VISUALDSP++™ 3.5 C Compiler and Library Manual for ADSP-218x DSPs

Revision 4.1, October 2003

Part Number:
82-000400-03

ANALOG DEVICES

Analog Devices, Inc.
One Technology Way
Norwood, Mass. 02062-9106

CONTENTS

PREFACE

Thank you for purchasing Analog Devices development software for digital signal processors (DSPs).

Purpose

The *VisualDSP++ 3.5 C Compiler and Library Manual for ADSP-218x DSPs* contains information about the C compiler and run-time library for ADSP-218x DSPs. It includes syntax for command lines, switches, and language extensions. It leads you through the process of using library routines and writing mixed C/assembly code.

Intended Audience

The primary audience for this manual is programmers who are familiar with Analog Devices DSPs. This manual assumes that the audience has a working knowledge of the ADSP-218x DSP architecture and instruction set and the C programming language.

Programmers who are unfamiliar with ADSP-218x DSPs can use this manual, but they should supplement it with other texts (such as the appropriate hardware reference and instruction set reference) that provide information about your ADSP-218x DSP architecture and instructions).

Manual Contents Description

This manual contains:

- Chapter 1, "Compiler"

  Provides information on compiler options, language extensions and C/assembly interfacing

- Chapter 2, "Achieving Optimal Performance from C Source Code" Shows how to optimize compiler operation.

- Chapter 3, "C Run-Time Library"

  Shows how to use library functions and provides a complete C library function reference

- Appendix A, "Compiler Legacy Support"

  Describes support for legacy code that was developed with previous releases of the development tools.

What's New in this Manual

This edition of the *VisualDSP++ 3.5 C Compiler and Library Manual for ADSP-218x DSPs* documents support for all ADSP-218x processors.

Refer to *VisualDSP++ 3.5 Product Bulletin for 16-Bit Processors* for information on all new and updated features and other release information.

Technical or Customer Support

You can reach DSP Tools Support in the following ways:

- Visit the DSP Development Tools website at
  `www.analog.com/technology/dsp/developmentTools/index.html`
- Email questions to
  `dsptools.support@analog.com`
- Phone questions to **1-800-ANALOGD**
- Contact your ADI local sales office or authorized distributor
- Send questions by mail to:

```
Analog Devices, Inc.
One Technology Way
P.O. Box 9106
Norwood, MA 02062-9106|
USA
```

Supported Processors

The name "ADSP-218x" refers to a family of Analog Devices 16-bit, fixed-point processors. VisualDSP++ currently supports the following ADSP-218x processors:

ADSP-2181, ADSP-2183, ADSP-2184, ADSP-2185, ADSP-2186, ADSP-2187, ADSP-2188, and ADSP-2189M

Product Information

You can obtain product information from the Analog Devices website, from the product CD-ROM, or from the printed publications (manuals).

Analog Devices is online at `www.analog.com`. Our website provides information about a broad range of products—analog integrated circuits, amplifiers, converters, and digital signal processors.

MyAnalog.com

MyAnalog.com is a free feature of the Analog Devices website that allows customization of a webpage to display only the latest information on products you are interested in. You can also choose to receive weekly email notification containing updates to the webpages that meet your interests. `MyAnalog.com` provides access to books, application notes, data sheets, code examples, and more.

Registration:

Visit `www.myanalog.com` to sign up. Click **Register** to use `MyAnalog.com`. Registration takes about five minutes and allows you to select the information you want to receive.

If you are already a registered user, just log on. Your user name is your email address.

DSP Product Information

For information on digital signal processors, visit our website at `www.analog.com`, which provides access to technical publications, datasheets, application notes, product overviews, and product announcements.

You may also obtain additional information about Analog Devices and its products in any of the following ways.

- Email questions or requests for information to
  `dsp.support@analog.com`
- Fax questions or requests for information to
  1-781-461-3010 (North America)
  089/76 903-557 (Europe)
- Access the Digital Signal Processing Division's FTP website at
  `ftp ftp.analog.com` or `ftp 137.71.23.21`
  `ftp://ftp.analog.com`

Related Documents

For information on product related development software, see the following publications:

VisualDSP++ 3.5 Getting Started Guide for 16-Bit Processors

VisualDSP++ 3.5 User's Guide for 16-Bit Processors

VisualDSP++ 3.5 Assembler and Preprocessor Manual for ADSP-218x and ADSP-219x DSPs

VisualDSP++ 3.5 Linker and Utilities Manual for 16-Bit Processors

VisualDSP++ 3.5 Loader Manual for 16-Bit Processors

VisualDSP++ 3.5 Product Bulletin 16-Bit Processors

VisualDSP++ Kernel (VDK) User's Guide

VisualDSP++ Component Software Engineering User's Guide

Quick Installation Reference Card

For hardware information, refer to the processor's *Hardware Reference Manual* and *Instruction Set Reference Manual*.

Online Technical Documentation

Online documentation comprises VisualDSP++ Help system and tools manuals, and FlexLM network license manager software documentation. You can easily search across the entire VisualDSP++ documentation set for any topic of interest. For easy printing, supplementary PDF files for the tools manuals are also provided.

If documentation is not installed on your system as part of the software installation, you can add it from the VisualDSP++ CD-ROM at any time.

Access the online documentation from the VisualDSP++ environment, Windows Explorer, or Analog Devices website.

A description of each documentation file type is as follows.

File	Description
`.CHM`	Help system files and VisualDSP++ tools manuals.
`.HTM` or `.HTML`	FlexLM network license manager software documentation. Viewing and printing the `.HTML` files require a browser, such as Internet Explorer 4.0 (or higher).
`.PDF`	VisualDSP++ tools manuals in Portable Documentation Format, one .PDF file for each manual. Viewing and printing the `.PDF` files require a PDF reader, such as Adobe Acrobat Reader 4.0 (or higher).

From VisualDSP++

Access VisualDSP++ online Help from the Help menu's **Contents**, **Search**, and **Index** commands.

Open online Help from context-sensitive user interface items (toolbar buttons, menu commands, and windows).

From Windows

In addition to any shortcuts you may have constructed, there are many ways to open VisualDSP++ online Help or the supplementary documentation from Windows.

Help system files (`.CHM` files) are located in the Help folder, and `.PDF` files are located in the **Docs** folder of your VisualDSP++ installation. The **Docs** folder also contains the FlexLM network license manager software documentation.

Using Windows Explorer

- Double-click any file that is part of the VisualDSP++ documentation set.
- Double-click the `vdsp-help.chm` file, which is the master Help system, to access all the other `.CHM` files.

Using the Windows Start Button

- Access the VisualDSP++ online Help by clicking the **Start** button and choosing **Programs**, **Analog Devices**, **VisualDSP++**, and **VisualDSP++ Documentation**.
- Access the `.PDF` files by clicking the **Start** button and choosing **Programs**, **Analog Devices**, **VisualDSP++**, **Documentation for Printing**, and the name of the book.

From the Web

To download the tools manuals, point your browser at
`http://www.analog.com/technology/dsp/developmentTools/gen_purpose.html`

Select a DSP family and book title. Download archive (`.ZIP`) files, one for each manual. Use any archive management software, such as WinZip, to decompress downloaded files.

Printed Manuals

For general questions regarding literature ordering, call the Literature Center at **1-800-ANALOGD** (**1-800-262-5643**) and follow the prompts.

VisualDSP++ Documentation Set

VisualDSP++ manuals may be purchased through Analog Devices Customer Service at **1-781-329-4700**; ask for a Customer Service representative. The manuals can be purchased only as a kit. For additional information, call **1-603-883-2430**.

If you do not have an account with Analog Devices, you will be referred to Analog Devices distributors. To obtain information on our distributors, log onto `http://www.analog.com/salesdir/continent.asp`.

Hardware Manuals

Hardware reference and instruction set reference manuals can be ordered through the Literature Center or downloaded from the Analog Devices website. The phone number is **1-800-ANALOGD** (**1-800-262-5643**). The manuals can be ordered by a title or by product number located on the back cover of each manual.

Datasheets

All data sheets can be downloaded from the Analog Devices website. As a general rule, any data sheet with a letter suffix (L, M, N) can be obtained from the Literature Center at **1-800-ANALOGD** (**1-800-262-5643**) or downloaded from the website. Data sheets without the suffix can be downloaded from the website only—no hard copies are available. You can ask for the data sheet by a part name or by product number.

If you want to have a data sheet faxed to you, call **1-800-446-6212**. Follow the prompts and a list of data sheet code numbers will be faxed to you. Call the Literature Center first to find out if requested datasheets are available.

Contacting DSP Publications

Please send your comments and recommendation on how to improve our manuals and online Help. You can contact us @ `dsp.techpubs@analog.com`.

Notation Conventions

The following table identifies and describes text conventions used in this manual. Additional conventions, which apply only to specific chapters, may appear throughout this document.

Example	**Description**
Close command (**File** menu)	Text in **bold** style indicates the location of an item within the VisualDSP++ environment's menu system. For example, the **Close** command appears on the **File** menu.
`{this | that}`	Alternative required items in syntax descriptions appear within curly brackets and separated by vertical bars; read the example as `this` or `that`.
`[this | that]`	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional `this` or `that`.
`[this,…]`	Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipsis; read the example as an optional comma-separated list of `this`.
`.SECTION`	Commands, directives, keywords, and feature names are in text with letter gothic font.
`filename`	Non-keyword placeholders appear in text with italic style format.

Example	Description
ⓘ	A note, providing information of special interest or identifying a related topic. In the online version of this book, the word **Note** appears instead of this symbol.
🛇	A caution, providing information about critical design or programming issues that influence operation of a product. In the online version of this book, the word **Caution** appears instead of this symbol.

1 COMPILER

The C compiler (`cc218x`) is part of Analog Devices development software for ADSP-218x DSPs.

This chapter contains:

- "C Compiler Overview" on page 1-2
  Provides an overview of C compiler for ADSP-218x DSPs.
- "Compiler Command-Line Reference" on page 1-6
  Describes the operation of the compiler as it processes programs, including input and output files, and command-line switches.
- "C Compiler Language Extensions" on page 1-52
  Describes the `cc218x` compiler's extensions to the ISO/ANSI standard for the C language.
- "Preprocessor Features" on page 1-125
  Contains information on the preprocessor and ways to modify source compilation.
- "C Run-Time Model and Environment" on page 1-132
  Contains reference information about implementation of C programs, data, and function calls in ADSP-218x DSPs.
- "C and Assembly Language Interface" on page 1-148
  Describes how to call an assembly language subroutine from C program, and how to call a C function from within an assembly language program.

C Compiler Overview

The C compiler (`cc218x`) is designed to aid your DSP project development efforts by:

- Processing C source files, producing machine level versions of the source code and object files
- Providing relocatable code and debugging information within the object files
- Providing relocatable data and program memory segments for placement by the linker in the processors' memory

Using C, developers can significantly decrease time-to-market since it gives them the ability to efficiently work with complex signal processing data types. It also allows them to take advantage of specialized DSP operations without having to understand the underlying DSP architecture.

The C compiler (`cc218x`) compiles ISO/ANSI standard C code for the ADSP-218x DSPs. Additionally, Analog Devices includes within the compiler a number of C language extensions designed to assist in DSP development. The `cc218x` compiler runs from the VisualDSP++ environment or from an operating system command line.

The C compiler (`cc218x`) processes your C language source files and produces ADSP-218x DSP's assembler source files. The assembler source files are assembled by the ADSP-218x assembler (`easm218x`). The assembler creates Executable and Linkable Format (ELF) object files that can either be linked (using the linker) to create an ADSP-218x executable file or included in an archive library (using `elfar`). The way in which the compiler controls the assemble, link, and archive phases of the process depends on the source input files and the compiler options used.

Source files contain C programs to be processed by the compiler. The compiler takes these source files for preprocessing first. Preprocessed source files are assembled by the ADSP-218x DSP assembler (`easm218x`).

The assembler creates Executable and Linkable Format object files that can either be linked (using the linker) to create an ADSP-218x executable file

The `cc218x` compiler supports the ANSI/ISO standard definitions of the C language. For information on these standards, see any of the many reference texts on the C language. In addition to ANSI standards, the compiler supports a set of C-language extensions. These extensions support hardware features of the ADSP-218x DSPs. For information on these extensions, see "C Compiler Language Extensions" on page 1-52.

Compiler options are set in the VisualDSP++ Integrated Development and Debug Environment (IDDE) from the Compile page of the Project Options dialog box (see "Specifying Compiler Options in VisualDSP++" on page 1-10). The selections control how the compiler processes your source files, letting you select features that include the language dialect, error reporting, and debugger output.

For more information on the VisualDSP++ environment, see the *VisualDSP++ 3.5 User's Guide for ADSP-21xx DSPs* and online Help.

Standard Conformance

Analog C compilers conform to the ISO/IEC 998:1990 C standard with a small number of currently unsupported features or areas of divergence.

Unsupported features are:

- ANSI features that require operating-system support are generally not supported. This includes `time.h` functionality in C.
- The `cc218x` compiler does not provide comprehensive support of NaN's, overflow and underflow conditions in their compiler support floating-point routines.

Areas of divergence from Standard:

- The `double` type is defined in terms of a single precision 32-bit `floats`, not double precision 64-bit `floats`.
- The `cc218x` compiler makes use of the DSP's double word (long) MAC instruction results to avoid having to explicitly promote integer operand multiplication to long. If the integer multiplication result overflows the `integer` type, then the result is not truncated as would be the case in strict ANSI terms. This behavior is disabled using the "-no-widen-muls" switch (on page 1-32).
- Normal ANSI C external linkage does not specifically require standard include files to be used, although it is recommended. In many cases, Analog C compilers do require standard include files to be used as build configurations. Instead, optimizations are used to select the correct and optimal implementation of C library functions. For example, the include files may redefine standard C functions to use optimal compiler built-in implementations.

The compilers also support a number of language extensions that are essentially aids to DSP programmers and would not be defined in strict ANSI conforming implementations. These extensions are usually enabled by default and in some cases can be disabled using a command-line switch, if required.

These extensions include:

- Inline (function) which directs the compiler to integrate the function code into the code of the callers. Disabled if the `-O` switches (described on page 1-32) are not used.
- Dual memory support keywords (`pm`/`dm`). Disabled using the `-no-extra-keywords` compiler switch (on page 1-30).
- Placement support keyword (`section`). Disabled using the `-no-extra-keywords` compiler switch (on page 1-30).

- Boolean type support keywords in C (`bool`, `true`, `false`). Disabled using the `-no-extra-keywords` compiler switch (see on page 1-30).
- Variable length array support
- Non-constant aggregate initializer support
- Indexed initializer support
- Preprocessor generated warnings
- Support for C++-style comments in C programs

For more information on these extensions, see the "C Compiler Language Extensions" on page 1-52.

Compiler Command-Line Reference

This section describes how the `cc218x` compiler is invoked from the command line, the various types of files used by and generated from the compiler, and the switches used to tailor the compiler's operation.

This section contains:

- "Running the Compiler" on page 1-7
- "Specifying Compiler Options in VisualDSP++" on page 1-10
- "Compiler Command-Line Switches" on page 1-11
- "Data Type Sizes" on page 1-47
- "Optimization Control" on page 1-49

By default, the `cc218x` compiler runs with Analog Extensions for C code enabled. This means that the compiler processes source files written in ANSI/ISO standard C language supplemented with Analog Devices extensions. Table 1-1 on page 1-9 lists valid extensions. By default, the compiler processes the input file through the listed stages to produce a `.DXE` file.

When developing a DSP project, you may find it useful to modify the compiler's default options settings. The way you set the compiler's options depends on the environment used to run the DSP development software.

See "Specifying Compiler Options in VisualDSP++" on page 1-10 for more information.

Running the Compiler

Use the following general syntax for the `cc218x` command line:

```
cc218x [-switch [-switch …]] sourcefile [sourcefile …]
```

where:

- `-switch` is the name of the switch to be processed. The compiler has many switches. These switches select the operations and modes for the compiler and other tools. Command-line switches are case sensitive. For example, `-O` is not the same as `-o`.
- The `sourcefile` is the name of the file to be preprocessed, compiled, assembled, and/or linked.
- A file name can include the drive, directory, file name and file extension. The compiler supports both Win32- and POSIX-style paths, using either forward or back slashes as the directory delimiter. It also supports UNC path names (starting with two slashes and a network name).
- The `cc218x` compiler uses the file extension to determine what the file contains and what operations to perform upon it. Table 1-1 on page 1-9 lists the allowed extensions.

For example, the following command line

```
cc218x -proc ADSP-2189 -O -Wremarks -o program.dxe source.c
```

runs `cc218x` with the following switches:

`-proc ADSP-2189`	Specifies the processor
`-O`	Specifies optimization for the compiler
`-Wremarks`	Selects extra diagnostic remarks in addition to warning and error messages
`-o program.dxe`	Selects a name for the compiled, linked output
`source.c`	Specifies the C language source file to be compiled

The normal function of the `cc218x` switch is to invoke the compiler, assembler, and linker as required to produce an executable file. The precise operation is determined by the extensions of the input filenames, and/or by various switches.

The compiler uses the following files to perform the specified action:

Extension	Action
`.c .C`	C source file is compiled, assembled, and linked
`.asm`, `.dsp`, or `.s`	Assembly language source file is assembled and linked
`.doj`	Object file (from previous assembly) is linked

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

You can stop this sequence at various points by using appropriate compiler switches, or by selecting options within the VisualDSP++ IDDE. These switches are: `-E`, `-P`, `-M`, `-H`, `-S` and `-c`.

Many of the compiler's switches take a file name as an optional parameter. If you do not use the optional output name switch, `cc218x` names the output for you. Table 1-1 lists the type of files, names, and extensions the compiler appends to output files.

File extensions vary by command-line switch and file type. These extensions 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 1-1 indicates the searches that the preprocessor, compiler, assembler, and linker support. The compiler supports relative and absolute directory names to define file search paths. 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 1-1. Input and Output Files

Input File Extension	Input File Description
`.c`	C source file
`.h`	Header file (referenced by a `#include` statement)
`.i`	Preprocessed C source, created when preprocess only (`-E` compiler switch) is specified
`.idl`	Interface definition language files for VCSE.
`.ipa, .opa`	Interprocedural analysis files — used internally by the compiler when performing interprocedural analysis.
`.pch`	Precompiled header file.
`.s,.dsp,.asm`	Assembly language source file
`.is`	Preprocessed assembly language source (retained when `-save-temps` is specified)
`.doj`	Object file to be linked
`.dlb`	Library of object files to be linked as needed
`.xml`	DSP system memory map file output
`.sym`	DSP system symbol map file output

For information on additional search directories, see the `-I directory` switch (on page 1-25) and `-L directory` switch (on page 1-27).

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

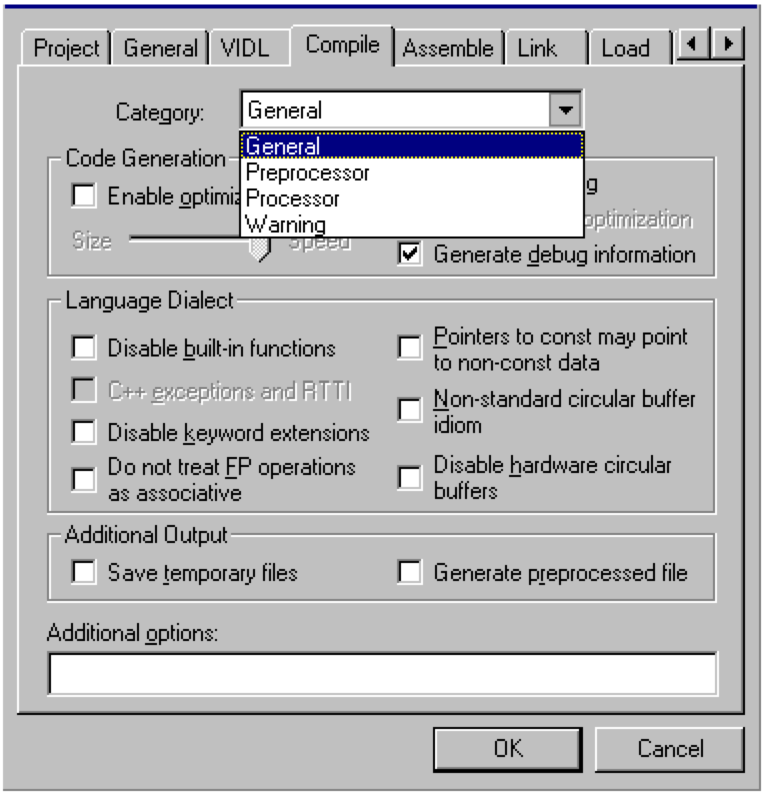
- Use a file name (include the file extension) with either an unambiguous relative path or an absolute path. A file name with an absolute path includes the drive, directory, file name, and file extension.

  Enclose long file names within straight quotes; for example, "*`long file name.c`*". The `cc218x` compiler uses the file extension convention listed in Table 1-1 to determine the input file type.

- Verify that the compiler is using the correct file. If you do not provide the complete path as part of the parameter or add additional search directories, `cc218x` looks for input in the current directory.

Specifying Compiler Options in VisualDSP++

When using the VisualDSP++ IDDE, use the **Compile** tab from the **Project Options** dialog box to set compiler functional options described on Figure 1-1.



Category: General

Code Generation
- [ ] Enable optimiz
- optimization
- [✓] Generate debug information

Language Dialect
- [ ] Disable built-in functions
- [ ] C++ exceptions and RTTI
- [ ] Disable keyword extensions
- [ ] Do not treat FP operations as associative
- [ ] Pointers to const may point to non-const data
- [ ] Non-standard circular buffer idiom
- [ ] Disable hardware circular buffers

Additional Output
- [ ] Save temporary files
- [ ] Generate preprocessed file

Figure 1-1. Project Options -- Compile Property Page

There are four sub-pages you can access—**General**, **Preprocessor**, **Processor**, and **Warning.** Most page options have a corresponding compiler command-line switch described in "Compiler Command-Line Switches". The **Additional options** field in each sub-page is used to enter the appropriate file names and options that do not have corresponding controls on the **Compile** tab but are available as compiler switches.

Use the VisualDSP++ online Help to get more information on compiler options you can specify from the VisualDSP++ environment.

Compiler Command-Line Switches

Table 1-2 lists the command-line switches accepted by the `cc218x` compiler. A detailed description of each of these switches follows the table.

Table 1-2. Compiler Command-Line Switches

Switch Name	Description
`-@ filename` (on page 1-18)	Reads command-line input from the file.
`-A name(tokens)` (on page 1-18)	Asserts the specified name as a predicate.
`-alttok` (on page 1-19)	Allows alternative keywords and sequences in sources.
`-bss` (on page 1-20)	Causes the compiler to put global zero-initialized data into a separate BSS-style section.
`-build-lib` (on page 1-20)	Directs the librarian to build a library file.
`-C` (on page 1-20)	Retains preprocessor comments in the output file; must run with the `-E` or `-P` switch.
`-c` (on page 1-20	Compiles and/or assembles only; does not link.
`-const-read-write` (on page 1-20	Specifies that data accessed via a pointer to const data may be modified elsewhere.

Table 1-2. Compiler Command-Line Switches (Cont'd)

Switch Name	Description
`-Dmacro[=definition]` (on page 1-20)	Defines a macro.
`-debug-types` (on page 1-21)	Supports building a `*.h` file directly and writing a complete set of debugging information for the header file.
`-default-linkage-{asm|C}` (on page 1-21)	Sets the default linkage type (`asm`, `C`).
`-dry` (on page 1-22)	Displays, but does not perform, main driver actions (verbose dry-run).
`-dryrun` (on page 1-22)	Displays, but does not perform, top-level driver actions (terse dry-run).
`-E` (on page 1-22)	Preprocesses, but does not compile, the source file.
`-ED` (on page 1-22)	Produce preprocessed file and compile source.
`-EE` (on page 1-22)	Preprocesses and compiles the source file.
`-extra-keywords` (on page 1-22)	Recognizes the Analog Devices extensions to ANSI/ISO standards for C. Default mode.
`-flags-{asm | compiler | | lib | link | mem} [, argument [,...]]` (on page 1-23)	Passes command-line switches through the compiler to other build tools.
`-fp-associative` (on page 1-23)	Treats floating-point multiplication and addition as associative.
`-full-version` (on page 1-23)	Displays version information for build tools.
`-g` (on page 1-23)	Generates DWARF-2 debug information.
`-H` (on page 1-24)	Outputs a list of header files, but does not compile.

Table 1-2. Compiler Command-Line Switches (Cont'd)

Switch Name	Description
`-HH` (on page 1-24)	Outputs a list of included header files and compiles.
`-h[elp]` (on page 1-24)	Outputs a list of command-line switches.
`-I-` (on page 1-25)	Establishes the point in the `include` directory list at which the search for header files enclosed in angle brackets should begin.
`-I directory` (on page 1-25)	Appends the specified search directory to the standard search path.
`-i` (on page 1-25)	Only output header details or makefile dependencies for include files specified in double quotes.
`-include filename` (on page 1-25)	Includes named file prior to processing each source file.
`-ipa` (on page 1-26)	Specifies that interprocedural analysis should be performed for optimization between translation units.
`-jump-{dm|pm|same}` (on page 1-26)	Specifies that the compiler should place jump tables in data memory, program memory, or the same memory section as the function to which it applies.
`-L directory` (on page 1-27)	Appends the specified directory to the standard library search path when linking.
`-l library` (on page 1-27)	Searches the specified library for functions when linking.
`-M` (on page 1-27)	Generates make rules only; does not compile.
`-MD` (on page 1-28)	Generates make rules and compiles.
`-MM` (on page 1-28)	Generates make rules and compiles.
`-Mo` *`filename`* (on page 1-28)	Specifies a file for output when using the `-ED` and `-MD` switches.

Table 1-2. Compiler Command-Line Switches (Cont'd)

Switch Name	Description
`-Mt filename` (on page 1-28)	Makes dependencies for the specified source file.
`-MQ` (on page 1-28)	Generates make rules only; does not compile.No notification when input files are missing.
`-make-autostatic` (on page 1-29)	Directs the compiler to place all automatic variables in static store.
`-map filename` (on page 1-29)	Directs the linker to generate a memory map of all symbols.
`-mem` (on page 1-29)	Causes the compiler to invoke the Memory Initializer after linking the executable.
`-no-alttok` (on page 1-29)	Does not allow alternative keywords and sequences in sources.
`-no-bss` (on page 1-30)	Causes the compiler to group global zero-initialized data into the same section as global data with non-zero initializers.
`-no-builtin` (on page 1-30)	Disables recognition of `__builtin` functions.
`-no-defs` (on page 1-30)	Disables preprocessor definitions: macros, include directories, library directories, run-time headers, or keyword extensions.
`-no-extra-keywords` (on page 1-30)	Does not define language extension keywords that could be valid C identifiers.
`-no-fp-associative` (on page 1-31)	Does not treat floating-point multiply and addition as an associative.
`-no-mem` (on page 1-31)	Causes the compiler to not invoke the Memory Initializer after linking. Set by default.
`-no-std-ass` (on page 1-31)	Disables any predefined assertions and system-specific macro definitions.
`-no-std-def` (on page 1-31)	Disables normal macro definitions; also disables Analog Devices keyword extensions that do not have leading underscores (__).

