

2 COMPILER

Overview

The C/C++ compiler (ccts) compiles ANSI/ISO standard C and C++ code for TigerSHARC family DSPs. Additionally, Analog Devices includes within the compiler a number of C language extensions designed to assist in DSP development. The ccts compiler runs from the VisualDSP++ environment or from an operating system command-line.

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

- [“Compiler Command-Line Reference” 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 2-53](#) explains the ccts compiler’s extensions to the ANSI/ISO standard for the C and C++ languages.
- [“Preprocessor Features” on page 2-91](#) contains information about the preprocessor and predefined macros.
- [“C/C++ Run-Time Model” on page 2-97](#) contains reference information about how C/C++ programs, data, and function calls are implemented on the TigerSHARC family of DSPs. This is important for low-level program analysis and interfacing assembly code with C/C++ programs.

Overview

- [“C/C++ and Assembly Interface” on pag e2-116](#) describes how to call an assembly language subroutine from within a C or C++ program, and how to call a C or C++ function from within an assembly language program.
- [“C/C++ Compiler Glossary” on page 2-121](#) contains a glossary of compiler-specific terms used this manual.

Your source files contain the program to be processed into object code by the compiler. By default, the ccts compiler supports the ANSI/ISO standard definition of the C and C++ languages. For information on these standards, see any reference texts on the C and C++ languages.

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 ccts 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, the ccts compiler supports templates and all other features of the full C++ standard with the exception of exception handling and run-time type identification. The additional supported features provide extra functionality without degrading the compiler performance.

The ccts compiler supports a set of C/C++ language extensions. These extensions support hardware features of the TigerSHARC family DSPs. For information on these extensions, see [“C/C++ Compiler Language Extensions” on page 2-53](#).

You set the compiler options from the `Project` menu, `Project Options` command, `Compile` tab of the VisualDSP++ IDE. 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 information about the VisualDSP++ environment, see the *VisualDSP++ 2.0 User's Guide for TigerSHARC DSPs* and online Help.

Compiler Command-Line Reference

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 extensions. Several options allow you to change the compilation mode and language dialect, thus enforcing certain standards and/or disabling the Analog Devices 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 only in C++ mode. [Table 2-3 on page 2-10](#) 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 the following sections:

- [“C/C++ Mode Selection Switch Descriptions” on page 2-18](#)
- [“C++ Compiler Switch Descriptions” on page 2-40.](#)

Compiler Command-Line Reference

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

Running the Compiler

Use the following general syntax for the `ccts` command line:

```
ccts [-switch [-switch ...]] sourcefile [sourcefile ...]
```

where:

- `-switch` is the name of the switch to be processed. The compiler has many optional 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, and file name and file extension. If a file name exceeds eight characters in length or contains spaces, enclose it within straight quotes: "long file name.c". The `ccts` compiler uses the file extension to determine what the file contains and what operations to perform. [Table 2-1](#) lists the allowed extensions. In the default mode of operation, the compiler processes the input file through the listed stages to produce a `.DXX` file.

The following command line, for example:

```
ccts -O -TS001 -Wremarks -o program.dxe source.c
```

runs ccts with:

-O	Specifies optimization for the compiler
TS001	Identifies the target processor type
-Wremarks	Selects extra diagnostic remarks in addition to warning and error messages
-o program.dxe	Specifies a name for the compiled, linked output
source.c	Specifies the C language source file to be compiled

The following command line, for example:

```
ccts -c++ fdot.cpp
```

runs ccts with:

-c++	Specifies that all of the source files are written in C++ and are compiled in C++ mode
fdot.cpp	Specifies the C++ language source file to be compiled

The normal function of ccts 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.

Compiler Command-Line Reference

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

Extension	Action
.c, .cpp, .cxx	source file is compiled, assembled, and linked
.asm or .s	assembly language source file is assembled and linked
.obj	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 using appropriate compiler switches or by selecting options in the compiler dialog boxes within the IDDE. These switches are: `-E`, `-P`, `-M`, `-H`, `-C`, and `-S`.

Because `ccts` 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, `ccts` names the output for you. [Table 2-1 on page 2-7](#) lists the type of files, names, and extensions `ccts` 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](#) 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 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". ccts 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, ccts 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. File Type Descriptions

Input File Extension	File Extension Description
.c	C/C++ language source file.
.cpp, .cxx	C++ language source file.
.h	Header file (referenced by a <code>#include</code> statement).
.pch	Precompiled 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.

Compiler Command-Line Reference

Table 2-1. File Type Descriptions (Cont'd)

Input File Extension	File Extension Description
.i	Preprocessed source file, created when preprocess only (-E compiler switch) is specified.
.s, .asm	Assembly language source file.
.is	Preprocessed assembly language source (retained when -save_temps is specified).
.obj	Object file, output from the assembler, ready to be linked.
.exe	Executable file produced by the compiler.
.ldf	Linker Description File (read by the linker via the -T compiler switch).
.dlb	Library of object files to be linked as needed.
.map	DSP system memory map file output (from the linker).
.sym	DSP system symbol map file output (from the linker).

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 2-10](#)
- [Table 2-4, “C++ Mode Compiler Switches,” on page 2-16](#)

A brief description of each switch appears in the section 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.
-ansi (page 2-18)	Supports ANSI/ISO standard C without extensions.
-c++ (page 2-19)	Supports ANSI/ISO standard C++ with Analog Devices extensions.
-traditional (page 2-19)	Supports pre-ANSI K&R C.

Compiler Command-Line Reference

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

Switch Name	Description
<i>sourcefile</i> (page 2-19)	Specifies file to be compiled.
<i>-@ filename</i> (page 2-20)	Reads command-line input from the file.
<i>-A name[tokens]</i> (page 2-20)	Asserts the specified name as a predicate.
<i>-build-lib</i> (page 2-20)	Directs the librarian to build a library file.
<i>-C</i> (page 2-20)	Retains preprocessor comments in the output file; active only with the <i>-E</i> or <i>-P</i> switch.
<i>-c</i> (page 2-21)	Compiles and/or assembles only; does not link.
<i>-Dmacro[=def]</i> (page 2-21)	Defines <i>macro</i> .
<i>-default-linkage</i> (page 2-21)	Sets the default linkage type (C, C++, asm).
<i>-dollar</i> (page 2-21)	Accepts dollar signs in symbols.
<i>-double-size-{32 64}</i> (page 2-21)	Selects 32- or 64-bit IEEE format for double. <i>-double-size-32</i> is the default mode.
<i>-dry</i> (page 2-23)	Displays, but does not perform, main driver actions (verbose dry-run).
<i>-dryrun</i> (page 2-23)	Displays, but does not perform, top-level driver actions (terse dry-run).
<i>-E</i> (page 2-23)	Preprocesses, but does not compile, the source file.

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

Switch Name	Description
-EE (page 2-23)	Preprocesses and compiles the source file.
-extra-keywords (page 2-23)	Recognizes ADI extensions to ANSI/ISO standards for C and C++. Default mode.
-flags-tool (page 2-24)	Passes command-line switches through the compiler to other build tools.
-g (page 2-24)	Generates DWARF-2 debug information.
-H (page 2-24)	Outputs a list of included header files, but does not compile.
-HH (page 2-25)	Outputs a list of included header files and compiles.
-h[elp] (page 2-25)	Outputs a list of command-line switches.
-I <i>directory</i> (page 2-25)	Appends <i>directory</i> to the standard search path.
-include <i>filename</i> (page 2-25)	Includes named file prior to each source file.
-ipa (page 2-25)	Enables interprocedural analysis.
-L <i>directory</i> (page 2-26)	Appends <i>directory</i> to the standard library search path when linking.
-l <i>library</i> (page 2-26)	Searches <i>library</i> for functions when linking.
-M (page 2-26)	Generates make rules only, but does not compile.

Compiler Command-Line Reference

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

Switch Name	Description
<code>-MM</code> (page 2-26)	Generates <code>make</code> rules and compiles.
<code>-map filename</code> (page 2-27)	Directs the linker to generate a memory map of all symbols.
<code>-no-builtin</code> (page 2-27)	Disables recognition of <code>__builtin</code> functions.
<code>-no-defs</code> (page 2-27)	Disables preprocessor definitions: macros, include directories, library directories, run-time headers, or keyword extensions.
<code>-no-dollar</code> (page 2-27)	Does not accept dollar signs in symbols. This is the default mode.
<code>-no-extra-keywords</code> (page 2-28)	Does not define language extension keywords that could be valid C/C++ identifiers.
<code>-no-inline</code> (page 2-28)	Ignores the <code>inline</code> keyword.
<code>-no-restrict</code> (page 2-28)	Disables the <code>restrict</code> keyword.
<code>-no-std-def</code> (page 2-28)	Disables normal macro definitions; also disables ADI keyword extensions that do not have leading underscores (<code>__</code>).
<code>-no-std-inc</code> (page 2-29)	Searches for preprocessor include header files only in the current directory and in directories specified with the <code>-I</code> switch.
<code>-no-std-lib</code> (page 2-29)	Searches for only those library files specified with the <code>-l</code> switch when linking.
<code>-nothreads</code> (page 2-29)	Specifies that no support is required for multi-threaded applications.

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

Switch Name	Description
-O (page 2-29)	Enables standard code optimizations.
-O0 (page 2-30)	Disables standard code optimizations.
-Os (page 2-30)	Optimizes to decrease code size.
-o <i>filename</i> (page 2-30)	Specifies the output filename.
-P (page 2-30)	Preprocesses, but does not compile, the source file. Output does not contain <code>#line</code> directives.
-PP (page 2-30)	Preprocesses and compiles the source file. Output does not contain <code>#line</code> directives.
-path-tool <i>executable</i> (page 2-31)	Uses the specified executable as the specified compilation tool (assembler, compiler, driver, librarian, or linker).
-path-install <i>directory</i> (page 2-31)	Uses the specified directory as the location of all compilation tools.
-path-output <i>directory</i> (page 2-31)	Specifies the location of non-temporary files.
-path-temp <i>directory</i> (page 2-31)	Specifies the location of temporary files.
-pch (page 2-31)	Generates and uses precompiled header files (*.pch).
-pchdir <i>directory</i> (page 2-32)	Specifies the location of PCHRepository.
-pedantic (page 2-32)	Issues compiler warnings for any constructs that are not strictly ANSI/ISO standard C/C++-compliant.

