiler to use the specified directory as the location of the specified compilation tool. Respectively, the tools are the assembler, compiler, librarian, linker, memory initializer, and prelinker. Use this switch when one or more of the tools is in a directory other than the directory that you name with the `-path-install` switch.

-path-install ***directory***

The `-path-install` (installation location) switch directs the compiler to use the specified directory as the location for all compilation tools instead of the default path. This is useful when working with multiple versions of the tool set.

You can selectively override this switch with the `-path-tool` switch.

-path-output ***directory***

The `-path-output` (non-temporary files location) switch directs the compiler to place final output files in the specified directory.

-path-temp ***directory***

The `-path-temp` (temporary files location) switch directs the compiler to place temporary files in the specified directory.

-pch

The `-pch` (precompiled header) switch directs the compiler to automatically generate and use precompiled header files. A precompiled output header has a `.pch` extension attached to the source file name. By default, all precompiled headers are stored in a directory called `PCHRepository`.

Precompiled header files can significantly speed compilation; precompiled headers tend to occupy more disk space.

-pchdir ***directory***

The `-pchdir` (locate PCHRepository) switch specifies the location of an alternative `PCHRepository` for storing and invocation of precompiled header files. If the directory does not exist, the compiler creates it. Note that `-o` (output) does not influence the `-pchdir` option.

-pedantic

The `-pedantic` (ANSI standard warnings) switch causes the compiler to issue a warning for each construct found in your program that does not strictly conform to ANSI/ISO standard C or C++. Note that the compiler may not detect all such constructs. In particular, the `-pedantic` switch does not cause the compiler to issue errors when Analog Devices keyword extensions are used.

-pedantic-errors

The `-pedantic-errors` (ANSI standard errors) switch causes the compiler to issue errors instead of warnings for cases described in the `-pedantic` switch.

-pplist ***filename***

The `-pplist` (preprocessor listing) directs the preprocessor to output a listing to the named file. When more than one source file has been preprocessed, the listing file contains information about the last file processed. The generated file contains raw source lines, information on transitions into and out of include files, and diagnostics generated by the compiler. Each listing line begins with a key character that identifies its type, as follows:

Character	Meaning
`N`	Normal line of source
`X`	Expanded line of source

Character	Meaning
S	Line of source skipped by #if or #ifdef
L	Change in source position
R	Diagnostic message (remark)
W	Diagnostic message (warning)
E	Diagnostic message (error)
C	Diagnostic message (catastrophic error)

-proc ***identifier***

The -proc (target processor) switch specifies that the compiler should produce code suitable for the identified DSP. If the processor identifier is unknown to the compiler it will attempt to read required switches for code generation from file <identifier>.ini. It will search for this file in the VisualDSP System folder.

-R directory[{**:**|**,**}***directory*** **…**]

The -R (add source directory) switch directs the compiler to add the specified directory to the list of directories searched for source files.

Multiple source directories are given as a colon-, comma-, or semicolon-separated (on Windows platforms) list. The compiler searches for the source files in the order specified on the command line. The compiler searches the specified directories before reverting to the current project directory. This option is position-dependent on the command line. That is, it affects only source files that follow the option.

Source files whose file names begin with /, ./, or ../ (or Windows equivalent) and contain drive specifiers (on Windows platforms) are not affected by this option

-reserve *register*[, *register* ...]

The `-reserve` (reserve register) switch directs the compiler not to use the specified registers. This guarantees that a known set of registers are available for inline assembly code or linked assembly modules. Separate each register name with a comma on the compiler command line. You can reserve the following registers: `b0`, `l0`, `m0`, `i0`, `b1`, `l1`, `m1`, `i1`, `b8`, `l8`, `m8`, `i8`, `b9`, `l9`, `m9`, `i9`, `ustat1`, and `ustat2`. When reserving an `L` (length) register, you must reserve the corresponding `I` (index) register; reserving an `L` register without reserving the corresponding `I` register may result in execution problems.

-restrict

The `-restrict` (restrict) switch directs the compiler to recognize the `restrict` keyword as a type qualifier for pointers and function parameter arrays that decay to pointers. This is the default setting.

-S

The `-S` (stop after compilation) switch directs cc21k to stop compilation before running the assembler. The compiler outputs an assembly file with a `.s` extension.

This switch can be invoked with the `Stop after: Compiler` check box located in the IDDE's `Compile` dialog box, `General` selection

-s

The -s (strip debug information) switch directs the compiler to remove debug information (symbol table and other items) from the output executable file during linking.

-save-temps

The `-save-temps`(save intermediate files) switch directs the compiler not to discard intermediate files. The compiler places the intermediate output (`*.i`, `*.is`, `*.s`, `*.doj`) files in the temp subdirectory of the current project directory. See Table 2-1 on page 2-7 for a list of intermediate files.

-show

The `-show` (display command line) switch directs the compiler to display the command-line arguments passed to the driver, including expanded option files and environment variables. This option allows you to ensure that command-line options have been successfully invoked by the driver.

-signed-char

The `-signed-char` (make char signed) switch directs the compiler to make the default type for `char` signed. The compiler also defines the `__SIGNED_CHARS__` preprocessor macro. This is the default mode.

-syntax-only

The `-syntax-only` (check syntax only) switch directs the compiler to check the source code for syntax errors but not to write any output.

-T *filename*

The `-T` (Linker Description File) switch directs the linker to use the specified Linker Description File (`.LDF`) as control input for linking. If `-T` is not specified, a default LDF is selected based on the processor variant.

-threads

The `-threads` (enable thread-safe build) specifies that the build and link should be thread-safe. The macro `_ADI_THREADS` is defined to one (1). It is used for conditional compilation by the preprocessor and by the default LDF files to link with thread-safe libraries.

This switch is only likely to be used by applications involving VDK.

-time

The `-time` (tell time) switch directs the compiler to display the elapsed time as part of the output information about each phase of the compilation process.

-Umacro

The `-U` (undefine macro) switch lets you undefine macros. Note that the compiler processes all `-D` (define macro) switches on the command line before any `-U` (undefine macro) switches.

This switch can be invoked with the `Undefines` dialog field located in the IDDE's `Compile` dialog box, `Preprocessor` selection.

-unsigned-char

The `-unsigned-char` (make char unsigned) switch directs the compiler to make the default type for `char` unsigned. The compiler also undefines the `__SIGNED_CHARS__` macro.

-v

The `-v` (version and verbose) switch directs the compiler to display both the version and command-line information for all the compilation tools as they process each file.

-verbose

The `-verbose` (display command line) switch directs the compiler to display command-line information for all the compilation tools as they process each file.

-version

The `-version` (display version) switch directs the compiler to display version information for all the compilation tools as they process each file.

-warn-protos

The `-warn-protos` (warn if incomplete prototype) switch directs the compiler to issue a warning when it calls a function for which an incomplete function prototype has been supplied. This option has no effect in C++ mode.

-W[error|remark|suppress|warn] ***number*****[,*****number*** **...]**

The `-W` (override error message) witch directs the compiler to override the severity of the specified diagnostic messages (errors, remarks, or warnings). The *`number`* argument specifies the message to override.

The number of a specific compiler diagnostic message is given at compilation time. The `-D` (discretionary) suffix attached to the message number marks the message whose severity can be overridden. The message representation numbers are constant between the compiler software releases.

-Wdriver-limit ***number***

The `-Wdriver-limit` (maximum process errors) switch sets a maximum number of driver errors (command-line, etc.) at which the driver aborts.

-Werror-limit *number*

The `-Werror-limit` (maximum compiler errors) switch sets a maximum number of errors for the compiler.

-Wremarks

The `-Wremarks` (enable diagnostic warnings) switch directs the compiler to issue remarks, which are diagnostic messages milder than warnings.

This switch can be invoked with the `Enable remarks` check box located in the IDDE's `Compile` dialog, `Warning` selection.

-Wterse

The `-Wterse` (enable terse warnings) switch directs the compiler to issue the briefest form of warnings. This also applies to errors and remarks.

-w

The `-w` (disable all warnings) switch directs the compiler not to issue warnings.

This switch can be invoked with the `Disable all warnings and remarks` check box located in the IDDE's `Compile` dialog box, `Warning` selection.

-write-files

The `-write-files` (enable driver I/O redirection) switch directs the compiler driver to redirect the file name portions of its command line through a temporary file. This technique helps to handle long file names, which can make the compiler driver's command line too long for some operating systems.

-xref *filename*

The `-xref` (cross-reference list) switch directs the compiler to write cross-reference listing information to the specified file. When more than one source file has been compiled, the listing contains information about the last file processed.

For each reference to a symbol in the source program, a line of the form:

```
symbol-id name ref-code filename line-number column-number
```

is written to the named file.

`symbol-id` represents a unique decimal number for the symbol, and `ref-code` is a character from one the following:

Character	Meaning
D	Definition
d	Declaration
M	Modification
A	Address taken
U	Used
C	Changed (used and modified)
R	Any other type of reference
E	Error (unknown type of reference)

C++ Mode Compiler Switch Descriptions

The following switches apply only to C++.

-explicit

The `-explicit` (explicit specifier) switch directs the compiler to enable support for the `explicit` specifier on constructor declarations. The compiler defines the `__EXPLICIT` preprocessor macro. This option is enabled by default.

-instant[**all | used**]

The default behavior that the compiler uses to perform template instantiation is to suppress the instantiation of any templates on the first compilation and let the prelinker decide which files need to be recompiled to instantiate the required templates. However, the`-instantused` switch automatically instantiates any template entities that are used in the first compilation and the `-instantall` switch automatically instantiates all template entities whether they are used or not. Both of these options can still be used in combination with the prelinker.

-namespace

The `-namespace` (namespace) switch directs the compiler to enables support for namespaces.

-newforinit

The `-newforinit` (new 'for' initialization) switch directs the compiler to limit a scope of any declaration within a 'for' statement to the block contained within that 'for' statement.

-newvec

The `-newvec` (new vector) switch directs the compiler to allow the overloading of the `new[]` and `delete[]` operators. The compiler also defines the `__ARRAY_OPERATORS` macro when this option, or another option that enables overloading of the dynamic memory allocation operators, is used. This is the default mode.

-no-demangle

The `-no-demangle` (disable demangler) switch directs the compiler to prevent the driver from filtering any linker errors through the demangler. The demangler's primary role is to convert the encoded name of a function into a more understandable version of the name.

-no-explicit

The `-no-explicit` (disable explicit specifier) switch directs the compiler to disable support for the explicit specifier on constructor declarations. For more information, see "-explicit" on page 2-43.

-no-namespace

The `-no-namespace` (disable namespace) switch directs the compiler to disable support for namespaces.

-no-newvec

The `-no-newvec` (disallow a new vector) switch directs the compiler to disallow the overloading of the `new[]` and `delete[]` operators. For more information, see "-newvec" on page 2-44.

-no-std

The `-no-std` (disable std namespace) switch directs the compiler to disable the implicit use of the `std` namespace when the standard header files are included. For more information, see "-std" on page 2-45.

-notstrict

The `-notstrict` (non-strict compilation) switch directs the compiler to omit diagnostic messages (warnings and errors) for any constructs in a C++ source file that do not conform to the ANSI standard for the C++ programming language.

-no-wchar

The `-no-wchar` (disable wide char type) switch directs the compiler to disable the `new wchar_t` construct.

-std

The `-std` (std namespace) switch directs the compiler to enable the implicit use of the `std` namespace when a standard header file is included. It also allows the inclusion of a standard header with an `.h` extension. The `-std` switch automatically enables the `-namespace` option. Note that `-std` is disabled by default.

-strict

The `-strict` (strict standard) switch directs the compiler to generate diagnostic error messages for any constructs of a source file that do not conform to the ANSI standard for the C++ programming language. Both `-strict` and `-ansi` define the `__STRICT_ANSI__` macro.

-strictwarn

The `-strictwarn` (warn if non-strict) switch directs the compiler to generate diagnostic warning messages for any constructs of a source file that do not conform to the ANSI standard for the C++ programming language. Both `-strictwarn` and `-ansi` define the `__STRICT_ANSI__` macro.

-tpautooff

The `-tpautooff` (disable automatic template instantiation) switch directs the compiler to disable automatic instantiation of templates and prevents implicit inclusion of source files as a method of finding definitions of template entities to be instantiated.

-trdforinit

The `-trdforinit` (traditional initialization) switch directs the compiler to limit a scope of any declaration within a 'for' statement to the block containing that 'for' statement.

-typename

The `-typename` (type name) switch directs the compiler to recognize the typename keyword and to define the `__TYPENAME` macro. This is the default mode.

-wchar

The `-wchar` (enable wide char type) switch directs the compiler to enable the `new wchar_t` construct.

Data Type Sizes

The sizes of intrinsic C/C++ data types are selected by Analog Devices so that normal C/C++ programs execute with hardware-native data types and therefore at high speed. Table 2-6 shows the size used for each of the intrinsic C/C++ data types.

Table 2-6. Data Type Sizes for the ADSP-21xxx DSPs

Type	Bit Size
int	32 bits signed
unsigned int	32 bits unsigned
long	32 bits signed
unsigned long	32 bits unsigned
char	32 bits signed
unsigned char	32 bits unsigned
short	32 bits signed
unsigned short	32 bits unsigned
pointer	32 bits
float	32 bits float
double	either 32 or 64 bits float (default 32)
long double	64 bits float
fract	32 bits fixed-point

Analog Devices does not support data sizes smaller than a single word location for a processor. For the ADSP-21xxx processors, this means that both `short` and `char` have the same size as `int`. Although 32-bit `chars` are unusual, they do conform to the standard. For information about the fract data type, refer to "C++ Fractional Type Support" on page 2-84.

Integer

On any platform the basic type `int` will be the native word size; on ADSP-21xxx DSPs, it is 32 bits. Many library functions are available for 32-bit integers, and these functions provide support for the C/C++ data types `int` and `long int`. Pointers are the same size as `int`s. The `long long int` data type is not supported.

Floating Point

On ADSP-21xxx DSPs, the `long` data type is 32 bits, as is `float`; `double` is option-selectable for 32- or 64-bits. The C/C++ language tends to default to `double` for constants and for many floating-point calculations. In general, `double` word data types run more slowly than 32-bit data types because they rely largely on software-emulated arithmetic.

Type `double` poses a special problem. Without some special handling, many programs would inadvertently end up using slow-speed, emulated, 64-bit floating point arithmetic, even when variables are declared consistently as `float`. In order to avoid this problem, Analog provides an option (switch) control: the size of `double` may be set to either 32- (default) or 64-bits. The 32-bit setting gives good performance and should be acceptable for most DSP programming. However, it does not conform fully to the ANSI C standard. For a larger floating-point type, `long double` provides 64-bit floating point.

For either size of `double`, the standard `#include` files automatically redefine the math library interfaces so that functions such as `sin` can be directly called with the proper size operands. Access to 64-bit floating point arithmetic and libraries is always provided via `long double`. Therefore:

```
float sinf (float);                /* 32-bit */
double sin (double);               /* 32 or 64-bit */
```

For full descriptions of these functions and their implementation, see "C/C++ Run-Time Library" on page 3-1.

Optimization Control

The cc21k compiler can operate at several different levels of optimization. The following list identifies these levels with least optimization listed first and most optimization listed last:

- **Debugging**. The compiler produces debug information to ensure that the object code can be matched to the appropriate source code line. See "-g" on page 2-27 for more information.
- **Default**. The compiler does basic high-level optimization, such as inlining functions that are explicitly marked for inlining.
- **Procedural optimization**. The compiler does advanced, aggressive optimization on each procedure in the file being compiled. If debugging is also requested, the optimization is given priority so that debugging functionality may be limited. See "-O" on page 2-33 for more information.
- **Interprocedural optimization**. The compiler does advanced, aggressive optimization over the whole program in addition to the per-file optimizations in procedural optimization. See "-ipa" on page 2-29 for more information.

Interprocedural analysis (see "Interprocedural Analysis" on page 2-51) allows the compiler to see all of the source files that are used and to use that information to enable the other optimizations to be exploited as fully as possible.

When no optimization switches are specified, cc21k effects only basic high level optimizations, such as inlining functions, which have been explicitly marked for inlining. When `-g` is specified, however, all inlining is suppressed to provide as comprehensive debugging information as possible. When `-inline` is specified with `-g`, then explicitly specified inlining is provided, which reduces the amount of source line debug information that is available. Therefore, the use of `-g` by itself effectively disables almost all optimizations.

Normally, a program is optimized to process the data as quickly as possible, but in some circumstances, the speed of the program may be less important than reducing the size of the generated code. When the `-Os` switch is specified, the compiler will only perform standard optimizations that do not significantly increase the size of the generated code. (See "-Os" on page 2-33 for more information.)

The `-O` switch requests the compiler to effect all normally safe optimizations. It also requests the compiler to generate the fastest possible executing code while conforming to standard language interpretations and a conservative view of any possible interactions between variables. The use of interprocedural analysis can be very useful in enabling the compiler to be more aggressive in optimizing the program since it has much greater knowledge of the overall structure of the program and the data being manipulated by the program.

For the ADSP-2116x DSPs, the optimizer always attempts to vectorize loops when it is safe to do so and uses information from the Interprocedural Analyzer to identify more opportunities to do so. In addition, there may be other loops that you know are safe candidates for the vectorizer; you can use pragmas to inform the optimizer of such loops (see "SIMD Support Annotation (#pragma SIMD_for)" on page 2-88).

Inlining Control

By default, cc21k inlines class members and those functions that are explicitly marked to be inlined. When the `-no-inline` switch is specified, then any explicit request for inlining is ignored.

When the `-O` switch has been specified or implied, then the optimizer also inlines some additional functions in cases where the reduction in execution time justifies the increase in code size.

Interprocedural Analysis

The cc21k compiler has a capability called *interprocedural analysis* (IPA), an optimization that allows the compiler to optimize across translation units instead of within just one translation unit. This capability effectively allows the compiler to see all of the source files that are used in a final link at compilation time and make use of that information when optimizing.

Interprocedural analysis is enabled by selecting the `Interprocedural analysis` option in the `Project Options` dialog box, on the `Compiler` tab in VisualDSP++, or by specifying the `-ipa` command-line switch.

The `-ipa` switch automatically enables the `-O` switch to turn on optimization. However, all object files that are supplied in the final link must have been compiled with the `-ipa` switch; otherwise, undefined behavior may result.

Use of the `-ipa` switch causes additional files to be generated along with the object file produced by the compiler. These files have `.ipa` and `.opa` filename extensions and should not be deleted manually unless the associated object file is also deleted. All of the `-ipa` optimizations are invoked after the initial link, whereupon a special program called the prelinker reinvokes the compiler to perform the new optimizations.

Because a file may be recompiled by the prelinker, you cannot use the `-S` option to see the final optimized assembler file when `-ipa` is enabled. Instead, you must use the `-save-temps` switch, so that the full compile/link cycle can be performed first.

Interaction with Libraries

When IPA is enabled, the compiler examines all of the source files to build up usage information about all of the function and data items. It then uses that information to make additional optimizations across all of the source files. One of these optimizations is to remove functions that are never called. This optimization can significantly reduce the overall size of the final executable.

Because IPA operates only during the final link, the `-ipa` switch has no benefit when initially compiling source files to object format for inclusion in a library. Although IPA generates usage information for potential additional optimizations at the final link stage, as normal, neither the usage information nor the module's source file are available when the linker includes a module from a library. Each library module has been compiled to the normal `-O` optimization level, but the prelinker cannot access the previously-generated additional usage information for an object in a library. Therefore, IPA cannot exploit the additional information associated with a library module.

If a library module has to make calls to a function in a user module in the program, IPA must be told that this call may occur. The reason IPA needs to know about the call is because IPA examines all the visible calls to the function and determines how best to optimize it based on that information. However, it cannot “see” the calls to the function from the library because the library code has no associated usage information to show that it uses the function.

There is a pragma, `retain_name`, that tells IPA that there are calls that it cannot see, as shown in the following example:

```
int delete_me(int x) {
       return x-2;
}

#pragma retain_name("keep_me")
int keep_me(int y) {
       return y+2;
}

int main(void) {
       return 0;
}
```

When this program is compiled and linked with the `-ipa` switch, IPA can see that there are no calls to `delete_me()` in any of the source files (one source file, in this case); therefore, IPA deletes it since it is unnecessary. IPA does not delete `keep_me()` because the `retain_name` pragma tells IPA that there are uses of the function not visible to IPA. No pragma is necessary for `main()` because IPA knows this is the entry-point to the program.