Table 1-2. Compiler Command-Line Switches (Cont'd)

Switch Name	Description
`-no-std-inc` (on page 1-31)	Searches for preprocessor include header files only in the current directory and in directories specified with the `-I` switch.
`-no-std-lib` (on page 1-32)	Searches only for those linker libraries specified with the `-l` switch when linking.
`-no-widen-muls` (on page 1-32)	Disable widening multiplications optimization.
`-O` (on page 1-32)	Enables optimizations.
`-Oa` (on page 1-33)	Enables automatic function inlining.
`-Os` (on page 1-33)	Optimize for code size.
`-Ov num` (on page 1-33)	Controls speed vs. size optimizations.
`-o filename` (on page 1-33)	Specifies the output file name.
`-P` (on page 1-34)	Preprocesses, but does not compile, the source file. Omits line numbers in the preprocessor output.
`-PP` (on page 1-34)	Similar to -P, but does not halt compilation after preprocessing.
`-path-{asm|compiler|def|lib|link|mem}` *`filename`* (on page 1-34)	Use the specified component in place of the default installed version of a compilation tool.
`-path-install` *`directory`* (on page 1-34)	Uses the specified directory as the location of all compilation tools.
`-path-output` *`directory`* (on page 1-35)	Specifies the location of non-temporary files.
`-path-temp` *`directory`* (on page 1-35)	Specifies the location of temporary files.
`-pch` (on page 1-35)	Generates and uses precompiled header files (`*.pch`)

Table 1-2. Compiler Command-Line Switches (Cont'd)

Switch Name	Description
`-pchdir directory` (on page 1-35)	Specifies the location of PCHRepository.
`-pedantic` (on page 1-35)	Issues compiler warnings for any constructs that are not strictly ISO/ANSI standard C-compliant.
`-pedantic-errors` (on page 1-36)	Issues compiler errors for any constructs that are not strictly ISO/ANSI standard C-compliant.
`-pplist filename` (on page 1-36)	Outputs a raw preprocessed listing to the specified file.
`-proc identifier` (on page 1-36)	Specifies that the compiler should produce code suitable for the specified DSP.
`-R directory` (on page 1-37)	Appends directory to the standard search path for source files.
`-R-` (on page 1-38)	Removes all directories from the standard search path for source files.
`-reserve <reg1>[,reg2..]` (on page 1-38)	Reserves specified registers for autobuffering.
`-S` (on page 1-38)	Stops compilation before running the assembler.
`-s` (on page 1-39)	Removes debugging information from the output executable file when linking.
`-save-temps` (on page 1-39)	Saves intermediate files.
`-show` (on page 1-39)	Displays the driver command-line information.
`-si-revision version` (on page 1-39)	Specifies a silicon revision of the specified processor.
`-signed-bitfield` (on page 1-40)	Treats bitfields which have not been declared with use of signed or unsigned keyword as signed. This is the default behavior.
`-signed-char` (on page 1-40)	Makes the `char` data type signed.

Table 1-2. Compiler Command-Line Switches (Cont'd)

Switch Name	Description
`-syntax-only` (on page 1-40)	Checks the source code for compiler syntax errors, but does not write any output.
`-sysdefs` (on page 1-41)	Defines the system definition macros.
`-T` *`filename`* (on page 1-41)	Specifies the Linker Description File.
`-time` (on page 1-41)	Displays the elapsed time as part of the output information on each part of the compilation process.
`-Umacro` (on page 1-42)	Undefines `macro`.
`-unsigned-bitfield` (on page 1-42)	Treats bitfields which have not been declared with use of signed or unsigned keyword as unsigned.
`-unsigned-char` (on page 1-43)	Makes the `char` data type unsigned.
`-v` (on page 1-43)	Displays both the version and command-line information.
`-val-global <name-list>` (on page 1-43)	Adds global names.
`-verbose` (on page 1-43)	Displays command-line information.
`-version` (on page 1-43)	Displays version information.
`-Wdriver-limit` *`number`* (on page 1-44)	Halts the driver after reaching the specified number of errors.
`-Werror-limit` *`number`* (on page 1-44)	Stops compiling after reaching the specified number of errors.
`-W{error|remark|suppress |warn}` <*num*>[,*num*...] (on page 1-44)	Overrides the severity of compilation diagnostic messages.
`-Wremarks` (on page 1-44)	Indicates that the compiler may issue remarks, which are diagnostic messages even milder than warnings.

Table 1-2. Compiler Command-Line Switches (Cont'd)

Switch Name	Description
`-Wterse` (on page 1-45)	Issues only the briefest form of compiler warnings, errors, and remarks.
`-w` (on page 1-45)	Does not display compiler warning messages.
`-warn-protos` (on page 1-45)	Produces a warning when a function is called without a prototype.
`-write-files` (on page 1-45)	Enables compiler I/O redirection.
`-write-opts` (on page 1-45)	Passes the user options (but not input filenames) via a temporary file.
`-xref` *`filename`* (on page 1-46)	Outputs cross-reference information to the specified file.

The following sections provide detailed comman-line switch descriptions.

-@ 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.

-A name(tokens)

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

system	`embedded`
machine	`adsp218x`

cpu	`adsp218x`
compiler	`cc218x`

The `-A` *`name(value)`* switch is equivalent to including

```
#assert name(value)
```

in your source file, and both may be tested in a preprocessor condition in the following manner.

```
#if #name(value)
   // do something
#else
   // do something else
#endif
```

For example, the default assertions may be tested as:

```
#if #machine(adsp218x)
   // do something
#endif
```

ⓘ The parentheses in the assertion should be quoted when using the `-A` switch, to prevent misinterpretation. No quotes are needed for a `#assert` directive in a source file.

-alttok

The `-alttok` (alternative tokens) switch directs the compiler to allow digraph sequences in C source files.

-bss

The `-bss` switch causes the compiler to place global zero-initialized data into a BSS-style section (called "`bsz`"), rather than into the normal global data section. This is default mode. See also the `-no-bss` switch (on page 1-30).

-build-lib

The `-build-lib` (build library) switch directs the compiler to use the `elfar` (librarian) to produce a library file (`.dlb`) as the output instead of using the linker to produce an executable file (`.dxe`). The `-o` option (on page 1-33) 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`, `-EE`, `-ED`, `-P` or `-PP` switches, directs the preprocessor to retain comments in its output.

-c

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

-const-read-write

The `-const-read-write` switch directs the compiler to specify that constants may be accessed as read-write data (as in ANSI C). The compiler's default behavior assumes that data referenced through `const` pointers will never change.

The `-const-read-write` switch changes the compiler's behavior to match the ANSI C assumption, which is that other `non-const` pointers may be used to change the data at some point.

-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`'. If definition is required to be a character string constant, it must be surrounded by escaped double quotes. Note that the compiler processes `-D` switches on the command line before any `-U` (undefine macro) switches.

This switch can be invoked with the **Definitions:** dialog field from the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Preprocessor** category.

-debug-types <file.h>

The `-debug-types` switch provides for building an `*.h` file directly and writing a complete set of debugging information for the header file. The `-g` option (on page 1-23) need not be specified with the `-debug-types` switch because it is implied. For example,

```
cc218x -debug-types anyHeader.h
```

The implicit `-g` option writes debugging information for only those `typedefs` that are referenced in the program. The `-debug-types` option provides complete debugging information for all `typedefs` and `structs`.

-default-linkage-{asm|C}

The `-default-linkage-asm` (assembler linkage) and `-default-linkage-C` (C linkage) switches direct the compiler to set the default linkage type. C is the default linkage type. Compatibility linkage (`OldAsmCall`) to previous generation calling conventions cannot be set as a default linkage.

This switch can be specified in the **Additional Options** box located in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **General** category.

-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 preprocessor runs (without compiling). The output (preprocessed source code) prints the standard output stream (`<stdout>`) unless the output file is specified with the `-o` switch. Note that the `-C` switch can be used in combination with the `-E` switch.

-ED

The `-ED` (run after preprocessing to file) switch directs the compiler to write the output of the C preprocessor to a file named `original_filename.i`. After preprocessing, compilation proceeds normally.

-EE

The `-EE` (run after preprocessing) switch directs the compiler to write the output of the C preprocessor to standard output. After preprocessing, compilation proceeds normally.

-extra-keywords

The `-extra-keywords` (enable short-form keywords) switch directs the compiler to recognize the Analog Devices keyword extensions to ISO/ANSI standard C language, including keywords such as `pm` and `dm` without leading underscores which could affect conforming to ISO/ANSI C programs. This is the default mode. The "-no-extra-keywords" switch (see on page 1-30) can be used to disallow support for the additional keywords. Table 1-5 on page 1-53 provides a list and a brief description of keyword extensions.

-flags-{asm|compiler|lib|link|mem} switch [,switch2 [,...]]

The `-flags` (command-line input) switch directs the compiler to pass each of the comma-separated arguments to the other build tools, such as:

Table 1-3. Build Tools' Options

Option	Tool
`-flags-asm`	`easm218x` Assembler
`-flags-compiler`	Compiler executable
`-flags-lib`	`elfar` Library Builder
`-flags-link`	Linker
`-flags-mem`	Memory Initializer

-fp-associative

The `-fp-associative` switch directs the compiler to treat floating-point multiplication and addition as associative.

-full-version

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

-g

The `-g` (generate debug information) switch directs the compiler to output symbols and other information used by the debugger.

When the `-g` switch is used in conjunction with the enable optimization (`-O`) switch, the compiler performs standard optimizations. The compiler also outputs symbols and other information to provide limited source level debugging through VisualDSP++. This combination of options provides line debugging and global variable debugging.

ⓘ You can invoke this switch by selecting the **Generate debug information** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **General** category.

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

-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. The `-o` switch (on page 1-33) may be used to specify the redirection of the list to a file.

-HH

The `-HH` (list headers and compile) switch directs the compiler to print to the standard output file 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-

The `-I-` (start include directory list) switch establishes the point in the `include` directory list at which the search for header files enclosed in angle brackets should begin. Normally, for header files enclosed in double quotes, the compiler searches in the directory containing the current input file; then the compiler reverts back to looking in the directories specified with the `-I` switch and then in the standard include directory.

ⓘ For header files in angle brackets the compiler performs the latter two searches only.

It is possible to replace the initial search (within the directory containing the current input file) by placing the `-I-` switch at the point on the command line where the search for all types of header file should begin. All `include` directories on the command line specified before the `-I-` switch will only be used in the search for header files that are enclosed in double quotes.

This switch removes the directory containing the current input file from the `include` directory list.

-I directory [{,|;} directory...]

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

-include filename

The `-include` *`filename`* (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 always processed before an `-include` file.

Include files, whose names are not absolute path names and that are enclosed in "..." when included, will be searched for in the following directories in this order:

1. The directory containing the current input file (the primary source file or the file containing the `#include`)
2. Any directories specified with the `-I` switch in the order they are listed on the command line
3. Any directories on the standard list:
   `<VisualDSP++ install dir>/.../include`

If a file is included using the <...> form, this file will only be searched for by using directories defined in items 2 and 3 above.

-ipa

The `-ipa` (interprocedural analysis) switch turns on Interprocedural Analysis (IPA) in the compiler. This option enables optimization across the entire program, including between source files that were compiled separately. The `-ipa` option should be applied to all C files in the program. For more information, see "Interprocedural Analysis" on page 1-50. Specifying `-ipa` also implies setting the `-O` switch (on page 1-32).

You can invoke this switch by selecting the **Interprocedural optimization** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **General** category.

-jump-{dm|pm|same}

The `-jump` (select jump table memory type) switch directs the compiler to place jump tables in data memory (`-jump-dm`) or program memory (`-jump-pm`), or the same memory section as the function to which it applies (`-jump-same`). Jump tables are storage that might be required to hold in memory target addresses for branch instruction used in complex `if-then-else` statements or switch statements. The default storage memory for jump tables is data memory.

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

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

-l library

The `-l` (link library) switch directs the linker to search the *`library`* for functions when linking. The library name is the portion of the file name between the `lib` prefix and `.dlb` extension. For example, the compiler command-line switch `-lc` directs the linker to search in the library named c for functions. This library resides in a file named `libc.dlb`.

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

-M

The `-M` (generate make rules only) switch directs the compiler not to compile the source file but to output a rule, which is 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 ...
```

-MD

The `-MD` (generate make rules and compile) switch directs the preprocessor to print to a file called `original_filename.d` a rule describing the dependencies of the main program file. After preprocessing, compilation proceeds normally. See also the `-Mo` switch.

-MM

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

-Mo filename

The `-Mo` *`filename`* (preprocessor output file) switch directs the compiler to use *`filename`* for the output of `-MD` or `-ED` switches.

-Mt filename

The `-Mt` filename (output make rule for the named source) switch specifies the name of the source file for which the compiler generates the make rule when you use the `-M` or `-MM` switch. If the named file is not in the current directory, you must provide the path name in double quotation marks (“”). The new file name will override the default base.doj. The `-Mt` option supports the `.IMPORT` extension.

-MQ

The `-MQ` switch directs the compiler not to compile the source file but to output a rule. In addition, the `-MQ` switch does not produce any notification when input files are missing.

-make-autostatic

The `-make-autostatic` (make automatic variables static) switch directs the compiler to place all automatic variables in static store. This may be beneficial in code that requires many accesses of automatic variables as an access to static store is done in one instruction, whereas an access to local automatic stack area may require three instructions.

Alternatively, this feature can be applied on a function-by-function basis using the `make_auto_static` pragma. See "Stack Usage Pragma" on page 1-108 for more information.

Do not use the `-make-autostatic` switch if the source being compiled contains any functions that are directly or indirectly recursive.

-map filename

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

-mem

The `-mem` (invoke memory initializer) switch causes the compiler to invoke the Memory Initializer tool after linking the executable. The MemInit tool can be controlled through the `-flags-mem` switch (on page 1-23).

-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. This is the default mode. For more information, see "-alttok" on page 1-19.

-no-bss

The `-no-bss` switch causes the compiler to keep zero-initialized and non-zero-initialized data in the same data section, rather than separating zero-initialized data into a different, BSS-style section. See also the `-bss` switch (on page 1-19).

-no-builtin

The `-no-builtin` (no builtin functions) switch directs the compiler to recognize only built-in functions that begin with two underscores (__). Note that this switch influences many functions. This switch also predefines the `__NO_BUILTIN` preprocessor macro.

For more information on built-in functions, see "Compiler Built-in Functions" on page 1-83 and "Using the Compiler's Built-In Functions" on page 3-2.

-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.

-no-extra-keywords

The `-no-extra-keywords` (disable short-form keywords) switch directs the compiler not to recognize the Analog Devices keyword extensions that might affect conformance to ISO/ANSI standards for the C language. These extensions include keywords, such as `asm`, which may be used as identifiers in conforming programs. Alternate keywords that are prefixed with two leading underscores, such as `__pm` and `__dm`, continue to work. The "-extra-keywords" switch (on page 1-22) can be used to explicitly request support for the additional keywords.

-no-fp-associative

The `-no-fp-associative` switch directs the compiler not to treat floating-point multiplication and addition as associative.

-no-mem

The `-no-mem` switch causes the compiler to not invoke the Memory Initializer tool after linking the executable. This is the default setting. See also "-mem" on page 1-29.

-no-std-ass

The `-no-std-ass` (disable standard assertions) switch prevents the compiler from defining the system, machine, cpu and compiler assertions and from defining system-specific assertions. See the `-A` switch (on page 1-18) for the list of standard assertions.

-no-std-def

The `-no-std-def` (disable standard macro definitions) prevents the compiler from defining any default preprocessor macro definitions.

-no-std-inc

The `-no-std-inc` (disable standard include search) switch directs the C preprocessor to search for header files in only the current directory and directories specified with `-I` switch (on page 1-25).

You can invoke this switch by selecting the **Ignore standard include paths** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Preprocessor** category.

-no-std-lib

The `-no-std-lib` (disable standard library search) switch directs the linker to limit its search for libraries to directories specified with the `-L` switch (on page 1-27). The compiler also defines `__NO_STD_LIB` during the linking stage and passes it to the linker, so that the `SEARCH_DIR` directives in the `.LDF` file can de disabled.

-no-widen-muls

The `-no-widen-muls` (disable widening multiplications) switch disables the compiler optimization which it performs on multiplication operations.

By default, the compiler, attempts to optimize integer multiplication operations which are stored in a long result to utilize the double-word MAC result registers of ADSP-218x DSPs. The code produced this way is better suited to the processor and therefore more efficient.

However, this optimization can generate overflow results which are not consistent in some cases and may differ from expected results depending on the optimizations enabled and the way that the source is written. The inconsistency and differences are seen if an overflow and truncation of the integer operands would normally occur.

When the optimization is applied, there is no truncation. When the optimization is disabled, the result of overflow will be truncated to integer size before being stored in the long result.

-O

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

You can invoke this switch by selecting the **Enable optimization** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **General** category.

-Oa

The `-Oa` (automatic function inlining) switch enables the inline expansion of C functions which are not necessarily declared inline in the source code. The amount of auto-inlining the compiler performs is controlled using the `-Ov` (optimize for speed versus size) switch (on page 1-33). Therefore, use of `-Ov100` indicates that as many functions as possible will be auto-inlined whereas `-Ov0` prevents any function from being auto-inlined. Specifying `-Oa` also implies the use of `-O`.

ⓘ When remarks are enabled, the compiler will produce a remark to indicate each function that is inlined.

-Os

The `-Os` (optimize for size) switch directs the compiler to produce code that is optimized for size. This is achieved by performing all optimizations except those that increase code size. The optimizations not performed include loop unrolling and jump avoidance. It also uses a function to save and restore preserved registers for a function instead of generating the more cycle-efficient inline default versions.

-Ov num

The `-Ov num` (optimize for speed versus size) switch directs the compiler to produce code that is optimized for speed versus size. The '`num`' should be an integer between 0 (purely size) and 100 (purely speed).

-o filename

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

-P

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

-PP

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

-path-{asm|compiler|def|lib|link|mem} filename

The `-path` (tool location) switch directs the compiler to use the specified component in place of the default installed version of the compilation tool. The component should comprise a relative or absolute path to its location. Respectively, the tools are the assembler, compiler, driver definitions file, librarian, linker or memory initializer.

Use this switch when you wish to override the normal version of one or more of the tools. The `-path` switch also overrides the directory specified by the `-path-install` switch (on page 1-34).

-path-install directory

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

ⓘ You can selectively override this switch with the `-path` switch (on page 1-34).

-path-output directory

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

-path-temp directory

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

-pch

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

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

-pch directory

The `-pch directory` (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 that do not strictly conform to the ISO/ANSI standard for C.

ⓘ The compiler may not detect all such constructs. In particular, the `-pedantic` switch does not cause the compiler to issue errors when Analog Devices keyword extensions are used.

-pedantic-errors

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

-pplist filename

The `-pplist` (preprocessor listing) switch 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:

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

-proc processor

The `-proc` (target processor) switch specifies that the compiler should produce code suitable for the specified processor. The *`processor`* identifiers directly supported in VisualDSP++ 3.5 are:

ADSP-2181, ADSP-2183, ADSP-2184, ADSP-2185, ADSP-2186, ADSP-2187, ADSP-2188, and ADSP-2189

For example,

```
cc218x -proc ADSP-2181 p1.c
```

If no target is specified with the `-proc` switch, the compiler will generate an error message.

If the processor identifier is unknown to the compiler, it attempts to read required switches for code generation from the file `<processor>.ini`. The assembler searches for the `.ini` file in the VisualDSP ++ System folder. For custom processors, the compiler searches the section "`proc`" in the `<processor>.ini` for key '`architecture`'. The custom processor must be based on an architecture key that is one of the known processors. For example, `-proc Customxxx` searches the `Customxxx.ini` file.

When compiling with the `-proc` switch, the compiler defines the corresponding `__ADSP21{81|83|84|85|86|87|88|89}__` macro for the variant selected in addition to the `__ADSP21XX__` and `__ADSP218X__` macros which are always are defined as 1. For example, `__ADSP2181__` and `__ADSP218X__` are pre-defined to 1 by the compiler.

See also "-si-revision version" on page 1-39 for more information on silicon revision of the specified processor.

-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.

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

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

-R-

The `-R-` (disable source path) switch removes all directories from the standard search path for source files, effectively disabling this feature.

This option is position-dependent on the command line; it only affects files following it.

-reserve register[,register...]

The `-reserve` (reserve register) switch directs the compiler *not* to use the specified register(s). This guarantees that a known register or set of registers is available for autobuffering.

You can reserve the `I2`, `I3`, `I5`, `I7` and `M0` registers. Separate register names with commas on the compiler command line. Reserving registers seriously reduces the effectiveness of compiler optimizations and should only be done when autobuffering (using circular buffers) is essential. For more information, refer to "Autobuffering Support" on page 1-134.

-S

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

You can invoke this switch by selecting the **Stop after: Compiler** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **General** category selection.

-s

The `-s` (strip debugging information) switch directs the compiler to remove debugging 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 to retain intermediate files generated and normally removed as part of the various compilation stages. These intermediate files are placed in the `-path-output` specified output directory or the build directory if the `-path-output` switch (on page 1-35) is not used. See Table 1-1 on page 1-9 for a list of input/output files (file extensions).

-show

The `-show` (display command line) switch directs the compiler to echo all command-line arguments, expanded option files switches and environment variables used by the compiler.

-si-revision version

The `-si-revision` *`version`* (silicon revision) switch sets the version of the hardware which is the required target for the build. It is used to enable inherent behavior relating to any errata in specific silicon revisions. The revision can be specified as "none" or a number of the form described by regular expression `[0-9]+\.[0-9]{1,3}` (for example 1.123). The compiler defines a macro `__SILICON_REVISION__` to a value specific to each silicon revision. For unknown revisions, the compiler will generate a warning and default to the latest known revision.

The parameter "*`version`*" represents a silicon revision of the processor specified by the `-proc` switch (on page 1-36). The "`none`" revision disables support for silicon errata.

For example,

```
cc218x -proc ADSP-2189 -si-revision 0.1
```

In the absence of silicon revision, the compiler selects the greatest silicon revision it "knows" about, if any.

A compiler will "pass along" the appropriate `-si-revision` switch setting when invoking another VisualDSP++ tool, for example, when the compiler driver invokes assembler and linker.

-signed-bitfield

The `-signed-bitfield` (make plain bitfields signed) switch directs the compiler to make bitfields which have not been declared with an explicit signed or unsigned keyword to be signed. This switch does not effect plain one-bit bitfields which are always unsigned. This is the default mode. See also the `-unsigned-bitfield` switch (on page 1-42).

-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__` macro. This is the default mode when the `-unsigned-char` (make char unsigned) switch (on page 1-43) is not used.

-syntax-only

The `-syntax-only` (just check syntax) switch directs the compiler to check the source code for syntax errors and warnings. No output files will be generated with this switch.

-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 are defined if the system returns information for them:

Macro	Description
`__HOSTNAME__`	The name of the host machine
`__MACHINE__`	The machine type of the host machine
`__SYSTEM__`	The OS name of the host machine
`__USERNAME__`	The current user's login name
`__GROUPNAME__`	The current user's group name
`__REALNAME__`	The current user's real name

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

-T filename

The `-T` (Linker Description File) switch directs that the linker, when invoked, will use the specified Linker Description File (LDF). If `-T` is not specified, a default `.LDF` file is selected based on the processor variant.

-time

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

-Umacro

The -U (undefine macro) switch lets you undefine macros. If you specify a macro name, it will be undefined. The compiler processes all -D (define macro) switches on the command line before any -U (undefine macro) switches.

You can invoke this switch by selecting the **Undefines** field in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Preprocessor** category.

-unsigned-bitfield

The -unsigned-bitfield (make plain bitfields unsigned) switch directs the compiler to make bitfields which have not been declared with an explicit signed or unsigned keyword to be unsigned. This switch does not effect plain one-bit bitfields which are always unsigned.

For example, given the declaration