Compiler Command-Line Reference

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

Switch Name	Description
<code>-pedantic-errors</code> (page 2-32)	Issues compiler errors for any constructs that are not strictly ANSI/ISO standard C/C++-compliant.
<code>-pplist filename</code> (page 2-32)	Outputs a raw preprocessed listing to the specified file.
<code>-R directory</code> (page 2-33)	Appends <i>directory</i> to the standard search path for source files.
<code>-restrict</code> (page 2-33)	Enables the <code>restrict</code> keyword.
<code>-S</code> (page 2-34)	Stops compilation before running the assembler.
<code>-s</code> (page 2-34)	Removes debug info from the output executable file when linking.
<code>-save-temps</code> (page 2-34)	Saves intermediate files.
<code>-show</code> (page 2-34)	Displays the driver command-line information.
<code>-signed-char</code> (page 2-34)	Makes the <code>char</code> data type signed.
<code>-syntax-only</code> (page 2-35)	Checks the source code for compiler syntax errors, but does not write any output.
<code>-sysdefs</code> (page 2-35)	Defines the system definition macros.
<code>-T filename</code> (page 2-35)	Specifies the Linker Description File.
<code>-threads</code> (page 2-36)	Specifies that support for multi-threaded applications is to be enabled.

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

Switch Name	Description
-time (page 2-36)	Displays the elapsed time as part of the output information on each part of the compilation process.
-TS001 (page 2-36)	Generates code for ADSP-TS001 DSPs.
-TS101 (page 2-36)	Generates code for ADSP-TS101 DSPs.
-U <i>macro</i> (page 2-36)	Undefines <i>macro</i> .
-unsigned-char (page 2-37)	Makes the <code>char</code> data type unsigned.
-v (page 2-37)	Displays both version and command-line information.
-verbose (page 2-37)	Displays command-line information.
-version (page 2-37)	Displays version information.
-Werror <i>number</i> (page 2-37)	Overrides the default severity of the specified error message.
-Wdriver-limit <i>number</i> (page 2-38)	Halts the driver after reaching the specified number of errors.
-Werror-limit <i>number</i> (page 2-38)	Stops compiling after reaching the specified number of errors.
-Wremarks (page 2-38)	Issues compiler remarks.
-Wterse (page 2-38)	Issues only the briefest form of compiler warning, errors, and remarks.

Compiler Command-Line Reference

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

Switch Name	Description
<code>-w</code> (page 2-38)	Disables all warnings.
<code>-write-files</code> (page 2-39)	Enables compiler I/O redirection.
<code>-xref filename</code> (page 2-39)	Outputs cross-reference information to the specified file.

Table 2-4. C++ Mode Compiler Switches

Switch Name	Description
<code>-alttok</code> (page 2-40)	Allows alternative keywords and sequences in sources.
<code>-bool</code> (page 2-40)	Enables the <code>bool</code> keyword.
<code>-explicit</code> (page 2-41)	Supports the <code>explicit</code> specifier on constructor declarations.
<code>-instant{all used}</code> (page 2-41)	Instantiates all or used members of a template class.
<code>-namespace</code> (page 2-41)	Supports namespaces.
<code>-newforinit</code> (page 2-41)	Limits the scope of any symbol declared within a “for” statement.
<code>-newvec</code> (page 2-41)	Allows the overloading of <code>new</code> and <code>delete</code> .
<code>-no-alttok</code> (page 2-42)	Does not allow alternative keywords and sequences in sources.

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

Switch Name	Description
<code>-no-bool</code> (page 2-42)	Disables the <code>bool</code> keyword.
<code>-no-demangle</code> (page 2-42)	Prevents filtering of any linker errors through the demangler.
<code>-no-explicit</code> (page 2-42)	Does not support the <code>explicit</code> specifier on constructor declarations.
<code>-no-namespace</code> (page 2-42)	Does not support namespaces.
<code>-no-newvec</code> (page 2-42)	Does not allow the overloading of <code>new</code> and <code>delete</code> .
<code>-no-std</code> (page 2-43)	Disables the implicit use of the <code>std</code> namespace.
<code>-notstrict</code> (page 2-43)	Omits warning and/or error messages for non-ANSI constructs.
<code>-no-wchar</code> (page 2-43)	Disables new <code>wchar_t</code> .
<code>-std</code> (page 2-43)	Enables the implicit use of the <code>std</code> namespace.
<code>-strict</code> (page 2-43)	Generates error messages for non-ANSI constructs.
<code>-strictwarn</code> (page 2-44)	Generates warning messages for non-ANSI constructs.
<code>-tpautooff</code> (page 2-44)	Disables automatic instantiation of templates.
<code>-trdforinit</code> (page 2-44)	Limits the scope of any symbol declared within a “for” statement.

Compiler Command-Line Reference

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

Switch Name	Description
-typename (page 2-44)	Recognizes the <code>typename</code> keyword.
-wchar (page 2-44)	Enables new <code>wchar_t</code> .

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 information on these extensions, see “[C/C++ Compiler Language Extensions](#)” on [page 2-53](#).

-ansi

The `-ansi` (ANSI standard C compilation) switch disables most Analog Devices extensions to ANSI/ISO C. It also defines the `__STRICT_ANSI__` preprocessor macro.

This switch influences many features of the compiler, such as disabling the `inline`, `dm`, and `pm` keywords. Note that the `-ansi` switch does not change the size of `double` to the default of 32 bits. To strictly conform to the ANSI/ISO standard, the `-double-size-64` switch must also be used.



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

Also see “[-no-extra-keywords](#)” on [page 2-28](#) for related information.

-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.

-traditional

The `-traditional` (traditional C 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
- 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. ccts 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.

Compiler Command-Line Reference

-@ *filename*

The @ *filename* (command file) switch directs the compiler to read command-line input from *filename*. The specified *filename* must contain driver options but may also contain source *filenames* and environment variables. It can be used to store frequently used options as well as to read from a file list.

filename can be a directory. In that case, all source files within that directory will be placed on the command line. A vertical bar (|) may be used in the file to indicate that all the following options and files to be placed at the end of the command line (dependent on option file nesting level). Multiple appearances of the vertical bar toggle this feature, but its state only continues to the end of the current option file.

-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.

-build-lib

The -build-lib (build library) switch directs the compiler to use the librarian to produce a library file (.d1b) 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 is only active in combination with the -E or -P switch, 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 (`.obj`) for each source file.

-Dmacro[=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.



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

-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.

-dollar

The `-dollar` (dollar format) switch directs the compiler to accept dollar signs (\$) in identifiers. This option is disabled by default.

-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`. `-double-size-32` is the default mode.

Compiler Command-Line Reference

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, ccts 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 standards require 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__` preprocessor macro, while the `-double-size-64` switch undefines the `__DOUBLES_ARE_FLOATS__` preprocessor macro.



This switch can be invoked with the `Double size` radio buttons located in the IDDE's `Compile` dialog box, `DSP Specific` selection.

-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`. The `-C` switch can be used in conjunction with `-E` to retain comments.



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 or C++ programs. This is the default mode.

Compiler Command-Line Reference

`-flags-{asm | compiler | 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.

-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 such 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 can be specified more than once; all specified directories to be 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 directs the compiler to turn on the interprocedural analysis option. This option enables optimization across the entire program, including source files that are compiled separately. For more information on the interprocedural analysis, see [“Interprocedural Analysis” on page 2-50](#).



This switch can be invoked with the Interprocedural Analysis check box located in the IDDE's **Compile** dialog box, **General** selection.

Compiler Command-Line Reference

For best results, `-ipa` should be applied to all files comprising the program.

`-L directory [{,|;} directory ...]`

The `-L` (library search directory) switch directs the linker to append the directory/directories to the search path for library files. This option may be specified more than once.

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

Normally, you should list all object files on the command line before the `-l` switch; this ensures that functions and global variables the object files refer to 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 on 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; if the argument is `test`, then the map file name is `test.map`. The `.map` extension is added where necessary.

-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 [“Run-time Library Reference” on page 3-29](#).



This switch can be invoked with the `Compiler Dialect: Disable built-in functions` check box located in the IDDE's `Compile dialog`, box `General` selection.

-no-defs

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

-no-dollar

The `-no-dollar` (disable dollar format) switch directs the compiler to not accept dollar signs (\$) in identifiers. This is the default mode.

Compiler Command-Line Reference

`-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 conforming to ANSI/ISO 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 and directs the compiler to ignore the `inline` keyword.

`-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) prevents the compiler from defining any 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 standard optimizations) switch directs the compiler to perform standard code optimizations.



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



Older versions of the compiler supported different levels of optimization via `-O1`, `-O2`, `-O3`. The current release of the compiler does not differentiate between levels of optimization; `-O` provides full optimization.

Compiler Command-Line Reference

-O0

The `-O0` (disable standard optimizations) switch directs the compiler to disable all standard code optimizations, including interprocedural analysis (`-ipa`).

-Os

The `-Os` (enable code size optimizations) switch directs the compiler to perform standard optimizations that are unlikely to increase code size significantly.

-o *filename*

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

-P

The `-P` (omit line numbers and halt) switch directs the compiler to stop after the C/C++ preprocessor runs (without compiling) and to omit the `#line` preprocessor directives 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 directs the compiler to omit the `#line` preprocessor directives with line number information from the preprocessor output. After preprocessing, compilation proceeds normally.

`-path-{asm | compiler | driver | lib | link | prelink} executable`

The `-path-tool` (tool location) switch directs the compiler to use the specified executable as the specified compilation tool. The executable should comprise a relative or absolute path to its location. Respectively, the tools are the assembler, compiler, librarian, linker, and prelinker. Use this switch when you wish to override the normal version of one or more tools. The `-path-tool` switch also overrides the directory specified by `-path-install`.

`-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 installation directory. This is useful when working with multiple versions of the development tool set. Note that 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; they also tend to occupy more disk space.

Compiler Command-Line Reference

`-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 warnings for any constructs found in your program which do not strictly conform to the ANSI/ISO standards for C and 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 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 <code>#if</code> or <code>#ifdef</code>
L	Change in source position
R	Diagnostic message (remark)
W	Diagnostic message (warning)
E	Diagnostic message (error)
C	Diagnostic message (catastrophic error)

-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 given 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.



This does not affect source files specified by an absolute path name (beginning with “/” or “\” or a drive letter in Windows OS) or “./” or “.\”.

-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.

Compiler Command-Line Reference

-S

The `-S` (stop after compilation) switch directs ccts to stop compilation before running the assembler. The compiler outputs an assembler 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 debugging 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. [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.

-syntax-only

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

-sysdefs

The `-sysdefs` (system definitions) switch directs the compiler to define several preprocessor macros describing the current user and user's system. The macros are defined as character string constants and are used in functions with null-terminated string arguments.

The following macros will be defined if the system returns information for them:

Macro	Value
<code>__HOSTNAME__</code>	The name of the host machine
<code>__MACHINE__</code>	The machine type of the host machine
<code>__SYSTEM__</code>	The OS name of the host machine
<code>__USERNAME__</code>	The current user's login name
<code>__GROUPNAME__</code>	The current user's group name
<code>__REALNAME__</code>	The current user's real name



Note that `__MACHINE__`, `__GROUPNAME__`, and `__REALNAME__` are not available on Windows platforms.

-T *filename*

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

Compiler Command-Line Reference

-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 likely to be used only 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.