IPA assumes that it can see all calls to a function and makes use of its knowledge of the parameters being passed to a function to effectively tailor the code generated for a function. If there are calls on a function from an object module in a library, then IPA will not have access to the information for that invocation of the function; this may cause it to incorrectly optimize the generated code.

C/C++ Compiler Language Extensions

The compiler supports a set of extensions to the ANSI standard for the C and C++ languages. These extensions add support for DSP hardware and allow some C++ programming features when compiling in C mode. The extensions are also available when compiling in C++ mode.

The additional keywords that are part of the C/C++ extensions do not conflict with any ANSI C/C++ keywords. The formal definitions of these extension keywords are prefixed with a leading double underscore (__). Unless the `-no-extra-keywords` command-line switch is used, the compiler defines the shorter form of the keyword extension that omits the leading underscores. For more information, see "C++ Mode Compiler Switch Descriptions" on page 2-43.

This section describes only the shorter forms of the keyword extensions, but in most cases you can use either form in your code. For example, all references to the `inline` keyword in this text appear without the leading double underscores, but you can interchange `inline` and `__inline` in your code.

You might need to use the longer form (such as `__inline`) exclusively if you are porting a program that uses the extra Analog Devices keywords as identifiers. For example, a program might declare local variables, such as `pm` or `dm`. In this case, use the `-no-extra-keywords` switch, and if you need to declare a function as inline, or allocate variables to memory spaces, you can use `__inline` or `__pm`/`__dm` respectively.

This section provides an overview of the extensions, with brief descriptions, and directs you to text with more information on each extension.

Table 2-7 provides a brief description of each keyword extension and directs you to sections of this chapter that document the extensions in more detail. Table 2-8 provides a brief description of each operational extension and directs you to sections that document these extensions in more detail.

Table 2-7. Keyword Extensions

Keyword extensions	Description
`inline(function)`	Directs the compiler to integrate the function code into the code of the callers. For more information, see "Inline Function Support Keyword (inline)" on page 2-57.
`asm()`	Places ADSP-21xxx family assembly language instructions directly in your C/C++ program. For more information, see "Inline Assembly Language Support Keyword (asm)" on page 2-58.
`dm`	Specifies the location of a static or global variable or qualifies a pointer declaration "*" as referring to Data Memory (DM). For more information, see "Dual Memory Support Keywords (pm dm)" on page 2-68.
`pm`	Specifies the location of a static or global variable or qualifies a pointer declaration "*" as referring to Program Memory (PM). For more information, see "Dual Memory Support Keywords (pm dm)" on page 2-68.
`section("string")`	Specifies the section in which an object or function is placed. The section keyword has replaced the segment keyword of the previous releases of the compiler software. For more information, see "Placement Support Keyword (section)" on page 2-73.
`bool, true, false`	A boolean type. For more information, see "Boolean Type Support Keywords (bool, true, false)" on page 2-74.
`restrict` keyword	Specifies restricted pointer features. For more information, see "Pointer Class Support Keyword (restrict)" on page 2-74.

Table 2-8. Operational Extensions

Operation extensions	Description
Variable-length arrays	Support for variable-length arrays lets you use arrays whose length is not known until run time. For more information, see "Variable-Length Array Support" on page 2-75.
Non-constant initializers	Support for non-constant initializers lets you use non-constants as elements of aggregate initializers for automatic variables. For more information, see "Non-Constant Initializer Support" on page 2-77.
Indexed initializers	Support for indexed initializers lets you specify elements of an aggregate initializer in an arbitrary order. For more information, see "Indexed Initializer Support" on page 2-77.
Aggregate constructor expressions	Support for aggregate assignments lets you create an aggregate array or structure value from component values within an expression. For more information, see "Aggregate Constructor Expression Support" on page 2-79.
`fract` data type (C++ mode)	Support for the fractional data type, fractional and saturated arithmetic. For more information, see "C++ Fractional Type Support" on page 2-84.
Preprocessor generated warnings	Lets you generate warning messages from the preprocessor.For more information, see "Preprocessor Generated Warnings" on page 2-80.
C++ style comments	Allows for "//" C++ style comments in C programs. For more information, see "C++ Style Comments" on page 2-80.

Inline Function Support Keyword (inline)

The cc21k `inline` keyword directs cc21k to integrate the code for the function you declare as `inline` into the code of its callers. Using this keyword eliminates the function-call overhead and therefore can increase the speed of your program's execution. Argument values that are constant and that have known values may permit simplifications at compile time. The example that follows shows a function definition that uses the `inline` keyword:

```
inline int add_one (int *a)
{
 (*a)++;
}
```

A function declared `inline` must be defined (its body must be included) in every file in which the function is used. The normal way to do this is to place the inline definition in a header file.

In some cases cc21k does not output assembler code for the function; for example, the address is not needed for an `inline` function called only from within the defining program. However, recursive calls, and functions whose addresses are explicitly referred to by the program, are compiled to assembler code.

The `-no-inline` and `-traditional` switches disable function inlining. For more information, see "C/C++ Compiler Common Switch Descriptions" on page 2-19.

Inline Assembly Language Support Keyword (asm)

The cc21k `asm()` construct lets you code ADSP-21xxx family assembly language instructions within a C or C++ function. The `asm()` construct is useful for expressing assembly language statements that cannot be expressed easily or efficiently with C or C++ constructs.

With `asm()` you can code complete assembly language instructions or you can specify the operands of the instruction using C or C++ expressions. When specifying operands with a C or C++ expression, you do not need to know which registers or memory locations contain C or C++ variables.

The compiler **does not analyze** code defined with the `asm()` construct; it passes this code directly to the assembler. The compiler **does** perform substitutions for operands of the formats `%0` through `%9`; however it passes **everything else** through to the assembler without reading or analyzing it.

Any `asm()` constructs defined before the variable declarations within `main()` are flagged as errors because executable statements are not allowed before declarations in C code.

An `asm()` construct without operands takes the form shown below:

```
asm("r0=0;");
```

The complete assembly language instruction, enclosed in quotes, is the argument to `asm()`. Using `asm()` constructs with operands requires some additional syntax. The following sections cover this syntax:

- "Assembly Construct Template" on page 2-59
- "Assembly Construct Operand Description" on page 2-62
- "Assembly Constructs With Multiple Instructions" on page 2-64
- "Assembly Construct Reordering and Optimization" on page 2-65

- "Assembly Constructs with Input and Output Operands" on page 2-66
- "Assembly Constructs and Macros" on page 2-67

Assembly Construct Template

Using `asm()` constructs, you can specify the operands of the assembly instruction using C or C++ expressions. You do not need to know which registers or memory locations contain C/C++ variables. Use the following general syntax for your `asm()` constructs:

```
asm(
   template
   [:[constraint(output operand)[,constraint(output operand)…]]
      [:[constraint(input operand)[,constraint(input operand)…]]
         [:clobber]]]
);
```

- *template*

  The template is a string containing the assembly instruction(s) with `%number` indicating where the compiler should substitute the operands. Operands are numbered in order of appearance from left to right, starting at 0. Separate multiple instructions with a semicolon, and enclose the entire string within double quotes. For more information on templates containing multiple instructions, see "Assembly Constructs With Multiple Instructions" on page 2-64.

- *constraint*

  The constraint string directs cc21k to use certain groups of registers for the input and output operands. Enclose the constraint string within double quotes. For more information on operand constraints, see "Assembly Construct Operand Description" on page 2-62.

- *output operand*

  The output operand is the name of a C/C++ variable that receives output from a corresponding operand in the assembly instruction.

- *input operand*

  The input operand is a C or C++ expression that provides an input to a corresponding operand in the assembly instruction.

- *clobber*

  The clobber list informs cc21k that a list of registers are overwritten by the assembly instructions. Use lowercase for clobbered register names. Enclose each name within double quotes, and separate each quoted register name with a comma.

The following rules apply to assembly construct template syntax:

- The template is the only mandatory argument to `asm()`. All other arguments are optional.
- An operand constraint string followed by a C or C++ expression in parentheses describes each operand. For output operands, it must be possible to assign to the expression, that is, the expression must be legal on the left side of an assignment statement.
- A colon separates the template from the first output operand, the last output operand from the first input operand, and the last input operand from the clobbered registers. If there are no output operands and there are input operands, there must be two consecutive colons separating the assembly template from the input operands.
- A comma separates operands and registers within arguments.
- The number of operands in arguments must match the number of operands in your template.

- The maximum permissible number of operands is ten (%0, %1, %2, %3, %4, %5, %6, %7, %8, and %9).

The compiler cannot check whether the operands have data types that are reasonable for the instruction being executed. The compiler does not parse the assembler instruction template, interpret the template, or verify whether the template contains valid input for the assembler.

The following example shows how to apply the asm() construct template to the ADSP-21xxx family assembly language clip instruction:

```
{
 int result, x, y;
 asm (
   "%0=clip %1 by %2;" :
   "=d" (result) :
   "d" (x), "d" (y)
  );
}
```

In the above example, note the following points:

- The template is "%0=clip %1 by %2;". The %0 is replaced with operand zero (result), the first operand, %1, is replaced with operand one (x), and %2 is replaced with operand two (y).
- The output operand is the C/C++ variable, result. The letter d is the operand constraint for the variable. This constrains the output to an r0 - r15 register. The compiler generates code to copy the output from the R register to the variable result, if necessary. The = in =d indicates that the operand is an output.
- The input operands are the C/C++ variables, x and y. The letter d in the operand constraint position for these variables constrains x and y, each to an r0 - r15 register. If x and y are stored in different kinds of registers or on the stack, the compiler generates code to copy the values into R registers before the asm() construct uses them.

Assembly Construct Operand Description

The second and third arguments to the `asm()` construct describe the operands in the assembly language template. There are several pieces of information that need to be conveyed for cc21k to know how to assign registers to operands. You convey this information with an operand constraint. Primarily, cc21k needs to know what kind of registers your assembly instructions can operate on, so that it can allocate the correct register type. You convey this information with a letter in the operand constraint string that describes the class of allowable registers.

Table 2-8 on page 2-56 describes the correspondence between constraint letters and register classes. Note that the use of any letter not listed in Table 2-8 results in unspecified behavior. The compiler does not check the validity of the code by using the constraint letter.

For example, if your assembly template contains "`r0 = dm(m5, %0);`" and the address from which you want to load is in the variable `p`, the compiler needs to know that it should put `p` in a DAG1 I register (`I0-I7`) before it generates your instruction. You convey this information to cc21k by specifying the operand "`w`" `(p)` where "`w`" is the constraint letter for DAG1 I registers.

To assign registers to the operands, cc21k must also be told which operands in an assembly language instruction are inputs, which are outputs, and which outputs may not overlap inputs. The compiler is told this in three ways.

- The output operand list appears as the first argument after the assembly language template. The list is separated from the assembly language template with a colon. The input operands are separated from the output operands with a colon and always follow the output operands.
- The operand constraints describe which registers are modified by an assembly language instruction. The = in `=constraint` indicates that the operand is an output; all output operand constraints must use =.

- The compiler may allocate an output operand in the same register as an unrelated input operand, unless the output operand has the `&=` constraint modifier. This is because cc21k assumes that the inputs are consumed before the outputs are produced. This assumption may be false if the assembler code actually consists of more than one instruction. In such a case, use `&=` for each output operand that must not overlap an input.

Table 2-9. ASM() Operand Constraints

Constraint[1]	Register type	Registers
a	DAG2 B registers	b8 — b15
b	Q2 R registers	r4 — r7
c	Q3 R registers	r8 — r11
d	All R registers	r0 — r15
e	DAG2 L registers	l8 — l15
F	Floating-point registers	F0 — F15
f	Accumulator register	mrf, mrb
h	DAG1 B registers	b0 — b7
j	DAG1 L registers	l0 — l7
k	Q1 R registers	r0 - r3
l	Q4 R registers	r12 - r15
r	All general registers	r0 — r15, i0 — i15, l0 — l15, m0 — m15, b0 — b15, ustat1, ustat2
u	User registers	ustat1, ustat2

Table 2-9. ASM() Operand Constraints (Cont'd)

Constraint[1]	Register type	Registers
w	DAG1 I registers	I0 — I7
x	DAG1 M registers	M0 — M7
y	DAG2 I registers	I8 — I15
z	DAG2 M registers	M8 — M15
=&constraint	Indicates that the constraint is applied to an output operand that may not overlap an input operand	
=constraint	Indicates that the constraint is applied to an output operand	

[1] The use of any letter not listed in Table 2-9 results in unspecified behavior. The compiler does not check the validity of the code by using the constraint letter.

Assembly Constructs With Multiple Instructions

There can be many assembly instructions in one template. The input operands are guaranteed not to use any of the clobbered registers, so you can read and write the clobbered registers as often as you like. If the `asm()` string is longer than one line, you may continue it on the next line by placing a backslash (\) at the end of the line. The following listing is an example of multiple instructions in a template:

```
/* (pseudo code) r9 = from; r10 = to; result = from + to; */
asm (“r9=%1; \
r10=%2; \
%0=r9+r10;”
: “=d” (result)             /* output */
: “d” (from), “d” (to)      /* input */
: “r9”, “r10”);             /* clobbers */
```

Assembly Construct Reordering and Optimization

For the purpose of optimization, the compiler assumes that the side effects of an `asm()` construct are limited to changes in the output operands. This assumption does not mean that you cannot use instructions with side effects, but you must be careful. The compiler may eliminate them if the output operands are not used, or move them out of loops, or replace two with one if they constitute a common subexpression. Also, if your instruction does have a side effect on a variable that otherwise appears not to change, the old value of the variable may be reused later if it happens to be found in a register.

Use the keyword `volatile` to prevent an `asm()` instruction from being moved, combined, or deleted. For example:

```
asm volatile("idle;": /* no outputs */ : /* no inputs */ : /* no clobbers */ );
```

A sequence of `asm volatile()` constructs is not guaranteed to be completely consecutive; it may be moved across jump instructions or in other ways that are not significant to the compiler. To force the compiler to keep the output consecutive, use only one `asm volatile()` construct, or use the output of the `asm()` construct in a C or C++ statement.

Restrictions on the Use of the asm Construct

Due to possible interactions between the assembler code and the code generated by the compiler, the `asm` statement has the following restrictions:

- Control flow through a procedure must not be changed using instructions contained in an `asm()` construct. Control flow should only be specified through use of the C/C++ source constructs.
- C variables should not be referenced explicitly in the `asm()` construct template code. Such references should be through the input and output operands of the construct.

- All registers changed in the `asm()` construct template must be listed in the clobber section.
- Pre-processor macros defined in the C/C++ source if used in the `asm()` construct template must be expanded in the C/C++ source or defined in an prior `asm()` construct to achieve the correct definition.

Assembly Constructs with Input and Output Operands

The output operands must be write only; cc21k assumes that the values in these operands do not need to be preserved. When the assembler instruction has an operand that is both read from and written to, you must logically split its function into two separate operands: one input operand and one write-only output operand. The connection between them is expressed by constraints that say they need to be in the same location when the instruction executes.

You can use the same C or C++ expression for both operands, or different expressions. For example, in the following statement, the modify instruction uses `ball` as its read only source operand and `foot` as its read write destination:

```
   /* (pseudo code) modify (foot,ball); */
asm("modify (%0,%2);":"=w"(foot):"0"(foot),"x"(ball));
```

The constraint `"0"` for operand 1 says that it must occupy the same location as operand `0`. A digit in an operand constraint is allowed only in an input operand, and it must refer to an output operand.

Only a digit in the constraint can guarantee that one operand is in the same place as another. Just because a variable (for example `foot` in the code that follows) is used for more than one operand does not guarantee that the operands are in the same place in the generated assembler code. The following does not work:

```
   /* Do NOT try to control placement with operand names, use
   the %digit. The following code does NOT work */
asm("modify (%0,%2);":"=w"(foot):"w"(foot),"x"(ball));
```

Various optimizations or reloading could cause operands `0` and `1` to be in different registers. For example, the compiler might find a copy of the value of `foot` in one register and use it for operand `1`, but generate the output operand `0` in a different register.

Be aware that `asm()` does not support input operands that are used as both read operands and write operands. The example below shows a dangerous use of such an operand. In this example, `my_variable` is modified during the `asm()` operation. The compiler only knows that the output, `result_asm`, has changed. Subsequent use of `my_variable` after the `asm()` instruction may yield incorrect results since those values may have been modified during the `asm()` instruction and may not have been restored.

```
int result_asm;
int *my_variable;
   /* NOT recommended
   --(pseudo code) result_asm=dm(*my_variable,3);
   --asm() operation changes value of my_variable */
asm("%0=dm(%1,3);":"=d"(result_asm):"w"(my_variable));
```

Assembly Constructs and Macros

A way to use `asm()` constructs is to encapsulate them in macros that look like functions. For example, the following shows macros that contain `asm()` constructs. This code defines a macro, `clip_macro()`, which uses the `asm()` instruction to perform an assembly-language clip operation of variable `x_var` by `y_var`, putting the result in `result_var`:

```
#define clip_macro(result,x,y) \
 asm("%0=clip %1 by %2;":"=d"(result):"d"(x),"d"(y))

main () {
 int result_var;
 int x_var=10;
 int y_var=2;
 clip_macro(result_var, 10, 2);
 /* or */
 clip_macro(result_var, x_var, y_var);
}
```

Dual Memory Support Keywords (pm dm)

This section describes cc21k language extension keywords to C and C++ that support the dual-memory space, modified Harvard architecture of the ADSP-21xxx family processors. There are two keywords used to designate memory space: `dm` and `pm`. They can be used to specify the location of a static or global variable or to qualify a pointer declaration.

The following rules apply to dual memory support keywords:

- The memory space keyword (`dm` or `pm`) refers to the expression to the right of the keyword.
- You can specify a memory space for each level of pointer. This corresponds to one memory space for each `*` in the declaration.
- The compiler uses Data Memory (DM) as the default memory space for all variables. All undeclared spaces for data are Data Memory spaces.
- The compiler always uses Program Memory (PM) as the memory space for functions. Function pointers always point to Program Memory.
- You cannot assign memory spaces to automatic variables. All automatic variables reside on the stack, which is always in Data Memory.
- Literal character strings always reside in Data Memory.

The following listing shows examples of dual memory keyword syntax.

```
int pm buf[100];
/* declares an array buf with 100 elements in Program Memory */

int dm samples[100];
/* declares an array samples with 100 elements in Data Memory */

int points[100];
/* declares an array points with 100 elements in Data Memory */

int pm * pm xy;
/* declares xy to be a pointer which resides in Program
   Memory and points to a Program Memory integer */

int dm * dm xy;
/* declares xy to be a pointer which resides in Data Memory and
   points to a Data Memory integer */

int *xy;
/* declares xy to be a pointer which resides in Data Memory
   and points to a Data Memory integer */

int pm * dm datp;
/* declares datp to be a pointer which resides in Data Memory
   and points to a Program Memory integer */

int pm * datp;
/* declares datp to be a pointer which resides in Data Memory
   and points to a Program Memory integer */

int dm * pm progd;
/* declares progd to be a pointer which resides in Program
   Memory and points to a Data Memory integer */

int * pm progd;
/* declares progd to be a pointer which resides in Program
   Memory and points to a Data Memory integer */

float pm * dm * pm xp;
/* The first *xp is in Program Memory,
   the following *xp in Data Memory, and xp itself is
   in Program Memory */
```

Memory space specification keywords cannot qualify type names and structure tags, but you can use them in pointer declarations. The following listing shows examples of memory space specification keywords in `typedef` and `struct` statements.

```
/* Dual Memory Support Keyword typedef & struct Examples */

typedef float pm * PFLOATP;
/* PFLOATP defines a type which is a pointer to a       */
/* float which resides in pm.                           */

struct s {int x; int y; int z;};
static pm struct s mystruct={10,9,8};
/* Note that the pm specification is not used in        */
/* the structure definition. The pm specification       */
/* is used when defining the variable mystruct          */
```

Memory Keywords and Assignments/Type Conversions

Memory space specifications limit the kinds of assignments your program can make, as follows:

- You may make assignments between variables allocated in different memory spaces.
- Pointers to Program Memory must always point to Program Memory. Pointers to Data Memory must always point to Data Memory. You may not mix addresses from different memory spaces within one expression. Do not attempt to explicitly cast one type of pointer to another.