```
struct {
   int a:2;
   int b:1;
   signed int c:2;
   unsigned int d:2;
} x;
```

the bitfield values are:

Field	-unsigned-bitfield	-signed-bitfield	Why
x.a	-2..1	0..3	Plain field
x.b	0..1	0..1	One bit
x.c	-2..1	-2..1	Explicit signed
x.d	0..3	0..3	Explicit unsigned

See also the -signed-bitfields switch (on page 1-40).

-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.

-val-global <name-list>

The `-val-global` (add global names) switch directs the compiler that the names given by <*name-list*> are present in all globally defined variables. The list is separated by double colons(::). In C, the names are prefixed and separated by underscores (_). The compiler will issue an error on any globally defined variable in the current source module(s) not using <*name-list*>. This switch is used to define VCSE components.

-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 compiler version) switch directs the compiler to display its version information.

-Wdriver-limit number

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

-Werror-limit number

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

-W{error|remark|suppress|warn} <num>[,num...]

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

At compilation time, the compiler produces a number for each specific compiler diagnostic message. The `{D}` (discretionary) string after the diagnostic message number indicates that the diagnostic may have its severity overridden. Each diagnostic message is identified by a number that is used across all compiler software releases.

-Wremarks

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

You can invoke this switch by selecting the **Enable remarks** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **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.

ⓘ You can invoke this switch by selecting the **Disable all warnings and remarks** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Warning** selection.

-warn-protos

The `-warn-protos` (prototypes warning) switch directs the compiler to produce a warning message when a function is called without a full prototype being supplied.

-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.

-write-opts

The `-write-opts` switch directs the compiler to pass the user-specified options (but *not* the input file names) to the main driver via a temporary file. This can be helpful if the resulting main driver command line would otherwise be too long.

-xref <file>

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.

The `ref-code` parameter is one of the following characters.

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)

Data Type Sizes

The sizes of intrinsic C data types are selected by Analog Devices so that normal C programs execute with hardware-native data types and therefore execute at high speed.

Table 1-4. Data Type Sizes for the ADSP-218x DSPs

Type	Bit Size	`sizeof` returns
`int`	16 bits signed	1
`unsigned int`	16 bits unsigned	1
`char`	16 bits signed	1
`unsigned char`	16 bits unsigned	1
`short`	16 bits signed	1
`unsigned short`	16 bits unsigned	1
`pointer`	16 bits	1
`fract16`	16 bits fractional (defined in `fract_typedef.h`)	1
`long`	32 bits signed	2
`unsigned long`	32 bits unsigned	2
`function pointer`	32 bits	2
`float`	32 bits float	2
`double`	32 bits float	2
`fract32`	32 bits fractional (defined in `fract_typedef.h`)	2

On any platform the basic type `int` is the native word size. The data type `long` is 32 bits, as is `float`. A pointer is the same size as an `int`.

On the ADSP-218x processor architecture, the `long long int`, `unsigned long long int`, and `long double` data types are not implemented (they are not redefined to other types). In general, `double` word data types should be expected to run more slowly, relying largely on software-emulated arithmetic.

Analog Devices does not support data sizes smaller than a single word location for ADSP-218x processors. For the current processors, this means that both `short` and `char` have the same size as `int`. Although 16-bit `chars` are unusual, they conform to the standard.

Type `double` poses a special problem. The C language tends to default to `double` for constants and in many floating-point calculations. 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 and provide the best performance, the size of `double` on the ADSP-218x DSPs is always 32 bits. This should be acceptable for most DSP programming. It is not, however, fully standard conforming.

The standard `include` files automatically redefine the math library interfaces such that functions like `sin` can be directly called with the proper size operands; therefore:

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

For full descriptions of these functions and their implementation, see Chapter 3, "C Run-Time Library".

Optimization Control

The general aim of compiler optimization is to generate correct code that executes fast and is small in size. Not all optimizations are suitable for every application or possible all the time so the compiler optimizer has a number of configurations, or optimization levels, which can be applied when suitable. Each of these levels are enabled by one or more compiler switches (and VisualDSP++ project options) or pragmas.

Refer to Chapter 2, "Achieving Optimal Performance from C Source Code" for information on how to obtain maximal code performance from the compiler.

The following list identifies several optimization levels. The levels are notionally ordered with least optimization listed first and most optimization listed last. The descriptions for each level outline the optimizations performed by the compiler and identifies any switches or pragmas required or that have direct influence on the optimization levels performed.

- **Debug**
  The compiler produces debug information to ensure that the object code matches the appropriate source code line. See "-g" on page 1-23 for more information.

- **Default**
  The compiler does not perform any optimizations by default when none of the compiler optimization switches are used (or enabled in VisualDSP++ project options). Default optimization level can be enabled using the `optimize_off` pragma (on page 1-105).

- **Procedural Optimizations**
  The compiler performs advanced, aggressive optimization on each procedure in the file being compiled. The optimizations can be directed to favor optimizations for speed (`-O`) or space (`-Os`) or a factor between speed and space (`-Ov`). If debugging is also

requested, the optimization is given priority, so the debugging functionality may be limited. See "-O" on page 1-32, "-Os" on page 1-33, and "-Ov num" on page 1-33.

Procedural optimizations for speed and space (using `-O` and `-Os`) can be enabled in C source using the pragma `optimize_{for_speed|for_space}` (see on page 1-105 for more information on optimization pragmas).

- **Automatic Inlining**
  The compiler automatically inlines C functions which are not necessarily declared as inline in the source code. It does this when it has determined that doing so will reduce execution time. How aggressively the compiler performs automatic inlining is controlled using the `-Ov` switch. Automatic inlining is enabled using the `-Oa` switch and additionally enables Procedural Optimizations (`-O`). See "-O" on page 1-32, "-Os" on page 1-33, and "-Ov num" on page 1-33. for more information.

- **Interprocedural optimizations (IPA)**
  The compiler performs advanced, aggressive optimization over the whole program, in addition to the per-file optimizations in procedural optimization. IPA is enabled using the `-ipa` switch and additionally enables Procedural Optimizations (`-O`). See "-ipa" on page 1-26 and "-O" on page 1-32 for more information.

Interprocedural Analysis

The compiler has an optimization capability called *Interprocedural Analysis* (IPA) that allows the compiler to optimize across translation units instead of within individual translation units. This capability allows the compiler to see all of the source files used in a final link at compilation time and to use that information while optimizing.

Interprocedural analysis is enabled by selecting the **Interprocedural analysis** option on the **Compiler** tab (accessed via the VisualDSP++ **Project Options** dialog box), or by specifying the `-ipa` command-line switch (see on page 1-26). The `-ipa` switch automatically enables the `-O` switch to turn on optimization.

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; when a special program has 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.

Because IPA operates only during the final link, the `-ipa` switch has no benefit when compiling the source files to object format for inclusion in a library. Although IPA will generate usage information for potential additional optimizations at the final link stage, neither the usage information nor the module's source file are available when the linker includes a module from a library. Therefore, each library module is compiled to the normal `-O` optimization level.

The prelinker inspects object modules included from libraries and other object files which were not compiled with the `-ipa` switch to see whether there are hidden references to the functions and variables defined in those objects which were compiled with the `-ipa` switch, and optimizes those variables and functions accordingly.

C Compiler Language Extensions

The compiler supports a set of extensions to the ISO/ANSI standards for the C language. These C extensions add support for DSP hardware and allow some C programming features.

This section contains:

The additional keywords that are part of these C extensions do not conflict with any ISO/ANSI 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 (see on page 1-30), the compiler defines shorter forms of the keyword extensions that omit the leading underscores.

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, and if you need to declare a function as inline, or allocate variables to memory spaces, you can use `__inline` or `__pm`/`__dm` respectively.

Table 1-5 provides a list and a brief description of keyword extensions. Table 1-6 provides a list and a brief description of operational extensions. Both tables direct you to sections of this chapter that document each extension in more detail.

Table 1-5. Keyword Extensions

Keyword extensions	**Description**
`inline (function)`	Directs the compiler to integrate the function code into the code of the callers. For more information, see "Inline Function Support Keyword (inline)" on page 1-55.
`dm`	Specifies the location of a static or global variable or qualifies a pointer declaration "*" as referring to data memory. For more information, see "Dual Memory Support Keywords (pm dm)" on page 1-71.

Table 1-5. Keyword Extensions (Cont'd)

Keyword extensions	Description
`pm`	Specifies the location of a static or global variable or qualifies a pointer declaration "*" as referring to program memory. For more information, see "Dual Memory Support Keywords (pm dm)" on page 1-71.
`section(string)`	Specifies the section in which an object or function is placed. For more information, see "Placement Support Keyword (section)" on page 1-76.
`bool, true, false`	A Boolean type. For more information, see "Boolean Type Support Keywords (bool, true, false)" on page 1-77.
`restrict` keyword	Specifies restricted pointer features. For more information, see "Pointer Class Support Keyword (restrict)" on page 1-77.

Table 1-6. Operational Extensions

Operation extensions	Description
Variable-length arrays	Support for 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 1-78.
Non-constant initializers	Support for non-constant initializers lets you use non-constants as elements of aggregate initializers for automatic variables. For more information, see "Non-Constant Aggregate Initializer Support" on page 1-80.
Indexed initializers	Support for indexed initializers lets you specify elements of an aggregate initializer in an arbitrary order. For more information, see "Indexed Initializer Support" on page 1-80.
Preprocessor generated warnings	Support for generating warning messages from the preprocessor. For more information, see "Preprocessor-Generated Warnings" on page 1-83.
C++-style comments	Support for C++-style comments in C programs. For more information, see "C++-Style Comments" on page 1-83.

Inline Function Support Keyword (inline)

The `inline` keyword directs `cc218x` to integrate the code for the function you declare as `inline` into the code of its callers. Inline function support and the `inline` keyword is a standard feature of C++; the compiler provides it as a C extension. Use of 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 following example shows a function definition that uses the `inline` keyword.

```
inline int max3 (int a, int b int c) {
   return max (a, max(b, c));
}
```

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. Usually, it will also be declared `static`.

In some cases, the compiler does not output object 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 compiler only inlines functions, even those declared using the inline keyword, when optimizations are enabled (using the `-O` switches, as described on page 1-32).

Inline Assembly Language Support Keyword (asm)

The `cc218x` `asm()` construct lets you code ADSP-218x processor's assembly language instructions within a C function and to pass declarations and directives through to the assembler. The `asm()` construct is useful for expressing assembly language statements that cannot be expressed easily or efficiently with C constructs.

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

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

The `asm()` constructs are executable statements, and as such, may not appear before declarations within C functions.

`asm()` constructs may also be used at global scope, outside function declarations. Such `asm` constructs are used to pass declarations and directives directly to the assembler. They are not executable constructs, and may not have any inputs or outputs, or affect any registers.

A simplified `asm()` construct without operands takes the form of:

```
asm(" ENA INTS;");
```

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

Note that the compiler generates a label before and after inline assembly instructions when generating debug code (`-g` switch). These labels are used to generate the debug line information used by the debugger. If the inline assembler inserts conditionally assembled code, then likely at link time, an undefined symbol error will occur. If the inline assembler changes the section, causing the compilers labels to be placed, for example, in a data section (instead of the default code section), then the debug line information will be incorrect for these lines.

Using `asm()` constructs with operands requires some additional syntax. The construct syntax is described in:

- "Assembly Construct Template" on page 1-57
- "Assembly Construct Operand Description" on page 1-61
- "Assembly Constructs With Multiple Instructions" on page 1-67
- "Assembly Construct Reordering and Optimization" on page 1-67
- "Assembly Constructs with Input and Output Operands" on page 1-68
- "Assembly Constructs and Macros" on page 1-70
- "Assembly Constructs and Flow Control" on page 1-70

Assembly Construct Template

Using `asm()` constructs, you can specify the operands of the assembly instruction using C expressions. You do not need to know which registers or memory locations contain C variables.

ASM() Construct Syntax:

Use the following general syntax for your `asm()` constructs.

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

The syntax elements are defined as:

template

The template is a string containing the assembly instruction(s) with `%number` indicating where the compiler should substitute the operands. Operands are numbered in order of occurrence from left to right, starting at 0. Separate multiple instructions with a semicolon; then enclose the entire string within double quotes.

For more information on templates containing multiple instructions, see "Assembly Constructs With Multiple Instructions" on page 1-67.

constraint

The constraint is a string that directs the compiler to use certain groups of registers for the input and output operands. Enclose the constraint string within double quotes. For more information on operand constraints, see "Assembly Construct Operand Description" on page 1-61.

output operand

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

input operand

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

`clobber`

The clobber notifies the compiler that a list of registers are overwritten by the assembly instructions. Use lowercase characters to name clobbered registers. Enclose each name within double quotes, and separate each quoted register name with a comma. The input and output operands are guaranteed not to use any of the clobbered registers, so you can read and write the clobbered registers as often as you like. See Table 1-8 on page 1-66.

ASM() Construct Syntax Rules

These rules apply to assembly construct template syntax:

- The template is the only mandatory argument to `asm()`. All other arguments are optional.
- An operand constraint string followed by a C expression in parentheses describes each operand. For output operands, it must be possible to assign to the expression—that is, the expression must be legal on the left side of an assignment statement.
- A colon separates:
  - The template from the first output operand
  - The last output operand from the first input operand
  - The last input operand from the clobbered registers
- A comma separates operands and registers within arguments.
- The number of operands in arguments must match the number of operands in your template.
- The maximum permissible number of operands is ten (`%0`, `%1`, `%2`, `%3`, `%4`, `%5`, `%6`, `%7`, `%8`, and `%9`).

The compiler cannot check whether the operands have data types that are reasonable for the instruction being executed. The compiler does not parse the assembler instruction template, interpret the template, nor verify whether the template contains valid input for the assembler.

ASM() Construct Template Example

The following example shows how to apply the `asm()` construct template to the ADSP-218x assembly language `abs` instruction:

```
{
int result, x;
…
asm (
   "%0 = abs %1;" :
   "=d" (result) :
   "d" (x)
   );
}
```

In the previous example, note the following points:

- The template is "`%0=abs %1;`". The `%0` is replaced with operand zero `(result)`, the first operand. The `%1` is replaced with operand one `(x)`.
- The output operand is the C variable `result`.

  The letter `c` is the operand constraint for the variable. This constrains the output to an `ALU` result register. The compiler generates code to copy the output from the register to the variable result, if necessary. The “=” in `=c` indicates that the operand is an output.
- The input operand is the C variable `x`.

  The letter `c` is the operand constraint for the variable. This constrains `x` to an `ALU` register. If `x` is stored in different kinds of registers or in memory, the compiler generates code to copy the values into an register before the `asm()` construct uses them.

Assembly Construct Operand Description

The second and third arguments to the `asm()` construct describe the operands in the assembly language template. There are several pieces of information that need to be conveyed for `cc218x` to know how to assign registers to operands. This information is conveyed with an operand constraint. The compiler needs to know what kind of registers the assembly instructions can operate on, so it can allocate the correct register type.

You convey this information with a letter in the operand constraint string which describes the class of allowable registers. Table 1-7 on page 1-65 describes the correspondence between constraint letters and register classes.

The use of any letter not listed in Table 1-7 results in unspecified behavior. The compiler does not check the validity of the code by using the constraint letter.

For example, if your assembly template contains "`ax1 = dm(%0 += m3);`" and the address you want to load from is in the variable `p`, the compiler needs to know that it should put `p` in a `DAG1 I` register (`I0-I3`) before it generates your instruction. You convey this information to `cc218x` by specifying the operand "`w" (p)` where "`w`" is the constraint letter for `DAG1I` registers.

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

- The output operand list appears as the first argument after the assembly language template. The list is separated from the assembly language template with a colon. The input operands are separated from the output operands with a colon and always follow the output operands.
- The operand constraints describe which registers are modified by an assembly language instruction. The "=" in `=constraint` indicates that the operand is an output; all output operand constraints must use `=`.
- The compiler may allocate an output operand in the same register as an unrelated input operand, unless the output operand has the `&=` constraint modifier. This situation can occur because the compiler assumes that the inputs are consumed before the outputs are produced.

  This assumption may be false if the assembler code actually consists of more than one instruction. In such a case, use `&=` for each output operand that must not overlap an input or supply an "&" for the input operand. Table 1-7 lists operand constrains.

Operand constraints indicate what kind of operand they describe by means of preceding symbols. The possible preceding symbols are: no symbol, `=`, `+`, `&`, `?`, and `#`.

- (no symbol)

  The operand is an input. It must appear as part of the third argument to the `asm()` construct. The allocated register will be loaded with the value of the C expression before the `asm()` template is executed. Its C expression will not be modified by the `asm()`, and its value may be a constant or literal. Example: `d`

- **= symbol**

  The operand is an output. It must appear as part of the second argument to the `asm()` construct. Once the `asm()` template has been executed, the value in the allocated register is stored into the location indicated by its C expression; therefore, the expression must be one that would be valid as the left-hand side of an assignment.
  Example: `=d`

- **+ symbol**

  The operand is both an input and an output. It must appear as part of the second argument to the `asm()` construct. The allocated register is loaded with the C expression value, the `asm()` template is executed, and then the allocated register's new value is stored back into the C expression. Therefore, as with pure outputs, the C expression must be one that is valid on the left-hand side of an assignment.
  Example: `+d`

- **? symbol**

  The operand is temporary. It must appear as part of the third argument to the `asm()` construct. A register is allocated as working space for the duration of the `asm()` template execution. The register's initial value is undefined, and the register's final value is discarded. The corresponding C expression is not loaded into the register, but must be present. This expression is normally specified using a literal zero. Example: `?d`

- **& symbol**

  This operand constraint may be applied to inputs and outputs. It indicates that the register allocated to the input (or output) may not be one of the registers that are allocated to

the outputs (or inputs). This operand constraint is used when one or more output registers are set while one or more inputs are still to be referenced. (This situation sometimes occurs if the `asm()` template contains more than one instruction.)
Example: `&d`

- **# symbol**

  The operand is an input, but the register's value is clobbered by the `asm()` template execution. The compiler may make no assumptions about the register's final value. The operand must appear as part of the second argument to the `asm()` construct.
  Example: `#d`

Table 1-7 lists the registers that may be allocated for each register constraint letter. The use of any letter not listed in the "Constraint" column of this table results in unspecified behavior. The compiler does not check the validity of the code by using the constraint letter. Table 1-8 on page 1-66 lists the registers that may be named as part of the clobber list.

It is also possible to claim registers directly, instead of requesting a register from a certain class using the constraint letters. You can claim the registers directly by simply naming the register in the location where the class letter would be. The register names are the same as those used to specify the clobber list; see Table 1-8.

For example,

```
asm("%0 = %1 + %2;"
:"=ar"(sum)       /* output */
:"g"(x),"G"(y)    /* input  */
);
```

would load `x` into `ALU-X` register, `y` into `ALU-Y` register, and `sum` will be calculated in register `AR`.

Table 1-7. ASM() Operand Constraints

Constraint[1]	Description	Registers
b	input xregs to MAC	`MX1, MX0, SR1, SR0, MR1, MR0, AR`
B	input yregs to MAC	`MY1, MY0`
c	results from ALU	`AR`
C	result from int multiplies	`MR0`
cc	Used in the clobber list to tell the compiler that condition codes have been clobbered	`ASTAT`
d	input xregs to SHIFTER	`SI, SR1, SR0, MR1, MR0, AR`
D	result from shift	`SR1`
e	data registers, size 16	`SI, AX1, AX0, MX1, MX0, MY0, MY1, AY1, AY0, MR1, MR0, SR1, SR0, AR`
f	shift amount	`SE`
g	ALU X registers	`AX1 AX0 AR SR1 SR0 MR1 MR0`
G	ALU Y registers	`AY1 AY0`
memory	Used in the clobber list to tell the compiler that the `asm()` statement writes to memory	
r	all registers	`SR1, SR0, SI, MY1, MX1, AY1, AX1, MY0, MX0, AY0, AX0, MR1, MR0, AR, I0-I7, M0-M7, L0-L7`
u	DAG1 L registers	`L0-L3`
v	DAG2 L registers	`L4-L7`
w	DAG1 I registers	`I0-I3`
x	DAG1 M registers	`M0-M3`
y	DAG2 I registers	`I4-I7`

Table 1-7. ASM() Operand Constraints (Cont'd)

Constraint[1]	Description	Registers
z	DAG2 M registers	`M4-M7`
=&constraint	Indicates that the constraint is applied to an output operand that may not overlap an input operand	
=constraint	Indicates that the constraint is applied to an output operand	
&constraint	Indicates the constraint is applied to an input operand that may not be overlapped with an output operand	
=&constraint	Indicates the constraint is applied to an output operand that may not overlap an input operand	
?constraint	Indicates the constraint is temporary	
+constraint	Indicates the constraint is both an input and output operand	

1 The use of any letter not listed here results in unspecified behavior. The compiler does not check the validity of the code by using the constraint letter.

Table 1-8. Register Names for asm() Constructs

Clobber String	Meaning
`"AX1", "AX0", "AY1", "AY0", "AR", "AF"`	ALU registers
`"MX1", "MX0", "MY1", "MY0", "MR1", "MR0", "MR2", MF`	MAC registers
`"SI", "SE", "SR1", "SR0", "SB", "SR2"`	SHIFTER registers
`"I0", "I1", "I2", "I3", "I6", "I7"`	DAG addressing registers
`"M0", "M3", "M5"`	Modifier registers
`"L0", "L1", "L2", "L3", "L5", "L6", "L7"`	Length register
`"PX"`	PMD-DMD bus exchange register
`"astat"`	ALU status registers
`"MSTAT", "MMODE", "SSTAT"`	Mode control registers
`"IMASK", "ICNTL", "IFC"`	Interrupt registers

Table 1-8. Register Names for asm() Constructs (Cont'd)

Clobber String	Meaning
`"CNTR", "TOPPCSTACK"`	Program sequencer registers
`"DMOVLAY", "PMOVLAY"`	Overlay registers
`"cc"`	Condition code register
`"memory"`	Unspecified memory location(s)

Assembly Constructs With Multiple Instructions

There can be many assembly instructions in one template. If the `asm()` string is longer than one line, you may continue it on the next line by placing a backslash (\) at the end of the line or by quoting each line separately.

This is an example of multiple instructions in a template:

```
asm ("se=exp %1 (hi); \
   "sr=norm %1 (hi);  \
   "%0=sr0;"
   : "=e" (normalized)          // output
   : "e" (inval) ;              // input
   : "se", "sr1", "sr0") ;      // clobbers
```

Assembly Construct Reordering and Optimization

For the purpose of optimization, the compiler assumes that the side effects of an `asm()` construct are limited to changes in the output operands or the items specified using the clobber specifiers. This does not mean that you cannot use instructions with side effects, but you must be careful to notify the compiler that you are using them by using the clobber specifiers (see Table 1-8 on page 1-66).

The compiler may eliminate supplied assembly instructions if the output operands are not used, move them out of loops, or replace two with one if they constitute a common subexpression. Also, if the instruction has a side effect on a variable that otherwise appears not to change, the old value of the variable may be reused later if it happens to be found in a register.

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

```
#define IOwrite(val,addr) \
asm volatile ("si="#val";IO("#addr")=si;": : :"si");
```

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

Assembly Constructs with Input and Output Operands

The output operands must be write only; `cc218x` assumes that the values in these operands do not need to be preserved. When the assembler instruction has an operand that is both read from and written to, you must logically split its function into two separate operands: one input operand and one write-only output operand. The connection between them is expressed by constraints that say they need to be in the same location when the instruction executes.

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

```
/* (pseudo code) modify (shoe += sock); */
asm("modify(%0 += %2);":"=w"(shoe):"0"(shoe),"x"(sock));
```

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

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

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

In some cases, operands `0` and `1` could be stored in different registers due to reloading or optimizations.

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

```
int result_asm;
int *my_variable;
/* NOT recommended */
/* (pseudo code) result_asm = dm(*my_variable += M3); */
/* asm() operation changes value of my_variable       */

asm("%0=DM(%1 += M3);":"=e"(result_asm):"w"(my_variable));
```

Assembly Constructs and Macros

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

```
#define abs_macro(result,x) \
asm("%0=abs %1;":"=c"(result):"c"(x))
/* (pseudo code) result = abs x */

main(){
int result_var=0;
int x_var=10;

abs_macro(result_var, 10);
/* or */
abs_macro(result_var, x_var);
}
```

Assembly Constructs and Flow Control

It is inadvisable to place flow control operations within an `asm()` construct that “leaves” the `asm()` construct, such as calling a procedure or performing a jump, to another piece of code that is not within the `asm()` construct itself. Such operations are invisible to the compiler and may violate assumptions made by the compiler.

For example, the compiler is careful to adhere to the calling conventions for preserved registers when making a procedure call. If an `asm()` construct calls a procedure, the `asm()` construct must also ensure that all conventions are obeyed, or the called procedure may corrupt the state used by the function containing the `asm()` construct.

Dual Memory Support Keywords (pm dm)

This section covers `cc218x` keyword extensions to the C language. These extensions support the dual-memory space, modified Harvard architecture of the ADSP-218x processors. There are two keywords used to designate memory space—`dm` and `pm`. They can be used to specify the location of a static or global variable or to qualify a pointer declaration.

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.

- A memory space keyword (`dm` or `pm`) refers to the expression to its right.
- You can specify a memory space for each level of pointer. This corresponds to one memory space for each `*` in the declaration.
- The compiler uses data memory as the default memory space for all variables. All undeclared spaces for data are data memory spaces.
- The compiler always uses program memory as the memory space for functions. Function pointers always point to program memory.
- You cannot assign memory spaces to automatic variables. All automatic variables reside on the stack, which is always in data memory.
- Literal character strings always reside in data memory.
- Although program memory on the ADSP-218x DSPs consists of 24-bit words, only 16 bits of each word are used when C data is stored in `pm`. (This is normally the case for assembly language programming as well.) If you need special access to all 24 bits, you should use an assembly language subroutine and work with the `PX` register.

The following listing shows examples of dual memory keyword syntax.

```
int pm abc[100];
   /* declares an array abc with 100 elements in program memory */
int dm def[100];
   /* declares an array def with 100 elements in data memory   */
int ghi[100];
   /* declares an array ghi with 100 elements in data memory   */
int pm * pm pp;
   /* declares pp to be a pointer which resides in program memory
   and points to a program memory integer                      */
int dm * dm dd;
   /* declares dd to be a pointer which resides in primary Data
   Memory and points to a data memory integer                  */
int *dd;
   /* declares dd to be a pointer which resides in data memory
   and points to a data memory integer                         */
int pm * dm dp;
   /* declares dp to be a pointer which resides in data memory
   and points to a program memory integer                      */
int pm * dp;
   /* declares dp to be a pointer which resides in data memory
   and points to a program memory integer                      */
int dm * pm pd;
   /* declares pd to be a pointer which resides in pm (secondary
   data memory) and points to a data memory integer            */
int * pm pd;
   /* declares pd to be a pointer which resides in Program memory
   and points to a data memory integer                         */
float pm * dm * pm fp;
   /* the first pm means that *fp is in program memory,
   the following dm puts *fp in data memory, and fp
   itself is in program memory                                 */
```

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

```
   /* Dual Memory Support Keyword typedef & struct Examples */
typedef float pm * PFLOATP;
   /* PFLOATP defines a type which is a pointer to a          */
   /* float which resides in pm                               */

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

Memory Keywords and Assignments/Type Conversions

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

- You may make assignments between variables allocated in different memory spaces.
- Pointers to program memory must always point to `pm`. Pointers to data memory must always 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.

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

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

/* Illegal Dual Memory Space Type Cast Example */
```

```
int pm *x;
int dm *y;
int dm a;
x = y;         /* Compiler will flag error */
x = &a;        /* Compiler will flag error */
```

Memory Keywords and Function Declarations/Pointers

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

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

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

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

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

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

Memory Keywords and Function Arguments

The compiler checks whether calls to prototyped functions for memory space specifications are consistent with the function prototype. The following example shows sample code that compiler 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
         n 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. cc218x 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 C preprocessor macros to define common memory spaces that must be coordinated. The following listing shows two code segments that are equivalent after preprocessing.

These code segments demonstrate how you can redefine the memory space specifications by redefining the macros `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);
```

PM and DM Compiler Support for Standard C Library Functions

There are a number of functions defined in the standard C library that take pointer input parameter types. These functions, which include for example `strlen()`, are implemented differently when the pointer input is to program memory (PM) or data memory (DM). The different implementations are called automatically by the compiler because it has specific in-built knowledge about the standard C functions that require pointer parameters. The support requires that the normal standard header file, for example `string.h`, is included prior to use of the function requiring PM and DM variants. The default library function variants are DM should the include file not be used.

Placement Support Keyword (section)

The `section` keyword directs the compiler to place an object or function in an assembly `.SECTION`, in the compiler's intermediate assembly output file. You name the assembly `.SECTION` with `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 only meaningful 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, meaning they are explicitly `static`, or are given as external-object definitions.

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

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

Boolean Type Support Keywords (bool, true, false)

The `bool`, `true`, and `false` keywords are extensions that support the C boolean type. The `bool` keyword is a unique signed integral type. 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`; 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 keywords 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 can not designate the same object and, therefore, are not aliases. The compiler is free to use the information about restricted pointers and aliasing in order to better optimize C code that uses pointers.

The restrict keyword is most useful when applied to function parameters about which the compiler would otherwise have little information. For example,

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

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

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

If your program 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. Unlike other automatic arrays, variable-length ones are declared with a non-constant length. This means that the space is allocated when the array is declared, and deallocated when the brace-level is exited.