-TS001

The `-TS001` switch directs the compiler and other build tools that are part of the compilation process to generate code suitable for the ADSP-TS001 DSP. This is the default behavior in the absence of any other option.

-TS101

The `-TS101` switch directs the compiler and other build tools that are part of the compilation process to generate code suitable for the ADSP-TS101 DSP. This option cannot be used in conjunction with the `-TS001` option.

-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__` preprocessor 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.

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

The `-W` (override error message) switch directs the compiler to override the severity of the specified diagnostic messages. For this compilation, they have the indicated severity: `error`, `warning`, or `remark`. `suppress` indicates that no diagnostic to be issued. 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.

Compiler Command-Line Reference

-Wdriver-limit *number*

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

-Werror-limit *number*

The `-Werror-limit` (maximum compiler errors) switch lets you set 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, `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 with handling 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)

Compiler Command-Line Reference

C++ Compiler Switch Descriptions

-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
<code>and</code>	<code>&&</code>
<code>and_eq</code>	<code>&=</code>
<code>bitand</code>	<code>&</code>
<code>bitor</code>	<code> </code>
<code>compl</code>	<code>~</code>
<code>or</code>	<code> </code>
<code>or_eq</code>	<code> =</code>
<code>not</code>	<code>!</code>
<code>not_eq</code>	<code>!=</code>
<code>xor</code>	<code>^</code>
<code>xor</code>	<code>^=</code>

-bool

The `-bool` (accept boolean) switch directs the compiler to predefine the `bool` keyword and the `true` and `false` constants. The compiler also defines the `__BOOL` preprocessor macro when this option (or another option which directs the compiler to recognize `bool`) is given. This switch is enabled by default in C++ mode.

-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}

By default, the compiler suppresses the instantiation of any templates on the first compilation and lets 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 enable support for namespaces. This is the default mode.

-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.

Compiler Command-Line Reference

-no-alttok

The `-no-alttok` (disable alternative tokens) switch directs the compiler to not accept alternative operator keywords and digraph sequences in the source files. For more information see [“-alttok” on page 2-40](#).

-no-bool

The `-no-bool` (disable boolean) switch directs the compiler to disable the `bool` keyword. Note that the compiler supports the `bool` data type in C++ mode by default. For more information, see [“-bool” on page 2-40](#).

-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 2-41](#).

-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 2-41](#).

-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 2-43](#).

-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.

Compiler Command-Line Reference

-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. It also 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 that contains 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 2-5](#) shows the size used for each of the intrinsic C/C++ data types.

Table 2-5. Data Type Sizes for the ADSP-TS001 and ADSP-TS101

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
long long int	64 bits signed
unsigned long long int	64 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 (written with “r” suffix) (C++ mode only)

Compiler Command-Line Reference

Analog Devices does not support data sizes smaller than a single word location for a processor. For the ADSP-TS001 and ADSP-TS101 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 2-88](#).

Integer

On any platform the basic type `int` will be the native word size; on TigerSHARC DSPs, it is 32 bits. Many library functions are available for both 32-bit and 64-bit integers. These functions provide support for the C/C++ data types `int` (or `long int`) and `long long int`. Pointers are the same size as `ints`. The `long long int` data type provides 64-bit integers.

Floating Point

On TigerSHARC DSPs, the `long` data type is a 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 */
long double sind (long double); /* 64-bit */
```

For full descriptions of these functions and their implementation, see [“Run-Time Library” on page 3-1](#).

Data Type Alignment

The TigerSHARC run-time model imposes some restrictions on data type alignment that are more stringent than the standard C/C++ alignment restrictions. Alignment of data objects can be further altered using the `#pragma align` option.

By default, non aggregate data objects are aligned on word boundaries (32 bits) for data types up to and including `long int`. The `long long int` and `long double` (and `double` if `double-size-64` is specified on the command line) data types are aligned on double-word boundaries (64 bits). The extended type `__builtin_quad` is aligned on a four-word boundary (128 bits).

Aggregate types have special rules:

- Top level arrays are aligned on 4-word boundaries; these are arrays that are not part of any other aggregate type. Statics and automatic arrays are aligned in this way to allow the option of using the TigerSHARC DSPs’ quad access instructions.
- `struct/union/class` objects are aligned to the maximal alignment of the members and have padding at the end of the structure to maintain alignment in arrays.

Compiler Command-Line Reference

The special `pragma align <alignment>` can be used to alter the alignment of data objects or type members. This allows the programmer to increase the alignment restriction for a variable or for an element declared within a structure. For example:

```
typedef struct {  
    #pragma align 4  
    int foo;  
    int bar;  
    #pragma align 4  
    int baz;  
} aligned_ints;
```

Any object declared of type `aligned_ints` will have the `foo` and `baz` members aligned on quad word boundaries. The size of this structure will be eight words with two words unused between `bar` and `baz`, and three words of padding at the end of the structure.

Optimization Control

The ccts 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 2-24 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 2-29 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 2-25](#) for more information.

Interprocedural analysis (see [“Interprocedural Analysis” on page 2-50](#)) 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, `ccts` affects 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 2-30](#) for more information.)

The `-O` switch requests the compiler to affect all generally 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.

Compiler Command-Line Reference

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.

Inlining Control

By default, ccts 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 ccts compiler has a capability called *interprocedural analysis*, 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 buildup 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

Compiler Command-Line Reference

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/ISO standards for the C and C++ language. 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/ISO 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 shorter forms of the keyword extensions that omit the leading underscores. [For more information, see “C/C++ Mode Selection Switch Descriptions” on page 2-18.](#)

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 use `inline` or `__inline` interchangeably in your code.

You might need to use the longer forms (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, you should use the `-no-extra-keywords` switch. If you need to declare a function as `inline` or allocate variables to memory spaces, you can use `__inline` or `__pm/__dm` respectively.

C/C++ Compiler Language Extensions

This section provides an overview of the extensions, brief descriptions of each, and directs you to text with further information on each extension. [Table 2-6](#) and [Table 2-7](#) summarize the extensions.

Table 2-6. Keyword Extensions

Keyword extensions	Description
<code>inline(function)</code>	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 2-56.
<code>long long [int]</code>	Provides 64-bit integer support, both signed and unsigned. For more information, see “64-Bit Integer Support (long long)” on page 2-57.
<code>dm</code>	Specifies the location of a static or global variable or qualifies a pointer declaration “*” as referring to primary data memory. For more information, see “Dual Memory Support Keywords (pm dm)” on page 2-58.
<code>pm</code>	Specifies the location of a static or global variable or qualifies a pointer declaration “*” as referring to secondary data memory. For more information, see “Dual Memory Support Keywords (pm dm)” on page 2-58.
<code>__regclass(unit)</code>	Gives the compiler hints about which TigerSHARC unit to use for a particular calculation.
<code>section("string")</code>	Specifies the section in which an object or function is placed. For more information, see “Placement Support Keyword (section)” on page 2-64.
<code>restrict keyword</code>	Specifies restricted pointer features. For more information, see “Pointer Class Support Keyword (restrict)” on page 2-65.
<code>bool, true, false</code>	A boolean type. For more information, see “Boolean Type Support Keywords (bool, true, false)” on page 2-65.

Table 2-7. Operational Extensions

Operation extensions	Description
Variable-length arrays	Lets you use automatic arrays whose length is not known until runtime. For more information, see “Variable Length Array Support” on page 2-66.
Non-constant aggregate initializers	Lets you use non-constants as elements of aggregate initializers for automatic variables. For more information, see “Non-Constant Aggregate Initializer Support” on page 2-68.
Indexed initializers	Lets you specify elements of an aggregate initializer in an arbitrary order. For more information, see “Indexed Initializer Support” on page 2-68.
Preprocessor generated warnings	Provides support for generating warning messages from the preprocessor. For more information, see “Preprocessor Generated Warnings” on page 2-87.
C++-style comments	Provides support for C++-style comments in C programs. For more information, see “C++-Style Comments” on page 2-87.
fract data type (C++ mode)	Provides support for the fractional data type, fractional and saturated arithmetic. For more information, see “C++ Fractional Type Support” on page 2-88.
Quad word support	Provides support for a quad word (128 bit) data type. For more information, see “Quad Word Support” on page 2-57.

Inline Function Support Keyword (inline)

The `ccts inline` keyword directs `ccts` 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 and to declare it `static`.

In some cases, `ccts` does not support 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 assembly code.



The `-no-inline` and `-traditional` switches disable function inlining. For more information, see [“C/C++ Mode Selection Switch Descriptions” on page 2-18.](#)

64-Bit Integer Support (long long)

In addition to the basic C/C++ data types, ccts provides an additional integral type that consists of 64-bit integers. This type is provided in both signed and unsigned forms. It is fully supported, including all arithmetic operations and relevant libraries.

Type Name	<code>long long [int]</code> <code>unsigned long long [int]</code>
Representation	64-bit 2's complement
Range	<code>long long int</code> : $-2^{63} \dots (2^{63})-1$ <code>unsigned long long int</code> : $0 \dots (2^{64}) - 1$

Normal library functions (as in `<stdlib.h>`) taking this type typically have an “ll” prefix. Formatting support (`printf` etc.) is available via the “ll” modifier, as in “%lld”.



This extension does not introduce additional keywords, so it does not interfere with porting standard-conforming programs onto TigerSHARC DSPs. The extension is always available; neither `-ansi` nor `-pedantic` affect it.

Quad Word Support

Certain features of TigerSHARC DSPs make use of 128-bit data types (a quad word). An extension to the language provides a data type of this size. Normal C/C++ arithmetic operators are not supported for the quad word type; therefore, supplied built-in functions must be used to process variables of this type. The type name is `__builtin_quad`.

Dual Memory Support Keywords (pm dm)

This section discusses ccts C language extension keywords that support the three-memory architecture of the TigerSHARC DSPs. 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.

These keywords allow you to control placement of data in primary (`dm`) or secondary (`pm`) data memory. No data is placed in the memory unit that holds programs.

The following rules apply to dual memory support keywords:

- The memory space keyword (`dm` or `pm`) refers to the expression that follows 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 primary Data Memory as the default memory space for all variables. All undeclared spaces for data are primary Data Memory space.
- You cannot assign memory spaces to automatic variables. All automatic variables reside on the primary stack, which is always in primary Data Memory.
- Literal character strings always reside in primary Data Memory.

[Listing 2-1 on page 2-59](#) shows examples of dual memory keyword syntax.

Listing 2-1. Dual Memory Keyword Syntax