The following listings show a code segment with variables in different memory spaces being assigned and a code segment with illegal mixing of memory space assignments.

```
/* Legal Dual Memory Space Variable Assignment Example */
int pm x;
int dm y;
x = y;       /* Legal code */

/* Illegal Dual Memory Space Type Cast Example */
int pm *x;
int dm *y;
int dm a;
x = y;       /* Compiler will flag error */
x = &a;      /* Compiler will flag error */
```

Memory Keywords and Function Declarations/Pointers

Functions always reside in Program Memory. Pointers to functions always point to Program Memory. The following listing shows some sample function declarations with pointers.

```
/* Dual Memory Support Keyword Function Declaration (With
Pointers) Syntax Examples */

int * y();       /* function y resides in*/
                 /* pm and returns a     */
                 /* pointer to an integer*/
                 /* which resides in dm  */

int pm * y();    /* function y resides in*/
                 /* pm and returns a     */
                 /* pointer to an integer*/
                 /* which resides in pm  */


int dm * y();    /* function y resides in*/
                 /* pm and returns a     */
                 /* pointer to an integer*/
                 /* which resides in dm  */
```

```
int * pm * y();  /* function y resides in*/
                 /* pm and returns a     */
                 /* pointer to a pointer */
                 /* residing in pm that  */
                 /* points to an integer */
                 /* which resides in dm  */
```

Memory Keywords and Function Arguments

cc21k checks calls to prototyped functions for memory space specifications consistent with the function prototype. The following listing shows sample code that cc21k flags as inconsistent use of memory spaces between a function prototype and a call to the function.

```
/* Illegal Dual Memory Support Keywords & Calls To Prototyped
Functions */

extern int foo(int pm*);
/* declare function foo() which expects a pointer to an int
residing in pm as its argument and which returns an int */

int x;    /* define int x in dm                              */

foo(&x);  /* call function foo()                             */
          /* using pm pointer (location of x) as the         */
          /* argument. cc21k FLAGS AS AN ERROR; this is an */
          /* inconsistency between the function's            */
          /* declared memory space argument and function     */
          /* call memory space argument                      */
```

Memory Keywords and Macros

Using macros when making memory space specification for variables or pointers can make your code easier to maintain. If you must change the definition of a variable or pointer (moving it to another memory space), declarations that depend on the definition may need to be changed to ensure consistency between different declarations of the same variable or pointer.

To make changes of this type easier, you can use C/C++ preprocessor macros to define common memory spaces that must be coordinated. The following listing shows two code segments that are equivalent after preprocessing. The segment on the right lets you redefine the memory space specifications by redefining the macro `SPACE1` and `SPACE2`.

```
/* Dual Memory Support Keywords & Macros */
   #define SPACE1 pm
   #define SPACE2 dm

char pm * foo (char dm *)   char SPACE1 * foo (char SPACE2 *)
char pm *x;   char SPACE1 *x;
char dm y;    char SPACE2 y;

x = foo(&y);   x = foo(&y);
```

Placement Support Keyword (section)

The `section` keyword directs the compiler to place an object or function in an assembly `.SECTION` of the compiler's intermediate output file. You name the assembly `.SECTION` with the `section()`'s string literal parameter. If you do not specify a `section()` for an object or function declaration, the compiler uses a default section. For information on the default sections, see "Memory Usage" on page 2-123.

Applying `section()` is meaningful only when the data item is something that the compiler can place in the named section. Apply `section()` only to top-level, named objects that have a static duration, are explicitly `static`, or are given as external-object definitions.

The following example shows the declaration of a static variable that is placed in the section called `bingo`:

```
static section("bingo") int x;
```

Note that `section` has replaced the `segment` keyword of the Release 4.x compiler. Although the `segment()` struct is supported by the compiler of the present release, we recommend you to revise the legacy code.

Boolean Type Support Keywords (bool, true, false)

The `bool`, `true`, and `false` keywords are extensions that support the C++ boolean type. The `bool` keyword is a unique signed integral type, just as the `wchar_t` is a unique unsigned type. There are two built-in constants of this type: `true` and `false`. When converting a numeric or pointer value to `bool`, a zero value becomes `false` and a nonzero value becomes `true`. A `bool` value may be converted to `int` by promotion, taking `true` to one and `false` to zero. A numeric or pointer value is automatically converted to `bool` when needed.

These keyword extensions behave more or less as if the declaration that follows had appeared at the beginning of the file, except that assigning a nonzero integer to a `bool` type always causes it to take on the value `true`.

```
typedef enum { false, true } bool;
```

Pointer Class Support Keyword (restrict)

The `restrict` operator keyword is an extension that supports restricted pointer features. The use of `restrict` is limited to the declaration of a pointer and specifies that the pointer provides exclusive initial access to the object to which it points. More simply, `restrict` is a way that you can identify that a pointer does not create an alias. Also, two different restricted pointers cannot designate the same object, and therefore, they are not aliases.

The compiler is free to use the information about restricted pointers and aliasing to better optimize C or C++ code that uses pointers.The keyword is most useful when applied to function parameters about which the compiler would otherwise have little information, as shown in the following example:

```
void fir(short *in, short *c, short *restrict out, int n)
```

The behavior of a program is undefined if it contains an assignment between two restricted pointers, except for the following cases:

- A function with a restricted pointer parameter may be called with an argument that is a restricted pointer.
- A function may return the value of a restricted pointer that is local to the function, and the return value may then be assigned to another restricted pointer.

If you have a program that uses a restricted pointer in a way that it does not uniquely refer to storage, then the behavior of the program is undefined.

Variable-Length Array Support

The compiler supports variable-length automatic arrays when in C mode (variable-length array are not supported for C++). Unlike other automatic arrays, variable-length arrays are declared with a non-constant length. This means that the space is allocated when the array is declared, and deallocated when the brace-level is exited.

The compiler does not allow jumping into the brace-level of the array, and produces a compile time error message if this is attempted. The compiler does allow breaking or jumping out of the brace-level, and it deallocates the array when this occurs.

You can use variable-length arrays as function arguments, as shown in the following example:

```
struct entry
tester (int len, char data[len][len])
{
 …
}
```

The compiler calculates the length of an array at the time of allocation. It then remembers the array length until the brace-level is exited and can return it as the result of the `sizeof()` function performed on the array.

Because variable-length arrays must be stored on the stack, it is impossible to have variable-length arrays in Program Memory. The compiler issues an error if an attempt is made to use a variable-length array in `pm`.

As an example, if you were to implement a routine for computation of a product of three matrices, you need to allocate a temporary matrix of the same size as the input matrices. Declaring an automatic variable size matrix is much easier then explicitly allocating it in a heap.

The expression declares an array with a size that is computed at run time. The length of the array is computed on entry to the block and saved in case `sizeof()` is applied to the array. For multidimensional arrays, the boundaries are also saved for address computation. After leaving the block, all the space allocated for the array is deallocated.

For example, the following program prints 10, not 50:

```
main ()
{
   foo(10);
}

void foo (int n)
{
    char c[n];
    n = 50;
    printf(“%d”, sizeof(c));
}
```

Non-Constant Initializer Support

The cc21k compiler includes support for the ISO/ANSI standard definition of the C and C++ language and includes extended support for initializers. The compiler does not require the elements of an aggregate initializer for an automatic variable to be constant expressions. The following example shows an initializer with elements that vary at run time:

```
void initializer (float a, float b)
{
   float the_array[2] = { a-b, a+b };
}


void foo (float f, float g)
{
   float beat_freqs[2] = { f-g, f+g };
}
```

Indexed Initializer Support

ANSI/ISO standard C/C++ requires the elements of an initializer to appear in a fixed order, the same as the order of the elements in the array or structure being initialized. The cc21k compiler C/C++, by comparison, supports labeling elements for array initializers. This feature lets you specify the array or structure elements in any order by specifying the array indices or structure field names to which they apply. All index values must be constant expressions, even in automatic arrays.

The following example shows equivalent array initializers, the first in standard C/C++ and the next using cc21k C/C++. Note that the `[index]` precedes the value being assigned to that element.

```
/* Example 1 Standard & cc21k C/C++ Array Initializer */

   /* Standard array initializer */

int a[6] = { 0, 0, 15, 0, 29, 0 };

   /* equivalent cc21k C/C++ array initializer */

int a[6] = { [4] 29, [2] 15 };
```

You can combine this technique of naming elements with standard C/C++ initialization of successive elements. The standard and cc21k instructions below are equivalent. Note that any unlabeled initial value is assigned to the next consecutive element of the structure or array.

```
/* Example 2 Standard & cc21k C/C++ Array Initializer */

   /* Standard array initializer */

int a[6] = { 0, v1, v2, 0, v4, 0 };

   /* equivalent cc21k C/C++ array initializer that uses
   indexed elements */

int a[6] = { [1] v1, v2, [4] v4 };
```

The following example shows how to label the array initializer elements when the indices are characters or an enum type.

```
/* Example 3 Array Initializer With enum Type Indices */

   /* cc21k C/C++ array initializer */

int whitespace[256] =
{
   [' '] 1, ['\t'] 1, ['\v'] 1, ['\f'] 1, ['\n'] 1, ['\r'] 1
};
```

In a structure initializer, specify the name of a field to initialize with `fieldname:` before the element value. The standard C/C++ and cc21k C/C++ struct initializers in the example below are equivalent.

```
/* Example 4 Standard C & cc21k C/C++ struct Initializer */

   /* Standard C struct Initializer */

struct point {int x, y;};
struct point p = {xvalue, yvalue};

   /* Equivalent cc21k C/C++ struct Initializer With Labeled
   Elements */

struct point {int x, y;};
struct point p = {y: yvalue, x: xvalue};
```

Aggregate Constructor Expression Support

Extended initializer support in cc21k C/C++ includes support for aggregate constructor expressions, which enable you to assign values to large structure types without requiring each element's value to be individually assigned.

The following example shows an ISO/ANSI standard C `struct` usage followed by equivalent cc21k C/C++ code that has been simplified using a constructor expression:

```
/* Standard C struct & cc21k C/C++ Constructor struct */

   /* Standard C struct */
struct foo {int a; char b[2];};
struct foo make-foo(int x, char *s)
{
   struct foo temp;
   temp.a = x;
   temp.b[0] = s[0];
   if (s[0] !=  '\0')
      temp.b[1] = s[1];
```

```
   else
      temp.b[1] = '\0';
   return temp;
}

   /* Equivalent cc21k C/C++ constructor struct */
struct foo make_foo(int x, char *s)
{
   return((struct foo) {x, {s[0], s[0] ? s[1] : '\0'}});
}
```

Preprocessor Generated Warnings

The preprocessor directive `#warning` causes the preprocessor to generate a warning and continue preprocessing. The text on the remainder of the line that follows `#warning` is used as the warning message.

C++ Style Comments

The compiler accepts C++ style comments in C programs, beginning with `//` and ending at the end of the line. This is essentially compatible with standard C, except for the following case:

```
a = b
//* highly unusual */ c
```

which a standard C compiler processes as:

```
a = b / c;
```

Compiler intrinsic Functions

The compiler supports intrinsic functions that enable efficient use of hardware resources. Knowledge of these functions is built into the cc21k compiler. Your program uses them via normal function call syntax. The compiler notices the invocation and generates one or more machine instructions, just as it does for normal operators, such as `+` and `*`.

Builtins have names which begin with `__builtin_`. Note that identifiers beginning with double underlines (__) are reserved by the C standard, so these names will not conflict with user program identifiers. The header files also define more readable names for the built-in functions without the `__builtin_` prefix. These additional names are disabled if the `-no-builtin` option is used.

The cc21k compiler provides built-in versions of some of the C library functions as described in "Using the Compiler's Built-In C library Functions" on page 3-19.

The `sysreg.h` header file defines a set of functions that provide efficient system access to registers, modes, and addresses not normally accessible from C source. These functions are specific to individual architectures. The built-in functions supported at this time on ADSP-21xxx family DSPs are listed in the sub-sections that follow.

Access to System Registers

This section describes the functions that provide access to system registers. These functions are based on the ADSP-21xxx family DSPs underlying hardware capabilities. The functions are defined in the header file `sysreg.h`. They allow direct read and write access, as well as, the testing and modifying of bit sets.

```
int sysreg_read (const int SR_number);
```

`sysreg_read` reads the value of the designated register and returns it.

```
void sysreg_write (const int SR_number, const int new_value);
```

`sysreg_write` stores the specified value in the nominated system register.

```
void sysreg_bit_clr (const int SR_number, const int bit_mask);
```

`sysreg_bit_clr` clears all the bits of the nominated system register that are set in the supplied bit mask.

```
void sysreg_bit_set (const int SR_number, const int bit_mask);
```

`sysreg_bit_set` sets all the bits of the nominated system register that are also set in the supplied bit mask.

```
void sysreg_bit_tgl (const int SR_number, const int bit_mask);
```

`sysreg_bit_tgl` toggles all the bits of the nominated system register that are set in the supplied bit mask.

```
int sysreg_bit_tst (const int SR_number, const int bit_mask);
```

`sysreg_bit_tst` returns a non-zero value if all of the bits that are set in the supplied bit mask are also set in the nominated system register.

```
int sysreg_bit_tst_all (const int SR_number, const int value);
```

`sysreg_bit_tst_all` returns a non-zero value if the contents of the nominated system register are equal to the supplied value.

The register names are defined in `sysreg.h` and must be a compile-time literal. The effect of using the incorrect function for the size of the register or using an undefined register number is undefined. The register names (and their values) used by the functions are as follows.

```
sysreg_USTAT1    = 0x7,
sysreg_USTAT2    = 0x8,

sysreg_IRPTL     = 0x2,
sysreg_MODE2     = 0x1,
sysreg_MODE1     = 0x0,
sysreg_ASTAT     = 0x5,
sysreg_IMASK     = 0x3,
sysreg_STKY      = 0x6,
sysreg_IMASKP    = 0x4
```

Due to hardware characteristics of the SHARC processor, the bit values for the `sysreg_bit_*` interrogation and manipulation functions must be compile-time constants. For the ADSP-2106x DSP, the header files `def21060.h`, `def21061.h`, and `def21065.h` provide symbolic names for the individual bits in the system registers. For the ADSP-2116x DSP the header files `def21160.h`, `def21161.h`, and `def21065.h` provide symbolic names for the individual bits in the system registers.

For the ADSP-2116x SHARC DSP, the compiler considers `USTAT1` and `USTAT2` to be "preserved" registers. If one is modified by a `sysreg` operation, it will be saved during the prolog of the containing function and restored to its old value on exit.

C++ Fractional Type Support

While in C++ mode, the cc21k compiler supports fractional (fixed-point) arithmetic that provides a way of computing with non-integral values within the confines of the fixed-point representation. Hardware support for the 32-bit fractional arithmetic is available on the ADSP-21xxx family DSPs.

Fractional values are declared with the `fract` data type. Ensure that your program incudes the `<fract>` header file. The `fract` data type is a C++ class that supports a set of standard arithmetic operators used in arithmetic expressions. Fractional values are represented as signed values in a range of [`-1 … 1`) with a binary point immediately after the sign bit. Other value ranges are obtained by scaling or shifting. In addition to the arithmetic, assignment, and shift operations, `fract` provides several type-conversion operations.

For more information about supported fractional arithmetic operators, see page 2-86. For sample programs demonstrating the use of the `fract` type, see Listing 2-5 on page 2-163 and Listing 2-6 and Listing 2-7 on page 2-164.

The current release of the software does not provide for automatic scaling of fractional values.

Format of Fractional Literals

Fractional literals use the floating-point representation with an "`r`" suffix to distinguish them from floating-point literals, for example, `0.5r`. The cc21k compiler validates fractional literal values at run time to ensure they reside within the valid range of values.

Fractional literals are written with the "`r`" suffix to avoid certain precision loss. Literals without an "`r`" are of the type `double`, and are implicitly converted to `fract` as needed. After the conversion of a 32-bit `double` literal to a `fract` literal, the value of the latter may retain only 25 bits of precision compared with the full 32 for a fractional literal with the "`r`" suffix.

Conversions Involving Fractional Values

The following notes apply to type-conversion operations:

- Conversion between a fractional value and a floating value is supported. The conversion to the floating-point type may result in some precision loss.
- Conversion between a fractional value and an integer value is supported. The conversion is not recommended because the only common values are `0` and `-1`.
- Conversion between a fractional value and a long double value is supported via `float` and may result in some precision loss.

Fractional Arithmetic Operations

The following notes summarize information about fractional arithmetic operators supported by the cc21k compiler:

- Standard arithmetic operations on two `fract` items include addition, subtraction, and multiplication.
- Assignment operations include `+=`, `-=`, and `*=`.
- Shift operations include left and right shifts. A left shift is implemented as a logical shift and a right shift is an arithmetic shift. Shifting left by a negative amount is not recommended.
- Comparison operations are supported between two `fract` items.
- Mixed-mode arithmetic has a preference for `fract`. For more information about the mixed-mode arithmetic, see page 2-87.
- Multiplication of a fractional and an integer produces an integer result or a fractional result. The program context determines which type of result is generated following the conversion algorithm of C++. When the compiler does not have enough context, it generates an ambiguous operator message, for example:

  ```
  error: more than one operator "*" matches these operands:
  …
  ```

  You cast the result of the multiply operation if the error occurs.

Mixed Mode Operations

Most operations supported for fractional values are supported for mixed fractional/float or fractional/double arithmetic expressions. At run time, a floating-point value is converted to a fractional value, and the operation is completed using fractional arithmetic.

The assignment operations, such as +=, are the exception to the rule. The logic of an assignment operation is defined by the type of a variable positioned on the left side of the expression.

Floating-point operations require an explicit cast of a fractional value to the desired floating type.

Saturated Arithmetic

The cc21k compiler supports saturated arithmetic for fractional data in the saturated arithmetic mode.

Whenever a calculation results in a bigger value than the `fract` data type represents, the result is truncated (wrapped around). An overflow flag is set to warn the program that the value has exceeded its limits. To prevent the overflow and to get the result as the maximum representable value when processing signal data, use saturated arithmetic. Saturated arithmetic forces an overflowed value to become the maximum representable value.

The mode is set to be saturated or default with the `set_saturate_mode()` and `reset_saturate_mode()` functions. Each arithmetic operator has its corresponding variant effected in the saturated mode, for example, `add_sat`, `sub_sat`, `neg_sat`, ….

SIMD Support Annotation (#pragma SIMD_for)

The ADSP-2116x processor supports Single-Instruction, Multiple-Data operations, which, under certain conditions, double the computational rate over ADSP-2106x family DSPs. It is important to gain some understanding of the DSP's architecture to take advantage of SIMD mode.

The ADSP-2116x processor includes a second processing element that has its own register file and computational units. When the second element is active, the processor operates in a "Single-Instruction, Multiple-Data" (SIMD) mode—each computation executes on both processing elements, and each element operates independently on different ("multiple") data.

SIMD is effective for performing exactly the same calculations simultaneously on two parallel sets of data. As a special case — which is what the compiler supports—programs can use the SIMD mechanism to perform a single task (such as summing a vector), by dividing it into two parts and doing both parts in parallel, simultaneously.

Also, there are situations where it is effective to perform two copies of a task simultaneously, such as summing two separate vectors.

SIMD processing is intimately linked with the memory model. In Single-Instruction, Single-Data (SISD) processing (ADSP-2106x compatible mode), the processor fetches single values from memory, performs single arithmetic operations, and stores single values back. In ADSP-2116x SIMD mode, each memory reference fetches a pair of values (from the designated address and the following address), one into each of the two compute blocks. Arithmetic instructions are done in pairs, and paired results are written back.

When compiling for the ADSP-2116x DSPs, the cc21k compiler automatically generates SIMD code wherever possible. However, there are occasions when the cc21k compiler does not generate SIMD code because the compiler does not have enough information to be sure that it is safe to use SIMD. If such a situation occurs, the compiler generates a warning, specifying the reason the automatic generation of SIMD code was disabled. This situation occurs most often in functions, where the data to be processed is passed as parameters.