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

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

```
struct entry
var_array (int array_len, char data[array_len][array_len])
{
        ...
    }
```

The compiler calculates the length of an array at the time of allocation. It then remembers the array length until the brace-level is exited and can return it as the result of the `SIZEOF()` function performed on the array. Because variable-length arrays must be stored on the stack, it is impossible to have variable-length arrays in program memory (`pm`). The compiler issues an error if an attempt is made to use a variable-length array in `pm`.

For 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 input matrices. Declaring automatic variable-size matrix is much easier then explicitly allocating it in a heap.

The expression declares an array with a size that is computed at run time. The length of the array is computed on entry to the block and saved in `sizeof()` that 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 are deallocated.

For example, the following program prints 40, not 50.

```
#include <stdio.h>
void foo(int);
main ()
{
   foo(40);
}

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

Non-Constant Aggregate Initializer Support

The compiler includes extended support for aggregate initializers. The compiler does not require the elements of an aggregate initializer for an automatic variable to be constant expressions. The following example shows an initializer with elements that vary at run time:

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

All automatic structures can be initialized by arbitrary expressions involving literals, previously declared variables and functions.

Indexed Initializer Support

ISO/ANSI Standard C requires the elements of an initializer to appear in a fixed order, the same as the order of the elements in the array or structure being initialized. The `cc218x` C compiler, 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. All index values must be constant expressions, even in automatic arrays.

For an array initializer, the syntax `[INDEX]` appearing before an initializer element value specifies the index to be initialized by that value. Subsequent initializer elements are then applied to sequentially following elements of the array, unless another use of the `[INDEX]` syntax appears. The index values must be constant expressions, even if the array being initialized is automatic.

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

```
/* Example 1. Standard C & cc218x C Array Initializer */
/* Standard C array initializer */

int abc[6] = { 0, 0, 12, 0, 14, 0 };

/* equivalent cc218x C array initializer */

int abc[6] = { [3] 12, [5] 14 };
```

You can combine this technique of naming elements with standard C initialization of successive elements. The standard C and `cc218x` instructions below are equivalent. Note that any unlabeled initial value is assigned to the next consecutive element of the structure or array.

```
/* Example 2. Standard C & cc218x C Array Initializer */
/* Standard C array initializer */

int abc[6] = { 0, 5, 6, 0, 12, 0 };

/* equivalent cc218x C array initializer that uses indexed elements */

int abc[6] = { [1] 5, 6, [4] 12 };
```

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

```
/* Example 3. C Array Initializer With enum Type Indices */
/* cc218x C array initializer */

int charsarray[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 *field name* before the element value. The standard C and cc218x C struct initializers in the example below are equivalent.

```
/* Example 4. Standard C & cc218x C struct Initializer */
/* Standard C struct Initializer */

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

/* Equivalent cc218x C struct Initializer With Labeled Elements */

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

Aggregate Constructor Expression Support

Extended initializer support includes support for aggregate constructor expressions, which enable you to assign values to large structure types without requiring each element's value to be individually assigned. The following example shows an ISO/ANSI standard C `struct` usage followed by equivalent `cc218x` code that has been simplified using an constructor expression.

```
/* Standard C struct & cc218x C Constructor struct */
/* Standard C struct */

struct foo {int a; char b[2];};
struct foo make_foo(int x, char *s)
{
struct foo temp;
temp.a = x;
temp.b[0] = s[0];
if (s[0] !=  '\0')
   temp.b[1] = s[1];
else
   temp.b[1] = '\0';
return temp;
}
```

```
/* Equivalent cc218x C constructor struct */
struct foo make_foo(int x, char *s)
{
return((struct foo) {x, {s[0], s[0] ? s[1] : '\0'}});
}
```

Preprocessor-Generated Warnings

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

C++-Style Comments

The compiler accepts C++-style comments, 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;
```

Compiler Built-in Functions

The compiler supports intrinsic functions that enable efficient use of hardware resources. Knowledge of these functions is built into the `cc218x` compiler. Your program uses them via normal function call syntax. The compiler notices the invocation and generates one or more machine instructions, just as it does for normal operators, such as `+` and `*`.

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

The `cc218x` compiler provides built-in versions of some of the C library functions as described in "Using the Compiler's Built-In Functions" on page 3-2.

The `sysreg.h` header file defines a set of functions that provide efficient system access to registers, modes and addresses not normally accessible from C source. These functions are specific to individual architectures and this section lists the built-in functions supported at this time on ADSP-218x DSPs.

The compiler supports:

- "I/O Space for Read/Write" on page 1-84
- "Read/Write of Non-Memory-Mapped Registers" on page 1-85
- "Interrupt Control" on page 1-85

I/O Space for Read/Write

The inclusion of `sysreg.h` allows the use of built-in functions that generate efficient inline instructions to implement read and write of values from and to I/O space addresses.

The prototypes for these functions are, as defined in `sysreg.h`:

```
void io_space_write(const unsigned int addr, const int value);
int io_space_read(const unsigned int addr);
```

These functions are fully described in "io_space_read" on page 3-75 and "io_space_write" on page 3-77.

Read/Write of Non-Memory-Mapped Registers

The inclusion of `sysreg.h` allows the use of built-in functions that generate efficient inline instructions to implement read and write of values from and to non-memory-mapped registers. The DSP has status registers which track and control machine status modes. These registers are:

`ASTAT`, `SSTAT`, `MSTAT`, `ICNTL`, `IMASK`, and `IFC`

The prototypes for these functions are, as defined in `sysreg.h`:

```
void sysreg_write(const int sysreg, const int value);
int sysreg_read(const int sysreg);
```

The `sysreg` parameter for these functions should be a member of the `SysReg` enumeration defined in `sysreg.h`. This enumeration is used to map the actual registers to a small constant defined as a user-friendly name.

An example call of `sysreg read` of `IMASK` might be:

```
#include <sysreg.h>
int value = sysreg_read(sysreg_IMASK);
```

These functions are fully described in "sysreg_read" on page 3-154 and "sysreg_write" on page 3-156.

Interrupt Control

The inclusion of `sysreg.h` allows the use of built-in functions that generate the instructions to enable and disable interrupts. The prototypes for these functions are, as defined in `sysreg.h`:

```
void enable_interrupts(void);
void disable_interrupts(void);
```

These functions are fully described in "enable_interrupts" on page 3-50 and "disable_interrupts" on page 3-48.

ETSI Support

The ETSI (European Telecommunications Standards Institute) support for ADSP-218x processors is a collection of functions that provides high performance implementations for operations commonly required by DSP applications. These operations provided by the ETSI library (`libetsi.dlb`) and compiler built-in functions (defined in `ETSI_fract_arith.h`) include support for fractional or fixed-point arithmetic. The results obtained from of use of these operations have well defined overflow and saturation conditions. The ETSI support operations are Analog Devices extensions to ANSI C standard.

The ETSI support contains functions that you can call from your source program. The following topics describe how to use this support.

- "ETSI Support Overview" on page 1-86
- "Calling ETSI Library Functions" on page 1-88
- "Using the ETSI Built-In Functions" on page 1-89
- "Linking ETSI Library Functions" on page 1-89
- "Working with ETSI Library Source Code" on page 1-90
- "ETSI Support for Data Types" on page 1-90
- "ETSI Header File" on page 1-91

ETSI Support Overview

The use of fractional arithmetic is vital for many applications on DSP processors as information can be held more compactly than in floating point. It would take 24 bits in floating-point format to match the precision of 16-bit fractional data. Also, control of normalization and precision is more complex with floating point. Many DSPs do not include hardware support for floating-point arithmetic and these operations are therefore very expensive in both code size and performance terms for such DSPs.

Fractional data has a representation similar to that of integers except that while an integer value is considered to have a decimal point to the right of the least significant bit, a fractional value is considered to have a decimal point to the left of the most significant bit. Fractional values are usually held in 16-bit or 32-bit "containers". In each case, signed values are in the range [-1.0, +1.0).

The bit operations on fractional data are identical to those on integer data, but there are three aspects of the result that are normally treated differently:

1. **MSB extraction:** Multiplication is a widening operation, thus multiplying a 16-bit value by another 16-bit value produces a 32-bit result. If a 16-bit integer result is required then this is taken to be the least significant 16 bits of the result, and the upper 16 bits are regarded as overflow. For a fractional operation the upper 16 bits would represent a 16-bit result, and the lower 16 bits would be regarded as an underflow.
2. **Duplicate sign bit elimination:** Following a multiplication of two 16-bit values the nature of the representation results in two "sign bits" in the result. For normal integer arithmetic this causes no problem, but for fractional arithmetic a shift left by one is required to normalize the result.
3. **Saturation:** If we perform an arithmetic operation that would cause us to overflow, it can be useful to return the maximum (appropriately signed) number that can be represented in the result register. The alternatives which include firing an interrupt, saying the result is undefined and is some other number, usually look less attractive to DSP programmers.

These fractional operations can often be done at no extra cost to normal integer operations on DSPs using special instructions or modes of operation.

The C programming language does not include a basic type for fractional data, and rather than introduce a non-standard type, Analog Devices defines `fract16` and `fract32` in terms of appropriately-sized integer data types and provides sets of basic intrinsic functions which perform the required operations. These look like library function calls, but are specially recognized by the compilers, which generate short sequences or single instructions, exploiting any specialized features, which may be available on the architecture. An important aspect of this is that the compiler optimizer is not inhibited in any way by the use of these intrinsics.

Because of the varying nature of the architectures, these basic intrinsic functions cannot be standardized across all the architectures. However, a set of standard functions for manipulating fractional data has been defined by the ITU (International Telecommunications Union) and ETSI (European Telecommunications Standards Institute).

Referred to as the ETSI Standard Functions, these have been very widely used to implement telecommunications packages such as GSM, EFR and AMR Vocoders, and have become a de-facto industry standard. These functions have been implemented on ADSP-218x DSPs.

The ETSI standard is aimed at DSP processors with 16-bit inputs, saturated arithmetic and 32-bit accumulators.

Calling ETSI Library Functions

To use an ETSI function, call the function by name and give the appropriate arguments. The names and arguments for each function appear on the function's reference page. The names and arguments for each function appear in "ETSI Header File" on page 1-91.

Like other functions you use, ETSI functions should be declared. Declarations are supplied in the header file `ETSI_fract_arith.h`, which must be included in any source files where ETSI functions are called. The function names are C function names. If you call C run-time library functions from an assembly language program, you must use the assembly version of the function name—prefix an underscore on the name. For more information on naming conventions, see "C and Assembly Language Interface" on page 1-148.

The standard reference code for the ETSI functions uses the `Carry` and `Overflow` flag global variables to keep track of any carry or overflow as a result of using on of the ETSI functions. With the ETSI functions provided by Analog Devices, this can be switched off by compiling with `__NO_ETSI_FLAGS` defined in the compiler command line. In fact, this is the default for the ADSP-218x DSP implementation.

If you wish to keep track of these flags, for debugging purposes, compile with `__NO_ETSI_FLAGS` set to zero. This stipulates the use of the functions in accordance with the ETSI standard, but will result in a reduced performance.

Using the ETSI Built-In Functions

Some of the ETSI functions have been implemented as part of `cc218x` compiler's set of built-in functions. For information on how to use these functions, refer to the section "Compiler Built-in Functions" on page 1-83. These built-in implementations will be automatically defined when header file `ETSI_fract_arith.h` is included.

Linking ETSI Library Functions

When your C code calls an ETSI function that is not implemented using a compiler built-in, the call creates a reference that the linker resolves when linking. This tells the linker to be directed to link with the ETSI library, `libetsi.dlb` in the `218x\lib` directory, which is a subdirectory of the VisualDSP++ installation directory. This is done automatically when using the default Linker Description File (LDF) for ADSP-218x DSP targets, as these specify that `libetsi.dlb` will be on each link line.

If not using default `.LDF` files, then either add `libetsi.dlb` to the `.LDF` file which is being used, or alternatively use the compiler's `-letsi` switch to specify that `libetsi.dlb` is to be added to the link line.

Working with ETSI Library Source Code

The source code for functions and macros in the ETSI library is provided with your VisualDSP++ software. By default, the installation program copies the source code to a subdirectory of the directory where the run-time libraries are kept named `218x\lib\src\libetsi_src`. Each function is kept in a separate file. The file name is the name of the function with the extension `.asm`. If you do not intend to modify any of the functions, you can delete this directory and its contents to conserve disk space.

The source code is provided so you can customize specific functions for your own needs. To modify these files, you need proficiency in ADSP-218x assembly language and an understanding of the run-time environment, as explained in "C and Assembly Language Interface" on page 1-148.

Before you make any modifications to the source code, copy the source code to a file with a different file name and rename the function itself. Test the function before you use it in your system to verify that it is functionally correct.

Analog Devices only supports the run-time library functions as provided.

ETSI Support for Data Types

ETSI functions support `fract16` and `fract32` data types as follows:

- `fract16` is a 16-bit fractional data type (1.15 format) having a range of [-1.0, +1.0). This is defined in the C language as:

  ```
  typedef short fract16
  ```

- `fract32` is a 32-bit fractional data type (1.31 format) having a range of [-1.0, +1.0). This is defined in the C language as:

  ```
  typedef long fract32
  ```

ETSI Header File

The following are summary descriptions of the functions provided by the ETSI library, as defined in the header file `ETSI_fract_arith.h`.

Short absolute

```
fract16 abs_s (fract16)
```

This function returns the 16-bit value that is the absolute value of the input parameter. Where the input is `0x8000`, saturation occurs and `0x7fff` is returned.

Short add

```
fract16 add (fract16, fract16)
```

This function returns the 16-bit result of addition of the two `fract16` input parameters. Saturation occurs with the result being set to `0x7fff` for overflow and `0x8000` for underflow.

Short division

```
fract16 div_s (fract16, fract16)
```

This function returns the 16-bit result of the fractional integer division of `f1` by `f2`. `f1` and `f2` must both be positive fractional values with `f2` greater than `f1`.

Long division

```
fract16 div_l (fract32, fract16)
```

This function produces a result which is the fractional integer division of the first parameter by the second. Both inputs must be positive and the least significant word of the second parameter must be greater or equal to the first; the result is positive (leading bit equal to 0) and truncated to 16 bits.

Extract high (most significant 16 bits)

```
fract16 extract_h (fract32)
```

This function returns the 16 most significant bits if the 32-bit `fract` parameter provided.

Extract low (least significant 16 bits)

```
fract16 (fract32)
```

This function returns the 16 least significant bits of the 32-bit `fract` parameter provided.

Multiply and accumulate with rounding

```
fract16 mac_r (fract32, fract16, fract16)
```

This function performs an `L_mac` operation using the three parameters provided. The result is the rounded 16 most significant bits of the 32-bit results from the `L_mac` operation.

Multiply and subtract with rounding

```
fract16 msu_r (fract32, fract16, fract16)
```

This function performs an `L_msu` operation using the three parameters provided. The result is the rounded 16 most significant bits of the 32-bit result from the `L_msu` operation.

Short multiply

```
fract16 mult (fract16, fract16)
```

This function returns the 16-bit result of the fractional multiplication of the input parameters. The result is saturated.

Multiply with rounding

```
fract16 mult_r (fract16, fract16)
```

This function performs a 16-bit multiply with rounding of the result of the fractional multiplication of the two input parameters.

Short negate

```
fract16 negate (fract16)
```

This function returns the 16-bit result of the negation of the input parameter. If the input is `0x8000`, saturation occurs and `0x7fff` is returned.

Long normalize

```
fract16 norm_l (fract16)
```

This function returns the number of left shifts required to normalize the input variable for positive values on the interval with minimum of `0x40000000` and maximum of `0x7fffffff`, and for negative values on the interval with minimum of `0x80000000` and maximum of `0xc0000000`.

Short normalize

```
fract16 norm_s (fract16)
```

This function returns the number of left shifts required to normalize the input 16 bit variable for positive values on the interval with minimum of `0x4000` and maximum of `0x7fff`, and for negative values on the interval with minimum of `0x8000` and maximum of `0xc000`.

Round

```
fract16 round (fract32)
```

This function rounds the lower 16-bits of the 32-bit input parameter into the most significant 16 bits with saturation. The resulting bits are shifted right by 16.

Saturate

```
fract16 saturate (fract32)
```

This function returns the 16 most significant bits of the input parameter. If the input parameter is greater than `0x7fff`, `0x7fff` is returned. If the input parameter is less than `0x8000`, `0x8000` is returned.

Short shift left

```
fract16 shl (fract16, fract16)
```

This function arithmetically shifts the first parameter left by second parameter bits. The empty bits are zero filled. If second parameter is negative the operation shifts right.

Short shift right

```
fract16 shr (fract16, fract16)
```

This function arithmetically shifts the first parameter right by second parameter bits with sign extension. If second parameter is negative the operation shifts left.

Shift right with rounding

```
fract16 shr_r (fract16, fract16)
```

This function performs a shift to the right as per the `shr()` operation with additional rounding and saturation of the result.

Short subtract

```
fract16 sub (fract16, fract16)
```

This function returns the 16-bit result of the subtraction of the two parameters. Saturation occurs with the result being set to `0x7fff` for overflow and `0x8000` for underflow.

Long absolute

```
fract32 L_abs (fract32)
```

This function returns the 32-bit absolute value of the input parameter. In cases where the input is equal to `0x80000000`, saturation occurs and `0x7fffffff` is returned.

Long add

```
fract32 L_add (fract32, fract32)
```

This function returns the 32-bit saturated result of the addition of the two input parameters.

Long add with carry

```
fract32 L_add_c (fract32, fract32)
```

This function performs 32-bit addition of the two input parameters. Uses the `Carry` flag as additional input when using the ETSI flag variables.

16-bit variable -> most significant bits (least significant bits zeroed)

```
fract32 L_deposit_h (fract16)
```

This function deposits the 16-bit parameter into the 16 most significant bits of the 32-bit result. The least 16 bits are zeroed.

16-bit variable -> least significant bits (sign extended)

```
fract32 L_deposit_l (fract16)
```

This function deposits the 16-bit parameter into the 16 least significant bits of the 32-bit result. The most significant bits are set to sign extension for the input.

Multiply and accumulate

```
fract32 L_mac (fract32, fract16, fract16)
```

This function performs a fractional multiplication of the two 16-bit parameters and returns the saturated sum of the multiplication result with the 32-bit parameter.

Multiply and accumulate without saturation

```
fract32 L_macNs (fract32, fract16, fract16)
```

This function performs a non-saturating version of the `L_mac` operation.

Multiply both the most significant bits and the least significant bits of a long, by the same short

```
fract32 L_mls (fract32, fract16)
```

Multiply and subtract

```
fract32 L_msu (fract32, fract16, fract16)
```

This function performs a fractional multiplication of the two 16-bit parameters and returns the saturated subtraction of the multiplication result with the 32-bit parameter.

Multiply and subtract without saturation

```
fract32 L_msuNs (fract32, fract16, fract16)
```

This function performs a non-saturating version of the `L_msu` operation.

Long multiply

```
fract32 L_mult (fract16, fract16)
```

This function returns the 32-bit result of the fractional multiplication of the two 16-bit parameters.

Long negate

```
fract32 L_negate (fract32)
```

This function returns the 32-bit result of the negation of the parameter. Where the input parameter is `0x80000000` saturation occurs and `0x7fffffff` is returned.

Long saturation

```
fract32 L_sat (fract32)
```

The resultant variable is set to `0x80000000` if `Carry` and `Overflow` ETSI flags are set (underflow condition), else if `Overflow` is set, the resultant is set to `0x7fffffff`. The default revision of the library simply returns as no checking or setting of the `Overflow` and `Carry` flags is performed.

Long shift left

```
fract32 L_shl (fract32, fract16)
```

This function arithmetically shifts the 32-bit first parameter to the left by the value given in the 16-bit second parameter. The empty bits of the 32-bit result are zero filled.

If the second parameter is negative, the shift performed is to the right with sign-extended. The result is saturated in cases of overflow and underflow.

Long shift right

```
fract32 L_shr (fract32, fract16)
```

This function arithmetically shifts the 32-bit first parameter to the right by the value given in the 16-bit second parameter with sign extension. If the shifting value is negative, the source is shifted to the left. The result is saturated in cases of overflow and underflow.

Long shift right with rounding

```
fract32 L_shr_r (fract32, fract16)
```

This function performs the shift-right operation as per `L_shr` but with rounding.

Long subtract

```
fract32 L_sub (fract32, fract32)
```

This function returns the 32-bit saturated result of the subtraction of two 32-bit parameters (first-second).

Long subtract with carry

```
fract32 L_sub_c (fract32, fract32)
```

This function performs 32-bit subtraction of the two input parameters. Uses the `Carry` flag as additional input when using the ETSI flag variables.

Compose long

```
fract32 L_Comp (fract16, fract16)
```

This function composes a `fract32` type value from the given `fract16` high (first parameter) and low (second parameter) components. The sign is provided with the low half, the result is calculated to be:

```
   high<<16 + low<<1
```

Multiply two longs

```
   fract32 Mpy_32 (fract16, fract16, fract16, fract16)
```

This function performs the multiplication of two `fract32` type variables, provided as high and low half parameters. The result returned is calculated as:

```
   Res = L_mult(hi1,hi2);
   Res = L_mac(Res, mult(hi1,lo2),1);
   Res = L_mac(Res, mult(lo1,hi2),1);
```

Multiply short by a long

```
   fract32 Mpy_32_16 (fract16, fract16, fract16)
```

Extract a long from two shorts

```
   void L_Extract(fract32 src,fract16 *hi,fract16 *lo)
```

This function extracts low and high halves of `fract32` type value into `fract16` variables pointed to by the parameters `hi` and `lo`. The values calculated are:

```
   Hi = bit16 to bit31 of src
   Lo = (src - hi<<16)>>1
```

Fract integer division of two longs

```
fract32 Div_32(fract32 L_num,fract16 denom_hi,fract16 denom_lo)
```

This is 32-bit fractional divide operation. The result returned is the `fract32` representation of `L_num` divided by `L_denom` (represented by `demon_hi` and `denom_lo`). `L_num` and `L_denom` must both be positive fractional values and `L_num` must be less that `L_denom` to ensure that the result falls within the fractional range.

Pragmas

The compiler supports a number of pragmas. Pragmas are implementation-specific directives that modify the compiler's behavior. There are two types of pragma usage: pragma directives and pragma operators.

Pragma directives have the following syntax:

```
#pragmapragma-directive pragma-directive-operands new-line
```

Pragma operators have the following syntax:

```
_Pragma (string-literal )
```

When processing a pragma operator, the compiler effectively turns it into a pragma directive using a non-string version of *`string-literal`*. This means that the following pragma directive

```
#pragmalinkage_name mylinkname
```

can also equivalently be expressed using the following pragma operator

```
_Pragma ("linkage_name mylinkname")
```

The examples in this manual use the directive form.

The C compiler supports pragmas for:

- Arranging alignment of data
- Defining functions that can act as interrupt handlers
- Changing the optimization level, midway through a module
- Changing how an externally visible function is linked
- Header file configurations and properties
- Giving additional information about loop usage to improve optimizations

The following sections describe the pragmas that support these features.

- "Data Alignment Pragmas" on page 1-101
- "Interrupt Handler Pragmas" on page 1-102
- "Loop Optimization Pragmas" on page 1-103
- "General Optimization Pragmas" on page 1-105
- "Linking Control Pragmas" on page 1-106
- "Stack Usage Pragma" on page 1-108
- "Function Side-Effect Pragmas" on page 1-109
- "Header File Control Pragmas" on page 1-115

The compiler will issue a warning when it encounters an unrecognized pragma directive or pragma operator. The compiler will not expand any preprocessor macros used within any pragma directive or pragma operator.

Data Alignment Pragmas

The data alignment pragmas include `align` and `pad` pragmas. Alignments specified using these pragmas must be a power of two. The compiler will reject uses of those pragmas that specify alignments that are not powers of two.

#pragma align (*num*)

The `align (`*`num`*`)` pragma may be used before variable and field declarations. It applies to the variable or field declaration that immediately follows the pragma. Use of this pragma causes the compiler to generate the next variable or field declaration aligned on a boundary specified by *`num`*.

The `align` pragma is useful for declaring arrays that need to be on a circular boundary. Such arrays might be required to make use of a bit-reversal sorting algorithm that is implemented using the ADSP-218x processor's DAG1 bit reversal mode.

```
#pragma align 256
int arr[128];
```

The `align` pragma is also useful for declaring arrays in C at correct base adresses. This way the arrays can be passed to assembly functions and used as circular buffers or storage for autobuffering.

If the `#pragma align` directives cause "insufficient memory" errors at link time due to fragmentation, compile using the `-flags-link -ip` switch options (described on page 1-23). See the `-ip` switch description in the *VisualDSP++ 3.5 Linker and Utilities Manual for 16-Bit Processors* for more information.

#pragma pad (*alignopt*)

The `#pragma pad(alignopt)` may be applied to `struct` definitions. It applies to `struct` definitions that follow, until the default alignment is restored by omitting *`alignopt`*, for example, by `#pragma pad()` with empty parentheses.

This pragma is effectively a shorthand for placing `#pragma align` before every field within the `struct` definition. The following example shows how to use `#pragma pad()`.

```
#pragma pad(4)
struct {
   int i;
   int j;
} s = {1,2};
#pragma pad()
```

Interrupt Handler Pragmas

The interrupt pragma s include `interrupt` and `altregisters` pragmas.

#pragma interrupt

The `interrupt` pragma may be used before a function declaration or definition. It applies to the function declaration or definition that immediately follows the pragma. Use of this pragma causes the compiler to generate the function code so that it may be used as a self dispatching interrupt handler.

The compiler arranges for the function to save its context above and beyond the usual caller-preserved set of registers, and to restore the context upon exit. The function will return using a return from interrupt (RTI) instruction.

```
#pragma interrupt
void field_SIG()
{
```

```
/* ISR code */
}
```

#pragma altregisters

The `altregisters` pragma may be used in conjunction to the `interrupt` pragma to indicate that the compiler can optimize the saving and restoring of registers through use of the secondary register sets. Note the use of the `altregisters` pragma is not safe when nested interrupts are enabled.

```
#pragma interrupt
#pragma altregisters
void field_SIG()
{
/* ISR code */
}
```

Loop Optimization Pragmas

Loop optimization pragmas give the compiler additional information about usage within a particular loop, which allows the compiler to perform more aggressive optimization. The pragmas are placed before the loop statement, and apply to the statement that immediately follows, which must be a `for`, `while` or `do` statement to have effect. In general, it is most effective to apply loop pragmas to inner-most loops, since the compiler can achieve the most savings there.