```
int pm abc[100];
/* declares an array abc with 100 elements in pm (secondary
   data memory) */

int dm def[100];
/* declares an array def with 100 elements in primary data
   memory */

int ghi[100];
/* declares an array ghi with 100 elements in primary data
   memory */

int pm * pm pp;
/* declares pp to be a pointer which resides in secondary data
   memory and points to a pm (secondary data memory) integer */

int dm * dm dd;
/* declares dd to be a pointer which resides in primary data
   memory and points to a primary data memory integer */

int *dd;
/* declares dd to be a pointer that points to a primary data
   memory integer */

int pm * dm dp;
/* declares dp to be a pointer which resides in primary data
   memory and points to a pm (secondary data memory) integer */

int pm * dp;
/* declares dp to be a pointer which resides in primary data
   memory and points to a pm (secondary data memory) integer */

int dm * pm pd;
/* declares pd to be a pointer which resides in pm (secondary
   data memory) and points to a primary data memory integer */

int * pm pd;
/* declares pd to be a pointer which resides in pm (secondary
   data memory) and points to a primary data memory integer */

float pm * dm * pm fp;
/* the first pm means that *fp is in pm (secondary data memory),
   the following dm puts *fp in primary data memory, and fp
   itself is in pm (secondary data memory) */
```

C/C++ Compiler Language Extensions

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 * PFL0ATP;
/* PFL0ATP 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 `pm` should point to `pm`. Pointers to `dm` should point to `dm`. You may not mix addresses from different memory spaces within one expression. Do not attempt to explicitly cast one type of pointer to another. If a qualified pointer does not actually refer to the specified memory, then less efficient code is generated.

The careful use of different memory spaces for data can substantially improve the execution time of your program. The processor can make two simultaneous references to memory when different spaces are used. If, through casting or other means, you confuse the association of a pointer with its memory space, the program still executes correctly, but any simultaneous accesses require an additional cycle.

Also, when working with pointers to different memory spaces, the compiler assumes they do not “alias,” which means that they cannot access the same data. This often increases the amount of optimization that can be done.

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 a separate memory block that is neither `dm` or `pm`. The following listing shows some sample function declarations with pointers.

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

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

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

C/C++ Compiler Language Extensions

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

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

Memory Keywords and Function Arguments

ccts checks calls to prototyped functions for memory space specifications consistent with the function prototype. The following listing shows sample code that ccts 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. ccts 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 kind easier, you can use 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);
```

__regclass Construct

The `__regclass(unit)` construct allows you to give the compiler hints about which TigerSHARC unit to use for a particular computation. Variables can be annotated with this construct, and the compiler then tries to perform computations involving those variables in the indicated units. The (*unit*) argument is a quoted string, naming one of the four major units: “X”, “Y”, “J”, “K”.

C/C++ Compiler Language Extensions

`__regclass` may be used anywhere that the `C register` qualifier may be used and is subject to the same restrictions: it cannot have its address taken. Effective use of this feature requires a good understanding of the TigerSHARC architecture. Typically, you will examine the assembly code produced by the optimizing compiler.

In the absence of specific `__regclass` hints, the TigerSHARC compiler uses the following criteria:

- Floating point calculations are done in compute block X.
- General integer calculations are done in compute block Y.
- Address calculations, including pointer arithmetic, are done in the JALU. If a multiplication is needed, it is done in Y.

A common way of achieving higher performance, particularly in tight loops, is to perform a part of the computation in a different unit. If the subcomputations assigned to different units are independent of each other, they will execute in parallel, resulting in increased throughput.

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. The LDF file supplied to the linker must also be updated to support the additional named sections.

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 static duration, i.e. are explicitly `static`, or are given as external-object definitions.

The example below 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` extension keyword. Although the `segment()` struct is supported by the compiler of the present release, we recommend that you 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. `bool` 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.

C/C++ Compiler Language Extensions

The compiler is free to use the information about restricted pointers and aliasing in order to better optimize C/C++ code that uses pointers.

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, which are declared like other automatic arrays with the exception of a length that is not a constant expression. The storage is allocated at the point of declaration and deallocated when the brace-level is exited.

Jumping or breaking out of the scope of the array deallocates the storage. Jumping into the scope is not allowed and produces a compile time error message.

You can also use variable-length arrays as arguments to functions:

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

The length of an array is computed once when the storage is allocated, and it is remembered for the scope of the array in case you access it with `sizeof()`. Because variable length arrays must be stored on the stack, it is impossible to have variable length arrays in secondary data memory (pm). 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 automatic variable size matrix is much easier then explicitly allocating it from the heap; and the space is to be automatically released.

The expression declares an array with a size that is computed at runtime. 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 and size information will be deallocated. For example, the following program prints 40, not 50:

```
main ()
{
    foo(40);
}

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

Non-Constant Aggregate Initializer Support

The ccts compiler provides support for initializers beyond that provided in standard C or C++. The elements of an aggregate initializer for an automatic variable are not required to be constant expressions in ccts. The example below shows an initializer with elements whose values must be computed at run-time:

```
void calc_freq (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. ccts C/C++, by comparison, supports labeling elements for array initializers. This feature lets you specify array or structure elements in any order by specifying the array indices or structure field names to which they apply. To specify an array index, write the `[index]` before the element value. The *index* values must be constant expressions, even if the array being initialized is automatic. The following example shows equivalent array initializers: the first one in is presented in standard C/C++ followed by one that uses ccts C/C++.

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

/* Standard C/C++ array initializer */

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

/* equivalent ccts 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. Each initializer element that does not have a label applies to the next consecutive element of the array or structure. The standard and ccts C/C++ instructions shown in the following example are equivalent:

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

/* Standard C/C++ array initializer */

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

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

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

Labeling the elements of an array initializer is especially useful when the indices are characters or belong to an `enum` type, as in the following example:

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

/* ccts 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 and ccts C/C++ struct initializers shown in the following example are equivalent:

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

/* Standard C/C++ struct Initializer */

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

C/C++ Compiler Language Extensions

```
/* Equivalent ccts C/C++ struct Initializer With Labeled  
Elements */  
  
struct point {int x, y;};  
struct point p = {y: yvalue, x: xvalue};
```

Built-In Functions

The compiler supports intrinsic (built-in) functions that enable you to make efficient use of hardware resources. The compiler is aware of the definition of these intrinsic functions without the use of a header file. 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 '+' or '*'.

Built-in functions have names that begin with `__builtin`.



Identifiers beginning with '`__`' are reserved by the C standard, so these names do not conflict with user-defined identifiers.

These functions are specific to individual architectures; the built-in functions supported at this time on TigerSHARC are described in subsections that follow.

Various system header files provide the user with definitions and access to the intrinsics. This feature may be disabled using the `-no-builtin` switch.

Data Types (16-Bit)

The functions and operators described in this section perform arithmetic on the `int2x16`, and `int4x16` data types.

- The `int2x16` data type consists of two 16-bit integers packed into a 32-bit structure.
- The `int4x16` data type consists of four 16-bit integers packed into a 64-bit item.

In the following table, the notation {A,B} represents an `int2x16` containing the two 16-bit values A and B. The TigerSHARC DSPs have a little endian architecture, so the “first” element is stored in the low order bits. A is considered to be the “first” element of {A,B}, so that the representation in a 32-bit register is as follows.

An `int2x16` has the following structure.

31	16	15	0
Value B		Value A	
Bit 15.....Bit 0		Bit 15.....Bit 0	

Similarly, {A,B,C,D} represent an `int4x16` containing the 16-bit values A, B, C, and D, with A in the lowest order bits, B next, followed by C, and D in the highest order bits. (This means that C and D will be stored in a register or memory location numbered one higher than A and B.)

Most operations on these types are element-wise. For example, the add operation for `int2x16` takes the first 16-bit quantity from each argument and adds them together to produce the first 16-bit quantity in the result. Likewise, the second 16-bit elements are added to produce the second element of the result. For example, the `int2x16` add operation on arguments {A,B} and {C,D} produces a result {A+C,B+D}.

A few operations which are not element wise are also supported. The sideways sum operation adds up all of the 16-bit elements in its single argument. A variety of packing and unpacking operations for accessing the individual elements of a packed value are also available.

These operations are very efficient for the ADSP-TS001 because they are directly supported in hardware. Most are implemented as a single machine instruction; a few which can take multiple machine instructions are provided for convenience.

C/C++ Compiler Language Extensions

Note that some operations are defined in terms of a 2x32 data type. This type and its operations are described starting on [page 2-81](#).

Packed 16-bit Integer Support Using C

The C language version used at Analog Devices lets you write your own program to operate on 16-bit packed data. To use the `int2x16` and `int4x16` types and operators in your Analog Devices C program, add a `#include <i16.h>` directive.

Constructors (int2x16 values)

The following functions create `int2x16` values.

Synopsis: `int2x16 compact_to_i2x16_from_i32 (int lo, int hi);`

Description: Constructs an `int2x16` by compacting two 32-bit integer values. The high order 16 bits of each original 32-bit value are lost.

Algorithm: `compact(A_32, B_32) => {A_16,B_16}`

Synopsis: `int2x16 compact_to_i2x16 (int2x32 val);`

Description: Construct an `int2x16` by compacting the two halves of a single `int2x32`. The high order 16 bits of each of the 32-bit values are lost.

Algorithm: `compact({A_32,B_32}) => {A_16,B_16}`

Extractors and Expanders (int2x16 values)

The following functions extract the latest or most significant 16-bit value from an `int2x16`.

Synopsis:

```
int expand_low_of_i2x16 (int2x16 val);
int expand_high_of_i2x16 (int2x16 val);
```

Description: The value returned is sign-extended to 32 bits and returned as an `int`.

Algorithm:

Given `val = {A_16,B_16}`:

```
expand_low_of_i2x16(val) => A_32
expand_high_of_i2x16(val) => B_32
```

See Also: `expand_i2x16_to_i2x32`

Arithmetic Operators

The following binary operators are defined on `int2x16`:

Synopsis:

```
int2x16 add_i2x16 (int2x16 a, int2x16 b);
int2x16 sub_i2x16 (int2x16 a, int2x16 b);
int2x16 mult_i2x16 (int2x16 a, int2x16 b);
```

Description: Perform element-wise arithmetic on `int2x16` values. The TigerSHARC does not have a 2x16 multiplication instruction; this multiply operation uses the 4x16 multiply and discards half of the result. It is still accomplished in one instruction.

C/C++ Compiler Language Extensions

Algorithm:

Addition: $\{A, B\} + \{X, Y\} \Rightarrow \{A+X, B+Y\}$

Subtraction: $\{A, B\} - \{X, Y\} \Rightarrow \{A-X, B-Y\}$

Multiplication: $\{A, B\} * \{X, Y\} \Rightarrow \{A*X, B*Y\}$

Unary minus is not yet supported on `int2x16`.

Bitwise Operators (`int2x16` values)

The following bitwise logical operators are defined on `int2x16`:

Synopsis:

```
int2x16 a, b;  
a ^ b  
a & b  
a | b  
a ^= b  
a &= b  
a |= b  
~a
```

Description: Perform logical operations on `int2x16` values. Note that the traditional infix operators are available here.