The primary causes of disabling automatic SIMD generation are a lack of alias information or a lack of alignment information. Normally, IPA helps to resolve these issues automatically. However, even with IPA, there may be times when the compiler cannot determine if it is safe to use SIMD code. If SIMD code is appropriate, then the SIMD_for pragma can be used to indicate this to the compiler.

Using SIMD Mode with Multichannel Data

When processing multichannel data in SIMD mode, the program essentially runs two copies of the algorithm simultaneously. Multichannel could be used for a whole program, for processing two modem channels, or for more local processes, such as calculating the sine of two values simultaneously.

Because there are two copies of the algorithm running, there must be two copies of the data as well. Due to the ADSP-2116x's SIMD memory architecture, the data must be interleaved in memory. The data for one channel uses only even locations, while data for the other channel uses the corresponding odd locations. Because the DSP implicitly doubles the memory references, arranging data in memory can be as simple as allocating twice as much space for all variables. Correct data arrangement in memory depends on the algorithm.

Such a program could increment loop indices by 2 instead of 1, as each fetch has consumed two words of memory. Data that is common to both could be loaded with a broadcast load or duplicated in memory.

The ADSP-2116x can handle conditional execution at a single-instruction level in SIMD mode and counted loops. Programs cannot do a conditional branch that depends on the result of a computation in SIMD mode because there would be two different results (one for each channel) and the branch must go one way or the other.

The compiler does not provide extended C/C++ support for SIMD mode and multichannel data.

Using SIMD Mode with Single Channel Data

When processing single channel data in SIMD mode, most of the program operates in SISD mode. At key places where the program loops over a collection of contiguous data elements, the program enters SIMD mode to perform computations on both processing elements.

For example, a program adds two vectors element by element. The normal SISD code picks up elements one at a time, evaluating

`c[j] = a[j] + b[j]` for each `j`.

Since each element is processed independently, the operation can as well be done in pairs in SIMD mode:

`c[j] = a[j] + b[j]` in one processing element,

`c[j+1] = a[j+1] + b[j+1]` in the other.

Note that the loop index now increments by 2.

This kind of processing is an effective use of the SIMD capability.

SIMD processing can also be used when one of the terms is a scalar. In that case, you should make sure that the same scalar value is loaded into both compute blocks, and the rest of the SIMD processing is the same.

A useful variant of the SIMD loop occurs in a form called a reduction. In such a loop, a vector is reduced to a scalar value by the action of the loop. For example, summing a vector or calculating the dot product of two vectors are areas where a program could use reduction. Again, SIMD processing lets the program process the vector on both processor elements at the same time (half in each).

Note that a reduction loop has to compute a single result. To use SIMD processing, the SISD algorithm would be transformed slightly. Two partial results are accumulated with all the even elements contributing to one result and the odds to the other. At the end of the loop, the program must combine the partial results into a final one. The reduction approach has two effects for which you must account:

- If the data is floating-point, the results will likely differ slightly due to floating-point round-off differences.
- The final combination of the partial results takes a little time that will detract from the SIMD performance gain.

In any of the single channel cases, if the array contains an odd number of elements, an extra step of processing will be required after the SIMD region. Note that when the number is not known at compile time, the extra step will be conditional on a run-time check.

The compiler provides extended C/C++ support for SIMD mode and single channel data. For more information, see “SIMD Support Annotation (#pragma SIMD_for)” on page 2-88.

Pitfalls in Using SIMD C/C++

Be aware that there are various pitfalls, as well as advantages, when a program is transformed to process single channel data in SIMD mode.

- The data and the access pattern must be arranged for SIMD fetches, which are always at immediately adjoining locations. For example, a program that sums every third element of an array or the first column of a multi-dimensional array would not be a good candidate for SIMD, because the second fetch does not pick up the element for the next iteration.
- The data must always be aligned on double-word boundaries when in SIMD mode. The compiler attempts to align arrays properly in memory, so that operations on whole arrays work.

You must be careful about situations that force misalignment. This can occur by calling a function with an argument that points to an arbitrary location in an array. For instance, a function `func(A[j]` where `func` expects a pointer or array parameter could have a data alignment problem if the value of `j` is odd. In this case, the parameter denotes a misaligned array, and an attempt to use SIMD processing within `func` fails.

Another SIMD failure situation arises when the reference pattern involves odd locations. A reference to `A[j-1]` fails. Similarly, programs cannot have locations that change between even and odd addresses (for example, `A[j+k]`). The FIR loop nest is an example of the latter.

You have to be careful about any interaction or dependency between different iterations of the loop in SIMD. This is important because the SIMD processing changes the order of evaluation. The data for iteration N+1 is actually fetched from memory before the results of iteration N are written back. Some programs get wrong answers if done in SIMD.

Looking at the example:

```
a[j] = a[j-1] + 2;
```

the current iteration uses the results from the previous one; if these results have not yet been written back, the current iteration is incorrect.

SIMD_for Syntax

The code transformation of a loop to run in SIMD mode involves one command. You indicate which loops are suitable for SIMD execution, and the compiler does the rest. Indicating that a loop should execute in SIMD mode takes the form of a `#pragma` command,

```
#pragma SIMD_for
```

which is placed ahead of the loop. As a preprocessing directive, this command must be alone on the line similar to a `#define` or `#include` command.

The compiler responds to the `#pragma` by first checking whether the loop meets the SIMD guidelines. If compliant, the compiler transforms the loop so that the processing is done in SIMD mode. Among other things, the transformation involves changing the loop increment to 2, so that the vector elements are processed in SIMD pairs.

If the loop performs a reduction, partial results are calculated and combined at the end. Also, the compiler takes care of duplicating scalar values used within the loop. The following loop uses `#pragma SIMD_for`:

```
float sum, c, A[N];
…
sum = 0;
#pragma SIMD_for
   for (j=0; j<N; j++) {
      sum += c * A[j];
      }
```

The compiler transforms this loop as follows:

```
   // declare SIMD temporaries
float t_sum[2], t_c[2];
   // initialize both partial sums
t_sum[0] = t_sum[1] = 0;
   // initialize both parts of scalar constant
t_c[0] = t_c[1] = c;
   // ENTER_SIMD_MODE -- set machine mode
for (j=0; j<N; j+=2) {
   t_sum[0] += t_c[0] * A[j];
      //  --  implicit SIMD processing performs:
      // t_sum[1] += t_c[1] * A[j+1];
   }
   // LEAVE_SIMD_MODE
   // combine partial sums
sum =   t_sum[0] + t_sum[1];
```

Constraints on Using SIMD C/C++

There are a number of conditions that limit when SIMD operations may occur. The compiler attempts to check these conditions and issues warnings or errors when it detects problems or possible problems.

The compiler usually avoids changing a program when the compiler is not certain that the transformed program produces the same results as the original. The SIMD transformations are handled a bit differently for two reasons:

- You provide explicit direction to use SIMD. Therefore, the compiler assumes that you are aware of what is needed for SIMD operation and that you share responsibility for correct operation.
- Some of the SIMD constraints are difficult or impossible to verify at compile time. Therefore, the compiler is not 100 percent conservative in checking, because if it were, few, if any loops would be accepted.

The compiler checks the conditions that can be checked. In many cases, the compiler can verify that a loop is unacceptable and rejects it for SIMD processing with a warning or error. Rejection occurs when there is an obvious dependency, obvious alignment problems, or a non-unit stride.

In other cases, the determining values are not available at compile time, and the compiler cannot be sure whether the loop can be safely transformed. In such cases, the compiler issues a warning and proceeds.

For some constraints—primarily the proper alignment of arrays that are parameters (arguments) of the function—the compiler assumes that conditions are acceptable and does not issue a warning.

There are two other restrictions on using `SIMD_for` loops:

- Function calls may not be made from within a `SIMD_for` loop.
- All data types used within a `SIMD_for` loop must have single-word base types. Long doubles, doubles in `double-size-64` mode, and `structs` should not to be used within a `SIMD_for loop`.

Impact of Anomaly #40 on SIMD

The SIMD read from internal memory with a Shadow Write FIFO hit does not always function correctly. This anomaly has been identified in the Shadow Write FIFOs that exist between the internal memory array of the ADSP-21160M and core /IOP busses that access the memory. (See page 7-73 of the *ADSP-21160 SHARC DSP Hardware Reference,* First Edition, November, 1999 for more details on shadow register operation). If performing SIMD reads which cross Long Word Address boundaries (i.e. odd Normal Word addresses or non-Long Word boundary aligned Short Word addresses) and the data for the read is in the Shadow Write FIFO, the read will result in revision 0.0 behavior for the read.

To avoid the anomaly, SIMD operations must always operate on double-word aligned vectors. To accomplish this type of operation the compiler:

- Allocates all static arrays on an appropriate double-word boundary.
- Ensures that arrays on the stack are double-word aligned by ensuring the stack is aligned on an even word boundary.
- The compiler only generates SIMD operations when it knows it is operating with double-word-aligned elements. It will be able to do this when arrays are defined statically or locally, or the arrays are arguments whose properties can be determined by Inter Procedural Analysis. A further requirement is that the initial index and increment are both explicit.

As a result of these precautions, SISD mode will be used when array properties and indexing are not "visible" to the optimizer.

The compiler generates a warning when it fails to automatically generate SIMD operation due to lack of information. The user can then add the `#pragma simd_for` in such cases where they can be sure that alignment requirements are satisfied.

Examples Using SIMD C (Problem Cases—Data Increments)

In SIMD mode, an assignment to or from a memory location refers to memory location [A] for the PEx processing element and memory location [A+1] for the PEy processing element. The `#pragma SIMD_for` takes advantage of these assignments by taking code containing a stride 1 loop which addresses contiguous memory locations and turning it into a stride 2 loop, addressing every second memory location.

In a potentially SIMD compatible loop, it is essential that the stride of a loop through an array is 1, so the compiler uses the correct memory locations. Any other value for the stride results in incorrect behavior.

The following matrix multiplication function demonstrates some stride issues.

This code is NOT a valid use of the `SIMD_for` pragma.

```
float *matmul(void *x_input,
              void *y_input,
              void *output,
              int r,
              int s,
              int t) {
   float *ipx, *ipy, *output_new;
   float tmp = 0;
   int i = 0, j = 0, k = 0;
   ipx = (float *) x_input;
   ipy = (float *) y_input;
   output_new = (float *) output;
   for (i = 0; i < r; i++)
   for (k = 0; k < t; k++) {
      tmp = 0;
      #pragma SIMD_for
         for (j = 0; j < s; j++)
         // The next two lines are wrong in SIMD mode
         tmp += ipx[j + (i * s)] * ipy[k + (j * t)];
         output_new[k + (i * r)] = tmp;
      }
      printf("SIMD\n");
      return output_new;
   }
```

The lines in `bold text` above read from the memory location `ipy[k+(j*t)]`. The loop counter, `j`, is multiplied to calculate the offset into the array. In this example, the stride through the array can not be 1, rather it is `t`.

The SIMD and non-SIMD versions of this code address the following memory locations:

non-SIMD	SIMD
`ipy[k]`	`ipy[k], ipy[k+1]`
`ipy[k + t]`	`ipy[k+(2*t)], ipy[k+(2*t)+1]`
`ipy[k + (2*t)]`	`ipy[k+(4*t)], ipy[k+(4*t)+1]`
`ipy[k + (3*t)]`	`ipy[k+(6*t)], ipy[k+(6*t)+1]`
`ipy[k + (4*t)]`	`ipy[k+(8*t)], ipy[k+(8*t)+1]`

Each version addresses different memory locations, giving different results.

SIMD C Loop Counter Rules:

- If the loop counter is multiplied within a loop to calculate an offset, do not use SIMD.
- If the [inner] loop counter is used to subscript a multi-dimensional array, it must only be used as the last subscript. Otherwise, do not use SIMD.

Examples Using SIMD C (Problem Cases—Data Alignment)

To work properly, SIMD calculations must only be used on arrays or other data that are double-word aligned. The compiler and libraries have some responsibility in ensuring that arrays meet this condition. All arrays, unions, and structures are guaranteed to be double-word aligned by the compiler, and functions such as `malloc()` only return double-word aligned memory.

There are some conditions in which you are responsible for determining whether the code and data are compatible with SIMD execution. This section describes some of the pitfalls of which you need to be aware.

Using Two-Dimensional Arrays

The following array,

```
int xyz[9][9];
```

would be double-word aligned by the compiler. Each sub-array, however, would start at an offset of 9 from the previous array, so `xyz[1]`, `xyz[3]`, and subsequent elements would not be double word aligned. Trying to use these sub-arrays in SIMD mode creates problems. If the function `sum()` contains SIMD code, then the following is incorrect:

```
for (i = 0; i < 10; i++)
   total += sum( xyz[i] );
      // leads to SIMD problems
```

SIMD C/C++ Data Alignment Rule:

- Two-dimensional arrays containing an odd number of rows or columns may lead to problems.

Adding To Array Offsets

The following is an example in which an offset is calculated using outer and inner loop counters. This code is NOT a valid use of the `SIMD_for` pragma.

```
for (k = 0; k < 20; ++k)
  #pragma SIMD_for
  for (i = 0; i < 20-k; ++i)
     output[i] += (input[i] + input[i+k]);
        // leads to SIMD problems
```

In this case, every second iteration of the outer loop (k=1, k=3, etc.) results in incorrect code as the expression `input[i+k]` is a non-double-word aligned location.

Note that this loop could be rewritten as:

```
for (k = 0; k < 20; ++k) {
   if (k % 2)
      for (i = 0; i < 20-k; ++i)
         output[i] += (input[i] + input[i+k]) ;
   else
      #pragma SIMD_for
      for (i = 0; i < 20-k; ++i)
         output[i] += (input[i] + input[i+k]) ;
   }
```

Now, the SIMD version is only used when k=0, k=2, and subsequent even elements. This technique does not offer the full performance benefits of SIMD, but does offer about a 50% improvement.

Performance When Using SIMD C/C++

When handling multichannel data in SIMD mode, the ADSP-2116x can accomplish roughly twice as much useful work as ADSP-2106x family DSPs. This is diluted slightly if data-dependent conditional blocks must be accommodated. The compiler does not support multichannel data operations in C or C++ code.

When handling single channel data in SIMD mode, C and C++ programs using single channel SIMD portions will usually show some performance improvement, but fall short of the double-performance level for a variety of reasons.

In measuring the performance improvement in single channel SIMD, it's useful to isolate the SIMD portion and verify that it is performing as intended. The overall program speedup is often bounded by factors outside of the SIMD portion.

Any parallel-processing situation requires a small amount of overhead to coordinate the parallelism, and the ADSP-2116x is no exception. Understanding this factor can help you evaluate where SIMD mode is likely to be most beneficial. Specific items include the following:

- Mode change: The processor must switch into SIMD mode and back out. Each change takes 2 cycles.
- Initialization of scalars: Non-array values must be duplicated in order to have correct values for both processing elements. This takes a few instructions per value. This is a compiler restriction only—assembly programmers may be able to use the broadcast load facility.
- Collection of partial results: For reductions such as vector dot-product, the two partial results must be combined at the end. This typically requires a move and a final add or multiply, another 2 or 3 cycles.

All of these are fairly small items and have little effect provided that the size of the SIMD loop is large.

Note, also, that the size of the inner loop is halved when working in SIMD mode. Consider the following example, a filter with 40 coefficients.

- Inner loop before SIMD: 40 iterations
- Inner loop with SIMD: 20 iterations

Because the ADSP-2106x family DSPs can do a dot-product with a single instruction, the loop represents 20 cycles. If the SIMD overhead is 6 cycles, this operation on the ADSP-2116x represents an overhead cost of 30%.

Actually, the true cost is a bit higher. Most loops require a little prologue code to achieve full processor efficiency. When the loop size is halved, the relative impact of the prologue—which remains a constant size—is increased. The prologue can lead to the loss of a few more percentage points off the performance.

Preprocessing a Program

The compiler includes a preprocessor that lets you use preprocessor commands within your C or C++ source. Table 2-10 lists these commands and provides a brief description of each. The preprocessor automatically runs before the compiler.

Table 2-10. Preprocessor Commands

Command	Description
`#define`	Defines a macro or constant.
`#elif`	Sub-divides an `#if ... #endif` pair.
`#else`	Identifies alternative instructions within an `#if ... #endif` pair.
`#endif`	Ends an `#if ... #endif` pair.
`#error`	Reports an error message.
`#if`	Begins an `#if ... #endif` pair.
`#ifdef`	Begins an `#ifdef ... #endif` pair and tests if macro is defined.
`#ifndef`	Begins an `#ifndef ... #endif` pair and tests if macro is not defined.
`#include`	Includes source code from another file.
`#line`	Outputs specified line number before preprocessing.

Table 2-10. Preprocessor Commands (Cont'd)

Command	Description
#undef	Removes macro definition.
#warning	Reports a warning message.
#	Converts a macro argument into a string constant.
##	Concatenates two strings.

Preprocessor commands are also useful for modifying the compilation. Using the #include command, you can include header files (.h) that contain code and/or data. A macro, which you declare with the #define preprocessor command, can specify simple text substitutions or complex substitutions with parameters. The preprocessor replaces each occurrence of the macro reference found throughout the program with the specified value.

The preprocessor is separate from the compiler and has some features that may not be used within your C or C++ source file. For more information, see the *VisualDSP++ 2.0 Assembler and Preprocessor Manual for ADSP-21xxx DSPs*.

Predefined Macros

The predefined macros that cc21k provides are listed below.

__2106x__

When compiling for the ADSP-21060, ADSP-21061, ADSP-21062, or the ADSP-21065L, cc21k defines `__ADSP2106x__` as 1.

__2116x__

When compiling for the ADSP-21160 or ADSP-21161, cc21k defines __2116x__ as 1.

__ADSP21000__

cc21k always defines `__ADSP21000__` as 1.

ADSP21000

cc21k defines `ADSP21000` as 1. The `__ADSP21000__` macro is defined unless you compile with `-no-extra-keywords`, `-pedantic`, or `-pedantic-errors`.

__ADSP21060__

cc21k defines `__ADSP21060__` as 1 when you compile with the `-21060` command-line switch.

ADSP21060

cc21k defines `ADSP21060` as 1 when you compile with the `-21060` command-line switch, but the compiler undefines this macro if you compile with `-pedantic` or `-pedantic-errors`.

__ADSP21061__

cc21k defines `__ADSP21061__` as 1 when you compile with the `-21061` command-line switch.

ADSP21061

cc21k defines `ADSP21061` as 1 when you compile with the `-21061` command-line switch, but the compiler undefines this macro if you compile with `-pedantic` or `-pedantic-errors`.

__ADSP21062__

cc21k defines `__ADSP21062__` as 1 when you compile with the `-21062` command-line switch.

ADSP21062

cc21k defines `ADSP21062` as 1 when you compile with the `-21062` command-line switch, but the compiler undefines this macro if you compile with `-pedantic` or `-pedantic-errors`.

__ADSP21065L__

cc21k defines `__ADSP21065L__` as 1 when you compile with the `-21065L` command-line switch.

ADSP21065L

cc21k defines `ADSP21065L` as 1 when you compile with the `-21065L` command-line switch, but the compiler undefines this macro if you compile with `-pedantic` or `-pedantic-errors`.

__ADSP21160__

cc21k defines `__ADSP21160__` as 1 when you compile with the `-21160` command-line switch.

ADSP21160

cc21k defines `ADSP21160` as 1 when you compile with the `-21160` command-line switch, but the compiler undefines this macro if you compile with `-pedantic` or `-pedantic-errors`.

__ADSP21161__

cc21k defines `__ADSP21161__` as 1 when you compile with the `-21161` command-line switch.

ADSP21161

cc21k defines `ADSP21161` as 1 when you compile with the `-21161` command-line switch, but the compiler undefines this macro if you compile with `-pedantic`, or `-pedantic-errors`.

__ANALOG_EXTENSIONS__

cc21k defines `__ANALOG_EXTENSIONS__` as 1, and the compiler undefines this macro if you compile with `-pedantic` or `-pedantic-errors`.

__cplusplus

cc21k defines `__cplusplus` as 1 when you compile in C++ mode.

__DATE__

The preprocessor expands this macro into the preprocessing date as a string constant. The date string constant takes the form `Mmm dd yyyy`. (ANSI standard).