The optimizer always attempts to vectorize loops when it is safe to do so. The optimizer exploits the information generated by the interprocedural analysis (see "Interprocedural Analysis" on page 1-50) to increase the cases where it knows it is safe to do so. Consider the following code:

```
void copy(short *a, short *b) {
     int i;
     for (i=0; i<100; i++)
         a[i] = b[i];
}
```

If you call copy with two calls, say copy(x,y) and later copy(y,z), the interprocedural analysis will not be able to tell that “a” never aliases “b”. Therefore, the optimizer cannot be sure that one iteration of the loop is not dependent on the data calculated by the previous iteration of the loop. If it is known that each iteration of the loop is not dependent on the previous iteration, then the vector_for pragma can be used to explicitly notify the compiler that this is the case.

#pragma loop_count(*min, max, modulo*)

The #pragma loop_count(min, max, modulo) appears just before the loop it describes. It asserts that the loop will iterate at least min times, no more than max times, and a multiple of modulo times. This information enables the optimizer to omit loop guards, to decide whether the loop is worth completely unrolling, and whether code need be generated for odd iterations. The last two arguments can be omitted if they are unknown.

For example,

```
int i;
#pragma loop_count(24, 48, 8)
for (i=0; i < n; i++)
```

#pragma vector_for

The #pragma vector_for notifies the optimizer that it is safe to execute two iterations of the loop in parallel. The vector_for pragma does not force the compiler to vectorize the loop; the optimizer checks various properties of the loop and does not vectorize it if it believes it is unsafe or if it cannot deduce that the various properties necessary for the vectorization transformation are valid.

Strictly speaking, the pragma simply disables checking for loop-carried dependencies.

```
void copy(short *a, short *b) {
int i;
#pragma vector_for
for (i=0; i<100; i++)
    a[i] = b[i];
}
```

In cases where vectorization is impossible (for example, if array `a` were aligned on a word boundary, but array `b` was not), the information given in the assertion made by `vector_for` may still be put to good use in aiding other optimizations.

#pragma no_alias

Use the `#pragma no_alias` to tell the compiler the following has no loads or stores that conflict due to references to the same location through different pointers, known as "aliases". In this example,

```
void vadd(int *a, int *b, int *out, int n) {
   int i;
#pragma no_alias
   for (i=0; i < n; i++)
       out[i] = a[i] + b[i];
}
```

the use of `#pragma no_alias` just before the loop informs the compiler that the pointers `a`, `b` and `out` point to different arrays, so no load from `b` or `a` will be using the same address as any store to `out`. Therefore, `a[i]` or `b[i]` is never an alias for `out[i]`. The use of the `no_alias` pragma can lead to better code because it allows the loads and stores to be reordered and any number of iterations to be performed concurrently, thus providing better software pipelining by the optimizer.

General Optimization Pragmas

There are three pragmas which can change the optimization level while a given module is being compiled. These pragmas must be used at global scope, immediately prior to a function definition.

These pragmas are:

- **#pragma optimize_off**

  This pragma turns off the optimizer, if it was enabled. This pragma has no effect if Interprocedural Optimization Analysis is enabled.

- **#pragma optimize_for_space**

  This pragma turns the optimizer back on, if it was disabled, or sets focus to give reduced code size a higher priority than high performance, where these conflict.

- **#pragma optimize_for_speed**

  This pragma turns the optimizer back on, if it was disabled, or sets focus to give high performance a higher priority than reduced code size, where these conflict.

- **#pragma optimize_as_cmd_line**

  This pragma resets the optimization settings to be those specified on the `cc218x` command line when the compiler was invoked.

The following shows example uses of these pragmas.

```
#pragma optimize_off
void non_op() { /* non-optimized code */ }

#pragma optimize_for_space
void op_for_si() { /* code optimized for size */ }

#pragma optimize_for_speed
void op_for_sp() { /* code optimized for speed */ }
/* subsequent functions declarations optimized for speed */
```

Linking Control Pragmas

Linking pragmas change how a given global function or variable is viewed during the linking stage.

#pragma linkage_name *identifier*

The `#pragma linkage_name` associates the *`identifier`* with the next external function declaration. It ensures that identifier is used as the external reference, instead of following the compiler's usual conventions.

If identifier is not a valid function name, as could be used in normal function definitions, the compiler will generate an error.

The following shows an example use of this pragma.

```
#pragma linkage_name realfuncname
void funcname ();
void func() {
   funcname();   /* compiler will generate a call to
                    realfuncname */
}
```

#pragma retain_name

The `#pragma retain_name` indicates that the external function or variable declaration that follows the pragma is not removed even though Interprocedural Analysis (IPA) sees that it is not used. Use this pragma for C functions that are only called from assembler routines, such as the startup code sequence invoked before `main()`. The following example shows how to use this pragma.

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

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

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

Since the program has no uses of either `delete_me()` or `keep_me()`, the compiler will remove `delete_me()`, but will keep `keep_me()` because of the pragma. You do not need to specify `retain_name` for `main()`.

For more information, see "Interprocedural Analysis" on page 1-50.

#pragma weak_entry

The `#pragma weak_entry` may be used before a static variable declaration or definition. It applies to the function or variable declaration or definition that immediately follows the pragma. Use of this pragma causes the compiler to generate the function or variable definition with weak linkage.

The following are example uses of the `pragma weak_entry` directive.

```
#pragma weak_entry
int w_var = 0;

#pragma weak_entry
void w_func(){}
```

Stack Usage Pragma

The C compiler for ADSP-218x DSPs supports the stack usage pragma `make_auto_static`.

#pragma make_auto_static

The `make_auto_static` pragma may be used before a function definition. The `make_auto_static` pragma directs the compiler to place all automatic variables used in the function in static store. This may be beneficial in code that requires many accesses of automatic variables. This is because an access to static store for the ADSP-218x DSPs is done in one instruction, whereas an access to local automatic stack are may require three instructions. The `-make-autostatic` switch (on page 1-29) can be used to notify the compiler to process all function definition in the current source being compiled as if pragma `make_auto_static` had been used.

Make sure that the `make_auto_static` pragma is not used on functions that are directly or indirectly recursive.

Function Side-Effect Pragmas

The function side-effect pragmas are used before a function declaration to give the compiler additional information about the function in order to enable it to improve the code surrounding the function call. These pragmas should be placed before a function declaration and apply to that function.

For example,

```
#pragma pure
long dot(short*, short*, int);
```

#pragma alloc

The `#pragma alloc` tells the compiler that the function behaves like the library function "`malloc`", returning a pointer to a newly allocated object. An important property of these functions is that the pointer returned by the function does not point at any other object in the context of the call. In the example,

```
#pragma alloc
int *new_buf(void);
int *vmul(int *a, int *b) {
   int *out = new_buf();
   for (i = 0; i < N; ++i)
      out[i] = a[i] * b[i];
   return out;
}
```

the compiler can reorder the iterations of the loop because the `#pragma alloc` tells it that `a` and `b` cannot overlap `out`.

The GNU attribute `malloc` is also be supported with the same meaning.

#pragma pure

The `#pragma pure` tells the compiler that the function does not write to any global variables, and does not read or write any volatile variables. Its result, therefore, is a function of its parameters or of global variables. If any of the parameters are pointers, the function may read the data they point at but it may not write it.

Therefore, because the function call will have the same effect every time it is called between assignments to global variables, the compiler need not generate the code for every call.

Therefore, in this example,

```
#pragma pure
long sdot(short *, short *, int);

long tendots(short *a, short *b, int n) {
   int i;
   long s = 0;
for (i = 1; i < 10; ++i)
      s += sdot(a, b, n);  // call can get hoisted out of loop
   return s;}
```

the compiler can replace the ten calls to `sdot` with a single call made before the loop.

#pragma const

The `#pragma const` is a more restrictive form of the `pure` pragma. It tells the compiler that the function does not read from global variables as well as not writing to them or reading or writing volatile variables. The result of the function is therefore a function of its parameters.

If any of the parameters are pointers, the function may not even read the data they point at.

#pragma regs_clobbered *string*

The `#pragma regs_clobbered` *`string`* may be used with a function declaration or definition to specify which registers are modified (or clobbered) by that function. The *`string`* contains a list of registers and is case-insensitive.

When used with an external function declaration, this pragma acts as an assertion telling the compiler something it would not be able to discover for itself. In the example,

```
#pragma regs_clobbered "ar m5"
void f(void);
```

the compiler knows that only registers `ar` and `m5` may be modified by the call to `f`, so it may keep local variables in other registers across that call.

The `regs_clobbered` pragma may also be used with a function definition, or a declaration preceding a definition, when it acts as a command to the compiler to generate register saves and restores on entry and exit from the function to ensure it only modifies the registers in string. For example,

```
#pragma regs_clobbered "ar m5"
int g(int a) {
   return a+3;
}
```

The `regs_clobbered` pragma may not be used in conjunction with `#pragma interrupt`. If both are specified, a warning is issued and the `regs_clobbered` pragma is ignored.

To obtain best results with the pragma, it is best to restrict the clobbered set to be a subset of the default scratch registers. The compiler is likely to produce more efficient code this way than if the scratch set is changed to use the same number of registers but which does not make a subset of the default volatile set usually scratch.

When considering when to apply the `regs_clobbered` pragma, it may be useful to look at the output of the compiler to see how many scratch registers were used. Restricting the volatile set to these registers will produce no impact on the code produced for the function but may free up registers for the caller to allocate across the call site.

String Syntax

A `regs_clobbered` string consists of a list of registers, register ranges, or register sets that are clobbered. The list is separated by spaces, commas, or semicolons.

A *register* is a single register name, which is the same as that which may be used in an assembly file.

A *register range* consists of `start` and `end` registers which both reside in the same register class, separated by a hyphen. All registers between the two (inclusive) are clobbered.

A *register set* is a name for a specific set of commonly clobbered registers that is predefined by the compiler. The following register sets are defined,

Set	Registers
`CCset`	ASTAT, condition codes
`MR`	MR0 - MR2
`SR`	SR0 - SR1
`DAG1scratch`	Members of DAG1 I, L, and M-registers that are scratch by default
`DAG2scratch`	Members of DAG2 I, L, and M-registers that are scratch by default
`DAGscratch`	`DAG1scratch` and `DAG2scratch`
`Dscratch`	Members of D-registers that are scratch by default
`ALLscratch`	Entire default volatile set

When the compiler detects an illegal string, a warning is issued and the default volatile set as defined in this compiler manual is used instead.

Unclobberable and Must Clobber Registers

There are certain caveats as to what registers may or must be placed in the clobbered set.

On ADSP-218x DSPs, the registers `I4`, `M4`, `MSTAT` and `M_MODE` may not be specified in the clobbered set, as the correct operation of the function call requires their value to be preserved. If the user specifies these registers in the clobbered set, a warning will be issued and they will be removed from the specified clobbered set.

The registers `AR` and `M5` are always clobbered. If the user specifies a function definition with the `regs_clobbered` pragma which does not contain these registers, a warning is issued and these registers are added to the clobbered set.

However, if the compiler sees an external function declaration with a `regs_clobbered` pragma that does not contain the `AR` and `M5` registers, a warning will not be issued as an assembly function may have been written which genuinely does not modify these registers.

Registers from these classes,

```
   D, I, ASTAT, PX, SE, SB, PMOVLAY, DMOVLAY, MF
```

may be specified in clobbered set and code will be generated to save them as necessary.

The L-registers are required to be zero on entry and exit from a function. A user may specify that a function clobbers the L-registers. If it is a compiler generated function, then it will in fact leave the L-registers zero at the end of the function. If it is an assembly function, then it may clobber the L-registers. In that case, the L-registers are re-zeroed after any call to that function.

The `IMASK`, `ICNTL`, `SSTAT`, `IFC`, and `OWNCNTR` registers are never used by the compiler and are never preserved.

User Reserved Registers
User reserved registers will never be preserved in the function wrappers whether in the clobbered set or not.

Function Results
The registers in which a function returns its result must always be clobbered by the callee and retain their new value in the caller. They may appear in the clobbered set of the callee but it will make no difference to the generated code; the return register will not be saved and restored. Only the return register used by the particular function return type is special. Return registers used by different return types will be treated in the clobbered list in the convention way.

For example,

```
typedef struct { int x, int y } Point;
typedef struct { int x[10] } Big;
int f();   // Result in AX1. SR1, SR0 and I0 may be
               preserved across call.
Point g(); // Result in SR1 and SR0. AX1 and I0 may be
               preserved across call.
Big f();   // Result pointer in I0. AX1, SR1 and SR0 may be
               preserved across call.
```

#pragma result_alignment (*n*)

The `#pragma result_alignment (n)` asserts that the pointer or integer returned by the function has a value that is a multiple of `n`. It is often used in conjunction with the `#pragma alloc` of custom allocation functions that return pointers that are more strictly aligned than be deduced from their type.

Header File Control Pragmas

The header file control pragmas help the compiler to handle header files.

#pragma hdrstop

The `#pragma hdrstop` is used in conjunction with the `-pch` (precompiled header) switch (on page 1-35). The switch tells the compiler to look for a precompiled header (`.pch` file), and, if it cannot find one, to generate a file for use on a later compilation. The `.pch` file contains a snapshot of all the code preceding the header stop point.

By default, the header stop point is the first non-preprocessing token in the primary source file. The `#pragma hdrstop` can be used to set the point earlier in the source file.

In the example,

```
#include "standard_defs.h"
#include "common_data.h"
#include "frequently_changing_data.h"

int i;
```

the default header stop point is start of the declaration of `i`. This might not be a good choice, as in this example, "`frequently_changing_data.h`" might change frequently, causing the `.pch` file to be regenerated often, and, therefore, losing the benefit of precompiled headers.

The `hdrstop` pragma can be used to move the header stop to a more appropriate place. In this case,

```
#include "standard_defs.h"
#include "common_data.h"
#pragma hdrstop
#include "frequently_changing_data.h"

int i;
```

#pragma no_pch

The `#pragma no_pch` overrides the `-pch` (precomiled headers) switch (on page 1-35) for a particular source file. It directs the compiler not to look for a `.pch` file and not to generate one for the specified source file.

#pragma once

The `#pragma once`, which should appear at the beginning of a header file, tells the compiler that the header is written in such a way that including it several times has the same effect as including it once.

For example,

```
#pragma once
#ifndef FILE_H
#define FILE_H
... contents of header file ...
#endif
```

In this example, the `#pragma once` is actually optional because the compiler recognizes the `#ifndef/#define/#endif` idiom and will not reopen a header that uses it.

#pragma system_header

The `#pragma system_header` identifies an `include` file as the file supplied with VisualDSP++. The pragma tells the compiler that every function and variable declared in the file (but not in files included in the file) is the variable or function with that name from the VisualDSP++ library.

The compiler will take advantage of any special knowledge it has of the behavior of the library.

GCC Compatibility Extensions

The compiler provides compatibility with the C dialect accepted by version 3.2 of the GNU C Compiler. Many of these features are available in the C99 ANSI Standard. A brief description of the extensions is included in this section. For more information, refer to the following web address:

`http://gcc.gnu.org/onlinedocs/gcc-3.2.1/gcc/C-Extensions.html#C%20Extensions`

Statement Expressions

A statement expression is a compound statement enclosed in parentheses. A compound statement itself is enclosed in braces { }, so this construct is enclosed in parentheses-brace pairs ({ }).

The value computed by a statement expression is the value of the last statement which should be an expression statement. The statement expression may be used where expressions of its result type may be used. But they are not allowed in constant expressions.

Statement expressions are useful in the definition of macros as they allow the declaration of variables local to the macro. In the following example,

```
#define min(a,b) ({                \
   short __x=(a),__y=(b),__res;   \
   if (__x > __y)                 \
      __res = __y;                \
   else                           \
   __res = __x;                   \
   __res;                         \
})

int use_min() {
   return  min(foo(), thing()) + 2;
}
```

The `foo()` and `thing()` statement get called once each because they are assigned to the variables `__x` and `__y` which are local to the statement expression that `min` expands to and `min()` can be used freely within a larger expression because it expands to an expression.

Labels local to a statement expression can be declared with the `__label__` keyword. For example,

```
({
   __label__ exit;
   int i;
   for (i=0; p[i]; ++i) {
      int d = get(p[i]);
      if (!check(d)) goto exit;
      process(d);
   }
exit:
   tot;
})
```

Statement expressions are an extension to C originally implemented in the GCC compiler. Analog Devices support the extension primarily to aid porting code written for that compiler. When writing new code consider using inline functions, which are compatible with ANSI/ISO C99 and are as efficient as macros when optimization is enabled.

Type Reference Support Keyword (typeof)

The `typeof( expression )` construct can be used as a name for the type of expression without actually knowing what that type is. It is useful for making source code that is interpreted more than once such as macros or include files more generic.

The `typeof` keyword may be used where ever a `typedef` name is permitted such as in declarations and in casts. For example,

```
#define abs(a) ({                    \
        typeof(a) __a = a;                  \
        if (__a < 0) __a = - __a;           \
        __a;                                \
    })
```

shows `typeof` used in conjunction with a statement expression to define a "generic" macro with a local variable declaration.

The argument to `typeof` may also be a type name. Because `typeof` itself is a type name, it may be used in another `typeof( type-name )` construct. This can be used to restructure the C type declaration syntax.

For example,

```
#define pointer(T)   typeof(T *)
#define array(T, N)  typeof(T [N])

array (pointer (char), 4) y;
```

declares `y` to be an array of four pointers to `char`.

The `typeof` keyword is an extension to C originally implemented in the GCC compiler. It should be used with caution because it is not compatible with other dialects of C and has not been adopted by the more recent C99 standard.

GCC Generalized Lvalues

A cast is an `lvalue` (may appear on the left hand side of an assignment) if its operand is an `lvalue`. This is an extension to C, provided for compatibility with GCC.

A comma operator is an `lvalue` if its right operand is an `lvalue`. This is an extension to C, provided for compatibility with GCC.

A conditional operator is an `lvalue` if its last two operands are `lvalues` of the same type. This is an extension to C, provided for compatibility with GCC.

Conditional Expressions with Missing Operands

The middle operand of a conditional operator can be left out. If the condition is non-zero (true), then the condition itself is the result of the expression. This can be used for testing and substituting a different value when a pointer is NULL. The condition is only evaluated once; therefore, repeated side effects can be avoided. For example,

```
printf("name = %s\n", lookup(key)?:"-");
```

calls `lookup()` once, and substitutes the string "-" if it returns NULL. This is an extension to C, provided for compatibility with GCC.

Hexadecimal Floating-Point Numbers

C99 style hexadecimal floating-point constants are accepted. They have the following syntax.

```
hexadecimal-floating-constant:
  {0x|0X} hex-significand binary-exponent-part [ floating-suffix ]
hex-significand: hex-digits [ . [ hex-digits ]]
binary-exponent-part: {p|P} [+|-] decimal-digits
floating-suffix: { f | l | F | L }
```

The hex-significand is interpreted as a hexadecimal rational number. the digit sequence in the exponent part is interpreted as a decimal integer. The exponent indicates the power of two by which the significand is to be scaled. The floating suffix has the same meaning it does for decimal floating constants: a constant with no suffix is of type `double`, a constant with suffix `F` is of type `float`, and a constant with suffix `L` is of type `long double`.

Hexadecimal floating-point constants enable the programmer to specify the exact bit pattern required for a floating-point constant. For example, the declaration

```
float f = 0x1p-126f;
```

causes f to be initialized with the value `0x800000`.

Zero Length Arrays

Arrays may be declared with zero length. This is an anachronism supported to provide compatibility with GCC. Use variable length array members instead.

Variable Argument Macros

The final parameter in a macro declaration may be followed by ... to indicate the parameter stands for a variable number of arguments.

For example,

```
#define trace(msg, args...) fprintf (stderr, msg, ## args);
```

can be used with differing numbers of arguments,

```
trace("got here\n");
trace("i = %d\n", i);
trace("x = %f, y = %f\n", x, y);
```

The `##` operator has a special meaning when used in a macro definition before the parameter that expands the variable number of arguments: if the parameter expands to nothing then it removes the preceding comma.

The variable argument macro syntax comes from GCC. It is not compatible with C99 variable argument macros.

Line Breaks in String Literals

String literals may span many lines. The line breaks do not need to be escaped in any way. They are replaced by the character `\n` in the generated string. The extension is not compatible with many dialects of C including ANSI/ISO C89 and C99. However, it is useful in `asm` statements, which are intrinsically non-portable.

Arithmetic on Pointers to Void and Pointers to Functions

Addition and subtraction is allowed on pointers to `void` and pointers to functions. The result is as if the operands had been cast to pointers to `char`. The `sizeof` operator returns one for `void` and `function` types.

Cast to Union

A type cast can be used to create a value of a union type, by casting a value of one of the unions member's types.

Ranges in Case Labels

A consecutive range of values can be specified in a single case, by separating the first and last values of the range with .... For example,

```
case 200 ... 300:
```

Declarations mixed with Code

In C mode the compiler will accept declarations in the middle of code as in C99 and C++. This allows the declaration of local variables to be placed at the point where they are required. Therefore, the declaration can be combined with initialization of the variable.

For example, in the following function

```
void func(Key k) {
Node *p = list;
while (p && p->key != k)
   p = p->next;
if (!p)
   return;
Data *d = p->data;
while (*d)
   process(*d++);
}
```

the declaration of `d` is delayed until its initial value is available, so that no variable is uninitialized at any point in the function.

Escape Character Constant

The character escape `'\e'` may be used in character and string literals and maps to the ASCII Escape code, 27.

Alignment Inquiry Keyword (__alignof__)

The `__alignof__ (type-name)` construct evaluates to the alignment required for an object of a type. The `__alignof__` *`expression`* construct can also be used to give the alignment required for an object of the *`expression`* type.

If expression is an `lvalue` (may appear on the left hand side of an assignment), the alignment returned takes into account alignment requested by pragmas and the default variable allocation rules.

Keyword for Specifying Names in Generated Assembler (asm)

The `asm` keyword can be used to direct the compiler to use a different name for a global variable or function. For example,

```
int N asm("C11045");
```

tells the compiler to use the label C11045 in the assembly code it generates wherever it needs to access the source level variable `N`. By default the compiler would use the label `_N`.

The `asm` keyword can also be used in function declarations but not function definition. However, a definition preceded by a declaration has the desired effect. For example,

```
extern int f(int, int) asm("func");
```

```
int f(int a, int b) {
. . .
}
```

Function, Variable and Type Attribute Keyword (__attribute__)

The `__attribute__` keyword can be used to specify attributes of functions, variables and types, as in these examples,

```
void func(void) __attribute__ ((section("fred")));

int a __attribute__ ((aligned (8)));

typedef struct {int a[4];} __attribute__((aligned (4))) Q;
```

The `__attribute__` keyword is supported, and therefore code, written for GCC, can be ported. All attributes accepted by GCC on ix86 are accepted. The ones that are actually interpreted by the compiler are described in the sections of this manual describing the corresponding pragmas (see "Pragmas" on page 1-99).

Preprocessor Features

The cc218x 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" on page 1-83 and "C++-Style Comments" on page 1-83.

This section contains:

- "Predefined Preprocessor Macros" on page 1-125
- "Header Files" on page 1-128
- "Writing Preprocessor Macros" on page 1-129
- "Preprocessing of .IDL Files" on page 1-131

Predefined Preprocessor Macros

The cc218x compiler defines a number of macros to produce information about the compiler, source file, and options specified. These macros can be tested, using the #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 "#" operator with a macro body is useful in converting such symbols into text constructs.

The predefined preprocessor macros are:

__ADSP21XX__ and __ADSP218X__

`cc218x` always defines `__ADSP21XX__` and `__ADSP218X__` as `1`.

__ADSP21{81|83|84|85|86|87|88|89}__

`cc218x` defines `__ADSP21{81|83|84|85|86|87|88|89}__` as `1` when you compile with the corresponding `-proc ADSP-21{81|83|84|85|86|87|88|89}` command-line switch.

__ANALOG_EXTENSIONS__

`cc218x` defines `__ANALOG_EXTENSIONS__` as `1` unless you compile with `-pedantic` or `-pedantic-errors`.

__DATE__

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

__DOUBLES_ARE_FLOATS__

`cc218x` always defines `__DOUBLES_ARE_FLOATS__` as `1`.

__ECC__

`cc218x` always defines `__ECC__` as `1`.

__EDG__

`cc218x` always defines `__EDG__` as `1`. This signifies that an Edison Design Group front end is being used.

__EDG_VERSION__

`cc218x` 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 `cc218x` command line or in a preprocessor `#include` command (ANSI standard).

_LANGUAGE_C

`cc218x` always defines `_LANGUAGE_C` as 1 when compiling C source.

__LINE__

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

__NO_BUILTIN

`cc218x` defines `__NO_BUILTIN` as `1` when you compile with the `-no-builtin` command-line switch.

__NO_LONG_LONG

`cc218x` defines `__NO_LONG_LONG` as `1` for C source. This definition signifies no support is present for the `long long int` type.

__SIGNED_CHARS__

`cc218x` defines `__SIGNED_CHARS__` as 1 unless you compile with the `-unsigned-char` command-line switch.

__STDC__

`cc218x` defines `__STDC__` as `1`.

__STDC_VERSION__

`cc218x` defines `__STDC__` as `199409L`.

__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).

__VERSION__

The preprocessor defines `__VERSION__` as a string constant giving the version number of the compiler used to compile this module.

Header Files

A header file contains C declarations and macro definitions. Use the `#include` C 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 parts of the operating system. Include these header files in your program for the definitions and declarations you need to access system calls and libraries. Use angle brackets to indicate a system header file. For example, `#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. For example, `#include "file"`.

Writing Preprocessor Macros

A macro is a name standing for a block of text that the preprocessor substitutes. Use the `#define` C 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, define a macro that expands into a compound statement. You can define such a macro to invoke it the same way you would call a function, making your source code easier to read and maintain.

The following two code segments define two versions of the macro `SKIP_SPACES`.

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

Enclosing the first definition within the `do {…} while (0)` pair changes the macro from expanding into a compound statement to expanding into a single statement. With the macro expansion into a compound statement, you must sometimes omit the semicolon after the macro call in order to have a valid 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.