Algorithm:

Xor: $\{A, B\} \wedge \{X, Y\} \Rightarrow \{A \wedge X, B \wedge Y\}$

And: $\{A, B\} \& \{X, Y\} \Rightarrow \{A \& X, B \& Y\}$

Or: $\{A, B\} | \{X, Y\} \Rightarrow \{A | X, B | Y\}$

Complement: $\{A, B\} \Rightarrow \{\sim A, \sim B\}$

Shift Operators

Not yet supported.

Comparison Operators (int2x16 values)

The following operators are used to compare `int2x16` values.

Synopsis:

```
int2x16 a, b;
a == b;
a != b;
```

Description: The `==` operator returns true if both elements are equal. The `!=` operator returns true unless both values are equal. Other comparison operators (`<`, `<=`, `>=`, `>`) are not supported.

Algorithm:

Equal: $\{A,B\} == \{C,D\} \Rightarrow (A == B \ \&\& \ C == D)$

Not Equal: $\{A,B\} != \{C,D\} \Rightarrow (A != B \ || \ C != D)$

Sideways Sum (int2x16 values)

This function adds together the two values in an `int2x16`.

Synopsis: `int sum_i2x16 (int2x16);`

Description: This operation produces a single 32-bit integer result.

Algorithm:

$\text{sum}(\{A,B\}) \Rightarrow A+B$

C/C++ Compiler Language Extensions

Example (int2x16 values)

This example shows some of the `int2x16` operations, including conversion to and from normal ints. Note that it is often helpful in debugging to print the packed values in hex, rather than decimal, as the two halves can be seen more easily.

```
#include <stdio.h>    // for printing at end
#include <i16.h>
void i2x16_example() {
    int u, v, s;
    int
        x = 3,
        y = 5,
        z = 7,
        w = 9;
    int2x16 aa, bb, cc, dd;
        // compact integers into packed 16 bit form
    aa = compact_to_i2x16_from_i32(x, y);
        //aa = {3, 5} or 0x00050003
    bb = compact_to_i2x16_from_i32(z, w);
        //bb = {7, 9} or 0x00090007
        // construct a packed constant
    cc = compact_to_i2x16_from_i32(1, -1);
        //cc = {1, -1} or 0xffff0001
        // a little arithmetic
    dd = mult_i2x16 (add_i2x16 (aa, bb), cc);
        //dd = (aa + bb) * cc;
        //dd = {10, -14} or 0xffff2000a
        // sideways sum
    s = sum_i2x16(dd);    //s = -4 or 0xffffffffc
        // extract components
    u = expand_low_of_i2x16(dd);    // low-order part
    v = expand_high_of_i2x16(dd);    // high-order part
    printf("results: s = %d; dd = %8x, u = %d, v = %d\n", s, dd,
u, v);
}
```

Prints:

results: s = -4; dd = fff2000a, u = 10, v = -14

Constructors (int4x16 values)

The following functions create `int4x16` values

Synopsis:

```
int4x16 compact_to_i4x16_from_i32 (int llo, int lhi, int hlo,
int hhi);
```

Description: Construct an `int4x16` by compacting four 32-bit integer values. The high order 16 bits of each original 32-bit value are lost.

Algorithm:

```
compact(A_32, B_32, C_32, D_32) => {A_16,B_16,C_16,D_16}
```

Synopsis:

```
int4x16 compose_i4x16_from_i32 (int2x16 low, int2x16 high);
```

Description: Construct an `int4x16` from two `int2x16`s.

Algorithm:

```
compose ({A, B}, {C, D}) => {A,B,C,D}
```

C/C++ Compiler Language Extensions

Extractors (2x16 from a 4x16 value)

This function extracts a 2x16 from a 4x16

Synopsis:

```
int2x16 extract_low_of_i4x16(int4x16 val);  
int2x16 extract_high_of_i4x16(int4x16 val);
```

Description: Extracts a int2x16 from a int4x16. Often it is convenient to use the various expand operators to break the i4x16 apart in steps, holding onto the intermediate forms, as shown in the following listing:

```
Given int4x16 val = {A_16,B_16,C_16,D_16}:  
int2x16 low_part = extract_low_of_i4x16(val);  
int2x16 high_part = extract_high_of_i4x16(val);  
int a = expand_low_of_i2x16(low_part);  
int b = expand_high_of_i2x16(low_part);  
int c = expand_low_of_i2x16(high_part);  
int d = expand_high_of_i2x16(high_part);
```

- or -

```
Given int4x16 val = {A_16,B_16,C_16,D_16}:  
int2x32 low_part = expand_i2x16(extract_low_of_i4x16(val));  
int2x32 high_part = expand_i2x16(extract_high_of_i4x16(val));  
int a = low_32(low_part);  
int b = high_32(low_part);  
int c = low_32(high_part);  
int d = high_32(high_part);
```

Split an int4x16 into component integer values

Algorithm:

```
extract_low ({A_16,B_16,C_16,D_16}) => {A_16,B_16}  
extract_high ({A_16,B_16,C_16,D_16}) => {C_16,D_16}
```

Synopsis:

```
void expand_i4x16_to_i32(int4x16 input,
    int *llo, int *lhi, int *hlo, int *hhi);
```

Description: Using a cascade of expansion and selection, this breaks a 4x16 apart into four separate integers.

Algorithm:

```
Given val = {A_16,B_16,C_16,D_16}:
int a, b, c, d;
expand_i4x16_to_i32 (val, &a, &b, &c, &d);
a => A_32
b => B_32
c => C_32
d => D_32
```

Arithmetic Operators (int4x16 values)

The following binary operators are defined on int4x16:

Synopsis:

```
int4x16 add_i4x16 (int4x16 a, int4x16 b);
int4x16 sub_i4x16 (int4x16 a, int4x16 b);
int4x16 mult_i4x16 (int4x16 a, int4x16 b);
```

Algorithm:

Addition: {A,B,C,D} + {W,X,Y,Z} => {A+W, B+X, C+Y, D+Z}
Subtraction: {A,B,C,D} - {W,X,Y,Z} => {A-W, B-X, C-Y, D-Z}
Multiplication: {A,B,C,D} * {W,X,Y,Z} => {A*W, B*X, C*Y, D*Z}

Description:

Perform element wise arithmetic on int4x16 values.



Unary minus is not yet supported on int4x16.

C/C++ Compiler Language Extensions

Sideways Sum (int4x16 values)

This function adds together the four values in an int4x16.

Synopsis:

```
int sum_i4x16(int4x16);
```

Description: This operation produces a single 32-bit integer result.

Algorithm: $\text{sum}(\{A,B,C,D\}) \Rightarrow A+B+C+D$

Example (int4x16 values)

The following shows some examples of int4x16 operations, including conversion to and from normal ints.

```
#include <stdio.h>
#include <i16.h>
void i4x16_example() {
    int w, x, y, z, s;
    int
        p = 3,
        q = 5,
        r = 7,
        s = 11,
        t = 13,
        u = 17,
        v = 19,
        g = 23;
    int4x16 aaaa, bbbb, cccc, dddd;

    // compact integers into packed 16 bit form
    aaaa = compact_to_i4x16_from_i32(p, q, r, s);
    bbbb = compact_to_i4x16_from_i32(t, u, v, g);
    cccc = compact_to_i4x16_from_i32(1, -1, 3, -3);
    // construct a packed constant
    // a little arithmetic
    dddd = mult_i4x16 (add_i4x16 (aaaa, bbbb), cccc);
    // dddd = (aaaa + bbbb) * cccc;
    s = sum_i4x16(ddd); // sideways sum
    // extract components
    w = dddd[0]; // low-order part
```

```

    x = dddd[1]; // high-order part
    y = dddd[2];
    z = dddd[3];
    printf("results: s = %d; w = %d, x = %d, y = %d, z = %d\n",
           w, x, y, z);
}

```

Prints:

results: sm = -30; w = 16, x = -22, y = 78, z = -102

Data Types (32-Bit)

The `int2x32` type is primarily used internally when converting between packed and expanded values. It often provides a useful way-station, allowing you to capture a partial expansion for later separation into smaller parts.

Constructors (`int2x32` values)

The following functions create `int2x32` values.

Synopsis: `int2x32 compose_64 (int lo, int hi);`

Algorithm: `compose(A_32, B_32) => {A_32, B_32}`

Description: Constructs an `int2x32` from two 32-bit integer values. The values are unchanged.

Synopsis: `int2x32 expand_i2x16 (int2x16 val)`

Algorithm: `expand ({A_16,B_16}) => {A_32, B_32}`

Description: Expands an `int2x16` to an `int2x32` sign extending each value to 32 bits.

C/C++ Compiler Language Extensions

Extractors (int2x32 values)

The following functions extract the least or most significant 32-bit value from an `int2x32`. The unchanged 32-bit value is returned as an `int`.

Synopsis: `int low_32 (int2x32 val);`
`int high_32 (int2x32 val);`

Algorithm:

```
low(A_32, B_32) => A_32  
high(A_32, B_32) => B_32
```

Math Intrinsics

Each of the built-in functions below map directly to a single machine instruction and are therefore very efficient.

```
/* Intrinsics defined in <stdlib.h> */  
/* "int" versions */  
int  abs (int j);           // absolute value  
int  avg (int a, int b);    // (a+b)/2  
int  clip (int a, int b);   // bound a by b (positive and negative)  
int  max (int a, int b);    // maximum  
int  min (int a, int b);    // minimum  
int  count_ones (int numb); // number of "1" bits  
  
/* "long int" versions */  
long  labs (long j);  
long  lavg (long a, long b);  
long  lclip (long a, long b);  
long  lmax (long a, long b);  
long  lmin (long a, long b);  
int   lcount_ones (long numb);  
  
/* "long long int" versions */  
long long int  llabs (long long int j);  
long long int  llavg (long long int a, long long int b);  
long long int  llclip (long long int a, long long int b);  
long long int  llmax (long long int a, long long int b);  
long long int  llmin (long long int a, long long int b);
```

```

int                llcount_ones (long long int numb);

/* bit-reversed addition */
int  addbitrev (int input_address, int number_of_bits);

/* Math intrinsics from <math.h> */
float  fabsf (float, float);
float  favgf (float a, float b);      // (a+b)/2
float  fclipf (float a, float b);    // bound a by b (pos & neg)
float  fmaxf (float, float);         // float maximum
float  fminf (float, float);         // float minimum
float  copysignf (float a, float b); // a with sign(b)

double  favgd (double, double);
double  fclipd (double, double);
double  fmaxd (double, double);
double  fmind (double, double);
double  copysignd (double, double);

long double  favgd (long double, long double);
long double  fclipd (long double, long double);
long double  fmaxd (long double, long double);
long double  fmind (long double, long double);
long double  copysignd (long double, long double);

```

Non-Standard Data Types

The ccts compiler supports access to machine-specific instructions via built-in functions. These functions are known to the compiler without the need to declare them (via a header file). However, for convenience, a header file is included with the other C-includes (`builtin.h`). This header file gives prototypes for all the built-in functions. This file is quite terse, but it does provide the correct call syntax for each function.