__DOUBLES_ARE_FLOATS__

cc21k defines `__DOUBLES_ARE_FLOATS__` as 1 when you compile with the `-double-size-32` command-line switch. Note that `-double-size-32` is the default switch. The compiler undefines this macro if the `-double-size-64` switch is used.

__ECC__

cc21k always defines `__ECC__` as 1.

__EDG__

cc21k always defines `__EDG__` as 1. This signifies that an Edison Design Group front end is being used.

__EDG_VERSION__

cc21k always defines `__EDG_VERSION__` as an integral value representing the version of the compiler's front end.

__FILE__

The preprocessor expands this macro into the current input file name as a string constant. The string matches the name of the file specified on the cc21k command-line or in a preprocessor `#include` command. (ANSI standard).

__LINE__

The preprocessor expands this macro into the current input line number as a decimal integer constant. (ANSI standard).

_NO_LONGLONG

cc21k always defines `_NO_LONGLONG` as 1.

__NO_BUILTIN

cc21k defines `__NO_BUILTIN` as 1 when you compile with the `-no-builtin` command-line switch.

__SIGNED_CHARS__

cc21k defines `__SIGNED_CHARS__` as 1. The macro is defined by default, but the compiler undefines this macro if you compile with the `-unsigned-char` command-line switch.

__STDC__

cc21k always defines `__STDC__` as 1, but the compiler undefines this macro if you compile with `-traditional`. (ANSI standard).

__STDC_VERSION__

cc21k always defines `__STD_VERSION__` as 199409L, but the compiler undefines this macro if you compile with `-traditional`. (ANSI standard).

__TIME__

The preprocessor expands this macro into the preprocessing time as a string constant. The date string constant takes the form `hh:mm:ss`. (ANSI standard).

Header Files

A header file contains C or C++ declarations and macro definitions. Use the `#include` preprocessor command to access header files for your program. Header file names have an `.h` or no extension. There are two main categories of header files:

- System header files declare the interfaces to the parts of the operating system. Include them in your program for the definitions and declarations you need to access system calls and libraries. Use angle brackets to indicate a system header file: `#include <file>`.
- User header files contain declarations for interfaces between the source files of your program. Use double quotes to indicate a user header file: `#include "file"`.

Writing Macros

A macro is a name standing for a block of text that the preprocessor substitutes. Use the `#define` preprocessor command to create a macro definition. When the macro definition has arguments, the block of text the preprocessor substitutes can vary with each new set of arguments.

Compound Statements as Macros. When writing macros, it can be useful to define a macro that expands into a compound statement. You can define such a macro so it can be invoked in the same way you would call a function, making your source code easier to read and maintain. The following two code segments define two versions of the macro `SKIP_SPACES`:

```
/* SKIP_SPACES, regular macro */
#define SKIP_SPACES ((p), limit)  \{
   char *lim = (limit); \
      while (p != lim)             { \
         if (*(p)++ != ' ')        { \
            (p)−; \
            break; \
         } \
      } \
} \

/* SKIP_SPACES, enclosed macro */
#define SKIP_SPACES (p, limit)  \
   do { \
      char *lim = (limit);      \
      while ((p) != lim)      { \
         if (*(p)++ != ' ')   { \
            (p)−;  \
            break; \
         }  \
      }  \
  } while (0)
```

Enclosing the first definition within the `do {…} while (0)` pair changes the macro from expanding to a compound statement to expanding to a single statement. With the macro expanding to a compound statement, you sometimes need to omit the semicolon after the macro call in order to have a legal program. This leads to a need to remember whether a function or macro is being invoked for each call and whether the macro needs a trailing semicolon or not. With the `do {…} while (0)` construct, you can pretend that the macro is a function and always put the semicolon after it. For example:

```
/* SKIP_SPACES, enclosed macro, ends without ‘;’ */
if (*p != 0)
   SKIP_SPACES (p, lim);
else …
```

This expands to:

```
if (*p != 0)
   do {
      …
  } while (0);  /* semicolon from SKIP_SPACES (…);  */
else …
```

Without the `do {…} while (0)` construct, the expansion would be:

```
if (*p != 0)
   {
      …
   }
;  /*  semicolon from SKIP_SPACES (…);  */
else
```

For more information on macros, see the *VisualDSP++ 2.0 Assembler and Preprocessor Manual for ADSP-21xxx DSPs*.

Support for Multiple Heaps

The ADSP-21xxx C/C++ run-time library supports the standard heap management functions `calloc`, `free`, `malloc`, and `realloc`. By default, these functions access the default heap, which is defined in the standard linker description file and the run-time header.

User written code can define any number of additional heaps, which can be located in any of the ADSP-21xxx memory blocks. These additional heaps can be accessed either by the standard `calloc`, `free`, `malloc`, and `realloc` functions, or via the Analog Devices extensions `heap_calloc`, `heap_free`, `heap_malloc`, and `heap_realloc`.

The primary use of alternate heaps is to allow dynamic memory allocation from more than one memory block. The ADSP-21xxx architecture allows two data accesses per cycle (in addition to a code access) if the memory locations are in different blocks.

Each heap used by a program is described by an entry in a heap descriptor table. The default heap is always described by the first entry in this table. User defined heaps are created by allocating a region of memory for the heap and by placing a descriptor entry for the heap in the heap descriptor table. See the section "Creating Heap Descriptor Records" on page 2-115 for details.

The default heap is initialized automatically if it is used by the program, but all user defined heaps must be initialized explicitly (with a call of the `heap_init`) function before they are used.

Heap Identifiers

All heaps have two identifiers. The primary heap ID is the index of the descriptor for that heap in the heap descriptor table. The primary heap ID of the default heap is always 0, and the primary IDs of user defined heaps will be 1, 2, 3, and so on.

Each heap also has a user ID. The user ID of a heap is specified in the heap descriptor for the heap. The user ID of the default heap is always 0, and user defined heaps must have distinct user IDs other than 0.

The only use for the heap user ID is to find the primary ID for a heap. The `heap_lookup` function takes the user ID as an argument and returns the primary identifier. All functions other than `heap_lookup` that take a heap ID argument must be given the primary ID, not the user ID.

Using Alternate Heaps with the Standard Interface

Alternate heaps can be accessed by the standard functions `calloc`, `free`, `malloc`, and `realloc`. The run-time library keeps track of a current heap, which initially is the default heap. The current heap can be changed any number of times at run time by calling the function `heap_switch`.

The standard functions `calloc` and `malloc` always allocate a new object from the current heap. If `realloc` is called with a null pointer, it too will allocate a new object from the current heap.

Previously allocated objects can be deallocated with `free` or `realloc`, or resized by `realloc`, even if the current heap is now different from when the object was originally allocated. When a previously allocated object is resized with `realloc`, the returned object will always be in the same heap as the original object.

Multithreaded programs (using VDK) cannot use `heap_switch` to change the current heap from the default. Such programs can access alternate heaps through the alternate interface described in the next section.

Using the Alternate Heap Interface

The C run-time library provides the alternate heap interface functions `heap_calloc`, `heap_free`, `heap_malloc`, and `heap_realloc`. These routines work exactly the same as the corresponding standard functions without the "`heap_`" prefix, except that they take an additional argument that specifies the heap ID. These functions are completely independent of the current heap setting.

Objects allocated with the alternate interface functions can be freed with either the `free` or `heap_free` (or `realloc` or `heap_realloc`) functions. The `heap_free` function is a little faster than `free` since it doesn't have to search for the proper heap. However, it is essential that the `heap_free` or `heap_realloc` functions be called with the same heap ID that was used to allocate the object being freed. If it is called with the wrong heap ID, the object won't be freed or reallocated.

The actual entry point names for the alternate heap interface routines have an initial underscore, that is, they are `_heap_lookup`, `_heap_switch`, `_heap_calloc`, `_heap_free`, `_heap_malloc`, and `_heap_realloc`. The `stdlib.h` standard header file defines macro equivalents without the leading underscores.

Re-initializing Heaps

Heaps can be re-initialized at any time by calling `heap_init`. This effectively invalidates the contents of any objects that have been allocated from that heap and not yet freed as well as all pointers to such objects. This is a very quick way to free all the objects currently allocated from the heap.

A heap should not be re-initialized unless it is known that the program will never again refer to any object currently allocated from that heap. Also, you should not re-initialize the default heap or any other heap that has been the current heap, since the run-time library or code generated by the compiler may have allocated objects from the current heap even in the absence of an explicit memory allocation call in the user's source code.

Creating Heap Descriptor Records

Every heap requires a heap descriptor, a heap storage area, and heap initialization. This section shows how to create heap descriptor records, while the following sections show how to allocate heap storage areas and how to initialize heaps.

The heap descriptor fields give the starting address, size, and user ID of the heap. The `_heap_descriptor_t` type declared in `stdlib.h` should be used for declaring a heap descriptor in a C or C++ program. Note that the size of a heap is measured in chars, the same unit of measure that the `sizeof` function uses.

The size of the heap descriptor table is fixed at link time, so the maximum number of different heaps that can be used at once is fixed when the program is built. However, the actual contents of a descriptor in this table are not referred to until the heap is initialized with a call of `heap_init`. Therefore, it is possible to delay specifying a heap's location and size until run time, as long as a heap descriptor for the heap has been placed in the heap descriptor table. This allows for techniques, such as, creating a new heap at run time from a storage area allocated from a different heap, or creating a heap from memory areas that are used for some other purpose in other phases of program execution. See Listing 2. on page 2-121 for examples.

The heap descriptor for an alternate heap can be placed in the heap descriptor table by editing a copy of the run-time header and placing the descriptor after the default heap descriptor, which immediately follows the `___heaptab_start` label. The default heap descriptor must always be first in the table.

It is probably more convenient to create the descriptor in C or C++ code as an instance of the `_heap_descriptor_t` struct. The segment (or section) directive is used to place the descriptor in an input section called "`heaptab`", and this causes the linker to place the descriptor in the heap descriptor table. See Listing 2. on page 2-121 for examples.

Allocating Heap Storage Areas

Heap storage areas do not have any alignment requirements; therefore, they can be located at any address. The `heap_init` function changes the start address and size in the heap descriptor if required to do so to satisfy alignment needs. Heap storage areas can be allocated either statically (at link time) or dynamically (at run time).

One way to create a static heap storage area is to imitate the method used to create the default heap. This procedure requires modifying the linker description file (LDF) and the run-time header. Briefly, the `MEMORY` section of the LDF declares a memory range, `seg_heap`, that is dedicated to the heap and an output section heap that is placed in `seg_heap`.

Two linker variables, `ldf_heap_space` and `ldf_heap_length`, are defined in the heap output section. These variables give the start address and size of `seg_heap`. At run-time startup, the heap descriptor is declared using `ldf_heap_space` (in source file `seg_init.asm`) and `ldf_heap_length` to initialize the start and size fields of the descriptor.

It is often more convenient to create heaps in C or C++ source code. Character arrays can be used for this purpose. An array declared outside a function, or inside a function using the static storage class, exists throughout the lifetime of the program. Therefore, such an array can be used as the storage area for a static heap.

Static heaps can be placed in particular memory blocks by declaring the storage area either with the `dm` and `pm` storage qualifiers or with the `segment` (or `section`) placement directive. The standard linker description file (LDF) causes variables declared with `dm` (the default) to be placed in block 1 and variables declared with `pm` to be placed in block 2. The directive `segment("seg_dmda")` is equivalent to using the qualifier `dm`, and `segment("seg_pmda")` is equivalent to using `pm`.

More generally, you can modify the standard LDF to allow the heap storage to be placed in any valid memory location (using the `segment` placement directive) by following the method used for locating variables placed in the `seg_dmda` or `seg_pmda` sections.

An array declared inside a function, but without the static storage class, exists only while that function is executing. Such an array can be used as the storage area for a dynamic heap. Note that this kind of dynamic heap can only be used while the function in which the heap storage was declared is still executing.

The storage for a dynamic heap can also be allocated from another heap, such as, the default heap or a different alternate heap that has already been initialized. This kind of dynamic heap can be used even after the function that allocated the heap storage has returned. Once such a heap's storage area has been deallocated, the heap can no longer be used.

See Listing 2-1 on page 2-121 for examples of static and dynamic storage areas.

Initializing Heaps

Heaps are usually initialized by a call of `heap_init` in the main function or in a function called by `main`.

Sometimes, however, it is necessary to initialize a heap before the main function is called by the run-time header (this is also true for the default heap). In C++, for example, you might have a class whose constructor allocates objects from an alternate heap. If your program has a static instance of this class, the C++ compiler ensures that the constructor for the object is called before the main function is called. You can't initialize the heap in the main function because this would be too late.

This problem can be solved by creating a simple initializer function dedicated to initializing that heap. A pointer to this initializer function is placed in an initializer table by using the `segment("ctor3")` placement directive. Use of the `ctor3` section name ensures that the heap's initializer function is called before any C++ constructors and after any initializers used by the run-time library.

Example C program

The C program in Listing 2-1 shows how to allocate and initialize alternate heaps. It creates three statically allocated heaps, two of which are in pm memory. It creates two dynamically allocated heaps, one from an automatic (stack) buffer and one from a buffer allocated from one of the statically allocated alternate heaps.

```
#include <stdio.h>
#include <stdlib.h>

// Define storage for three static heaps

static char heap1[1000]; // heap 1 is in block 1 (dm)
static pm char heap2[500]; // heap 2 is in block 2 (pm)
static segment("data2") char heap3[200];
                        // heap 3 also in block 2

// heap user IDs

#define UID1 10
#define UID2 20
#define UID3 30
#define UID4 40
```

```
static int my_heap_init(int heap_index)
{
   int ret = heap_init(heap_index);
   if (ret)
   printf("*** Failure initializing heap with index %d\n",
   heap_index);
  else
    printf("Success initializing heap with index %d\n",
heap_index);
  return ret;
}

// descriptor records for static heaps

segment("heaptab") _heap_descriptor_t heap1_dsc =
    {(dm char*)heap1, sizeof(heap1), UID1};
segment("heaptab") _heap_descriptor_t heap2_dsc =
    {(dm char*)heap2, sizeof(heap2), UID2};
segment("heaptab") _heap_descriptor_t heap3_dsc =
    {(dm char*)heap3, sizeof(heap3), UID3};

// descriptor record for dynamic heap

segment("heaptab") _heap_descriptor_t heap4_dsc ={0, 0, UID4};
static int heap1_stat = -10;

// initializer function for heap 1

static void init_heap1(void)
{
  heap1_stat = my_heap_init(heap_lookup(UID1));
}

static void auto_heap(void);
static void dynamic_heap(void);
static int index1, index2, index3;
```

```
int main()
{
  printf("first statement in main()\n");
  init_heap1();
  index1 = heap_lookup(UID1);
  index2 = heap_lookup(UID2);
  index3 = heap_lookup(UID3);
  if (!heap1_stat && !my_heap_init(index2) &&
!my_heap_init(index3)) {
    printf("main(): Now heaps 1, 2, and 3 can be used\n");
  }
  else {
    printf("main(): *** Problem initializing heaps 1, 2, or
3\n");
  }
  auto_heap();
  dynamic_heap();
  return 0;
}

static void auto_heap(void)
{ // storage for 4 heap is allocated on stack
  char buf[500];
  int index4 = heap_lookup(UID4);
  heap4_dsc.start = buf;
  heap4_dsc.size  = sizeof(buf);
  if (!my_heap_init(index4)) {
    printf("auto_heap(): Now heap 4 can be used\n");
  }
  else {
    printf("auto_heap(): *** Problem initializing heap 4\n");
  }
  // Heap 4 disappears when auto_heap() returns
}

static void dynamic_heap(void)
{ // storage for heap 4 is allocated from heap 1
  const int bufsize = 500;
  char *buf = heap_malloc(index1, bufsize);
  int index4 = heap_lookup(UID4);
  heap4_dsc.start = buf;
  heap4_dsc.size  = bufsize;
```

```
  if (buf && !my_heap_init(index4)) {
    printf("dynamic_heap(): Now heap 4 can be used\n");
  }
  else {
    printf("dynamic_heap(): *** Problem initializing heap
4\n");
  }
  if (buf) heap_free(index1, buf); // Return heap 4 storage to
heap 1
  // Heap 4 cannot be used after this point
}
```

Listing 2-1. Allocating and Initializing Alternate Heaps

C/C++ Run-Time Model

This section describes the conventions that you must follow as you write assembly code that can be linked with C or C++ code. The description of how C or C++ constructs appear in assembly language are also useful for low-level program analysis and debugging.

This section provides a full description of the ADSP-21xxx run-time model, including the layout of the stack, data access, and call/entry sequence.

This model applies to the compiler-generated code. Assembly programmers are encouraged to maintain stack conventions.

C/C++ Run-Time Environment

The C/C++ run-time environment is a set of conventions that C and C++ programs follow to run on ADSP-21xxx DSPs. Assembly routines that you link to C or C++ routines must follow these conventions.

Figure 2-1 on page 2-123 shows an overview of the run-time environment issues that you must consider as you write assembly routines that link with C/C++ routines. These issues include the following:

- Register usage conventions (see the following sections)

  "Compiler Registers" on page 2-129

  "Miscellaneous Information" on page 2-130

  "Call Preserved Registers" on page 2-131

  "Scratch Registers" on page 2-132

  "Stack Registers" on page 2-133

- Memory usage conventions (see the following sections)

  "Memory Usage" on page 2-123

  "Using Data Storage Formats" on page 2-143

- Program control conventions (see the following sections)

  "Managing the Stack" on page 2-135

  "Transferring Function Arguments and Return Value" on page 2-140

  "Using Macros to Manage the Stack" on page 2-167

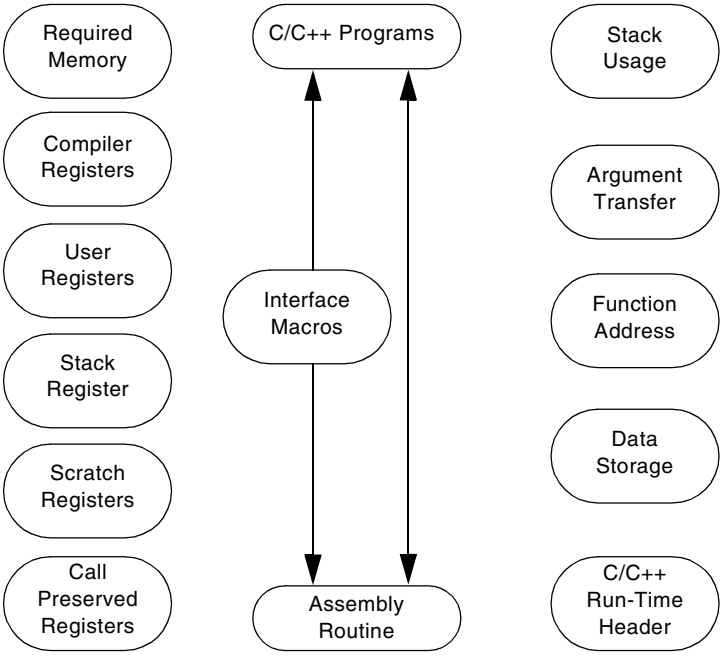


Figure 2-1. Assembly Language Interfacing Overview

Memory Usage

The cc21k C/C++ run-time environment requires that a specific set of memory section names be used for placing code in memory. In assembly language files, these names are used as labels for the `.SECTION` directive. In the linker description file, these names are used as labels for the output section names within the `SECTIONS{}` command. For information on syntax for the Linker Description File and other information on the linker, see the *VisualDSP++ 2.0 Linker and Utilities Manual for ADSP-21xxx DSPs*. Table 2-11 lists the memory section and output section names.

Because the compiler and linker must know the processor type to create code for the correct memory model, you must specify the processor for which you are developing. If you are using the VisualDSP++ IDDE, you specify the processor in the `Project Options` dialog box. If you are running the compiler from the command line, you specify the processor with a compiler switch. For more information on processor selection switches, see "C/C++ Compiler Common Switch Descriptions" on page 2-19.