With the `do {…} while (0)` construct, you can treat the macro as a function and 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
```

This is not legal C syntax. For more information on macros, see the *VisualDSP++ 3.5 Assembler and Preprocessor Manual for ADSP-218x and ADSP-219x DSPs.*

Preprocessing of .IDL Files

Every VisualDSP++ Interface Definition Language (VIDL) specification is analyzed by the C language preprocessor prior to syntax analysis.

The `#include` directive is used to control the inclusion of additional VIDL source text from a secondary input file that is named in the directive. Two available forms of `#include` are shown in Figure 1-2.

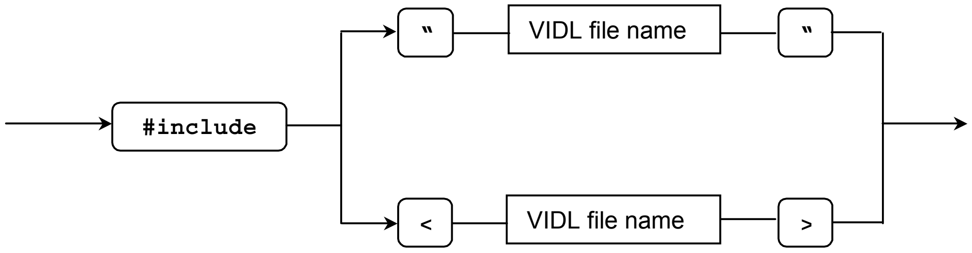


Figure 1-2. #INCLUDE Syntax Diagram

The file identified by the file name is located by searching a list of directories. When the name is delimited by quote characters, the search begins in the directory containing the primary input file, then proceeds with the list of directories specified by the `-I` command-line switch. When the name is delimited by angle-bracket characters, the search proceeds directly with the directories specified by `-I`. If the file is not located within any directory on the search list, the search may be continued in one or more platform dependent system directories.

For more information, refer to the *VisualDSP++ Component Software Engineering User's Guide*.

C Run-Time Model and Environment

This section provides a full description of the ADSP-218x DSP run-time model and run-time environment. The run-time model, which applies to compiler-generated code, includes descriptions of the layout of the stack, data access, and call/entry sequence. The C run-time environment includes the conventions that C routines must follow to run on ADSP-218x DSPs. Assembly routines linked to C routines must follow these conventions.

ADI recommends that assembly programmers maintain stack conventions.

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

This section contains:

Using the Run-Time Header

The run-time header is an assembly language procedure that initializes the processor and sets up processor features to support the C run-time environment. The default run-time header source code for the ADSP-218x is in the `218x_hdr.asm` file. This run-time header performs the following operations:

- Initializes the C run-time environment
- Calls your `main` routine
- Calls `exit` routine, defined in the C run-time library (`libc.dlb`), if `main` returns.
- Defines system halt instruction called from `exit` routine.

Interrupt Table and Interface

The interrupt table is an assembly language set of functions defined in named sections. These get placed appropriately in the Linker Description File (`.LDF`) to be executed at interrupt vector addresses. The default code for the ADSP-218x DSP interrupt table is defined in `218x_int_tab.asm`.

The default interrupt table uses the following external symbols,

`_lib_int_table`	Static table holding interrupt information defined in the C run-time library
`__lib_int_determiner`	An interrupt dispatcher defined in the C run-time library
`_____system_start`	C run-time initialization defined in the run-time header

The `218x_int_tab` file contains a section of code for each hardware interrupt. The `.LDF` file places these code sections in the correct interrupt vector slots for each interrupt.

If an interrupt occurs, program execution begins at the interrupt vector addresses. Program execution causes a jump to `__lib_int_determiner` in the default vector code. If `__lib_int_determiner` finds (by inspecting `__lib_int_table`) a handler set for the interrupt, it will call the handler. `__lib_int_determiner` saves and restores all scratch registers around the handler call. The function `__lib_int_determiner` terminates by executing a return from interrupt (RTI) instruction, which restores program execution to the point at which the interrupt was raised.

A handler for an interrupt or signal is set using the `interrupt` or `signal` C run-time library functions. These functions pass the signal name and a handler function pointer as parameters. The signal macro names are defined in `signal.h`.

The default interrupt vector code may be replaced with custom code by modifying or creating a new piece of code to be placed at the vector addresses. This is usually done by copying the default `218x_int_tab.asm` file and `.LDF` file into your project and modifying them as required.

An `interrupt` pragma defined function can be placed in the interrupt vector code directly or be jumped to from the vector if it does not fit in the interrupt vector space (see "Interrupt Handler Pragmas" on page 1-102).

The reset vector code, which is placed at address zero (0) and does a jump to `_____system_start`, should not be replaced.

Autobuffering Support

The `cc218x` compiler allows registers `I2`, `I3`, `I5`, `I7` and `M0` to be reserved from compiler use so that serial ports (SPORTS) can be configured to use autobuffering. Registers are reserved using the `-reserve` switch (on page 1-38). This support makes it possible to to enable both SPORT for transmit and receive autobuffering.

In addition to the compiler not using any reserved registers, the run-time libraries must do likewise. The libraries normally avoid the use of registers `I2`, `I3` and `M0` by default. Making the library code avoid registers `I5` and `I7` can add an extra complexity and in some cases introduces a performance penalty. The aim is to avoid this penalty for applications that do not require autobuffering or these extra registers.

To this end, autobuffering variants of the libraries are provided (`libcab.dlb` and l`ibioab.dlb)` which avoid `I5` and `I7` and will be used in place of the defaults. These libraries will be added to the link line by the default .`LDF` files when macro `__RESERVE_AUTOBUFFER_REGS__` is defined. The compiler driver will define this macro in compile, assemble and link phases when the compilers `-reserve` switch is used.

The compiler will also plant a common symbol, `___reserved_bitmask`, into modules compiled with the `-reserve` switch. This symbol can be used to determine which registers are reserved accross an application. This is sometimes required when saving and restoring registers in an interrupt service routine.

Bits, set in `___reserved_bitmask`, correspond to the following reserved registers:

```
bit 0 set = I2 reserved
bit 1 set = I3 reserved
bit 2 set = I5 reserved
bit 3 set = I7 reserved
bit 4 set = M0 reserved
```

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 is not a leaf procedure (a procedure calling 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.

Figure 1-3 provides a general overview of the ADSP-218x DSP stack.

ⓘ The stack grows downward on the page. Because the stacks grow towards smaller addresses, higher addresses are found in the upwards direction.

The stack resides in primary data memory (`DM`). It 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 C compiler environment defines register `I4` as the `SP` (stack pointer) and `M4` as the `FP` (frame pointer).

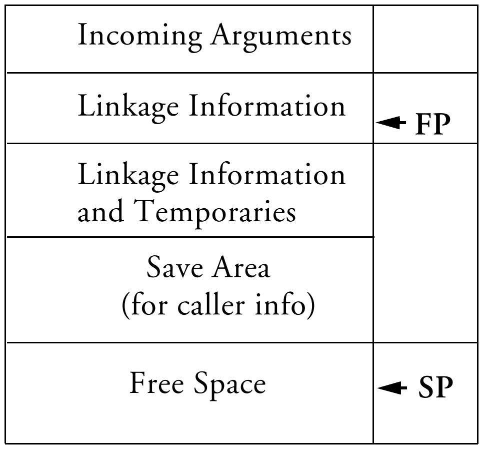

Incoming Arguments
Linkage Information
FP
Linkage Information and Temporaries
Save Area (for caller info)
Free Space
SP

Figure 1-3. ADSP-218x DSP Stack

Stack Frame Description

This section describes each part of the stack frame as shown in Figure 1-3.

Incoming Arguments

The memory area for incoming arguments begins at the `FP` value `+1`. Argument words are mapped by ascending addresses, so the second argument word is mapped at `FP+2`.

Linkage Information

The return address is saved on the hardware stack by the `CALL` instruction. In the called function, the address can then be popped from the hardware stack and saved as part of the stack frame. This information is used by the debugger to generate call stack debug information for source level debugging. Saving the return address on the stack frame is also useful in avoiding overflowing the finite hardware call stack; for example, when using recursion. The value stored on the stack gets pushed back on the hardware call stack before the function returns. The compiler detects recursive routines.

Local Variables and Temporaries

Space for a register save area and local variables/temporaries is allocated on the stack by the function prologue. Local variables and temporaries are typically placed first in this area, so they can be addressed with the smallest offsets from `FP`. The register save area is located at the farthest end of this area and can be accessed by `SP`-relative addressing.

Outgoing Arguments

Outgoing arguments are located at the top of the stack prior to the call. Space may be pre-allocated or claimed at the time of each call.

Free Space

Space below `SP` 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 on the ADSP-218x DSPs.

General System-Wide Specifications

The following list contains some general specifications that apply to the stacks.

- The stacks grow down in memory from higher to lower addresses.
- The current frame's "base" is addressed by the `FP` register.
- The first free 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 like 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+1.

Data can be pushed onto the stack by executing an instruction like the following one for the ADSP-218x DSPs:

```
DM (I4 += M7) = rej.
```

- The return address of the caller is stored at offset -1 from the address carried by the current FP if it is stored on the stack.
- The linkage back to the previous stack frame is stored at offset 0 from the current FP.

At a procedure call, the following must be true:

- There must be at least one free slot on the PC stack to hold the return address.

At an interrupt, the following must be true:

- Space beyond the SP must be available.

Return Values

Return values always use registers. Single-word return values come back in register AX1 (for OldAsmCall, register AR is used). Double-word return values are stored in SR1:0, with the most significant part in SR1.

If the return value is larger than two words, then the caller must allocate space and pass the address as a "hidden argument". The register I0 is used for this purpose.

Procedure Call and Return

To call a procedure:

1. Evaluate the arguments and push them onto the stack.
2. Call the procedure.
3. On return, remove arguments if desired.

On Entry:

1. Save the old `FP`, then set `FP` to the current `SP`.
2. If debugging, pop the PC stack and store it on the main stack.
3. Move the `SP` to create space for the new frame.
4. If `-g` is specified, push the return address back onto the hardware stack.

   After this step, the new frame is officially in place.
5. Continue saving registers, and then execute the procedure.

A leaf procedure, which does not require much stack space, might choose to omit steps (1) and (2), operating without its own stack frame.

To Return from a Procedure:

1. Restore miscellaneous saved registers.
2. Place the return value in the correct register (if not there already).
3. Restore `FP` for the previous frame.
4. Reset `SP` to remove the frame for procedure.
5. Return to the caller.

File I/O Support

The VisualDSP++ environment provides access to files on a host system, using `stdio` functions. File I/O support is provided through a set of low-level primitives that implement the `open`, `close`, `read`, `write`, and `seek` operations.The `stdio` functions make use of these primitives to provide conventional C input and output facilities. The source files for the I/O primitives are available under the VisualDSP++ installation in the subdirectory `218x\lib\src\libio_src`.

Refer to "stdio.h" on page 3-11 for more information.

Extending I/O Support To New Devices

The I/O primitives are implemented using an extensible device driver mechanism. The default start-up code includes a device driver that can perform I/O through the VisualDSP++ simulator and EZ-KIT Lite. Other device drivers may be registered and then used through the normal `stdio` functions.

A device driver is a set of primitive functions, grouped together into a `DevEntry` structure. This structure is defined in `device.h`:

```
struct DevEntry {
   int   DeviceID;
   void  *data;

   int   (*init)(struct DevEntry *entry);
   int   (*open)(const char *name, int mode);
   int   (*close)(int fd);
   int   (*write)(int fd, unsigned char *buf, int size);
   int   (*read)(int fd, unsigned char *buf, int size);
   long  (*seek)(long fd, long offset, int whence);
}

typedef struct DevEntry DevEntry;
typedef struct DevEntry *DevEntry_t;
```

The `DeviceID` field is a unique identifier for the device, known to the user. Device IDs are used globally across an application. The data field is a pointer for any private data the device may need; it is not used by the run-time libraries. The function pointed to by the `init` field is invoked by the run-time library when the device is first registered. It returns a negative value for failure, positive value for success.

The functions pointed to by the `open`, `close`, `write` and `read` fields are the functions that provide the same functionality used in the default I/O device. `Seek` is another function at the same level, for those devices which support such functionality. If a device does not support an operation (such as seeking, writing on read-only devices or reading write-only devices), then a function pointer must still be provided; the function must arrange to always return failure codes when the operation is attempted.

A new device can be registered with the following call:

```
int add_devtab_entry(DevEntry_t entry);
```

If the device is successfully registered, the `init()` routine of the device is called, with `entry` as its parameter. `add_devtab_entry()` returns the DeviceID of the device registered.

If the device is not successfully registered, a negative value is returned. Reasons for failure include, but are not limited to:

- The DeviceID is the same as another device, already registered
- There are no more spaces left in the device registry table
- The DeviceID is less than zero
- Some of the function pointers are NULL
- The device's `init()` routine returned a failure result

Once a device is registered, it can be made the default device, using the following function:

```
void set_default_io_device(int);
```

The user passes the `DeviceID`. There is a corresponding function for retrieving the current default device:

```
int get_default_io_device(void);
```

The default device is used by `fopen()` when a file is first opened. The `fopen()` function passes the open request to the `open()` function of the device indicated by `get_default_io_device()`. The device file identifier (`dfid`) returned by the `open()` function is private to the device; other devices may simultaneously have other files open which use the same identifier. An open file is uniquely identified by the combination of `DeviceID` and `dfid`.

The `fopen()` function records the `DeviceID` and `dfid` in the global open file table, and allocates its own internal `fid` to this combination. All future operations on the file reads, writes, seeks and close —use this `fid` to retrieve the `DeviceID`—and thus direct the request to the appropriate device's primitive functions, passing the `dfid` along with other parameters. Once a file has been opened by `fopen()`, the current value of `get_default_io_device()` is irrelevant to that file.

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. Prototypes are always good programming practice. Programs that call library subroutines should always include header files.

- There is no special interface for calling system library functions. They use the standard calling convention.

Register Classification

This section describes the ADSP-218x registers. Registers are listed in order of preferred allocation by the compiler.

Callee Preserved Registers (“Preserved”)

Registers `I2`, `I3`, `I5`, `I7`, and `M0` are *preserved*. A subroutine which uses any of these registers must save (preserve) and restore it.

Dedicated Registers

Certain registers have dedicated purposes and are not used for other things. Compiled code and libraries expect the dedicated registers to be correct.

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.

Circular Buffer Length Registers

Registers `L0` through `L7` are the circular buffer length registers. The compiler assumes that these registers contain zero, which disables circular buffering; they *must* be set to zero when compiled code is executing, to avoid incorrect behavior. There is no restriction on the value of an `L` register when the corresponding `I` register has been reserved from compiler use.

See “-reserve register[,register...]” on page 1-38 for more information about reserving registers.

Mode Status (MSTAT) Register

The C runtime initializes the MSTAT register as part of the run-time header code. The compiler and run-time libraries assume to be running in these preset modes. If you change any of the modes listed in Table 1-9, ensure that they are reverted before calling C compiled functions or functions from the C run-time library. Failure to revert to the default modes may cause applications to fail when running.

Table 1-9. MSTAT Register Modes

Mode	Description	State
SEC_REG	Secondary Data Registers	disabled
BIT_REV	Bit-reversed address output	disabled
AR_SAT	ALU saturation mode	disabled
M_MODE	MAC result mode	Integer Mode, 16.0 format

Complete List of Registers

The following tables describe all of the registers for the ADSP-218x DSPs.

- Table 1-10 lists the data register's file registers
- Table 1-11 lists the DAG1 registers
- Table 1-12 lists the DAG2 registers

Table 1-10. Data Register File Registers

Register	Descriptuon	Notes
AX0	scratch	
AX1	scratch; single-word return	
AY0	scratch	

Table 1-10. Data Register File Registers (Cont'd)

Register	Descriptuon	Notes
AY1	scratch	Argument 2 for compatibility call
AR	scratch;	Argument 1 for compatibility call
AF	scratch	
MX0	scratch	
MX1	scratch	
MY0	scratch	
MY1	scratch	
MR1:0	scratch	
MR2	scratch	
MF	scratch	
SB	scratch	
SE	scratch	
SI	scratch	
SR1:0	scratch; double-word return	

Table 1-11. DAG1 Registers

Register	Descriptuon
I0	scratch
I1	scratch
I2	preserved
I3	preserved

Table 1-11. DAG1 Registers

Register	Descriptuon
M0	preserved
M1	dedicated: +1
M2	dedicated: 0
M3	scratch
L0-3	not used, must be zero

Table 1-12. List of DAG2 Registers

Register	Descriptuon
I4	dedicated: SP
I5	preserved
I6	scratch
I7	preserved
M4	dedicated: FP
M5	scratch
M6	dedicated: 0
M7	dedicated: -1
L4-7	not used, must be zero

C and Assembly Language Interface

This section describes how to call assembly language subroutines from within C programs, and how to call C functions from within assembly language programs. Before attempting to do either of these, be sure to familiarize yourself with the information about the C run-time model (including details about the stack, data types, and how arguments are handled) in "C Run-Time Model and Environment" on page 1-132.

This section contains:

Calling Assembly Subroutines from C Programs

Before calling an assembly language subroutine from a C program, create a prototype to define the arguments for the assembly language subroutine and the interface from the C program to the assembly language subroutine. Even though it is legal to use a function without a prototype in C, prototypes are a strongly recommended practice for good software engineering. When the prototype is omitted, the compiler cannot perform argument type checking and assumes that the return value is of type integer and uses K&R promotion rules instead of ANSI promotion rules.

The run-time model defines some registers as *scratch* registers and others as *preserved* or *dedicated* registers. Scratch registers can be used within the assembly language program without worrying about their previous contents. If more room is needed (or an existing code is used) and you wish to use the preserved registers, you *must save* their contents and then *restore* those contents before returning.

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

- Familiarize yourself with the general features of the 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.
- Create an interface definition, or "prototype", so that the 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. When using the function from assembly programs, you might want your function's name to be just as you write it. Then you will also need to tell the C compiler that it is an `asm` function, by placing `'extern "asm" {}'` around the prototype.
- The C run time determines that all function parameters are passed on the stack. A good way to observe and understand how arguments are passed is to write a dummy function in C and compile it using the `-save-temps` command-line switch (on page 1-39). The resulting compiler generated assembly file (`.s`) can then be viewed.

The following example includes the global volatile variable assignments to indicate where the arguments can be found upon entry to `asmfunc`.

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

int global_a;
float global_b;
int *  global_p;

// the function itself

int asmfunc(int a, float b, int * p, int d, int e) {
// do some assignments so that .s file will show where args are
   global_a = a;
   global_b = b;
   global_p = p;
   //value gets loaded into the return register
   return 12345;
}
```

When compiled with the `-save-temps` option set (see on page 1-39), this produces the following:

```
// PROCEDURE: _asmfunc
   .global _asmfunc;
_asmfunc:
      SI = DM(I4 +  4);
      I0 = SI ;
      AX1 = DM(I4 +  2);
      SI = DM(I4 +  1);
      AX0 = DM(I4 +  3);
      DM(_global_b) = AX1;
      DM(_global_a) = SI;
      DM(_global_b+1) = AX0;
      RTS (DB);
      AX1 = 12345;
      DM(_global_p) = I0;
_asmfunc.end
```

For a more complicated function, you might find it useful to follow the general run-time model, and use the run-time stack for local storage, etc. A simple C program, passed through the compiler, will provide a good template to build on. Alternatively, you may find it just as convenient to use local static storage for temporaries.

Calling C Routines from Assemby Programs

You may want to call a C-callable library and other functions from within an assembly language program. As discussed in "Calling Assembly Subroutines from C Programs" on page 1-148, you may want to create a test function to do this in C, and then use the code generated by the compiler as a reference when creating your assembly language program and the argument setup. Using volatile global variables may help clarify the essential code in your test function.

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

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

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

If arguments are on the stack, they are addressed via an offset from the stack pointer or frame pointer. Explore how arguments are passed between an assembly language program and a function by writing a dummy function in C and compiling it with the `save temporary files` option (the `-save-temps` switch on page 1-39). By examining the contents of volatile global variables in `*.s` file, you can determine how the C function passes arguments, and then duplicate that argument setup process in the assembly language program.

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

Using Mixed C/Assembly Support Macros

This section describes the C/Assembly interface support macros available via the `asm_sprt.h` system header file. Use these macros for interfacing assembly language modules with C functions.

Your software package includes a version of the `asm_sprt.h` file. Table 1-13 lists and the following section describes the macros.

Table 1-13. Interface Support Macros

`function_entry`	`exit`	`leaf_entry`	`leaf_exit`
`alter(x)`			
`save_reg`	`restore_reg`	`readsfirst(x)`	`readsnext`
`putsfirst`	`putsnext`	`getsfirst(x)`	`getsnext`

function_entry

The `function_entry` macro expands into the function prologue for non-leaf functions. This macro should be the first line of any non-leaf assembly routine.

exit

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

leaf_entry

The `leaf_entry` macro expands into the function prologue for leaf functions. This macro should be the first line of any leaf assembly routine.

ⓘ This macro is currently null, but should be used for future compatibility.

leaf_exit

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

alter(x)

The `alter` macro expands into an instruction that adjusts the stack pointer by adding the immediate value `x`. With a positive value for `x`, alter pops `x` words from the top of the stack. You could use alter to clear `x` number of parameters off the stack after a call.

save_reg

The preprocessor expands the `save_reg` macro into a series of assembly language commands that push the following registers (on the ADSP-218x architecture) onto the C run-time stack:

```
AY0, AX0, AX1, MY0, MX0, MX1, MR1, MR0, SR1, SR0, I0, I1, M0, M3, I5
```

restore_reg

The `restore_reg` macro expands into a series of instructions that pop the stored registers off of the C run-time stack.

readsfirst(register)

The preprocessor expands the `readsfirst` macro into a series of assembly language commands that read the value off the top of the stack, write the value to `register`, and set up for a read of the next stack entry with the `readsnext` macro. The `readsfirst` macro references the stack-pointer (`I4`) and might be used to read values that were placed on the stack using the `putsfirst` and `putsnext` macros.

register = readsnext

The preprocessor expands the `readsnext` macro into a series of assembly language commands. These commands continue the read process set up by the `readsfirst` macro by reading the next value off the top of the stack and writing it to `register`.

putsfirst = register

The preprocessor expands the `putsfirst` macro into a series of assembly language commands. These commands write the contents of `register` to the top of the stack and set up for a write of the next stack entry with the `putsnext` macro.

putsnext = register

The preprocessor expands the putsnext macro into a series of assembly language commands. These commands continue the write process set up by the putsfirst macro by writing the next register to the top of the stack.

getsfirst(register)

The preprocessor expands the getsfirst macro into a series of assembly language commands that read the value off the top of the stack, write the value to register, and set up for a read of the next stack entry with the getsnext macro. The preprocessor expands the getsfirst macro into a series of assembly language instructions that read a value from the top of a function frame, write the value to register and set up a read of the next value with getsnext. The getsfirst macro references the frame-pointer (M4) and would be used to read function parameters.

register = getsnext

The preprocessor expands the getsnext macro into a series of assembly language commands. These commands continue the read process set up by the getsfirst macro by reading the next value off the top of the stack and writing it to *register*.

Using Mixed C/Assembly Naming Conventions

It is necessary to be able to use C symbols (function or variable names) in assembly routines and use assembly symbols in C routines. This section describes how to name C and assembly symbols and how to use C and assembly symbols.

To name an assembly symbol that corresponds to a C symbol, add an underscore prefix to the C symbol name when declaring the symbol in assembly. For example, the C symbol main becomes the assembly symbol _main.

To use a C function or variable in your assembly routine, declare it as global in the C program and import the symbol into the assembly routine by declaring the symbol with the `.EXTERN` assembler directive.

To use an assembly function or variable in your C program, declare the symbol with the `.GLOBAL` assembler directive in the assembly routine and import the symbol by declaring the symbol as `extern` in the C program.

Alternatively, the `cc218x` compiler provides an "`asm`" linkage specifier (used similarly to the "C" linkage specifier of C++), which when used, removes the need to add an underscore prefix to the symbol that is defined in assembly.

Table 1-14 shows the C/Assembly interface naming conventions.

Table 1-14. Naming Conventions for Symbols

In The C Program	In The Assembly Subroutine

`int c_var;`
`/* declared global */`
	`.extern _c_var;`

`void c_func();`	`.extern _c_func;`
`extern int asm_var;`	`.global _asm_var;`

`extern void asm_func();`	`.global _asm_func;`
	`_asm_func:`

`extern "asm" void asm_func();`	`.global asm_func;`
	`asm_func:`

Compatibility Call

The `cc218x` compiler in VisualDSP++ 3.5 produces code that is not fully compatible with the Release 6.1 run-time model. However, the new compiler is superior in many ways to the old one, and your programs will be faster, smaller, and more reliable after the C code is converted to the new system.

The `cc218x` compiler provides a compatibility call to enable usage of existing libraries and special-purpose assembly language subroutines with the new compiler. This feature is available with a small amount of source code modification by adding an `'extern "OldAsmCall"` ' specification to the prototype in the source program, similar to what is done when calling between C and C++ source programs. There is no compiler option for compatibility calls.

This feature provides full compatibility with the following restrictions:

- You cannot mix old and new compiled modules
- Old code is not allowed to call into a new compiled module
- A procedure pointer from a new compiled module is not allowed as an argument to an old routine

Some programs may not have any declarations of external assembly language functions. This is not good programming practice and should be fixed.

The effect of the `OldAsmCall` declaration is as follows:

- Pass the first two arguments in registers `AR` and `AY1`.

The C run-time stack for compatibility calls is normally used to pass the third and subsequent parameters to a called function. This changes if either of the first two parameters is a multi-word parameter, in which case, it and all subsequent parameters are passed on the stack. Functions that take variable arguments (varargs functions) will have the last named parameter and subsequent parameters passed on the stack.

- Postpend an underscore onto the external name.
- Look in the `AR` register (instead of `AX1`) for a one-word return value.

The `OldAsmCall` extern declaration can encompass one or more prototypes that define external entry points, as shown in the following example.

```
extern "OldAsmCall" {
int libfn(int flag, int * a);
void resetmach(int idle);
}
```

2 ACHIEVING OPTIMAL PERFORMANCE FROM C SOURCE CODE

This chapter provides guidance to help you to tune your application to achieve the best possible code from the compiler. Some implementation choices are available when coding an algorithm, and understanding their impact is crucial to attaining optimal performance.

The focus of what follows is on how to obtain maximal code performance from the compiler. Most of these guidelines also apply when optimizing for minimum code size, although some techniques specific to that goal are also discussed. The first section looks at some general principles, and how the compiler can lend the most help to your optimization effort. Optimal coding styles are then considered in detail. Special features such as compiler switches, built-in functions, and pragmas are also discussed. The chapter ends with a short example to demonstrate how the optimizer works.

Small examples are included throughout this chapter to demonstrate points being made. Some show recommended coding styles, others identify styles to be avoided or code that it may be possible to improve. These are marked as "**Good**" and "**Bad**", respectively.

This chapter contains:

- “Pragmas” on page 2-29
- “Useful Optimization Switches” on page 2-35

General Guidelines

It is useful to bear in mind the following basic strategy when writing an application:

1. Try to choose an algorithm that is suited to the architecture being targeted. For example, there may be a trade-off between memory usage and algorithm complexity that may be influenced by the target architecture.

2. Code the algorithm in a simple, high-level generic form. Keep the target in mind, especially with respect to choices of data types.

3. You can then turn your attention towards code tuning. For critical code sections, think more carefully about the strengths of the target platform, and make any non-portable changes where necessary.

Tip: Choose the language as appropriate.

This section contains:

How the Compiler Can Help

The compiler provides many facilities designed to help the programmer.