Every function name is prefixed with `__builtin_` followed by a functional name, which usually corresponds directly to the machine instruction that the builtin generates. Some functions, such as the `compose` or `extract` functions, may not generate code under optimization since the compiler may be able to arrange register assignments to eliminate the need for these functions.

C/C++ Compiler Language Extensions

The list of functions presented here may not be exhaustive since later compiler releases may add more functionality. Check release notes for any additions, or browse the `builtins.h` file for the complete list supported by the current version of the compiler. The functions are presented below without their `__builtin_` prefix.

The following functions generate ALU operations with non-standard types. They have been grouped alphabetically by operation, disregarding any prefix or suffixes that provide type information.

```
long long abs_4x16(long long );
int abs_4x8(int );
long long abs_8x8(long long );
int abs_2x16(int );

int add_2x16(int ,int );
int add_sat(int ,int );
int add_2x16_sat(int ,int );
long long add_4x16(long long ,long long );
long long add_4x16_sat(long long ,long long );
int add_4x8(int ,int );
int add_4x8_sat(int ,int );
long long add_8x8(long long ,long long );
long long add_8x8_sat(long long ,long long );

float fclipf(float ,float );
long long llclip(long long ,long long );
int clip_2x16(int ,int );
long long clip_4x16(long long ,long long );

long long max_4x16(long long ,long long );
int max_4x8(int ,int );
long long max_8x8(long long ,long long );
int max_2x16(int ,int );

long long merge_4x8(int ,int );
long long merge_2x16(int ,int );

int min_2x16(int ,int );
long long min_8x8(long long ,long long );
long long min_4x16(long long ,long long );
int min_4x8(int ,int );
```

```
int frmul(int ,int );
int frmul_sat(int ,int );

int mult_fr2x16(int ,int );
int cmult_i2x16(int ,int );
int cmult_fr2x16(int ,int );
int cmult_conj_i2x16(int ,int );
int cmult_conj_fr2x16(int ,int );
int mult_fr2x16_sat(int ,int );
int cmult_fr2x16_sat(int ,int );
long long mult_fr4x16(long long ,long long );
long long mult_fr4x16_sat(long long ,long long );

int neg_sat(int );
int neg_2x16_sat(int );
long long neg_4x16_sat(long long );

long long sub_8x8(long long ,long long );
long long sub_8x8_sat(long long ,long long );
int sub_sat(int ,int );
int sub_2x16(int ,int );
int sub_2x16_sat(int ,int );
long long sub_4x16(long long ,long long );
long long sub_4x16_sat(long long ,long long );
int sub_4x8(int ,int );
int sub_4x8_sat(int ,int );

int sum_8x8(long long );
int sum_2x16(int );
int sum_4x16(long long );
int sum_4x8(int );
```

C/C++ Compiler Language Extensions

The following functions are used to compact or merge values to create larger or smaller types. In some case the optimizer is able remove instructions that may be generated by these functions.

To combine smaller objects into a single larger object:

```
__builtin_quad compose_128(long long ,long long );  
unsigned long long compose_64u(unsigned int ,unsigned int );  
long double f_compose_64(float ,float );
```

To create an object of size n by s; n number of parts of size s bits:

```
int compact_to_fr2x16(long long );  
int compact_to_i4x8(long long );
```

To expand an object of size n by s into the next size up.

Into to fracts:

```
long long expand_fr2x16(int );
```

Into two 2x16's:

```
long long expand_i4x8(int );
```

To extract the low/high half out of various larger objects:

```
float f_high_32(long double );  
unsigned int high_32u(unsigned long long );  
long long high_64(__builtin_quad );  
unsigned int low_32u(unsigned long long );  
float f_low_32(long double );  
long long low_64(__builtin_quad );
```

To access to various system registers:



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.

```
int sysreg_read(int );
long long sysreg_read2(int );
__builtin_quad sysreg_read4(int );
void sysreg_write(int ,int );
void sysreg_write2(int ,long long );
void sysreg_write4(int ,__builtin_quad );
```

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, beginning with `//` and ending at the end of the line, in C programs. 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;
```

and a C++ compiler and ccts process as:

```
a = b;
```

C++ Fractional Type Support

While in C++ mode, the ccts 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-TigerSHARC DSPs.

Fractional values are declared with the `fract` data type. Ensure that your program includes the `<fract>` header file. `fract` 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 $[-1.0 \dots +1.0)$ 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.



Note that this implementation 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 ccts compiler validates fractional literal values 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 retains only 24 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 not 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 ccts 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.

C/C++ Compiler Language Extensions

- When arithmetic expressions contain a mixture of `fract` and `float` (or `double`) items, then the `float` (or `double`) items are normally converted to `fract` representation. For more information about mixed-mode arithmetic, see [page 2-90](#).
- 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 must explicitly cast the result of the multiply operation if the error occurs.

Mixed Mode Operations

Most operations that are supported for fractional values are supported for mixed fractional/float or fractional/double arithmetic expressions. At runtime, 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.

Preprocessor Features

Several features of the C/C++ compiler are used by VisualDSP++ to control the programming environment.

The ccts compiler provides standard preprocessor functionality, as described in any C text. The following extensions to standard C are also supported:

// end of line (C++-style) comments

#warning directive

For more information about these extensions refer to [“Preprocessor Generated Warnings”](#) and [“C++-Style Comments”](#) on page 2-87.

Predefined Macros

The ccts compiler defines a number of macros to produce information about the compiler, source file, and options specified. These macros can be tested using `#ifdef` and related directives to support your program’s needs. Similar tailoring is done in the system header files.

Macros such as `__DATE__` can be useful to incorporate in text strings. The `#` (to string) operator with a macro body is useful in converting symbols into text constants.

The ccts compiler provides the following macros:

`__ADSPTS__`

The compiler driver always defines `__ADSPTS__` as 1. This indicates that an ADSP_TS family program is being compiled.

Preprocessor Features

`__ADSPTS001__`

The compiler driver defines `__ADSPTS001__` as 1 unless you compile with the `-TS101` switch. This macro defines the chip model number.

`__ADSPTS101__`

The compiler driver defines `__ADSPTS101__` as 1 if you compile with the `-TS101` switch. This macro defines the chip model number.

`__ANALOG_EXTENSIONS__`

ccts defines `__ANALOG_EXTENSIONS__` as 1 unless you compile with the `-ansi`, `-pedantic` or `-pedantic-errors` switch.

`__CPLUSPLUS__`

ccts defines `__CPLUSPLUS__` as 1 when you compile in C++ mode.

`__DATE__`

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

`__DOUBLES_ARE_FLOATS`

ccts 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.

`__ECC__`

ccts always defines `__ECC__` as 1.

`__EDG__`

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

`__EDG_VERSION__`

ccts 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 ccts 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_BUILTIN`

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

`__SIGNED_CHARS__`

ccts 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__`

ccts defines `__STDC__` as 1 unless you compile with the `-traditional` switch. (ANSI standard).

Preprocessor Features

`__STDC_VERSION__`

`ccts` defines `__STDC__` as 199409L unless you compile with the `-traditional` switch. (ANSI standard).

`__TIME__`

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

Header Files

A header file contains C/C++ declarations and macro definitions. Use the `#include` preprocessor directive to access header files for your program. Header file names have an `.h` extension. There are two main categories of header files:

- System header files declare the interfaces to the libraries supplied with the TigerSHARC product. Include them in your program for the definitions and declarations you need to access. 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`:

Listing 2-2. Skip Spaces Macro

```
/* 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)
```

Preprocessor Features

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 would 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...
```

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.

The C/C++ run-time model is a set of conventions that C and C++ programs follow to run on TigerSHARC DSPs. Assembly routines that you link to C or C++ routines must follow these conventions. This section provides a full description of the ccts 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.

Stack Frame

The Stack frame (or activation record) provides for the following activities:

- Space for local variables for the current procedure. For the compiler, this includes temporary storage as well as that required for explicitly declared user automatic variables.
- Place to save linkage information such as return addresses, location information for the previous caller's stack frame and to allow this procedure to return to its caller.
- Space to save information that must be preserved and restored.
- Arguments passed to the current procedure. In addition, if this procedure is going to call other procedures, its stack frame also contains "outgoing" linkage and parameter space.

C/C++ Run-Time Model

In addition, if this is not a leaf procedure (if it is going to call other procedures), its stack frame also contains outgoing linkage and parameter space:

- Space for the arguments to the called procedure.
- Space for the callee to save basic linkage information.

i [Figure 2-1](#) is a general picture of the stack. Note that the stack grows downward on the page. Because the stacks grow towards smaller addresses, higher addresses are found in the upwards direction. Each row is a quad word group; the higher addresses are on the left since this is a little-endian machine.

Incoming Arguments	
Linkage Information	← FP
Linkage Information and Temporaries	
<u>Save Area (for caller info)</u>	
Outgoing Arguments	
Free Space	← SP

Figure 2-1. ADSP-TS001 Stack

There are two stacks and thus two current stack frames at any point on the ADSP-TS001 and ADSP-TS101. One stack resides in each of the data memory banks. One stack is addressed from the j iALU registers, the other stack is addressed from the k registers. [For more information, see “Parts of the Stack Frame” on page 2-100.](#)

Each stack is controlled by a pair of pointers: a Stack Pointer (SP), which identifies the boundary of the in-use portion of the stack space; and a Frame Pointer (FP), which provides stable addressing to the current frame.



The second stack is not yet used for local variables.

The following design attributes have been included in the ADSP-TS001 and ADSP-TS101 run-time model for additional efficiency:

- The frame pointers are offset by -0×40 from the actual base of the frame. This provides greater addressing range for larger negative offsets (for local variables) than positive ones (for arguments). Since all FP-based references are relative (i.e., with an offset), there's no run-time cost and little extra complexity. This offset is constant throughout the system; it does not change for particular routines. However, this may cause the Frame Pointer address to actually be off the end of the stack.
- Optionally, a copy of the j frame pointer may be kept in register $k24$. This increases performance by allowing simple j FP-relative memory references to occur in the same instruction line as other [address] calculations in the j ALU. This copy is made at the discretion of the particular procedure. A similar copy of kSP into j is also possible.
- A routine that does not wish to use the k stack need not do so. If i_r does not use the k stack, it can dispense with all operations, including linkage related to the k stack and k stack pointers.



At present, a k frame must always be allocated in order for the debugger to be able to walk back up the stack. At some future point, it is expected that the debug information will be enriched to include information on whether or not a k frame is present.

- Space is allocated in the caller's frame for at least four “argument words” (see detailed in the following section, “[Parts of the Stack Frame](#)”). For routines with few arguments, this space is available for use as temporary storage. That may be of use for small, leaf routines, which may be able to avoid having to create a stack frame.