Table 2-11. Memory .SECTION and SECTION{} Names

Names	Usage Description
`seg_pmco`	This section must be in Program Memory, holds code, and is required by some functions in the C/C++ run-time library. For more information, see "Program Memory Code Storage"on page 2-125.
`seg_dmda`	This section must be in Data Memory, is the default location for global and static variables and string literals, and is required by some functions in the C/C++ run-time library. For more information, see "Data Memory Data Storage"on page 2-125.
`seg_pmda`	This section must be in PM, holds PM data variables, and is required by some functions in the C/C++ run-time library. For more information, see "Program Memory Data Storage"on page 2-125.
`seg_stak`	This section must be in DM, holds the run-time stack, and is required by the C/C++ run-time environment. For more information, see "Run-Time Stack Storage"on page 2-126.
`seg_heap`	This section must be in DM, holds the default run-time heap, and is required by the C/C++ run-time environment. For more information, see "Run-Time Heap Storage"on page 2-126.

Table 2-11. Memory .SECTION and SECTION{} Names (Cont'd)

Names	Usage Description
`seg_init`	This section must be in PM, holds system initialization data, and is required for system initialization. For more information, see "Initialization Data Storage"on page 2-127.
`seg_rth`	This section must be in the interrupt table area of PM, holds system initialization code and interrupt service routines, and is required for system initialization. For more information, see "Run-Time Header Storage"on page 2-128.

Program Memory Code Storage. The Program Memory code section, `seg_pmco`, is where the compiler puts all the program instructions that it generates when you compile your program. When linking, use your linker description file to map this section to Program Memory space.

Data Memory Data Storage. The Data Memory data section, `seg_dmda`, is where the compiler puts global and static data in Data Memory. When linking, use your linker description file to map this section to Data Memory space.

By default, the compiler stores static variables in the Data Memory data section. The compiler's `dm` and `pm` keywords (memory type qualifiers) let you override this default. If a memory type qualifier is not specified, the compiler places static and global variables in Data Memory. For more information on type qualifiers, see "Dual Memory Support Keywords (pm dm)" on page 2-68. The following example allocates an array of 10 integers in the Data Memory data section:

```
static int data [10];
```

Program Memory Data Storage. The Program Memory data section, `seg_pmda`, is where the compiler puts global and static data in Program Memory. When linking, use your linker description file to map this section to Program Memory space.

By default, the compiler stores static variables in the Data Memory data section. The compiler's `pm` keyword (memory type qualifier) lets you override this default and place variables in the Program Memory data section. If a memory type qualifier is not specified, the compiler places static and global variables in Data Memory. For more information on type qualifiers, see "Dual Memory Support Keywords (pm dm)" on page 2-68. The following example allocates an array of 10 integers in the Program Memory data section:

```
static int pm coeffs[10];
```

Run-Time Stack Storage. The run-time stack section, `seg_stak`, is where the compiler puts the run-time stack in Data Memory. When linking, use your linker description file to map this section to Data Memory space. Because the run-time environment cannot function without this section, you must define it, and the section must be in Data Memory space. A typical size for the run-time stack is 4K 32-bit words of Data Memory.

The run-time stack is a 32-bit wide structure, growing from high memory to low memory. The compiler uses the run-time stack as the storage area for local variables and return addresses. During a function call, the calling function pushes the return address onto the stack. For more information, see "Managing the Stack" on page 2-135.

Run-Time Heap Storage. The run-time heap section, `seg_heap`, is where the compiler puts the run-time heap in Data Memory. When linking, use your Linker Description File (`.ldf`) to map the `seg_heap` section to Data Memory space. A typical size for the run-time heap is 60K 32-bit words of Data Memory.

To dynamically allocate and deallocate memory at run-time, the C or C++ run-time library includes five functions: `malloc`, `calloc`, `realloc` and `free`. These functions allocate memory from the `seg_heap` section of memory by default.

The run-time library also provides support for multiple heaps, which allow dynamically allocated memory to be located in different blocks. See "Support for Multiple Heaps" on page 2-111 for more information on the use of multiple heaps.

The Linker Description File requires the `seg_heap` declaration for every DSP project whether a program dynamically allocates memory at run-time or not.

Initialization Data Storage. The initialization section, `seg_init`, is where the compiler puts the initialization data in Program Memory. When linking, use your linker description file to map this section to Program Memory space.

The initialization section may be processed by two different utility programs: `mem21k` or `elfloader`.

- If you are producing boot-loadable executable file for your DSP system, you should use the `elfloader` utility to process your executable. The `elfloader` utility processes your executable file, producing an ADSP-2106x boot-loadable file which you can use to boot a target hardware system and initialize its memory.

  The boot loader, `elfloader`, operates on the executable file produced by the linker. When you run `elfloader` as part of the compilation process (using the `-no-mem` switch), the linker (by default) creates a `*.dxe` file for processing with `elfloader`.

  When preparing files for the `ld21k` loader, the system configuration file's `seg_init` section needs only 16 slots/locations of space.

- If you are producing an executable file that is not going to be boot loaded into the DSP (almost exclusively an ADSP-21020 issue), you should use the `mem21k` utility to process your executable. The `mem21k` utility processes your executable file, producing an optimized executable file in which all RAM memory initialization is stored in the `seg_init` PM ROM section. This optimization has the advantage of initializing all RAM to its proper value before the call to `main()` and reducing the size of an executable file by combining contiguous, identical initializations into a single block.

  The memory initializer, `mem21k`, operates on the executable file produced by the linker. When you run `mem21k` as part of the compilation process, the linker (by default) creates a `*.lnk` file for processing with `mem21k`.

  The C run-time header reads the `seg_init` section generated by `mem21k` to determine which memory locations should be initialized to what values. This process occurs during the `__lib_setup_processor` routine that is called from the run-time header.

Run-Time Header Storage. The run-time header section, `seg_rth`, is where the compiler puts the system initialization code and interrupt table in Program Memory. When linking, use your linker description file to map this section to the interrupt vector table area of Program Memory space.

If you do not specify a run-time header file, the compiler uses a default run-time header from the `21k\lib` or `211xx\lib` directory. The default header file is `060_hdr.doj` in the `21k\lib` directory for the `-21060`, `-21061`, `-21062`, `-21065L` switches or `160_hdr.doj` in the `211xx\lib` directory for the `-21160` and `-21161` switch. Note that if the compiler finds a copy of `xxx_hdr.obj` in the current directory, the compiler uses the copy instead of the default from the `21k\lib` or `211xx\lib` directory.

The source files for many run-time header files (including `060_hdr.asm` and `160_hdr.asm`) come with the development tools package. Keep the following points in mind if you prefer to write your own interrupt handlers in C or C++:

- Note that the library functions `signal`, `raise`, `interrupt`, and their variants are based on the run-time header used.
- Note that on both the ADSP-2106x and ADSP-2116x, each interrupt is allocated four words.

Compiler Registers

The cc21k C/C++ run-time environment reserves a set of registers for its own use. Table 2-12 lists these registers and the values the C/C++ run-time environment expects to be in them. Do not modify these registers, except as noted in the table.

Table 2-12. Compiler Registers

Register	Value	Modification Rules
m5, m13	0	Do not modify
m6, m14,	1	Do not modify
m7, m15	-1	Do not modify
b6, b7	stack base	Do not modify
l6, l7	stack length	Do not modify
l0, l1, l2, l3, l4, l5, l8, l9, l10, l11, l12, l13, l14, l15	0	Modify for temporary use, restore when done

Miscellaneous Information

The following is some miscellaneous information that you might find helpful in understanding register functionality:

- All of the L registers, except `L6` and `L7`, are required to be zero at any call/return point.
- When you either make a function call or return to your caller and have modified any of the L registers, you must reset them to zero.
- Interrupt routines must save and set to zero the L register before using its corresponding I register for any post-modify instruction.

User Registers

The `-reserve` command-line switch lets you reserve registers for your inline assembly code or assembly language routines. If reserving an L register, you must reserve the corresponding I register; reserving an L register without reserving the corresponding I register can result in execution problems.

You must reserve the same list of registers in all linked files; the whole project must use the same `-reserve` option. Table 2-13 lists these registers. Note that the C run-time library does not use these registers.

Table 2-13. User Registers

Register	Value	Modification Rule
i0, b0, l0, m0, i1, b1, l1, m1, i8, b8, l8, m8, i9, b9, l9, m9, mrb, ustat1, ustat2	user defined	If not reserved, modify for temporary use, restore when done If reserved, usage is not limited

When you reserve a register, you are asking the compiler to avoid using the register. If the compiler requires a register you have reserved, the compiler ignores your reservation request. Reserving registers can negatively influence the efficiency of compiled C or C++ code; use this option infrequently.

Call Preserved Registers

The cc21k C/C++ run-time environment specifies a set of registers whose contents must be saved and restored. Your assembly function must save these registers during the function's prologue and restore the registers as part of the function's epilogue. These registers must be saved and restored if they are modified within the assembly function; if a function does not change a particular register, it does not need to save and restore the register. Table 2-14 lists these registers.

Table 2-14. Call Preserved Registers [1]

b0	b1	b2	b3	b5	b8
b9	b10	b11	b14	b15	
i0	i1	i2	i3	i5	i8
i9	i10	i11	i14	i15	mode1
mode2	mrb	mrf	m0	m1	m2
m3	m8	m9	m10	m11	r3
r5	r6	r7	r9	r10	r11
r13	r14	r15	ustat1	ustat2	

[1] If you use a call preserved I register, you must save and zero (clear) the corresponding L register as part of the function prologue. Then, restore the L register as part of the function epilogue.

Many functions in the C/C++ run-time library expect the processor to be in a specific mode and may not operate correctly if the processor is in a different mode. If you need to change processor modes, save the old values in the `mode1` and `mode2` registers and restore these registers before calling or returning to calling functions. The C/C++ run-time environment operates in the following mode:

- Uses default bit order for DAG operations (no bit reversal)
- Uses the primary register set (not background set)
- Uses `.PRECISION=32` (32-bit floating-point) and `.ROUND_NEAREST` (round-to-nearest value)
- Disables ALU saturation (`mode1` register, `ALUSAT` bit = 0)
- Uses default FIX instruction rounding to nearest (`MODE1` register, `TRUNCATE`=0)

Scratch Registers

The cc21k C/C++ run-time environment specifies a set of registers whose contents do not need to be saved and restored. Note that the contents of these registers are not preserved across function calls. Table 2-15 lists these registers.

Table 2-15. Scratch Registers

b4	b12	b13	r0	r1	r2	r4	r8	r12
i4	i12	m4	m12	i13				

In addition, for ADSP-2116x DSPs the PEy data registers are all scratch registers. Table 2-16 lists these registers.

Table 2-16. Additional ADSP-2116x scratch Registers

s0	s1	s2	s3	s4	s5	s6	s7	s8
s9	s10	s11	s12	s13	s14	s15	USTAT2	USTAT4
ASTATy	STKy							

Stack Registers

The cc21k C/C++ run-time environment reserves a set of registers for controlling the run-time stack. These registers may be modified for stack management, but they must be saved and restored. Table 2-17 lists these registers.

Table 2-17. Pointer Registers

Register	Value	Modification Rules
i7	Stack pointer	Modify for stack management, restore when done
i6	Frame pointer	Modify for stack management, restore when done
i12	Return address	Load with function call return address on function exit

Alternate Registers

The C/C++ run-time environment model does not use any of the alternate registers because these registers are available for use in assembly language only. To use these registers, several aspects of the C/C++ run-time model must be understood.

The C/C++ run-time model uses register `I6` as the frame pointer and register `I7` as the stack pointer. Setting the DAG register that contains `I6` and `I7` from a background register to an active register directly affects the stack operation. The C/C++ run-time model does not have an understanding of background registers. If the background `I6` and `I7` registers are active and an interrupt occurs, the C/C++ run-time model still uses `I6` and `I7` to update the stack. This results in faulty stack handling.

The background register set containing DAG registers `I6` and `I7` should only be used in assembly routines if interrupts are not enabled.

The super fast interrupt dispatcher uses context switching rather than saving registers on the run-time stack. To ensure no register conflicts, do not use the super fast interrupt dispatcher or disable interrupts when using secondary registers in an assembly routine.

Managing the Stack

The cc21k C/C++ run-time environment uses the run-time stack for storage of automatic variables and return addresses. The stack is managed by a frame pointer and a stack pointer and grows downward in memory, moving from higher to lower addresses.

A stack frame is a section of the stack used to hold information about the current context of the C or C++ program. Information in the frame includes local variables, compiler temporaries, and parameters for the next function.

The frame pointer serves as a base for accessing memory in the stack frame. Routines refer to locals, temporaries, and parameters by their offset from the frame pointer.

Figure 2-2 on page 2-136 shows an example section of a run-time stack. In the figure, the currently executing routine, `Current()`, was called by `Previous()`, and `Current()` in turn calls `Next()`. The state of the stack is as if `Current()` has pushed all the arguments for `Next()` onto the stack and is just about to call `Next()`.

Stack usage for passing any or all of a function's arguments depends on the number and types of parameters to the function.

The prototypes for the functions in Figure 2-2 are as follows:

```
void Current(int a, int b, int c, int d, int e);
void Next(int v, int w, int x, int y, int z);
```

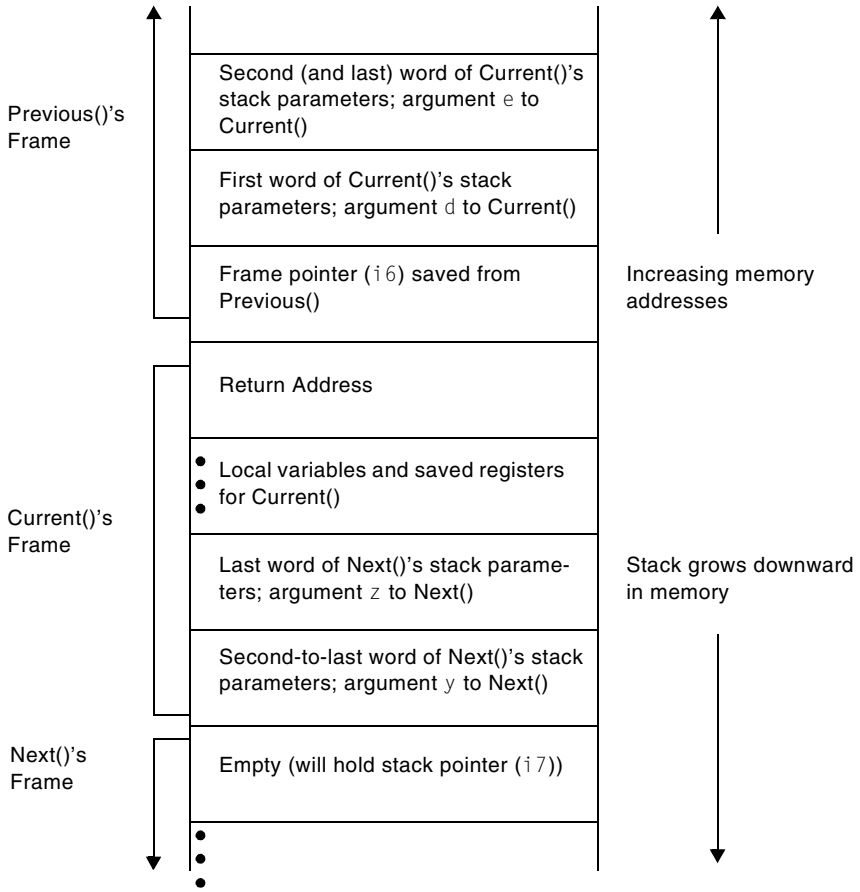


Figure 2-2. Example Run-Time Stack

In generating code for a function call, the compiler produces the following operations to create the called function's new stack frame:

1. Loads the `r2` register with the frame pointer (in the `i6` register)
2. Sets the frame pointer, `i6` register, equal to the stack pointer (in the `i7` register)
3. Uses the delayed-branch instruction to pass control to the called function
4. Pushes the frame pointer, `r2`, onto the run-time stack during the first branch delay slot
5. Pushes the return address, `pc`, onto the run-time stack during the second delay branch slot

For ADSP-2106x and ADSP-2116x DSPs, the following instructions create a new stack frame. Note how the two initial register moves are incorporated into the `cjump` instruction:

```
cjump my_function (DB);
   /* where my_function is the called function */
dm(i7, m7) = r2;
dm(i7, m7) = pc;
```

As you write assembly routines, note that the operations to create a stack frame are the responsibility of the called function, and you can use the `entry` or `leaf_entry` macros to perform these operations. For more information on these macros, see "Using Mixed C/C++ and Assembly Support Macros" on page 2-153.

In generating code for a function return, the compiler produces the following operations to restore the calling function's stack frame:

1. Pops the return address off the run-time stack and loads it into the `i12` register
2. Uses the delayed-branch instruction to pass control to the calling function and jumps to the return address (`i12 + 1`)
3. Restores the caller's stack pointer, `i7` register, by setting it equal to the frame pointer, `i6` register, during the first branch delay slot
4. Restores the caller's frame pointer, `i6` register, by popping the previously saved frame pointer off the run-time stack and loading the value into `i6` during the second delay branch slot

For ADSP-2106x DSPs, the following instructions return from the function and restore the stack and frame pointers. Note that the restoring of the stack pointer and frame pointer are incorporated into the `rframe` instruction:

```
i12 = dm(-1, i6);
jump (m14, i12) (DB);
nop;
rframe;
```

As you write assembly routines, note that the operations to restore stack and frame pointers are the responsibility of the called function, and you can use the `exit` or `leaf_exit` macros to perform these operations. For more information on these macros, see "Using Mixed C/C++ and Assembly Support Macros" on page 2-153.

In the following code examples (Listing 2-2 and Listing 2-3), observe how the function calls in the C code translate to stack management tasks in the compiled (assembly) version of the code. The comments have been added to the compiled code to indicate the function prologue and function epilogue.

```
/* Stack management — C code */

int my_func(int, int);
int arg_a, return_c;

main()
{
   static int arg_b;
   arg_b = 0;
   return_c = my_func(arg_a, arg_b);
}

int my_func(int arg_1, int arg_2)
{
   return (arg_1 + arg_2)/2;
}
```

Listing 2-2. Stack Management, Example C Code

```
/* Stack management — C compiled (2106x assembly) code */
.section /pm seg_pmco;
.global _main;
_main:
    .def end_prologue; .val .; .scl 109; .endef;
  r4=dm(_arg_a);
  /* r4, the first argument register, which is arg_a */
  r8=0;
  /* r8, the second argument register, which is arg_b */
  dm(arg_b)=r8;

/* The next three lines are the function call sequence */
  cjump (pc,_my_func) (DB);
  dm(i7,m7)=r2;
  dm(i7,m7)=pc;

  dm(_return_c)=r0;

  /* The next four lines are main's function epilogue */
  i12=dm(-1,i6);
  jump (m14, i12) (DB);
  nop;
  rframe;
```

```
.global _my_func;
_my_func:
  .def end_prologue; .val .; .scl 109; .endef;
  r0=(r4+r8)/2;

  /* The next four lines are my_func's function epilogue */
  i12=dm(-1,i6);
  jump (m14, i12) (DB);
  nop;
  rframe;
.endseg;
```

Listing 2-3. Stack Management, Example ADSP-2106x Assembly Code

The next two sections, "Transferring Function Arguments and Return Value" on page 2-140 and "Using Macros to Manage the Stack" on page 2-167, provide additional detail on function call requirements.

Transferring Function Arguments and Return Value

The C/C++ run-time environment uses a set of registers and the run-time stack to transfer function parameters to assembly routines. Your assembly language functions must follow these conventions when they call or when they are called by C or C++ functions.

Because it is most efficient to use registers for passing parameters, the run-time environment attempts to pass the first three parameters in a function call using registers; it then passes any remaining parameters on the run-time stack.

The convention is to pass the function's first parameter in `r4`, the second parameter in `r8`, and the third parameter in `r12`. The following exceptions apply to this convention:

- If any parameter is larger then a single 32-bit word, then that parameter and all subsequent parameters are passed on the stack.
- If the function is declared to take a variable number of arguments (has ... in its prototype), then the last named parameter and any subsequent parameters are passed on the stack.

Table 2-18 lists the rules that cc21k uses for passing parameters in registers to functions and the rules that your assembly code must use for returns.