Using the Compiler Optimizer

There is a vast difference in performance between code compiled optimized and code compiled non-optimized. In some cases optimized code can run ten or twenty times faster. Optimization should always be used when measuring performance or shipping code as product.

The optimizer in the C compiler is designed to generate efficient code from source that has been written in a straightforward manner. The basic strategy for tuning a program is to present the algorithm in a way that gives the optimizer excellent visibility of the operations and data, and hence the greatest freedom to safely manipulate the code. Future releases of the compiler will continue to enhance the optimizer, and expressing algorithms simply will provide the best chance of benefiting from such enhancements.

Note that the default setting (or "debug" mode within the VisualDSP++ IDDE) is for non-optimized compilation in order to assist programmers in diagnosing problems with their initial coding. The optimizer is enabled in VisualDSP++ by checking the **Enable optimization** check-box under the **Project Options ->Compile** tab. This adds the `-O` (enable optimization) switch (on page 1-32) to the compiler invocation. A "release" build from within VisualDSP++ will automatically enable optimization.

Using the Statistical Profiler

Tuning an application begins with an understanding of which areas of the application are most frequently executed and therefore where improvements would provide the largest gains. The statistical profiling feature provided in VisualDSP++ is an excellent means for finding these areas. More details about how to use it may be found in the *VisualDSP++ 3.5 User's Guide*.

The particular advantage of statistical profiling is that it is completely unobtrusive. Other forms of profiling insert instrumentation into the code, perturbing the original optimization, code size and register allocation to some degree.

The best methodology is usually to compile with both optimization and debug information generation enabled. In this way, you can obtain a profile of the optimized code while retaining function names and line number information. This will give you accurate results that correspond directly to the C source. Note that the compiler optimizer may have moved code between lines.

You can obtain a more accurate view of your application if you build it optimized but without debug information generation. You will then obtain statistics that relate directly to the assembly code. The only problem with doing this may be in relating assembly lines to the original source. Do not strip out function names when linking, since keeping function names means you can scroll through the assembly window to instructions of interest.

In very complicated code, you can locate the exact source lines by counting the loops, unless they are unrolled. Looking at the line numbers in the assembly file (use the `-save-temps` switch (on page 1-39) to retain compiler generated assembly files, which will have the `.s` filename extension) may also help. The compiler optimizer may have moved code around so that it does not appear in the same order as in your original source.

Using Interprocedural Optimization

To obtain the best performance, the optimizer often requires information that can only be determined by looking outside the function that it is working on. For example, it helps to know what data can be referenced by pointer parameters, or whether a variable actually has a constant value. The `-ipa` compiler switch (on page 1-26) enables interprocedural analysis

(IPA), which can make this available. When this switch is used the compiler will be called again from the link phase to recompile the program using additional information obtained during previous compilations.

Because it only operates at link time, the effects of IPA will not be seen if you compile with the `-S` switch (on page 1-38). To see the assembly file when IPA is enabled, use the `-save-temps` switch (on page 1-39), and look at the `.s` file produced after your program has been built.

As an alternative to IPA, you can achieve many of the same benefits by adding pragma directives and other declarations such as `__builtin_aligned` to provide information to the compiler about how each function interacts with the rest of the program.

These directives are further described "Using Built-in Functions in Code Optimization" on page 2-24 and "Pragmas" on page 2-29.

Data Types

The compiler directly supports the following scalar data types.

Single-Word Fixed-Point Data Types: Native Arithmetic	
`char`	16-bit signed integer
`unsigned char`	16-bit unsigned integer
`short`	16-bit signed integer
`unsigned short`	16-bit unsigned integer
`int`	16-bit signed integer
`unsigned int`	16-bit unsigned integer
Double-Word Fixed-Point Data Types: Emulated Arithmetic	
`long`	32-bit signed integer
`unsigned long`	32-bit unsigned integer
Floating-Point Data Types: Emulated Arithmetic	
`double`	32-bit float
`float`	32-bit float

Fractional data types are represented using the integer types. Manipulation of these is best done by use of built-in functions, described in "System Support Built-in Functions" on page 2-25.

Avoiding Emulated Arithmetic

Arithmetic operations for some types are implemented by library functions because the DSP hardware does not directly support these types. Consequently, operations for these types are far slower than native operations-sometimes by a factor of a hundred-and also produce larger code. These types are marked as "Emulated Arithmetic" in "Data Types" on page 2-7.

The hardware does not provide direct support for division, so division and modulus operations are almost always multi-cycle operations, even on integral type inputs. If the compiler has to issue a full division operation, it will usually need to generate a call to a library function. One notable situation in which a library call is avoided is for integer division when the divisor is a compile-time constant and is a power of two. In that case the compiler generates a shift instruction. Even then, a few fix-up instructions are needed after the shift if the types are signed. If you have a signed division by a power of two, consider whether you can change it to unsigned in order to obtain a single-instruction operation.

When the compiler has to generate a call to a library function for one of these arithmetic operators that are not supported by the hardware, performance will suffer not only because the operation will take multiple cycles, but also because the effectiveness of the compiler optimizer will be reduced.

For example, such an operation in a loop can prevent the compiler from making use of efficient zero-overhead hardware loop instructions. Also, calling the library to perform the required operation can change values held in scratch registers before the call, so the compiler will have to generate more stores and loads from the data stack to keep values required after the call returns. Emulated arithmetic operators should therefore be avoided where possible, especially in loops.

Getting the Most from IPA

Interprocedural analysis (IPA) is designed to try to propagate information about the program to parts of the optimizer that can use it. This section looks at what information is useful, and how to structure your code to make this information easily accessible to the analysis.

Initializing Constants Statically

IPA will identify variables that have only one value and replace them with constants, resulting in a host of benefits for the optimizer's analysis. For this to happen a variable must have a single value throughout the program. If the variable is statically initialized to zero, as all global variables are by default, and is subsequently assigned some other value at another point in the program, then the analysis sees two values and will not consider the variable to have a constant value.

For example,

```
#include <stdio.h>
int val;           // initialized to zero

void init() {
   val = 3;        // re-assigned
}

void func() {
   printf("val %d",val);
}

int main() {
   init();
   func();
}
```

Bad: IPA cannot see that `val` is a constant is better written as

```
#include <stdio.h>
const int val = 3;    // initialized once

void init() {
}

void func() {
   printf("val %d",val);
}

int main() {
   init();
   func();
}
```

Good: IPA knows `val` is 3.

Avoiding Aliases

It may seem that the iterations may be performed in any order in the following loop:

```
void fn(char a[], char b[], int n) {
int i;
for (i = 0; i < n; ++i)
   a[i] = b[i];
}
```

Bad: `a` and `b` may alias each other.

but `a` and `b` are both parameters, and, although they are declared with `[]`, they are in fact pointers, which may point to the same array. When the same data may be reachable through two pointers, they are said to alias each other.

If IPA is enabled, the compiler will look at the call sites of `fn` and try to determine whether `a` and `b` can ever point to the same array.

Even with IPA, it is quite easy to create what appear to the compiler as aliases. The analysis works by associating pointers with sets of variables that they may refer to at some point in the program. If the sets for two pointers are found to intersect, then both pointers are assumed to point to the union of the two sets.

If `fn` above were called in two places with global arrays as arguments, then IPA would have the results shown below:

```
fn(glob1, glob2, N);
fn(glob1, glob2, N);
```

Good: sets for `a` and `b` do not intersect: `a` and `b` are not aliases.

```
fn(glob1, glob2, N);
fn(glob3, glob4, N);
```

Good: sets for `a` and `b` do not intersect: `a` and `b` are not aliases.

```
fn(glob1, glob2, N);
fn(glob3, glob1, N);
```

Bad: sets intersect - both `a` and `b` may access `glob1`; `a` and `b` may be aliases.

The third case arises because IPA considers the union of all calls at once, rather than considering each call individually, when determining whether there is a risk of aliasing. If each call were considered individually, IPA would have to take flow control into account and the number of permutations would make compilation time impracticably long.

The lack of control flow analysis can also create problems when a single pointer is used in multiple contexts. For example, it is better to write

```
int *p = a;
int *q = b;
   // some use of p
   // some use of q
```

Good: p and q do not alias.

than

```
int *p = a;
   // some use of p
p = b;
   // some use of p
```

Bad: uses of p in different contexts may alias.

because the latter may cause extra apparent aliases between the two uses.

Indexed Arrays vs. Pointers

C allows a program to access data from an array in two ways: either by indexing from an invariant base pointer, or by incrementing a pointer. These two versions of vector addition illustrate the two styles:

Style 1: using indexed arrays

```
void va_ind(const short a[], const short b[], short out[], int n) {
   int i;
   for (i = 0; i < n; ++i)
      out[i] = a[i] + b[i];
}
```

Style 2: using pointers

```
void va_ptr(const short a[], const short b[], short out[], int n) {
   int i;
   short *pout = out;
   const short *pa = a, *pb = b;
   for (i = 0; i < n; ++i)
      *pout++ = *pa++ + *pb++;
}
```

Trying Pointer and Indexed Styles

One might hope that the chosen style would not make any difference to the generated code, but this is not always the case. Sometimes, one version of an algorithm will generate better optimized code than the other, but it is not always the same style that is better.

Tip: Try both pointer and index styles.

The pointer style introduces additional variables that compete with the surrounding code for resources during the compiler optimizer's analysis. Array accesses, on the other hand, must be transformed to pointers by the compiler and sometimes it does not do the job as well as you could do by hand.

The best strategy is to start with array notation. If the generated code looks unsatisfactory, try using pointers. Outside the critical loops, use the indexed style, since it is easier to understand.

Function Inlining

The function inlining may be used in two ways

- By annotating functions in the source code with the inline keyword. In this case, function inlining is only performed when optimization is enabled.
- By turning on automatic inlining with the `-Oa` switch (on page 1-33). This switch automatically enables optimization.

Tip: Inline small, frequently executed functions.

You can use the compiler's `inline` keyword to indicate that functions should have code generated inline at the point of call. Doing this avoids various costs such as program flow latencies, function entry and exit instructions and parameter passing overheads. Using an inline function also has the advantage that the compiler can optimize through the inline code and does not have to assume that scratch registers and condition states are modified by the call. Prime candidates for inlining are small, frequently used functions because they will cause the least code-size increase while giving most performance benefit.

As an example of the usage of the `inline` keyword, the function below sums two input parameters and returns the result.

```
inline int add(int a, int b) {
   return (a+b);
}
```

Good: use of the `inline` keyword.

Inlining has a code-size to performance trade-off that should be considered when it is used. With `-Oa`, the compiler will automatically inline small functions where possible. If the application has a tight upper code-size limit, the resulting code-size expansion may be too great. It is worth considering using automatic inlining in conjunction with the `-Ov n` switch (on page 1-33) to restrict inlining (and other optimizations with a code-size cost) to parts of the application that are performance-critical. This will be considered in more detail later in this chapter.

Using Inline asm Statements

The compiler allows use of inline `asm` statements to insert small sections of assembly into C code.

Tip: Avoid use of inline `asm` statements where built-in functions may be used instead

The compiler does not intensively optimize code that contains inline `asm` statements because it has little understanding about what the code in the statement does. In particular, use of an `asm` statement in a loop may inhibit useful transformations.

The compiler has been enhanced with a large number of built-in functions. These generate specific hardware instructions and are designed to allow the programmer to more finely tune the code produced by the compiler, or to allow access to system support functions. A complete list of compiler's built-in functions is given in "Compiler Built-in Functions" on page 1-83.

Use of these builtins is much preferred to using inline `asm` statements. Since the compiler knows what each builtin does, it can easily optimize around them. Conversely, since the compiler does not parse asm statements, it does not know what they do, and so is hindered in optimizing code that uses them. Note also that errors in the text string of an `asm` statement will be caught by the assembler and not the compiler.

Examples of efficient use of built-in functions are given in "System Support Built-in Functions" on page 2-25.

Memory Usage

The compiler, in conjunction with the use of the linker description file (`.LDF`), allows the programmer control over where data is placed in memory. This section describes how to best lay out data for maximum performance.

Tip: Try to put arrays into different memory sections.

The DSP hardware can support two memory operations on a single instruction line, combined with a compute instruction. However, two memory operations will only complete in one cycle if the two addresses are situated in different memory blocks; if both access the same block, then a stall will be incurred.

Take as an example the dot product loop below. Because data is loaded from both array `a` and array `b` in every iteration of the loop, it may be useful to ensure that these arrays are located in different blocks.

```
for (i=0; i<100; i++)
   sum += a[i] * b[i];
```

Bad: compiler assumes that two memory accesses together may give a stall.

This is done by using the "Dual Memory Support Language Keywords" compiler extension. Placing a `pm` qualifier before the type definition tells the compiler that the array is located in "Program Memory" (PM). The memory may be accessed in parallel with the usual "Data Memory" (DM).

The array declaration of one of either `a` or b is modified to

```
pm int a[100];
```

and any pointers to the buffer a become, for example,

```
pm int *p = a;
```

to allow simultaneous accesses to the two buffers.

Note that the explicit placement of data in PM can only be done for global data.

Loop Guidelines

Loops are where an application will ordinarily spend the majority of its time. It is therefore useful to look in detail at how to help the compiler to produce the most efficient code possible for them.

Keeping Loops Short

For best code efficiency, loops should be as small as possible. Large loop bodies are usually more complex and difficult to optimize. Additionally, they may require register data to be stored in memory. This will cause a decrease in code density and execution performance.

Avoiding Unrolling Loops

Tip: Do not unroll loops yourself.

Not only does loop unrolling make the program harder to read but it also prevents optimization by complicating the code for the compiler.

```
void va1(const short a[], const short b[], short c[], int n)
{
   int i;
   for (i = 0; i < n; ++i) {
      c[i] = b[i] + a[i];
   }
}
```

Good: the compiler will unroll if it helps.

```
void va2(const short a[], const short b[], short c[], int n)
{
   short xa, xb, xc, ya, yb, yc;
   int i;
   for (i = 0; i < n; i+=2) {
      xb = b[i]; yb = b[i+1];
      xa = a[i]; ya = a[i+1];
```

```
            xc = xa + xb; yc = ya + yb;
            c[i] = xc; c[i+1] = yc;
        }
    }
```

Bad: harder for the compiler to optimize.

Avoiding Loop Rotation by Hand

Tip: Do not rotate loops by hand.

Programmers are often tempted to "rotate" loops in DSP code by "hand" attempting to execute loads and stores from earlier or future iterations at the same time as computation from the current iteration. This technique introduces loop-carried dependencies that prevent the compiler from rearranging the code effectively. However, it is better to give the compiler a "normalized" version, and leave the rotation to the compiler.

```
int ss(short *a, short *b, int n) {
   int sum = 0;
   int i;
   for (i = 0; i < n; i++) {
      sum += a[i] + b[i];
   }
   return sum;
}
```

Good: will be rotated by the compiler.

```
int ss(short *a, short *b, int n) {
   short ta, tb;
   int sum = 0;
   int i = 0;
   ta = a[i]; tb = b[i];
   for (i = 1; i < n; i++) {
      sum += ta + tb;
      ta = a[i]; tb = b[i];
   }
   sum += ta + tb;
```

```
    return sum;
}
```

Bad: rotated by hand—hard for the compiler to optimize.

By rotating the loop, the scalar variables `ta` and `tb` have been added, introducing loop-carried dependencies.

Avoiding Array Writes in Loops

Other dependencies can be caused by writes to array elements. In the following loop, the optimizer cannot determine whether the load from a reads a value defined on a previous iteration or one that will be overwritten in a subsequent iteration.

```
for (i = 0; i < n; ++i)
    a[i] = b[i] * a[c[i]];
```

Bad: has array dependency.

The optimizer can resolve access patterns where the addresses are expressions that vary by a fixed amount on each iteration. These are known as "induction variables".

```
for (i = 0; i < n; ++i)
    a[i+4] = b[i] * a[i];
```

Good: uses induction variables.

Inner Loops vs. Outer Loops

Tip: Inner loops should iterate more than outer loops.

The optimizer focuses on improving the performance of inner loops because this is where most programs spend the majority of their time. It is considered a good trade-off for an optimization to slow down the code before and after a loop if it is going to make the loop body run faster. Therefore, try to make sure that your algorithm also spends most of its time in the inner loop; otherwise it may actually be made to run slower by optimization. If you have nested loops where the outer loop runs many times and the inner loop a small number of times, it may be possible to rewrite the loops so that the outer loop has the fewer iterations.

Avoiding Conditional Code in Loops

If a loop contains conditional code, control-flow latencies may incur large penalties if the compiler has to generate conditional jumps within the loop. In some cases, the compiler will be able to convert `IF-THEN-ELSE` and `?:` constructs into conditional instructions. In other cases, it will be able to relocate the expression evaluation outside of the loop entirely. However, for important loops, linear code should be written where possible.

The compiler will not perform the loop transformation that interchanges conditional code and loop structures. Instead of writing

```
for (i=0; i<100; i++) {
   if (mult_by_b)
      sum1 += a[i] * b[i];
   else
      sum1 += a[i] * c[i];
}
```

Bad: loop contains conditional code.

it is better to write

```
if (mult_by_b) {
   for (i=0; i<100; i++)
      sum1 += a[i] * b[i];
} else {
   for (i=0; i<100; i++)
      sum1 += a[i] * c[i];
}
```

Good: two simple loops can be optimized well.

if this is an important loop.

Avoiding Placing Function Calls in Loops

The compiler will not usually be able to generate a hardware loop if the loop contains a function call due to the expense of saving and restoring the context of a hardware loop. In addition to obvious function calls, such as `printf()`, hardware loop generation can also be prevented by operations such as division, modulus, and some type coercions. These operations may require implicit calls to library functions. For more details, see "Data Types" on page 2-7.

Avoiding Non-Unit Strides

If you write a loop such as:

```
for (i=0; i<n; i+=3) {
        // some code
}
```

Bad: non-unit stride means division may be required.

then in order for the compiler to turn this into a hardware loop, it will need to work out the loop trip count. To do so, it must divide `n` by 3. The compiler will decide that this is worthwhile as it will speed up the loop, but as discussed above, division is an expensive operation. Try to avoid creating loop control variables with strides of non-unit magnitude.

Loop Control

Tip: Use `int` types for loop control variables and array indices.

Tip: Use automatic variables for loop control and loop exit test.

For loop control variables and array indices, it is always better to use signed `ints` rather than any other integral type. The C standard requires various type promotions and standard conversions that complicate the code for the compiler optimizer. Frequently, the compiler is still able to deal with such code and create hardware loops and pointer induction variables. However, it does make it more difficult for the compiler to optimize and may occasionally result in under-optimized code.

The same advice goes for using automatic (local) variables for loop control. It is easy for a compiler to see that an automatic scalar, whose address is not taken, may be held in a register during a loop. But it is not as easy when the variable is a global or a function static. Therefore, code such as

```
for (i=0; i<globvar; i++)
   a[i] = 10;
```

Bad: may need to reload `globvar` on every iteration.

may not create a hardware loop if the compiler cannot be sure that the write into the array `a` does not change the value of the global variable. The `globvar` must be re-loaded each time around the loop before performing the exit test.

In this circumstance, the programmer can make the compiler's job easier by writing

```
int upper_bound = globvar;
for (i=0; i<upper_bound; i++)
   a[i] = 10;
```

Good: easily becomes hardware loop.

Using the Restrict Qualifier

The `restrict` qualifier provides one way to help the compiler resolve pointer aliasing ambiguities. Accesses from distinct `restricted` pointers do not interfere with each other. The loads and stores in the following loop

```
for (i=0; i<100; i++)
   a[i] = b[i];
```

Bad: possible alias of arrays `a` and `b`.

may be disambiguated by writing

```
int * restrict p = a;
int * restrict q = b;
for (i=0; i<100; i++)
   *p++ = *q++;
```

Good: `restrict` qualifier tells compiler that memory accesses do not alias.

The `restrict` keyword is particularly useful on function parameters.

Using the Const Qualifier

By default, the compiler assumes that the data referenced by a pointer to `const` type will not change. Therefore, another way to tell the compiler that the two arrays a and b do not overlap is to use the `const` keyword.

```
void copy(short *a, const short *b) {
   int i;
   for (i=0; i<100; i++)
      a[i] = b[i];
}
```

Good: pointers disambiguated via `const` qualifier.

The above example will have a similar effect on the `no_alias` pragma (see in "#pragma no_alias" on page 2-34). In fact, the `const` implementation is better since it also allows the optimizer to use the fact that accesses via a and b are independent in other parts of the code, not just the inner loop.

In C, it is legal, though bad programming practice, to use casts to allow the data pointed to by pointers to `const` type to change. This should be avoided since, by default, the compiler will generate code that assumes `const` data does not change. If you have a program that modifies `const` data through a pointer, you can generate standard-conforming code by using the compile-time flag `-const-read-write`.

Using Built-in Functions in Code Optimization

Built-in functions, also known as compiler intrinsics, provide a method for the programmer to efficiently use low-level features of the DSP hardware while programming in C. Although this section does not cover all the built-in functions available (for more information, refer to "Compiler Built-in Functions" on page 1-83), it presents some code examples where implementation choices are available to the programmer.

Fractional Data

Fractional data, represented as an integral type, can be manipulated in two ways: one way is the use of long promoted shifts and multiply constructs, and the other is the use of compiler built-in functions. The built-in functions are recommended as they give you the most control over your data. Let's consider the fractional dot product algorithm. This may be written as:

```
long dot_product (short *a, short *b) {
   int i;
   long sum=0;
   for (i=0; i<100; i++) {
      /* this line is performance critical */
      sum += (((long)a[i]*b[i]) << 1);
   }
   return sum;
}
```

Bad: uses shifts to implement fractional multiplication.

This presents some problems to the optimizer. Normally, the code generated here would be a multiply, followed by a shift, followed by an accumulation. However, the DSP hardware has a fractional multiply/accumulate instruction that performs all these tasks in one cycle.

In the example code, the compiler recognizes this idiom and replaces the multiply followed by shift with a fractional multiply. In more complicated cases, where perhaps the multiply is further separated from the shift, the compiler may not detect the possibility of using a fractional multiply.

The recommended coding style is to use built-in functions. In the following example, a builtin is used to multiply fractional 16-bit data.

```
#include <math.h>
fract32 dot_product(fract16 *a, fract16 *b) {
   int i;
   fract32 sum=0;
   for (i=0; i<100; i++) {
      /* this line is performance critical */
      sum += __builtin_mult_fr1x32(a[i],b[i]);
   }
   return sum;
}
```

Good: uses builtins to implement fractional multiplication.

Note that the `fract16` and `fract32` types used in the example above are merely `typedefs` to C integer types used by convention in standard include files. The compiler does not have any in-built knowledge of these types and treats them exactly as the integer types that they are `typedefed` to.

System Support Built-in Functions

Built-in functions are also provided to perform low-level system management, in particular for the manipulation of system registers (defined in `sysreg.h`). It is usually better to use these built-in functions rather than inline `asm` statements. The built-in functions cause the compiler to generate efficient inline instructions and their use often results in better optimization of the surrounding code at the point where they are used. Using builtins will also usually result in improved code-readability. For more information on built-in functions supported by the compiler, refer to "Compiler Built-in Functions" on page 1-83.

Examples of the two styles are:

```
int read_io() {
   int ret_val;
   asm("%0 = IO(0x20);" : "=e"(ret_val) : :);
   return ret_val;
}
```

Bad: uses inline `asm` statement.

```
#include <sysreg.h>
#define ADDR 0x20
int read_io() {
   return io_space_read(ADDR);
}
```

Good: uses `sysreg.h`.

Smaller Applications: Optimizing for Code Size

The same ethos for producing fast code also applies to producing small code. You should present the algorithm in a way that gives the optimizer excellent visibility of the operations and data, and hence the greatest freedom to safely manipulate the code to produce small applications.

Once the program is presented in this way, the optimization strategy will depend on the code-size constraint that the program must obey. The first step should be to optimize the application for full performance, using `-O` or `-ipa` switches. If this obeys the code-size constraints, then no more need be done.

The "optimize for space" switch `-Os` (on page 1-33). which may be used in conjunction with IPA, will perform every performance-enhancing transformation except those that increase code-size. In addition, the `-e` linker switch (`-flags-link -e` if used from the compiler command line) may be helpful (on page 1-23). This performs section elimination in the linker to remove unneeded data and code. If the code produced with `-Os` and `-e` does not meet the code-size constraint, some analysis of the source code will be required to try to reduce the code-size further.

Note that loop transformations such as unrolling and software pipelining increase code size. But it is these loop transformations that also give the greatest performance benefit. Therefore, in many cases compiling for minimum code size will produce significantly slower code than optimizing for speed.

The compiler provides a way to balance between the two extremes of `-O` and `-Os`. This is the sliding-scale `-Ov num` switch (adjustable using the optimization slider bar under **Project Options** in the VisualDSP++ IDDE), described on page 1-33. The `n` is a value between 0 and 100, where the lower value corresponds to minimum code size and the upper to maximum performance. A value in between will try to optimize the frequently executed regions of code for maximum performance, while keeping the infrequently executed parts as small as possible. The switch is most reliable when using profile-guided optimization, since the execution counts of the various code regions have been measured experimentally. Without PGO, the execution counts are estimated, based on the depth of loop nesting.

Tip: Avoid the use of inline code.

Avoid using the `inline` keyword to inline code for functions that are used a number of times, especially if they not very small functions. The `-Os` switch does not have any effect on the use of the `inline` keyword. It does, however, prevent automatic inlining (using the `-Oa` switch described on page 1-33) from increasing the code size. Macro functions can also cause code expansion and should be used with care.

Pragmas

Pragmas can assist optimization by allowing the programmer to make assertions or suggestions to the compiler. This section looks at how they can be used to finely tune source code. Refer "Pragmas" on page 1-99 for full details of how each pragma works; the emphasis here will be in considering under what circumstances they are useful during the optimization process.

In most cases the pragmas serve to give the compiler information which it is unable to deduce for itself. It must be emphasized that the programmer is responsible for making sure that the information given by the pragma is valid in the context in which it is used. Use of a pragma to assert that a function or loop has a quality that it does not in fact have is likely to result in incorrect code and hence a malfunctioning application.

An advantage of the use of pragmas is that they allow code to remain portable, since they will normally be ignored by a compiler that does not recognize them.

Function Pragmas

Function pragmas include:
`#pragma const`, `#pragma pure`, `#pragma alloc`, `#pragma regs_clobbered`, and `#pragma optimize_{off|for_speed|for_space}`,.

#pragma const

The `pragma const` pragma asserts to the compiler that a function does not have any side effects (such as modifying global variables or data buffers), and the result returned is only a function of the parameter values. The pragma may be applied to a function prototype or definition. It helps the compiler since two calls to the function with identical parameters will always yield the same result. In this way, calls to `#pragma const` functions may be hoisted out of loops if their parameters are loop independent.