Parts of the Stack Frame

This section describes each part of the stack frame as shown in [Figure 2-1 on page 2-98](#).

- Incoming Arguments

The memory area for incoming arguments begins at the effective `jFP` value `+8`, or actual `j26 + 32`. Argument words are mapped by ascending addresses, so the second argument word is mapped at `j26+33`. Note that the first four words arrive in registers, although space is allocated in memory as well.

- Linkage Information

The return address is saved in the `CJMP` register by the `CALL` instruction. This is saved at `jSP+0 (j27+0)` on entry. It is retrieved from effective `jFP + 0 (j26+64)` at exit.

- Local Variables and Temporaries

Space for a register save area and local variables/temporaries is allocated on both stacks by the procedure prologue. The save area begins after the outgoing argument space, typically at `jSP+12` and `kSP+8`. This can also be addressed from `jFP` and `kFP`. Local variable/temp storage begins after the save area. It is addressed at effective `jFP-offset` or effective `kSP-offset`. Since the actual `FP` registers are offset by a constant `0x40`, the actual offsets might be positive for some cases. This should cause no problems.

- **Outgoing Arguments**

Space for outgoing arguments is allocated on the stack at $jSP+8$ ($j27+8$). Four words are always available as part of the basic stack frame. If more are needed, they can either be allocated as part of the prologue or by local extension of the stack frame for the purpose of a particular call.

- **Outgoing Linkage**

Four words, at $jSP+4$ ($j27+4$) and $kSP+4$ ($k27+4$), are available immediately to the callee for saving linkage information. These words are designated for holding the quad registers containing the j and k stack pointer registers. Four words, at $jSP+0$ ($j27+0$) and $kSP+0$ ($k27+0$), are also available at any time, and may be used by the callee. [For more information, see “Linkage Information” on page 2-100.](#)

- **Free Space**

Space below jSP and kSP is considered free and unprotected. It is available for use (in growing the stack) at any time, synchronously or asynchronously (the latter for interrupt handling). This space should never be used.

General System-wide Specifications

The following lists some general specifications that apply to stacks:

- The stacks grow down in memory from higher to lower addresses.
- The current frames' "base" are addressed by the effective `FP` registers (the value in `FP` plus `0x40`);
- The first free quad-word in each stack is addressed by the `SP` register. Locations at that point and beyond are vulnerable and must not be used. These locations may be clobbered by asynchronous activities, such as, interrupt service routines. Alternatively, locations at `SP` and beyond are always available if additional space is needed, but `SP` must be moved to guard the space. Therefore, the first "used" (protected) word is at `SP+4`.



Data can be pushed onto the stack by executing an instruction like `[jSP += -4] = reg;` for either `j` or `k`.

- The address of each `FP` must always be four-word aligned, i.e., the low-order two address bits are always zero.
- Likewise, the `SP` registers should also be kept four-word aligned at all times. This allows interrupt routines to save registers without having to first verify stack alignment.
- The return address of the caller is stored at offset zero from the address carried by the current effective `jFP`.
- The linkage back to the previous stack frame is stored at offset `+6` and `+7` from each current effective `FP`.

At a procedure call, the following must be true:

- Each `SP` must be four-word aligned (as it will form the basis for the new frame's `FP`).

- There must be eight words available starting at $jSP+4$ and 4 words available starting at $kSP+4$. The first four words on each stack are used for storing linkage information, and the remaining four (and perhaps others) on the j stack hold arguments. There is always space allocated for at least four arguments, even if the current procedure does not have that many. That way, the callee can always store the argument registers. Small routines are free to use unneeded argument slots for local storage.
- The standard calling convention further specifies that the previous frame's linkage information will already be stored in the standard slots by the time control reaches the new procedure. The callee is responsible for restoring this information prior to exit, however.

The saving of the frame linkage can be performed as part of the instruction line containing the procedure call. In some instances this information will remain constant for several outgoing calls, in which case it need not be repeatedly stored.

Argument Passage

A contiguous block of memory, near the end of the current frame, is provided to hold all arguments. Argument handling begins by conceptually “mapping” all the arguments into this area. The arguments are laid out into ascending addresses. They must obey alignment constraints based on their size. An argument might occupy more than one word; there might also be a vacant word, or “hole”, in the argument area to maintain alignment. This is the definition of “argument words”.

Data corresponding to the first four words of the argument area are normally passed in registers, rather than on the stack, which increases efficiency. For a common situation where each argument occupies a single word, the first four arguments can be passed in registers. There is also space allocated on the stack so that these register values can be stored back in the prologue in their canonical places.

C/C++ Run-Time Model

The choice of register file used for argument passage is based on the argument's declared type. Although two register files are available, only four argument words are passed. The corresponding word in the other register file is unused. (See [Table 2-8 on page 2-104](#)). Rules governing argument words are as follows:

- Pointers and integral types of one word are passed in the *j* registers
- Floating types, integral types of more than one word, and structures (or unions) that fit are passed in the *x* registers. This use of registers to pass structures or unions operates independently of the type of the structure or union fields. (A structure composed entirely of pointers is still passed in *x*). The alignment constraints are still observed.
- If an argument requires sufficient space that causes it to span over the end of the register argument area (typically, if it requires more than two words), it and all succeeding arguments are passed in memory in the usual place rather than in the registers.
- If the prototype specifies `varargs` (“...”), then the argument immediately preceding the ellipsis and all succeeding arguments are passed in memory. Note that `varargs` routines **MUST** have prototypes in order to function correctly.
- If there is no prototype, the actual types of the arguments are used to determine how they are passed.

Table 2-8. Register–Argument Word Correspondence

Argument Word	Register Used	
	if pointer or int	if float or double word
Arg word 1	j4	xR4
Arg word 2	j5	xR5

Table 2-8. Register–Argument Word Correspondence (Cont'd)

Argument Word	Register Used	
	if pointer or int	if float or double word
Arg word 3	j6	xR6
Arg word 4	j7	xR7

- Large structures may be passed by value. The structures are copied into the argument area, just as with other arguments.

Return Values

Return values always use registers. The type of the value determines, as with arguments, whether to use the *j* or *x* register file (*j*8, or *x*R8 and *x*R9).



Two *x* return registers are identified to accommodate double-word result values.

If the return value is larger than two words, then the caller must allocate space and pass the address in, as a “hidden argument”. The *j* return register, *j*8, is used for this purpose. The procedure must also return the [same] pointer value in that register on return. This is an optimization to allow more efficient referencing of the returned value.

Procedure Call and Return

To call a procedure:

1. Expand the current stack frame, if necessary, to provide space for the outgoing arguments and linkage information.

The number of arguments must be rounded up to a multiple of four in order to maintain proper alignment.

It is recommended that you establish and retain a basic call area during prologue. This saves you from creating and removing the area at each procedure call. If the number of arguments is large or a nested call occurs you must do a full expand-remove cycle.

2. Evaluate the arguments and set them up in the argument area and/or registers.

If another function must be called as part of computing an argument, then the stack can be extended (temporarily) a second time. If you extend the stack a second time, it's probably best to store even the register arguments of the outer procedure into the stack (for safekeeping), then load them just prior to the actual call.

3. Call the procedure, saving the current frame pointers in the appointed slots.
4. On return, remove the frame expansion if desired.

On Entry:

1. Set the new frame pointers (in both `iALU`'s) to the value of the current stack pointer, less the `0x40` offset. This can be done simultaneously in both `iALU`'s.
2. Set up a new frame and continue saving context: store the `cjmp` (return) register on the `j` stack at offset `+0` from the current (old) stack pointer. By using post-modify addressing on the store instruction, the stack pointer may simultaneously be moved down to create a new frame.

Registers are saved on the `k` stack as needed while the frame is created.



After this step, the new frame is officially in place.

3. If debug is specified (`-g` switch), arguments arriving in registers should be stored back into memory so the debugger can find them.



This restriction may be lifted in future releases when the debug tables are more expressive.

4. If desired, transfer a copy of the new `jFP` register to `k`.
5. Continue saving registers and then executing the procedure.

A leaf procedure that does not require much stack space might choose to omit steps (1) and (2), operating without its own stack frame. A small amount of local storage is available in any unused argument slots since there are always at least four slots, as well as, the two quad words reserved for saving linkage information.

C/C++ Run-Time Model

To Return from a Procedure:

1. If a call was made to another procedure, then `CJMP` must be restored.

For best performance, restore `CJMP` as soon as possible after the last procedure call or computed jump, (such as a switch). If the return jump is predicted, stalls can be avoided provided that `CJMP` is reloaded four cycles plus 12 words prior to the return.

2. Restore miscellaneous saved registers.
3. Place the return value in the correct register (if not there already).
4. Restore the stack pointers for the previous frame. This can be accomplished, for each stack, with a single quad load.
5. Return to the caller.

Code Sequences

[Listing 2-3](#) shows the normal code sequences used for the various actions involved in calling a procedure and returning from a procedure.

Listing 2-3. Procedure Call

```
// Extend stack frame if necessary (more than 4 arguments)
jsp = jsp - nn; ksp = ksp - mm;; // must be multiple of 4
//Evaluate arguments. Store into memory, or into arg
registers.
...
// Now do the call.
call subr; q[jSP+4]=j27:24; q[kSP+4]=k27:24;;
```

The fourth slot might be available for some computation or it may be used by the `IMEX` forming the full address for the call.

```
// if the stack was extended, cut it back.
jSP = jSP + nn; kSP = kSP + mm;; // must be multiple of 4
```

Prologue

```
jFP = jSP - 0x40; kFP = kSP - 0x40;; [jSP += -nnj]=CJMP;
    q[kSP += -nnk]=<some regs>;
    //At this point, the new frame is intact.
    q[jSP+n]=<some quad>; kjFPcopy = jFP;
```

Epilogue

```
//Restore CJMP after last outgoing call
CJMP = [jFP+0x40];
```

More code

```
// Nearing end, restore other saved registers.
...
// Exit: Restore stack linkage and return.
jump CJMP; j27:24=q[jFP+0x44]; k27:24=q[kFP+0x44];;
```



There is a four-cycle wait before either frame pointer can be used again. If the return jump is not predicted correctly, the four-cycle wait will cover the delay. Otherwise, callers might want to refrain from immediately accessing the frame pointers on return.

You should not reset the stack pointers prior to return. That would create a time window in which the stack has been reset to a frame not corresponding to the current procedure.

Miscellaneous Information

This section contains a number of miscellaneous aspects of the design that may be helpful in understanding stack functionality.

- Procedures without prototypes can be called successfully, provided the argument types correspond properly. Since pointers and `ints` are handled in the same way, some common interface errors are tolerated. Mixing up float and integer arguments generally results in incorrect behavior.

C/C++ Run-Time Model

- Procedures that use the `stdargs` (`varargs`) mechanism need to have prototypes.
- There is no special interface for calling system library functions. They are called using the standard calling convention.
- The `k` stack has a slightly irregular existence at present. There must always be a frame, but it cannot be used for local variables. These restrictions are both based on limitations in the debug information.