Table 2-18. Parameter and Return Value Transfer Registers

Register	Parameter Type Passed Or Returned
r4	Pass first 32-bit data type parameter
r8	Pass second 32-bit data type parameter
r12	Pass third 32-bit data type parameter
stack	Pass fourth and remaining parameters; see exceptions to this rule on page 2-141
r0	Return `int`, `long`, `char`, `float`, `short`, `pointer`, and one-word structure parameters
r0, r1	Return long double and two-word structure parameters. Place MSW in r0 and LSW in r1
r1	Return the address of results that are longer than two words; r1 contains the first location in the block of memory containing the results

Consider the following function prototype example:

```
pass(int a, float b, char c, float d);
```

The first three arguments, `a`, `b`, and `c` are passed in registers `r4`, `r8`, and `r12`, respectively. The fourth argument, `d`, is passed on the stack.

This next example illustrates the effects of passing doubles.

```
count(int w, long double x, char y, float z);
```

The first argument, `w`, is passed in `r4`. Because the second argument, `x`, is a multi-word argument, `x` is passed on the stack. As a result, the remaining arguments, `y` and `z`, are also passed on the stack.

The following illustrates the effects of variable arguments on parameter passing:

```
compute(float k, int l, char m,…);
```

Here, the first two arguments, `k` and `l`, are passed in registers `r4` and `r8`. Because `m` is the last named argument, `m` is passed on the stack, as are all remaining variable arguments.

When arguments are placed on the stack, they are pushed on from right to left. The right-most argument is at a higher address than the left-most argument passed on the stack.

The following example shows how to access parameters passed on the stack:

```
tab(int a, char b, float c, int d, int e, long double f);
```

Parameters `a`, `b`, and `c` are passed in registers because they are single-word parameters. The remaining parameters, `d`, `e`, and `f`, are passed on the stack.

All parameters passed on the stack are accessed relative to the frame pointer, register `i6`. The first parameter passed on the stack, `d`, is at address `i6 + 1`. To access it, you could use this assembly language statement:

```
r3=dm(1,i6);
```

The second parameter passed on the stack, `e`, is at `i6 + 2` and can be accessed by the statement:

```
r3=dm(2,i6);
```

The third parameter passed on the stack, `f`, is a `long double` that has its most significant word at `i6 + 3` and its least significant word at `i6 + 4`. The most significant word of `f` can be accessed by the statement:

```
r3=dm(3,i6);
```

Using Data Storage Formats

The C/C++ run-time environment uses the data formats that appear in the Table 2-19, Table 2-20, Figure 2-3, and Figure 2-4.

Table 2-19. Data Storage Formats and Data Type Sizes

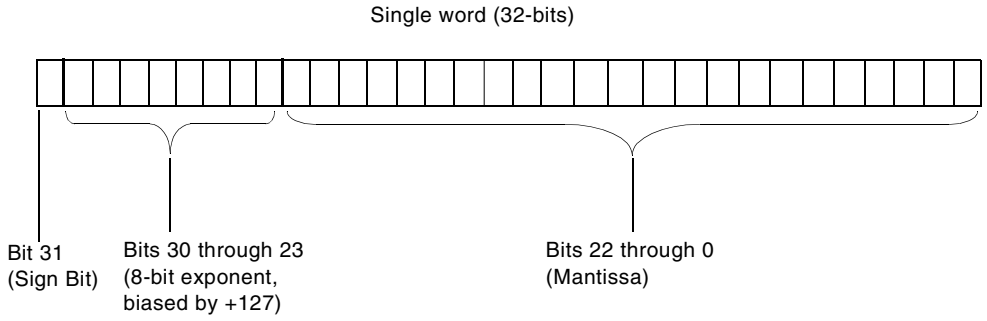
Applied Type	Number Representation
int	32-bit two's complement
long int	32-bit two's complement
short int	32-bit two's complement
unsigned int	32-bit unsigned magnitude
unsigned long int	32-bit unsigned magnitude
char	32-bit two's complement

Table 2-19. Data Storage Formats and Data Type Sizes (Cont'd)

Applied Type	Number Representation
unsigned char	32-bit unsigned magnitude
float	32-bit IEEE single-precision
double	32-bit IEEE single-precision or 64-bit IEEE double-precision if you compile with the -double-size-64 switch
long double	64-bit IEEE double-precision

Table 2-20. Data Storage Formats and Data Storage

Data	Big Endian Storage Format
long double	Writes 64-bit IEEE double-precision data with the most significant word closer to address 0x0000, proceeds toward the top of memory with the rest (see Figure 2-4 for details).



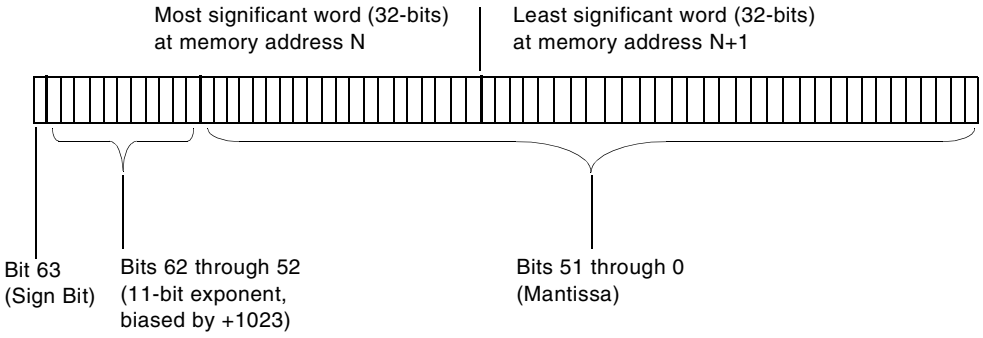
The single word (32-bit) data storage format equates to:

$$-1^{Sign} \times 1.Mantissa \times 2^{(Exponent - 127)}$$

Where:

Sign	Comes from the Sign Bit
Mantissa	Represents the fractional part of the Mantissa (23-bits). The 1. is assumed in this format.
Exponent	Represents the 8-bit exponent

Figure 2-3. Floating-Point (32-Bit IEEE Single-Precision) Storage



The two word (64-bit) data storage format equates to:

$$-1^{Sign} \times 1.Mantissa \times 2^{(Exponent\ -1023)}$$

Where:

Sign	Comes from the Sign Bit
Mantissa	Represents the fractional part of the Mantissa (52-bits). The 1. is assumed in this format.
Exponent	Represents the 11-bit exponent

Figure 2-4. Floating-Point (64-Bit IEEE Double-Precision) Storage

Using the Run-Time Header

The run-time header is an assembly language procedure that initializes the processor and sets up processor features to support the C/C++ run-time environment. The source code for the default run-time header is in the `060_hdr.asm` file for ADSP-2106x DSPs and in the `160_hdr.asm` file for ADSP-2116x DSPs. This run-time header performs the following operations:

- Initializes the C/C++ run-time environment
- Sets up the interrupt table
- Calls your `main()` routine

C/C++ and Assembly Interface

This section describes how to call assembly language subroutines from within C or C++ programs and how to call C or C++ functions from within assembly language programs. Before attempting to do either of these calls, be sure to familiarize yourself with the information about the C/C++ run-time model (including details about the stack, data types, and how arguments are handled) in "C/C++ Run-Time Environment" on page 2-122. At the end of this reference, a series of examples demonstrate how to mix C/C++ and assembly code.

Calling Assembly Language Subroutines from C/C++ Programs

Before calling an assembly language subroutine from a C or C++ program, you should create a prototype to define the arguments for the assembly language subroutine and the interface from the C or C++ program to the assembly language subroutine. You can legally use a function without a prototype in C. However, using prototypes is strongly recommended for good software engineering. When the prototype is omitted, the compiler cannot do argument type-checking and assumes that the return value is of type integer.

The compiler prefaces the name of any external entry point with an underscore. You should either declare your assembly language subroutine's name with a leading underscore or define it within an `'extern "asm" {}'` format to tell the compiler that it is an assembly language subroutine.

The run-time model defines some registers as *scratch* registers and others as *preserved* or *dedicated* registers. The scratch registers can be used within the assembly language program without worrying about their previous contents. If you need more room (or are working with existing code) and wish to use the preserved registers, you **must** first **save** their contents and then **restore** those contents before returning. Do **not** use the dedicated registers for other than their intended purpose; the compiler, libraries, interrupt routines, debugger, and interrupt routines all depend on having a stack available as defined by those registers.

The compiler also assumes that the machine state does not change during execution of the assembly language subroutine.

Do not change any machine modes (for example, the machine may have an integer/fractional mode, or it may use certain registers to indicate circular buffering when those register values are zero).

If arguments are on the stack, they are addressed via an offset from the stack pointer or frame pointer.

A good way to explore how arguments are passed between a C/C++ program and an assembly language subroutine is to write a dummy function in C/C++ and compile it with the `save temporary files` option (the `-save-temps` command-line option). The following example includes the global volatile variable assignments to indicate where the arguments can be found upon entry to `asmfunc`.

```
// Sample file for exploring compiler interface …
  // global variables … assign arguments there just so
  // we can track which registers were used
  // (type of each variable corresponds to one of arguments):

   int global_a;
   float global_b;
   int * global_p;

// the function itself:

int asmfunc(int a, float b, int * p)
{
  // do some assignments so .s file will show where args are:
  global_a = a;
  global_b = b;
  global_p = p;

  // value gets loaded into the return register:
  return 12345;
}
```

When compiled with the `-save-temps -O0` option set, this produces the following:

```
// PROCEDURE: asmfunc
.global _asmfunc;

_asmfunc:
     modify(i7,-7);
     dm(-8,i6)=r3;
     dm(-7,i6)=r6;
     r2=i0;
     dm(-6,i6)=r2;
     dm(-4,i6)=r4;
     dm(-3,i6)=r8;
     dm(-2,i6)=r12;
     r3=r4;
     r6=r8;
     i0=r12;
     dm(_global_a)=r3;
     dm(_global_b)=r6;
     r2=i0;
     dm(_global_p)=r2;
     r0=12345;
```

Calling C/C++ Functions from Assembly Language Programs

You may want to call C or C++-callable library and other functions from within an assembly language program. As discussed in "Calling Assembly Language Subroutines from C/C++ Programs" on page 2-148, you may wish to create a test function to do this in C or C++, and then use the code generated by the compiler as a reference when creating your assembly language program and the argument setup. Using volatile global variables may help clarify the essential code in your test function.

The run-time model defines some registers as *scratch* registers and others as *preserved* or *dedicated*. The contents of the scratch registers may be changed without warning by the called C/C++ function; if the assembly language program needs the contents of any of those registers, you **must** first **save** their contents before the call to the C/C++ function and then **restore** those contents after returning from the call.

Do **not** use the dedicated registers for other than their intended purpose; the compiler, libraries, interrupt routines, debugger, and interrupt routines all depend on having a stack available as defined by those registers.

Preserved registers can be used; their contents will not be changed by calling a C/C++ function. The function will always save and restore the contents of preserved registers if they are going to change.

If arguments are on the stack, they are addressed via an offset from the stack pointer or frame pointer. You can explore how arguments are passed between an assembly language program and a function by writing a dummy function in C or C++ and compiling it with the `save temporary files` option (the `-save-temps` command line option). By examining the contents of volatile global variables in `*.s` file, you can determine how the C/C++ function passes arguments and then duplicate that argument setup process in the assembly language program.

The stack must be set up correctly before calling a C/C++-callable function. If you call other functions, maintaining the basic stack model also facilitates the use of the debugger. The easiest way to do this is to define a C or C++ main program to initialize the run-time system; hold the stack until it is needed by the C/C++ function being called from the assembly language program; and then hold that stack until it is needed to call back into C/C++, making sure the dedicated registers are correct. You do not need to set the `FP` prior to the call; the caller's `FP` is never used by the callee.

The following example demonstrates the features described in this section. Because so many different features are combined into a single example, this procedure as a whole should not be viewed as an example of good assembly programming.

```
// PROCEDURE: memalloc
.global _memalloc;
_memalloc:
  r5=0xffff;        // Assign a value to preserved reg r5
     r8=0xffff;     // Assign a value to scratch reg r8

     r0=dm(-3,i6);  // Read a value from the stack
     r4=r0;         // Put this value in a parameter register

     // Save value of scratch register prior to function call
     r7=r8;

     // Call the C function malloc()
     r2=i6;
     i6=i7;
     jump _malloc (DB);
     dm(i7,m7)=r2;
     dm(i7,m7)=pc;

     // Check the result of the function call
     r0=pass r0;
     if eq jump(pc,_error);

     // Check that the preserved register did not change over
     // the function call
     r4=0xffff;
     comp(r4,r5);
     if ne jump(pc, _error);

     // Restore value of scratch register after function call
     r8=r7;

     i6 = 0x123;    // PROGRAMMING ERROR! Do not change
                    // dedicated registers
     rts;
```

Using Mixed C/C++ and Assembly Support Macros

This section lists, describes, and shows syntax for the C/C++ and assembly interface support macros in the `asm_sprt.h` system header file. Use these macros for interfacing assembly language modules with C or C++ functions. Table 2-21 lists the macros.

Although the syntax for each macro does not change, the listing of `asm_sprt.h` in this section may not be the most recent version. To see the current version, check the `asm_sprt.h` file that came with your software package.

Table 2-21. Interface Support Macros, Summary

`entry`	`exit`	`leaf_entry`	`leaf_exit`
`ccall(x)`	`reads(x)`	`puts`	`gets(x)`
`alter(x)`	`save_reg`	`restore_reg`	

Interface Support Macros, Defined

entry

The `entry` macro expands into the function prologue for non-leaf functions. This macro should be the first line of any non-leaf assembly routine. Note that this macro is currently null, but it should be used for future compatibility.

exit

The `exit` macro expands into the function epilogue for non-leaf functions. This macro should be the last line of any non-leaf assembly routine. Exit is responsible for restoring the caller's stack and frame pointers and jumping to the return address. Note that this macro is currently null, but it should be used for future compatibility.

leaf_entry

The `leaf_entry` macro expands into the function prologue for leaf functions. This macro should be the first line of any non-leaf assembly routine. Note that this macro is currently null, but it should be used for future compatibility.

leaf_exit

The `leaf_exit` macro expands into the function epilogue for non-leaf functions. This macro should be the last line of any leaf assembly routine. `leaf_exit` is responsible for restoring the caller's stack and frame pointers and jumping to the return address.

ccall(x)

The `ccall` macro expands into a series of instructions that save the caller's stack and frame pointers and then jump to function `x()`.

reads(x)

The `reads` macro expands into an instruction that reads a value from the stack. The value is located at an offset `x` from the frame pointer.

puts=x

The `puts` macro expands into an instruction that pushes the value in register `x` onto the stack.

gets(x)

The `gets` macro expands into an instruction that pops a value off of the stack and puts the value in the indicated register:

```
register = gets(x);
```

The value is located at an offset `x` from the stack pointer.

alter(x)

The `alter` macro expands into an instruction that adjusts the stack pointer by adding the immediate value `x`. With a positive value for `x`, `alter` pops x words from the top of the stack. You could use `alter` to clear `x` number of parameters off the stack after a call.

save_reg

The `save_reg` macro expands into a series of instructions that push the register file registers (`r0-r15`) onto the run-time stack.

restore_reg

The `restore_reg` macro expands into a series of instructions that pop the register file registers (`r0-r15`) off of the run-time stack.

```
/* asm_sprt.h — C/C++/Assembly Interface Support Macros */
/* asm_sprt.h - $Date: 10/09/97 6:28p $ */

#ifndef __ASM_SPRT_DEFINED
#define __ASM_SPRT_DEFINED

#define entry/* nothing */
#define leaf_entry/* nothing */

#ifdef __ADSP21020__
#define ccall(x) \
   r2=i6; i6=i7; \
   jump (pc, x) (db); \
   dm(i7,m7)=r2; \
   dm(i7,m7)=PC
#define leaf_exit \
   i12=dm(m7,i6);\
   jump (m14,i12) (db); \
   i7=i6; i6=dm(0,I6)
#define exit leaf_exit
#else
```

```
#define ccall(x) \
   cjump (x) (DB); \
   dm(i7,m7)=r2; \
   dm(i7,m7)=PC

#define leaf_exit \
   i12=dm(m7,i6); \
   jump (m14,i12) (db); \
   nop; \
      RFRAME
#define exit leaf_exit
#endif

#define reads(x)dm(x, i6)
#define putsdm(i7, m7)
#define gets(x)dm(x, i7)
#define alter(x) modify(i7, x)

#define save_reg \
       puts=r0;\
       puts=r1;\
       puts=r2;\
       puts=r3;\
       puts=r4;\
       puts=r5;\
       puts=r6;\
       puts=r7;\
       puts=r8;\
       puts=r9;\
       puts=r10;\
       puts=r11;\
       puts=r12;\
       puts=r13;\
       puts=r14;\
       puts=r15

#define restore_reg \
       r15=gets(1);\
       r14=gets(2);\
       r13=gets(3);\
       r12=gets(4);\
       r11=gets(5);\
       r10=gets(6);\
       r9 =gets(7);\
```

```
        r8 =gets(8);\
        r7 =gets(9);\
        r6 =gets(10);\
        r5 =gets(11);\
        r4 =gets(12);\
        r3 =gets(13);\
        r2 =gets(14);\
        r1 =gets(15);\
        r0 =gets(16);\
    alter(16)

#endif
```

Listing 2-4. asm_sprt.h — C/C++/Assembly Interface Support Macros

Using Mixed C/C++ and Assembly Naming Conventions

It is necessary to be able to use C or C++ symbols (function or variable names) in assembly routines and use assembly symbols in C or C++ code. This section describes how to name and use C/C++ and assembly symbols.

To name an assembly symbol that corresponds to a C or C++ symbol, add an underscore prefix to the C/C++ symbol name when declaring the symbol in assembly. For example, the C/C++ symbol `main` becomes the assembly symbol `_main`.

To use a C function or variable in an assembly routine, declare it as global in the C program. Import the symbol into the assembly routine by declaring the symbol with the `.EXTERN` assembler directive.

The external name of a C++ function encodes information about its type and parameters. Function "signature" enables the overloading of functions and operators that C++ language supports. To reference a function in a C++ module, declare it with the `extern "C"` specifier in order to use the naming convention of C. Note that C++ data symbols use the same convention as C.

To use an assembly function or variable in your C program, declare the symbol with the `.GLOBAL` assembler directive in the assembly routine and import the symbol by declaring the symbol as `extern` in the C program.

To use an assembly function in your C++ module, declare the symbol with the `.GLOBAL` assembler directive in the assembly routine and import the symbol by declaring the symbol as `extern "C"` in the C++ program. For example, to reference the `_funcmult` assembly routine from a C++ program, you declare it as `extern "C" int funcmult(int a, int b)` in the C++ program.

Table 2-22 shows several examples of the C/Assembly interface naming conventions. Each row shows how assembler code can reference the corresponding C item.

Table 2-22. C Naming Conventions For Symbols

In the C Program	In the Assembly Subroutine
`int c_var; /*declared global*/`	`.extern _c_var;`
`void c_func(){...}`	`.extern _c_func;`
`extern int asm_var;`	`.global _asm_var;`
`extern void asm_func();`	`.global _asm_func;` `_asm_func:`

Table 2-23 shows several examples of the C++/Assembly interface naming conventions.Each row shows how assembler code can reference the corresponding C++ item.

Table 2-23. C++ Naming Conventions for Symbols

In the C++ Program	In the Assembly Subroutine
`int c_var; /*declared global*/`	`.extern _c_var;`
`extern "C" void c_func(void){...}`	`.extern _c_func;`
`extern int asm_var;`	`.global _asm_var;`
`extern "C" void asm_func(void);`	`.global _asm_func;` `_asm_func:`

Implementing C++ Member Functions in Assembly

If an assembly language implementation is desired for a C++ member function, the simplest way is to use C++ to provide the proper interface between C++ and assembly.

In the class definition, write a simple member function to call the assembly-implemented function (subroutine). This call can establish any interface between C++ and assembly, including passing a pointer to the class instance. Since the call to the assembly subroutine resides in the class definition, the compiler inlines the call (inlining adds no overhead to compiler performance). From an efficiency point of view, the assembly-language function is called directly from the user code.