Register Classification

This section provides a list of ADSP-TS001 and ADSP-TS101 registers. `RES` denotes a reserved register.

Registers are listed in order of preferred allocation by the compiler

Callee Preserved Registers (“Preserved”)

Registers `j16` through `j25`, `k16` through `k25`, `x24` through `x31`, `y24` through `y31` are “preserved”. A subroutine that uses any of these registers must save (preserve) it and restore it.

Caller Save Registers (“Scratch”)

All registers not preserved or dedicated are “scratch” registers. A subroutine may use a scratch register without having to save it.

ADSP-TS001 and ADSP-TS101 Registers

The following is a list of registers for the ADSP-TS001 and ADSP-TS101 DSPs. The registers are listed in order of preferred allocation by the compiler.

Table 2-9. List of iALU (j and k) General Registers

j31:	j30:	j29:	j28:	k31:	k30:	k29:	k28:
zero ¹	scratch	scratch	scratch	zero ¹	scratch	scratch	scratch

j27:	j26:	j25:	j24:	k27:	k26:	k2:	k2:
Dedicated jSP	Dedicated jFP	RES (or GOT)	RES	Dedicated kSP	Dedicated kFP	RES (or GOT)	RES kjFP Copy

j23:	j22:	j21:	j20:	k23:	k22:	k21:	k20:
RES	RES	RES	RES	RES	RES	RES	RES

j19:	j18:	j17:	j16:	k19:	k18:	k17:	k16:
RES	RES	RES	RES	RES	RES	RES	RES

j15:	j14:	j13:	j12:	k15:	k14:	k13:	k12:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

C/C++ Run-Time Model

Table 2-9. List of iALU (j and k) General Registers (Cont'd)

j11:	j10:	j9:	j8:	k11:	k10:	k9:	k8:
scratch	scratch	scratch	scratch RTN0 ²	scratch	scratch	scratch	scratch

j7:	j6:	j5:	j4:	k7:	k6:	k5:	k4:
scratch ARG4	scratch ARG3	scratch ARG2	scratch ARG1	scratch	scratch	scratch	scratch

j3:	j2:	j1:	j0:	k3:	k2:	k1:	k0:
scratch (circ)	scratch (circ)	scratch (circ)	scratch (circ)	scratch (circ)	scratch (circ)	scratch (circ)	scratch (circ)

¹ j/k 31 are zero registers when used for addressing, or as operands to iALU operations; but when used in a store or load (or ureg transfer), they become the j/k status register.

² j8 is also used for passing the address of a hidden argument.

Table 2-10. Compute Block (x & y) General Registers

x31:	x30:	x29:	x28:	y31:	y30:	y29:	y28:
RES	RES	RES	RES	RES	RES	RES	RES

x27:	x26:	x25:	x24:	y27:	y26:	y25:	y24:
RES	RES	RES	RES	RES	RES	RES	RES

x23:	x22:	x21:	x20:	y23:	y22:	y21:	y20:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

Table 2-10. Compute Block (x & y) General Registers

x19:	x18:	x17:	x16:	y19:	y18:	y17:	y16:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

x15:	x14:	x13:	x12:	y15:	y14:	y13:	y12:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

x11:	x10:	x9:	x8:	y11:	y10:	y9:	y8:
scratch	scratch	scratch RTN 1	scratch RTN 0	scratch	scratch	scratch	scratch

x7:	x6:	x5:	x4:	y7:	y6:	y5:	y4:
scratch ARG4	scratch ARG3	scratch ARG2	scratch ARG1	scratch ARGn	scratch ARGn	scratch ARGn	scratch ARGn

x3:	x2:	x1:	x0:	y3:	y2:	y1:	y0:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

Table 2-11. iALU (j & k) Special Registers: Circular Buffering –B (base) and L (length)

jB3:	jB2:	jB1:	jB0:	kB3:	kB2:	kB1:	kB0:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

C/C++ Run-Time Model

Table 2-11. iALU (j & k) Special Registers: Circular Buffering –B (base) and L (length) (Cont'd)

jL3:	jL2:	jL1:	jL0:	kL3:	kL2:	kL1:	kL0:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

Table 2-12. Compute Block X MAC Registers

xMR4:		xMR3:	xMR2:	xMR1:	xMR0:
scratch		scratch	scratch	scratch	scratch

Table 2-13. Compute Block Y MAC Registers

yMR4:		yMR3:	yMR2:	yMR1:	yMR0:
scratch		scratch	scratch	scratch	scratch

Table 2-14. Compute Block ALU Summation Registers

xPR1:	xPR0:		yPR1:	yPR0:
scratch	scratch		scratch	scratch

Table 2-15. Loop Counters

LC1:		LC0:
scratch		scratch

Table 2-16. Data Alignment Registers

xDAB:		yDAB:
scratch		scratch

Table 2-17. Bit FIFO Temporaries

xBFOTMP:		yBFOTMP:
scratch		scratch

Table 2-18. Static Condition Flags

iSF0:	iSF1:		xF0:	xF1:		ySF0:	ySF1:
scratch	scratch		scratch	scratch		scratch	scratch

Table 2-19. Return Address Registers

CJMP:		RETI:
scratch – for caller reserved – by callee		Not Available

Table 2-20. Enhanced Communications Registers (ADSP-TS101 only)¹

xt15:	xt14:	xt13:	xt12:	yt15:	yt14:	yt13:	yt12:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

xt11:	xt10:	xt9:	xt8:	yt11:	yt10:	yt9:	yt8:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

xt7:	xt6:	xt5:	xt4:	yt7:	yt6:	yt5:	yt4:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

C/C++ and Assembly Interface

Table 2-20. Enhanced Communications Registers (ADSP-TS101 only)¹

xt3:	xt2:	xt1:	xt0:	yt3:	yt2:	yt1:	yt0:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

xth1:	xth0:	yth1:	yth0:
scratch	scratch	scratch	scratch

¹ The registers are listed in order of preferred allocation by the compiler.

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 perform either or 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) contained in the section, “[C/C++ Run-Time Model](#)” on page 2-97.

Writing C/C++-Callable Assembly Subroutines

In general, you should perform the following steps when writing C/C++-callable assembly subroutines:

1. Familiarize yourself with the general features of the C/C++ run-time model. This should include the general notion of a stack, how arguments are handled, and also the various data types and their sizes.

2. Create an interface definition or “prototype” so that the C or C++ program knows the name of your function and the types of its arguments. The prototype also determines how the arguments are passed.



In C mode, the compiler allows you to use a function without a prototype. In this case, the compiler assumes that all the arguments, as they appear in the call, are of the proper type even though this may not be desired. The compiler also assumes that the return type is integer.

The compiler normally prefaces the name of external entry points with an underscore. You can simply declare the function with an underscore as the compiler does. If you're using the function from assembly programs as well, you might want your function's name to be just as you write it. Then you'll also need to tell the C/C++ compiler that it's an `asm` function, by placing `'extern "asm" {}'` around the prototype.

3. A good way to determine how arguments are passed is to write a dummy function in C/C++ and compile it with the `stop after compile` option set (`-S` command-line option). [Listing 2-4](#) provides an example.



The assignments to the global variables have been put in to help show where the arguments can be found upon entry to `asmfunc`.

Listing 2-4. Sample File for Exploring Compiler Interface

```
//global variables ... assign arguments to there just so
// you can track which registers were used
// the 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, int d, int e) {
    // do some assignments so .s file shows 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 `-S` and `-O` switches, this example produces the following code:

 TigerSHARC passes up to four arguments in registers. (The optimizer has gratuitously reversed the order of the assignments.)

```
// PROCEDURE: asmfunc
.global _asmfunc;
_asmfunc:
    J8 = j31 + 12345;;
    [j31 + _global_p] = J6;;
    [j31 + _global_b] = XR5;;
    cjmp (NP) (ABS); [j31 + _global_a] = J4;;
```

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

4. For a simple function, this may be all the information you need.
Do the computation, set up a return value, and return to the caller.

Some General Cautions:

- The run-time model defines certain registers as “scratch” and others as “preserved” or “dedicated”. The scratch registers are available for immediate use without concern. You may also use the preserved registers if you need more room (or if you're working with existing code), but you **MUST** save them first and then restore them before returning.
- Dedicated registers should be used only for their intended purposes. The run-time model conventions apply not only to the compiler, but also the libraries, interrupt routines, and the debugger. The interrupt routines in particular rely on having a stack available.
- The compiler assumes that the overall machine state does not change during execution. This is processor specific. If the machine has modes, such as integer/fractional, they must not be changed. If the machine performs circular buffering when certain registers are non-zero, those registers must not be altered (unless suitable reservations have been made on the related registers).
- For a more complicated function, you might find it useful to follow the general run-time model and use the runtime stack for local storage, etc. Again, a simple C/C++ program, passed through the compiler, provides a good template to build on. Alternatively, you may find it just as convenient to use local static storage for temporaries.

Calling C/C++ Routines from the Assembler

If you call other functions, maintaining the basic stack model facilitates the use of the debugger.

Calling “out” from assembly language to C/C++ is a bit more complex, but it can be managed successfully. This kind of call is particularly useful for calling C/C++-callable functions, most notably in the library. Please keep the following in mind when making these calls:

- It is essential that you know the run-time model well.
- A sample program which calls the function in question provides an excellent guide to argument setup. (Again, the trick of using volatile global variables helps clarify the essential code).
- To call a C/C++-callable function, the stack must be set up correctly. The easiest way to do this is to have a C/C++ main program, which initializes the runtime system. Then hold that stack until it's needed to call back into C or C++.

Here, again, an example program may provide the necessary template. Even a simple function has the code to set up and tear down a stack frame.

Keep in mind the division of registers between scratch and preserved. You must assume that every scratch register is modified when a C/C++ function is called. If there is something in a scratch register that you need after the call, you must save that register prior to making the call.

C/C++ Compiler Glossary

Assembler (easmts) — Assembles your TigerSHARC Family assembly source code or code output from the C/C++ compiler (ccts) into linkable object files.

C/C++ Run-time model (for ccts C/C++ compiler) — The C/C++ run-time model is a set of rules that the ccts C/C++ Compiler uses to operate on the TigerSHARC Family processors. The environment defines register use (compiler, user, scratch, and stack registers), run-time stack operation, and C/C++/assembly program interfacing requirements.

ccts, TigerSHARC family C/C++ compiler — ccts is an ANSI/ISO compliant C/C++ compiler for the TigerSHARC Family digital signal processors.

Leaf routines — Leaf routines are routines that return without making any calls.

Linker — The VisualDSP++ linker links your object files and libraries into executable DSP programs.

Macros and the 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.

Non-leaf routines — Non-leaf routines call other routines before returning to the caller.