As for any C++ function, ensure that a prototype for the assembly-implemented function is included in your program. As discussed in "Using Mixed C/C++ and Assembly Naming Conventions" on page 2-157, you declare your assembly language subroutine's name with the `.GLOBAL` directive in the assembly portion and import the symbol by declaring it as `extern "C"` in the C++ portion of the code.

Note that using this method you avoid name mangling — you choose your own identifier for the external function. Access to internal class information can be done either in the C++ portion or in the assembly portion. If the assembly subroutine needs only limited access to the class members, it is easier to select those in the C++ code and pass them as explicit arguments. This way the assembly code does not need to know how data is allocated within a class instance.

```
#include <stdio.h>

/* Prototype for external assembly routine: */
/* C linkage does not have name mangling */
extern "C" int cc_array(int);

class CC {
    private:
        int av;
    public:
        CC(){};
        CC(int v) : av(v){};
        int a() {return av;};
        /* Assembly routine call: */
        int array() {return cc_array(av);};
};

int main()
{
    CC samples(11);
    CC points;
    points = CC(22);
    int j, k;
    j = samples.a();
    k = points.array();    // Test asm call
```

```
    printf ( "Val is %d\n", j );
    printf ( "Array is %d\n", k );

    return 1;
}

/* In a separate assembly file: */
     .section /pm seg_pmco;
     .global _cc_array;
     _cc_array:
          modify(i7,-3);
          dm(-4,i6)=r3;
          dm(-2,i6)=r4;
          r3=r4;
          r0=r3+r3;
          r3=dm(-4,i6);
          i12=dm(m7,i6);
          jump(m14,i12)(DB);
          rframe;
          nop;
```

Writing C/C++ Callable SIMD Subroutines

You can write assembly subroutines that use the ADSP-2116x DSP's SIMD mode and call them from your C programs. The routine may use SIMD mode (`PEYEN` bit=1) for all code between the function prologue and epilogue, placing the chip in SISD mode (`PEYEN` bit =0) before the function epilogue or returning from the function.

While it is possible to write subroutines that can be called in SIMD mode (the chip is in SIMD mode before the call and after the return), the compiler does not support a SIMD call interface at this time. For example, trying to call a subroutine from a `#pragma SIMD_for` loop prevents the compiler from executing the loop in SIMD mode because the compiler does not support SIMD mode calls.

Because transfers between memory and data registers are doubled in SIMD mode (each explicit transfer has a matching implicit transfer), it is recommended that you access the stack in SISD mode to prevent corrupting the stack. For more information on SIMD mode memory accesses, see the Memory chapter in the appropriate ADSP-2116x *SHARC DSP Hardware Reference*.

If you are using SIMD subroutines, your interrupt handler must provide additional support. This support in the interrupt service routine entails saving-restoring the `PEYEN` bit and placing the DSP in the mode (SISD or SIMD) that the interrupt service routine needs. Interrupt handlers often use the `MMASK` register to expedite these mode changes.

C++ Programming Examples

This section provides the following examples for C++-specific features:

- "Using Fract Support" on page 2-163
- "Using Complex Support" on page 2-163

Note that the cc21k compiler runs in C mode by default. To run the compiler in C++ mode, you select the corresponding option on the command line, or check it in the `Project Options` dialog box of the VisualDSP++ environment. The following command line, for example:

```
cc21k -c++ fdot.c -T062.ldf
```

runs cc21k with:

`-c++`	Specifies that the following source file is written in ANSI/ISO standard C++ extended with the Analog Devices keywords.
`fdot.c`	Specifies the source file for your program.
`-T 062.ldf`	Specifies the Linker Description File for the ADSP-21062 DSP system.

Using Fract Support

Listing 2-5 demonstrates the compiler support for the `fract` type and associated arithmetic operators, such as `+` and `*`. The dot product algorithm is expressed using the standard arithmetic operators. The code demonstrates how two variable-length arrays are initialized with fractional literals. For more information about the fractional data type and arithmetic, see "C++ Fractional Type Support" on page 2-84.

```
fract fdot (int array_size, fract *x, fract *y)
   {
      int j;
      fract s;
      s = 0;
      for (j=0; j < array_size; j++)
         {
            s += x[j] * y[j];
         }
      return s;
   }
int main(void)
   {
      set_saturate_mode();
      fdot (N,x,y);
   }
```

Listing 2-5. Dot Product Using Fract Arithmetic Example — C++ Code

Using Complex Support

The Mandelbrot fractal set is defined by the following iteration on complex numbers:

```
z := z * z + c
```

The `c` values belong to the set for which the above iteration does not diverge to infinity. The canonical set is defined when `z` starts from `zero`.

Listing 2-6 demonstrates the Mandelbrot generator expressed in a simple algorithm using the C++ library `complex` class:

```
#include <complex>

int iterate (complex<double> c, complex<double> z, int max)
{
   int n;

   for (n = 0; n<max && abs(z)<2.0; n++)
   {
      z = z * z + c;
   }
   return (n == max ? 0 : n);
}
```

Listing 2-6. Mandelbrot Generator Example — C++ code

Listing 2-7 shows a C version of the inner computational function of the Mandelbrot generator, which extracts performance and programming penalties (compared with the C++ version).

```
int    iterate (double creal, double cimag,
double zreal, double zimag, int max)

{
   double real, imag;
   int n;
   real = zreal * zreal;
   imag = zimag * zimag;

   for (n = 0; n<max && (real+imag)<4.0; n++)
   {
      zimag = 2.0 * zreal * zimag + cimag;
      zreal = real - imag + creal;
      real = zreal * zreal;
      imag = zimag * zimag;
   }
   return (n == max ? 0 : n);
}
```

Listing 2-7. Mandelbrot Generator Example — C code

Mixed C/C++/Assembly Programming Examples

This section shows examples of types of mixed C/C++/assembly programming in order of increasing complexity. The examples in this section are as follows:

Note that leaf assembly routines are routines that return without making any calls. Non-leaf assembly routines call other routines before returning to the caller.

Note that you can use cc21k to compile your C or C++ program and assemble your assembly language modules. This ensures that the assembly of your modules complies with the C/C++ run-time environment. The following cc21k command line, for example:

```
cc21k my_prog.c my_sub1.asm -T 062.ldf -Wremarks
```

runs cc21k with:

`my_prog.c`	Selects a C language source file for your program
`my_sub1.asm`	Selects an assembly language module to be assembled and linked with your program
`-T 062.ldf`	Selects a linker description file describing your DSP system
`-Wremarks`	Selects diagnostic compiler warnings

Using Inline Assembly (Add)

The following example shows how to write a simple routine in ADSP-21xxx family assembly code that properly interfaces to the C/C++ environment. It uses the `asm()` construct to pass inline assembly code to the compiler.

```
int i,j,k,l;
main()
{
   l = add(i,j,k);
}

/* Per the run-time environment, function add() passes arg x in
   r4, arg y in r8, and arg z in r12. Then, func adds args and
   puts return in r0. */

add(int x, int y, int z)
{
   asm("r0=%0+%1; r0=r0+%2"::"d"(x),"d"(y),"d"(z));
}
```

Using Macros to Manage the Stack

Listing 2-8 and Listing 2-9 show how macros can simplify function calls between C, C++, and assembly functions. The assembly function uses the `entry`, `exit`, and `ccall` macros to keep track of return addresses and manage the run-time stack. For more information, see "Managing the Stack" on page 2-135.

```
/* Subroutine Return Address Example—C code: */

/* assembly and c functions prototyped here */
void asm_func(void);
void c_func(void);

/* c_var defined here as a global */
/* used in .asm file as _c_var */
int c_var=10;

/* asm_var defined in .asm file as _asm_var */
extern int asm_var;

main ()
{
   asm_func();  /* call to assembly function */
}

/* this function gets called from asm file */
void c_func(void)
{
   if (c_var != asm_var)
      exit(1);
   else
      exit(0);
}
```

Listing 2-8. Subroutine Return Address Example — C Code

```
/* Subroutine Return Address Example—Assembly code: */
#include <asm_sprt.h>
.section/dm seg_dmda;
.var _asm_var=0;/* asm_var is defined here */
.global _asm_var;/* global for the C function */
.endseg;

.section/pm seg_pmco;
.global _asm_func;/* _asm_func is defined here */
.extern _c_func;/* c_func from the C file */
.extern _c_var;/* c_var from the C file */

_asm_func:
entry;/* entry macro from asm_sprt */

   r8=dm(_c_var/* access the global C var */
   dm(_asm_var)=r8;/* set _asm_var to _c_var) */

   ccall(_c_func);/* call the C function */

   exit;/* exit macro from asm_sprt */
.endseg;
```

Listing 2-9. Subroutine Return Address Example—Assembly Code

Using Scratch Registers (Dot Product)

To write assembly functions that can be called from a C or C++ program, your assembly code must follow the conventions of the C/C++ run-time environment and use the conventions for naming functions. The following assembly function demonstrates how to comply with these specifications.

This function computes the dot product of two vectors. The two vectors and their lengths are passed as arguments. Because the function uses only scratch registers (as defined by the run-time environment) for intermediate values and takes advantage of indirect addressing, the function does not need to save or restore any registers.

```
/* dot(int n, dm float *x, pm float *y);
Computes the dot product of two floating-point vectors of
length n. One is stored in dm and the other in pm. Length n
must be greater than 2.*/

#include <asm_sprt.h>

.section/pm seg_pmco;
   /* By convention, the assembly function name is the C func-
   tion name with a leading underscore; "dot()" in C becomes
   "_dot" in assembly */

.global _dot;
_dot:

leaf_ entry;

r0=r4-1,i4=r8;
   /* Load first vector address into I register, and load r0
   with length-1 */

r0=r0-1,i12=r12;
   /* Load second vector address into I register and load r0
   with length-2 (because the 2 iterations outside feed and
   drain the pipe */

f12=f12-f12,f2=dm(i4,m6),f4=pm(i12,m14);
   /* Zero the register that will hold the result and start
   feeding pipe */

f8=f2*f4, f2=dm(i4,m6),f4=pm(i12,m14);
   /* Second data set into pipeline, also do first multiply */

lcntr=r0, do dot_loop until lce;
   /* Loop length-2 times, three-stage pipeline: read, mult,
   add */

dot_loop:
f8=f2*f4, f12=f8+f12,f2=dm(i4,m6),f4=pm(i12,m14);
f8=f2*f4, f12=f8+f12;
f0=f8+f12;
   /* drain the pipe and end with the result in r0, where it'll
   be returned */
```

```
leaf_exit;
   /* restore the old frame pointer and return */

.endseg;
```

Using Void Functions (Delay)

The simplest kind of assembly routine is one with no arguments and no return value, which corresponds to C/C++ functions prototyped as `void my_function(void)`. Such routines could be used to monitor an external event or used to perform an operation on a global variable.

It is important when writing such assembly routines to pay close attention to register usage. If the routine uses any call-preserved or compiler-reserved registers (as defined in the run-time environment), the routine must save the register and restore it before returning. Because the following example does not need many registers, this routine uses only scratch registers (also defined in the run-time environment) that do not need to be saved.

Note that in the example all symbols that need to be accessed from C/C++ contain a leading underscore. Because the assembly routine name, `_delay`, and the global variable `_del_cycle` must both be available to C and C++ programs, they contain a leading underscore in the assembly code.

```
/* Simple Assembly Routines Example – _delay */
/* void delay (void);
An assembly language subroutine to delay N cycles, where N is
the value of the global variable del_cycle */

 #include <asm_sprt.h>;

.section/pm seg_pmco;
.extern _del_cycle;
.global _delay;
_delay:
   leaf_entry;     /* first line of any leaf func */
   R4 = DM (_del_cycle);
   /* Here, register r4 is used because it is a scratch register
     and doesn't need to be preserved */
```

```
   LCNTR = R4, DO d_loop UNTIL LCE;
   d_loop:
   nop;
   leaf_exit;     /* last line of any leaf func */
.endseg;
```

Using the Stack for Arguments and Return (Add 5)

A more complicated kind of routine is one that has parameters but no return values. The following example adds together the five integers passed as parameters to the function.

```
/* Assembly Routines With Parameters Example — _add5 */
/* void add5 (int a, int b, int c, int d, int e);
An assembly language subroutine that adds 5 numbers */

#include <asm_sprt.h>
.section/pm seg_pmco;
.extern _sum_of_5; /* variable where sum will be stored */
.global _add5;

_add5:
leaf_entry;
   /* the calling routine passes the first three parameters in
   registers r4, r8, r12 */

r4 = r4 + r8; /* add the first and second parameter */
r4 = r4 + r12; /* adds the third parameter */

   /* the calling routine places the remaining parameters
   (fourth/fifth) on the run-time stack; these parameters can
   be accessed using the reads() macro */

r8 = reads(1); /* put the fourth parameter in r8 */
r4 = r4 + r8;  /* adds the fourth parameter */
r8 = reads(2); /* put the fifth parameter in r8 */
r4 = r4 + r8;  /* adds the fifth parameter */

dm(_sum_of_5) = r4;
   /* place the answer in the global variable */

leaf_exit;
.endseg;
```

Using Registers for Arguments and Return (Add 2)

There is another class of assembly routines in which the routines have both parameters and return values. The following example of such a routine adds two numbers and returns the sum. Note that this routine follows the run-time environment specification for passing function parameters (in registers `r4` and `r8`) and passing the return value (in register `r0`).

```
/* Routine With Parameters & Return Value —add2_ */
/* int add2 (int a, int b);
An assembly language subroutine that adds two numbers and
returns the sum */

#include <asm_sprt.h>

.section/pm seg_pmco;
.global _add2;
_add2:
leaf_entry;

/* per the run-time environment, the calling routine passes the
first two parameters passed in registers r4 and r8; the return
value goes in register r0  */

r0 = r4 + r8;
   /* add the first and second parameter, store in r0*/

leaf_exit;
.endseg;
```

Using Non-leaf Routines That Make Calls (RMS)

A more complicated example, which calls another routine, computes the root mean square of two floating-point numbers, as follows:

$$z = \sqrt{x^2 + y^2}$$

Although it is straight forward to develop your own function that calculates a square-root in ADSP-21xxx assembly language, the following example demonstrates how to call the square root function from the C/C++ run-time library, `sqrtf`. In addition to demonstrating a C run-time library call, this example shows some useful calling macros.

```
/*  Non-Leaf Assembly Routines Example – _rms */
/* float rms(float x, float y);
An assembly language subroutine to return the rms
z = (x^2 + y^2)^(1/2)  */

#include <asm_sprt.h>

.section/pm seg_pmco;
.extern _sqrtf;
.global _rms;
_rms:
   entry;   /* first line of non-leaf routine */

   f4 = f4 * f4;
   f8 = f8 * f8;
   f4 = f4 + f8;
   /* f4 contains argument to be passed to sqrtf function */

   ccall (_sqrtf);
   /* use the ccall() macro to make a function call in a C envi-
   ronment; f0 contains the result returned by the _sqrtf func-
   tion. In turn, _rms returns the result to its caller in f0
   (and it is already there) */
   exit;   /* last line of non-leaf routine */
.endseg;
```

If a called function takes more than three single word parameters, the remaining parameters must be pushed on the stack and popped off the stack after the function call. The following function could call the `_add5` routine shown in "Using the Stack for Arguments and Return (Add 5)" on page 2-171. Note that the last parameter must be pushed on the stack first.

```
/* Non-Leaf Assembly Routines Example – _calladd5 */
/* int calladd5 (void);
An assembly language subroutine that calls another routine with
more than 3 parameters. This example adds the numbers 1, 2, 3,
4, and 5. */

#include <asm_sprt.h>
.section/pm seg_pmco;
.extern _add5;
.extern _sum_of_5;
.global _calladd5;
_calladd5:

entry;
   r4 = 5;
/* the fifth parameter is stored in r4 for pushing onto stack */
   puts=r4; /* put fifth parameter in stack */
   r4 = 4;
   /* the fourth parameter is stored in r4 for pushing onto
   stack */
   puts=r4; /* put fourth parameter in stack */
   r4 = 1;    /* the first parameter is sent in r4 */
   r8 = 2;    /* the second parameter is sent in r8 */
   r12 = 3;   /* the third parameter is sent in r12 */

   ccall (_add5);
   /* use the ccall macro to make a function call in a C envi-
   ronment */
   alter(2);
   /* call the alter() macro to remove the two arguments from
   the stack */
   r0 = dm(_sum_of_5);
   /* _sum_of_5 is where add5 stored its result */
   exit;
.endseg;
```

Using Call Preserved Registers (Pass Array)

Some functions need to make use of registers that the run-time environment defines as *call preserved registers*. These registers, whose contents are preserved across function calls, are useful for variables whose lifetime spans a function call. The following example performs an operation on the elements of a C array using call preserved registers.

```
/* Non-Leaf Assembly Routines Example — _pass_array */
/* void pass_array(
float function(float),
float *array,
int length);
An assembly language routine that operates on a C array */

#include <asm_sprt.h>
.section/pm seg_pmco;
.global _pass_array;
_pass_array:
             entry;
             puts = i8;
   /* This function uses a call preserved register, i8, because
   it could be used by multiple functions, and this way it does
   not have to be stored for every function call */

   r0 = i1;
   puts = r0; /* i1 is also call preserved */

i8 = r4;
   /* read the first argument, the address of the function to
   call */

i1 = r8;
   /* read the second argument, the C array containing the data
   to be processed */

r0 = r12;
   /* read third argument, the number of data points in the
   array */

lcntr=r0, do pass_array_loop until lce;
   /* loop through data points */
```

```
f4=dm(i1,m5);
   /* get data point from array, store it in f4 as a parameter
   for the function call */

ccall(m13,i8);/* call the function */
pass_array_loop:
   dm(i1,m6)=f0;
   /* store the return value back in the array */
   i1 = gets(1);/* restore the value of i1 */
   i8 = gets(2);/* restore the value of i8 */

exit;

.endseg;
```

C/C++ Compiler Glossary

Assembler (easm21k) — Assembles your ADSP-21xxx family assembly source code or code output from the C/C++ compiler (cc21k) into linkable object files.

B (Base) registers — Data Address Generator (DAG) B registers (`B0-B7` in `DAG1`, `B8-B15` in `DAG2`) contain the base address of a circular buffer.

C/C++ Run-time environment (for cc21k C/C++ compiler) — The C/C++ run-time environment is a set of rules that the cc21k C/C++ Compiler uses to operate on the ADSP-21xxx family processors. The environment defines register use (compiler, user, scratch, and stack registers), run-time stack operation, and C/C++/assembly program interfacing requirements.

cc21k, ADSP-21xxx family C/C++ compiler — cc21k is an ANSI compliant C/C++ compiler for the ADSP-21xxx family Digital Signal Processors.

I (Index) registers — Data Address Generator (DAG) I registers (`I0- I7` in `DAG1`, `I8-I15` in `DAG2`) contain the actual address used to access memory.

L (Length) registers — Data Address Generator (DAG) L registers (`L0-L7` in DAG1, `L8-L15` in DAG2) contain–for the corresponding I (index) register–either zero for linear addressing or the number of words (length) of a circular buffer.

Leaf Assembly routines — Leaf assembly routines are routines that return without making any calls.

Linker — The VisualDSP++ linker links your object files and libraries into executable DSP programs.

M (Modify) registers — Data Address Generator (DAG) M registers (`M0-M7` in DAG1, `M8-M15` in DAG2) contain a value used to post-modify the I register specified in an indirect memory access operation.

Macros and the C/C++ preprocessor — Macros are blocks of text substituted by the C/C++ preprocessor during the process of building your program. The C/C++ preprocessor is a macro processor. Use the preprocessor to make transformations and text substitutions in your source code. The C/C++ preprocessor provides header file inclusion, macro expansion, and conditional compilation for C/C++ and assembly files.

MODE (1 and 2) registers — MODE registers control many of the ADSP-21xxx DSP's operational features: addressing, interrupts handling, rounding, bit-reversing, and others.

MRF and MRB (multiplier results, foreground and background) registers — MR registers can hold the 80-bit result of a fixed-point multiply-accumulate operation.

Non-leaf assembly routines — Non-leaf assembly routines call other routines before returning to the caller.

R (register file) registers — Register file registers (called `R0-R15` when fixed-point and `F0-F15` when floating-point) can hold the input or output of any computational unit in the DSP.

USTAT (1 and 2) registers — USTAT registers hold user defined status flags.