#pragma pure

Like #pragma const, this pragma asserts to the compiler that a function does not have any side effects (such as modifying global variables or data buffers). However, the result returned may be a function of both the parameter values and any global variables. The pragma may be applied to a function prototype or definition. Two calls to the function with identical parameters will always yield the same result provided that no global variables have been modified between the calls. Hence, calls to #pragma pure functions may be hoisted out of loops if their parameters are loop independent and no global variables are modified in the loop.

#pragma alloc

The pragma alloc pragma asserts that the function behaves like the malloc library function. In particular, it returns a pointer to new memory that cannot alias any pre-existing buffers. In the following code,

```
#pragma alloc
int *new_buf(void);
int *vmul(int *a, int *b) {
   int i;
   int *out = new_buf();
   for (i=0; i<100; i++)
      out[i] = a[i] * b[i];
}
```

Good: uses #pragma alloc to disambiguate out from a and b.

the use of the pragma allows the compiler to be sure that the write into buffer out does not modify either of the two input buffers a or b, and, therefore, the iterations of the loop may be re-ordered.

#pragma regs_clobbered

The `regs_clobbered` pragma is a useful way to improve the performance of code that makes function calls. The best use of the pragma is to increase the number of call-preserved registers available across a function call. There are two complementary ways in which this may be done.

First of all, suppose that you have a function written in assembly that you wish to call from C source code. The `regs_clobbered` pragma may be applied to the function prototype to specify which registers are "clobbered" by the assembly function, that is, which registers may have different values before and after the function call. Consider for example an simple assembly function to add two integers and mask the result to fit into 8 bits:

```
_add_mask:
   I6 = I4;
   M5 = 1;
   MODIFY(I6 += M5);
   AY1 = DM(I6 += M5);
   AX1 = DM(I6 += M6);
   AY0 = 255;
   AR = AX1 + AY1;
   AR = AR AND AY0;
   AX1 = AR;
   RTS;
._add_mask.end
```

Clearly, the function does not modify the majority of the scratch registers available and thus these could instead be used as call-preserved registers. This way, fewer spills to the stack would be needed in the caller function. Using the prototype

```
#pragma regs_clobbered "I6, M5, AX1, AY1, AY1, AR, ASTAT"
int add_mask(int, int);
```

Good: uses `regs_clobbered` to increase call-preserved register set.

the compiler is told which registers are modified by a call to the `add_mask` function. The registers not specified by the pragma are assumed to preserve their values across such a call and the compiler may use these spare registers to its advantage when optimizing the call sites.

The pragma is also powerful when all of the source code is written in C. In the above example, a C implementation might be:

```
int add_mask(int a, int b) {
   return ((a+b)&255);
}
```

Bad: function thought to clobber entire volatile register set.

Since this function will not need many registers when compiled, it can be defined using:

```
#pragma regs_clobbered "AX1, AY0, AY1, AR, M5, I6, CCset"
int add_mask(int a, int b) {
   return ((a+b)&255);
}
```

Good: function compiled to preserve most registers.

to ensure that any other registers aside from `AX1, AY0, AY1, AR, M5, I6` and the condition codes will not be modified by the function. If any other registers are used in the compilation of the function, they will be saved and restored during the function prologue and epilogue.

In general, it is not very helpful to specify any of the condition codes as call-preserved as they are difficult to save and restore and are usually clobbered by any function. Moreover, it is usually of limited benefit to be able to keep them live across a function call. Therefore, it is better to use `CCset` (all condition codes) rather than `ASTAT` in the clobbered set above.

For more information, refer to "#pragma regs_clobbered string" on page 1-111.

#pragma optimize_{off|for_speed|for_space|as_cmd_line}

The `optimize_` pragma may be used to change the optimization setting on a function-by-function basis. In particular, it may be useful to optimize functions that are rarely called (for example, error handling code) for space (using `#pragma optimize_for_space`), whereas functions critical to performance should be compiled for maximum speed (`#pragma optimize_for_speed`). The `#pragma optimize_off` is useful for debugging specific functions without increasing the size or decreasing the performance of the overall application unnecessarily.

For more information, refer to "General Optimization Pragmas" on page 1-105.

Loop Optimization Pragmas

Many pragmas are targeted towards helping to produce optimal code for inner loops. These are the `loop_count` and `no_alias` pragmas.

#pragma loop_count

The `loop_count` pragma enables the programmer to inform the compiler about a loop's iteration count. The compiler is able to make more reliable decisions about the optimization strategy for a loop if it knows the iteration count range. If you know that the loop count is always a multiple of some constant, this can also be useful as it allows a loop to be partially unrolled or vectorized without the need for conditionally-executed iterations. Knowledge of the minimum trip count may allow the compiler to omit the guards that are usually required after software pipelining. Any of the parameters of the pragma that are unknown may be left blank.

An example of the use of the `loop_count` pragma might be:

```
#pragma loop_count(/*minimum*/ 40, /*maximum*/ 100, /*modulo*/ 4)
for (i=0; i<n; i++)
   a[i] = b[i];
```

Good: the `loop_count` pragma gives compiler helpful information to assist optimization.

For more information, refer to "#pragma loop_count(min, max, modulo)" on page 1-104.

#pragma no_alias

When immediately preceding a loop, the `no_alias` pragma asserts that no load or store in the loop accesses the same memory as any other. This helps to produce shorter loop kernels as it permits instructions in the loop to be rearranged more freely. See "#pragma no_alias" on page 1-105 for more information.

Useful Optimization Switches

These are the compiler switches useful during the optimization process.

Switch Name	Description
`-const-read-write` on page 1-20	Specifies that data accessed via a pointer to `const` data may be modified elsewhere.
`-flags-link -e` on page 1-23	Specifies linker section elimination.
`-ipa` on page 1-26	Turns on inter-procedural optimization. Implies use of `-O`. May be used in conjunction with `-Os` or `-Ov`.
`-no-fp-associative` on page 1-31	Does not treat floating-point multiply and addition as an associative.
`-O` on page 1-32	Enables code optimizations and optimizes the file for speed.
`-Os` on page 1-33	Optimizes the file for size.
`-Ov num` on page 1-33	Controls speed vs. size optimizations (sliding scale).
`-save-temps` on page 1-39	Saves intermediate files (for example, `.s`).

3 C RUN-TIME LIBRARY

The C run-time library is a collection of functions that you can call from your C programs. Many of these functions are implemented in ADSP-218x DSP's assembly language. C programs depend on library functions to perform operations that are basic to the C language. These operations include memory allocation, character and string conversions, and math calculations.

This chapter describes the current release of the run-time library. Future releases may include more functions. You may use the object files of the library functions in systems based on ADSP-218x processors.

This chapter contains:

- "C Run-Time Library Guide" on page 3-2
- "Documented Library Functions" on page 3-14
- "C Run-Time Library Reference" on page 3-18

For more information on the C standard library, see The Standard C Library by P.J. Plauger, Prentice Hall, 1992. For more information on the algorithms on which many of the library's math functions are based, see Cody, W. J. and W. Waite, "*Software Manual For The Elementary Functions*", Englewood Cliffs, New Jersey: Prentice Hall,1980.

C Run-Time Library Guide

The C run-time library contains functions that you can call from your C program. This section describes how to use the library.

Calling Library Functions

To use a C library function, call the function by name and give the appropriate arguments. The names and arguments for each function appear on the function's reference page. The reference pages appear in "C Run-Time Library Reference" on page 3-18.

Like other functions you use, library functions should be declared. Declarations are supplied in header files, as described in the section, "Working with Library Header Files" on page 3-5.

Function names are C function names. If you call C run-time library functions from an assembly language program, you must use the assembly version of the function name: prefix an underscore on the name. For more information on naming conventions, see the section, "C and Assembly Language Interface" on page 1-148.

You can use the archiver, described in the *VisualDSP++ 3.5 Linker and Utilities Manual for 16-Bit Processors,* to build library archive files of your own functions.

Using the Compiler's Built-In Functions

The C compiler's built-in functions are a set of functions that the compiler immediately recognizes and replaces with inline assembly code instead of a function call. Typically, in-line assembly code is faster than a library routine, and it does not incur the calling overhead. For example, the absolute value function, `abs()`, is recognized by the compiler, which subsequently replaces a call to the C run-time library version with an in-line version.

To use built-in functions, your source must include the required standard include file. For the `abs` functions this would require `stdlib.h` to be included. There are built-in functions used to define some ANSI C `math.h`, `string.h` and `stdlib.h` functions. There are also built-in functions to support various ANALOG extensions to the ANSI standard defined in the include file `math_builtins.h`. Not all built-in functions have a library alternate definition. Therefore, the failure to use the required include files can result in your program build failing to link.

If you want to use the C run-time library functions of the same name, compile with the `-no-builtin` (no builtin functions) compiler switch.

Linking Library Functions

The C run-time library is organized as several libraries which are catalogued in Table 3-15 on page 3-4. The libraries and start-up files are installed within the subdirectory `...218x\lib` of your VisualDSP++ installation. When you call a run-time library function, the call creates a reference that the linker resolves. One way to direct the linker to the library's location is to use the default Linker Description File (`ADSP-21<your_target>.ldf`).

If you are not using the default `.LDF` file, then either add the appropriate library/libraries to the `.LDF` file used for your project, or use the compiler's `-l` switch to specify the library to be added to the link line. For example, the switches `-lc -letsi` will add the C library `libc.dlb` and the ETSI support library `libetsi.dlb` to the list of libraries to be searched by the linker. For more information on the ETSI support library, see "ETSI Support" on page 1-86. For more information on the `.LDF` file, see the *VisualDSP++ 3.5 Linker and Utilities Manual for 16-Bit Processors.*

Table 3-15. C Run-Time Library Files

218x\lib	Directory Description
`218x_hdr.doj`	Default C run-time initialization code . Calls `main()`.
`218x_ezkit_hdr.doj`	C run-time initialization code with EZ-Kit monitor program support. Calls `main()`.
`218x_int_tab.doj`	Default interrupt vector code
`218x_ezkit_int_tab.doj` `2189_int_tab.doj`	Object files for backward compatibility
`218x_exit.doj`	Dummy exit object for backwards compatibility.
`libc.dlb`	C run-time library
`libcab.dlb`	C run-time library code with autobuffering support
`libetsi.dlb`	ETSI library support
`libio.dlb`	I/O library
`libioab.dlb`	I/O library with autobuffering support

Working With Library Source Code

The source code for some functions in the C run-time libraries is provided with your VisualDSP++ software. By default, the installation program copies the source code to a subdirectory of the directory where the run-time libraries are kept, named `218x\lib\src`. Each function is kept in a separate file. The filename is the name of the function with the appropriate extension for C or assembler source. If you do not intend to modify any of the run-time library functions and are not interested in using the source for reference, you can delete this directory and its contents to conserve disk space.

The source code is provided so you can customize specific functions for your own needs. To modify these files, you need proficiency in ADSP-218x DSP assembly language and an understanding of the run-time environment, as explained in "C Run-Time Model and Environment" on page 1-132. Before you make any modifications to the source code, copy the source code to a file with a different filename and rename the function itself. Test the function before you use it in your system to verify that it is functionally correct.

Analog Devices only supports the run-time library functions as provided.

Working with Library Header Files

When you use a library function in your program, you should also include the function's header with the `#include` preprocessor command. The header file for each function is identified in the **Synopsis** section of the function's reference page. Header files contain function prototypes. The compiler uses these prototypes to check that each function is called with the correct arguments.

A list of the header files that are supplied with this release of the `cc218x` compiler appears in Table 3-16. You should use a C standard text to augment the information supplied in this chapter.

Table 3-16. C Run-Time Library Header Files

Header	Purpose	Standard
`assert.h`	Diagnostics	ANSI
`circ.h`	Support for circular buffers	C Extension
`ctype.h`	Character handling	ANSI
`def2181.h`	Memory map definitions	C Extension
`def218x.h`	Memory map definitions	C Extension
`errno.h`	Error handling	ANSI
`ffts.h`	FFT functions	C Extension
`filters.h`	DSP filters	C Extension
`float.h`	Floating-point types	ANSI

Table 3-16. C Run-Time Library Header Files (Cont'd)

Header	Purpose	Standard
`fract.h`	Macros to use fract	C Extension
`iso646.h`	Boolean operators	ANSI
`limits.h`	Limits	ANSI
`locale.h`	Localization	ANSI
`math.h`	Mathematics	ANSI
`misc.h`	Timer	C Extension
`setjmp.h`	Non-local jumps	ANSI
`signal.h`	Signal handling	ANSI
`sport.h`	Serial port	C Extension
`stdarg.h`	Variable arguments	ANSI
`stddef.h`	Standard definitions	ANSI
`stdio.h`	Input/Output	ANSI
`stdlib.h`	Standard library	ANSI
`string.h`	String handling	ANSI
`sysreg.h`	Efficient system access	C Extension

The following sections provides descriptions of all header files.

assert.h

The `assert.h` header file contains the `assert` macro.

ctype.h

The `ctype.h` header file contains functions for character handling, such as `isalpha`, `tolower`, and others.

def2181.h – Memory Map Definitions

The `def2181.h` header file contains macros that define reserved memory addresses for a ADSP-2181 processor. These memory addresses are used as system registers.

def2181x.h – Memory Map Definitions

The `def2181.h` header file contains macros that define reserved memory addresses for a ADSP-218x processor, apart from a ADSP-2181 processor. These memory addresses are used as system registers.

errno.h

The `errno.h` header file provides access to `errno`. This facility is not, in general, supported by the rest of the library.

ffts.h – Fast Fourier Transforms

This category includes the Fast Fourier Transform functions of the C library. These functions are Analog Devices extensions to the ANSI standard.

`fftN` stands for the entire family of Fast Fourier Transform functions: `fft1024`, `fft512`, etc.

The `fftN` functions compute the Fast Fourier Transform (FFT) of their N-point complex input signal. The `ifftN` functions compute the Inverse Fast Fourier Transform of their N-point complex input signal. The input to each of these functions is two integer arrays (real and imaginary) of N elements. The routines output two N-element arrays, and an associated block exponent for them.

If you only wish to input the real part of a signal, make sure that the imaginary input array is filled with zeros before calling the function.

The functions first bit-reverse the input arrays and then process them with an optimized block floating-point FFT (or IFFT) routine.

filters.h – DSP Filters

This category includes the digital signal processing filter functions of the C library. These functions are Analog Devices extensions to the ANSI standard.

float.h – Floating Point

The `float.h` file defines the format of floating-point data types. The `FLT_ROUNDS` macro, defined in the header file, is set to the C run-time environment definition of the rounding mode for `float` variables, which is *round-towards-nearest*.

fract.h – ADSP-218x DSP Macro Fract Definitions

The `fract.h` header file defines macros used to manipulate `fract` fixed point types.

Improved fractional support is provided through the new ETSI operators added for VisualDSP++ 3.5 (see "ETSI Support" on page 1-86).

iso646.h

The `iso646.h` header file defines symbolic names for certain C operators; the symbolic names and their associated value are shown in Table 3-17.

Table 3-17. Symbolic Names Defined in iso646.h

Symbolic Name	Equivalent
`and`	&&
`and_eq`	&=

Table 3-17. Symbolic Names Defined in iso646.h (Cont'd)

Symbolic Name	Equivalent
`bitand`	&
`bitor`	|
`compl`	~
`not`	!
`not_eq`	!=
`or`	||
`or_eq`	|=
`xor`	^
`xor_eq`	^=

locale.h

The `locale.h` header file contains definitions for expressing numeric, monetary, time, and other data.

math.h

This category includes the floating-point mathematical functions of the C run-time library. The mathematical functions are ANSI standard. The `math.h` header file contains prototypes for functions used to calculate mathematical properties of single-precision floating type variables. On the ADSP-218x processors, `double` and `float` are both single-precision floating point types. Additionally, some functions support a 16-bit fractional data type.

The `math.h` file also defines the macro `HUGE_VAL`. `HUGE_VAL` evaluates to the maximum positive value that the type `double` can support. The macros `EDOM` and `ERANGE`, defined in `errno.h`, are used by `math.h` functions to indicate domain and range errors.

Some of the functions in this header file exist as both integer and floating point. The floating-point functions typically have an `f` prefix. Make sure you are using the correct one.

The C language provides for implicit type conversion, so the following sequence produces surprising results with no warnings:

```
float x,y;
y = abs(x);
```

The value in `x` is truncated to an integer prior to calculating the absolute value, then reconverted to floating point for the assignment to `y`.

misc.h – ADSP-218x DSP Timer Functions

The `misc.h` header file includes processor-specific timer functions of the C library, such as `timer_set()`, `timer_on()`, and `timer_off()`.

setjmp.h

The `setjmp.h` header file contains `setjmp` and `longjmp` for non-local jumps.

signal.h

The `signal.h` header file provides function prototypes for the standard ANSI `signal.h` routines and also for several ADSP-218x DSP extensions such as `interrupt()` and `clear_interrupt()`.

The signal handling functions process conditions (hardware signals) that can occur during program execution. They determine the way that your C program responds to these signals. The functions are designed to process such signals as external interrupts and timer interrupts.

sport.h – ADSP-218x DSP Serial Ports

The `sport.h` header file contains macros and subroutines to support the serial ports of the ADSP-218x DSPs.

stdarg.h

The `stdarg.h` header file contains definitions needed for functions that accept a variable number of arguments. Programs that call such functions must include a prototype for the functions referenced.

stdio.h

The `stdio.h` header file defines a set of functions, macros, and data types for performing input and output. Applications that use the facilities of this header file should link with the I/O library `libio.dlb` in the same way as linking with the C run-time library `libc.dlb`. The library is not interrupt-safe and should not therefore be called either directly or indirectly from an interrupt service routine.

The implementation of the `stdio.h` routines is based on a simple interface with a device driver that provides a set of low-level primitives for `open`, `close`, `read`, `write`, and `seek` operations. By default, these operations are provided by the VisualDSP++ simulator and EZ-kits. However, alternative device drivers may be registered (see "Extending I/O Support To New Devices" on page 1-141) that can then be used through the `stdio.h` functions.

The following restrictions apply to this software release:

- Functions `tmpfile()` and `tmpnam()` are not available,
- Functions `rename()` and `remove()` are only supported under the default device driver supplied by the VisualDSP++ simulator and EZ-kits, and they only operate on the host file system,

- Positioning within a file that has been opened as a text stream is only supported if the lines within the file are terminated by the character sequence `\r\n`.

When using the default device driver, all I/O operations are channeled through the C function `_primIO()`. The assembly label has two underscores, `__primIO`. The `_primIO()` function accepts no arguments. Instead, it examines the I/O control block at label `_primIOCB`. Without external intervention by a host environment, the `_primIO` routine simply returns, which indicates failure of the request.

At program termination, the host environment will close down any physical connection between the application and an opened file. However, the I/O library will not implicitly close any opened streams to avoid an unnecessary overheads (particularly with respect to memory occupancy).

Therefore, unless explicit action is taken by an application any unflushed output may be lost. Any output generated by `printf` is always flushed but output generated by other library functions, such as `putchar`, `fwrite`, `fprintf`, will not be automatically flushed. Applications should therefore arrange to close down any streams that they open. Note that the function reference `fflush (NULL);` will flush the buffers of all opened streams.

stdlib.h

The `stdlib.h` header file contains general utilities specified by the C standard. These include some integer math functions, such as `abs`, `div`, and `rand`; general string-to-numeric conversions; memory allocation functions, such as `malloc` and `free`; and termination functions, such as `exit`. This header file also contains prototypes for miscellaneous functions such as `bsearch` and `qsort`.

string.h

The `string.h` header file contains string handling functions, including `strcpy` and `memcpy`.

sysreg.h

The `sysreg.h` header file defines a set of functions that provide efficient system access to registers, modes and addresses not normally accessible from C source. See "Compiler Built-in Functions" on page 1-83 for more information on these functions.

Documented Library Functions

The following tables list the library functions documented in this chapter.

These tables list the functions for each header file separately; however, the reference pages for these library functions are in alphabetical order.

Table 3-18. Library Functions in the `ctype.h` Header File

isalnum	isalpha	iscntrl
isdigit	isgraph	islower
isprint	ispunct	isspace
isupper	isxdigit	tolower
toupper		

Table 3-19. Library Functions in the `ffts.h` Header File

fftN (fft1024, fft512, fft256, fft128, fft64, fft32, fft16, fft8)
ifftN (ifft1024, ifft512, ifft256, ifft128, ifft64, ifft32, ifft16, ifft8)

Table 3-20. Library Functions in the `filters.h` Header File

biquad	demean_buffer	fir
iir		

Table 3-21. Library Functions in the `math.h` Header File

acos	asin	atan
atan2	ceil	cos
cosh	cot	exp
fabs	floor	fmod

Table 3-21. Library Functions in the `math.h` Header File (Cont'd)

frexp	isinf	isnan
ldexp	log	log10
modf	pow	sin
sinh	tan	tanh

Table 3-22. Library Functions in the `misc.h` Header File

timer_off	timer_on	timer_set

Table 3-23. Library Functions in the `setjmp.h` Header File

longjmp	setjmp

Table 3-24. Library Functions in the `signal.h` Header File

clear_interrupt	interrupt	raise
signal		

Table 3-25. Library Functions in the `stdarg.h` Header File

va_arg	va_end	va_start

Table 3-26. Supported Library Functions in the `stdio.h` Header File

clearerr	fclose	feof
ferror	fflush	fgetc
fgetpos	fgets	fopen
fprintf	fputc	fputs
fread	freopen	fscanf
fseek	fsetpos	ftell
fwrite	getc	getchar

Table 3-26. Supported Library Functions in the `stdio.h` Header File

gets	perror	putc
putchar	puts	remove
rename	rewind	scanf
setbuf	setvbuf	sprintf
sscanf	ungetc	vfprintf
vprintf	vsprintf	

Table 3-27. Library Functions in the `stdlib.h` Header File

abort	abs	atexit
atof	atoi	atol
bsearch	calloc	div
exit	free	labs
ldiv	malloc	qsort
rand	realloc	srand
strtod	strtodf	strtol
strtoul		

Table 3-28. Library Functions in the `string.h` Header File

memchr	memcmp	memcpy
memmove	memset	strcat
strchr	strcmp	strcoll
strcpy	strcspn	strerror
strlen	strncat	strncmp
strncpy	strpbrk	strrchr
strspn	strstr	strtok
strxfrm		

Table 3-29. Library Functions in the `sysreg.h` Header File

disable_interrupts	enable_interrupts
io_space_read	io_space_write
sysreg_read	sysreg_write

C Run-Time Library Reference

The C run-time library is a collection of functions that you can call from your C programs.

The information that follows applies to all of the functions in the library.

Notation Conventions

An interval of numbers is indicated by the minimum and maximum, separated by a comma, and enclosed in two square brackets, two parentheses, or one of each. A square bracket indicates that the endpoint is included in the set of numbers; a parenthesis indicates that the endpoint is not included.

Reference Format

Each function in the library has a reference page. These pages have the following format:

Name and Purpose of the function

Synopsis—Required header file and functional prototype

Description—Function specification

Error Conditions—How the function indicates an error

Example—Typical function usage

See Also—Related functions

abort

abnormal program end

Synopsis

```
#include <stdlib.h>
void abort(void);
```

Description

The `abort` function causes an abnormal program termination by raising a `SIGABRT` signal. If the `SIGABRT` handler returns, `abort()` calls `exit()` to terminate the program with a failure condition

Error Conditions

The `abort` function does not return.

Example

```
#include <stdlib.h>
extern int errors;

if(errors)          /* terminate program if */
   abort();         /* errors are present   */
```

See Also

atexit, exit

abs

absolute value

Synopsis

```
#include <stdlib.h>
int abs(int j);
```

Description

The `abs` function returns the absolute value of its `int` input. The `abs` function is implemented through a built-in. The built-in causes the compiler to emit an inline instruction to perform the required function at the point where `abs` is called.

ⓘ `abs(INT_MIN)` returns `INT_MIN`.

Error Conditions

The `abs` function does not return an error condition.

Example

```
#include <stdlib.h>
int i;
i = abs(-5);       /* i == 5 */
```

See Also

fabs, labs

acos

arc cosine

Synopsis

```
#include <math.h>
double acos(double x);
float acosf (float x);
fract16 acos_fr16 (fract16 x);
```

Description

The `acos` function returns the arc cosine of the argument. The input must be in the range [-1, 1]. The output, in radians, is in the range [0, π].

The `acos_fr16` function is only defined for input values between 0 and 0.9 (=`0x7333`). The input argument is in radians. Output values range from `acos(0)*2π(= 0x7FFF)` to `acos(0.9)*2/π(= 0x24C1)`.

Error Conditions

The `acos` function returns a zero if the input is not in the range [-1, 1].

Example

```
#include <math.h>
double y;
y = acos(0.0);      /* y = π/2 */
```

See Also

cos

asin

arc sine

Synopsis

```
#include <math.h>
double asin(double x);
float asinf (float x);
fract16 asin_fr16(fract16 x);
```

Description

The `asin` function returns the arc sine of the argument. The input must be in the range [-1, 1]. The output, in radians, is in the range -π/2 to π/2.

The `asin_fr16` function is only defined for input values between -0.9 (=`0X8CCD`) and 0.9 (=`0x7333`). The input argument is in radians. Output values range from `asin(-0.9)*2/π (= 0XA4C1)` to `asin(0.9)*2/π(= 0x5B3F)`.

Error Conditions

The `asin` function returns a zero if the input is not in the range [-1, 1].

Example

```
#include <math.h>
double y;
y = asin(1.0);      /* y = π/2 */
```

See Also

sin

atan

arc tangent

Synopsis

```
#include <math.h>
double atan(double x);
float atanf (float x);
fract16 atan_fr16 (fract16 x);
```

Description

The `atan` function returns the arc tangent of the argument. The output, in radians, is in the range -π/2 to π/2.

The `atan_fr16` function covers the output range from -π/4 (input value `0x8000`, output value `0x9B78`) to π/4 (input value `0x7FFF`, output value `0x6488`). The input argument is in radians.

Error Conditions

The `atan` function does not return an error condition.

Example

```
#include <math.h>
double y;
y = atan(0.0);      /* y = 0.0 */
```

See Also

atan2, tan

atan2

arc tangent of quotient

Synopsis

```
#include <math.h>
double atan2 (double x, double y);
float atan2f (float x, float y);
fract16 atan2 (fract16 x, fract16 y);
```

Description

The atan2 function computes the arc tangent of the input value x divided by input value y. The output, in radians, is in the range [-π, π].

The atan2_fr16 function uses the full range from -π/4 to π/4 (0x8000 to 0x7FFF) for both input and output arguments. This corresponds to a scaling by π compared to the floating-point function. The input argument is in radians.

Error Conditions

The atan2 function returns a zero if x = 0 and y < > 0.

Example

```
#include <math.h>
double a;
float b;

a = atan2(0.0, 0.5);    /* the error condition: a = 0.0 */
b = atan2(1.0, 0.0);    /* b = π/2                      */
```

See Also

atan, tan

atexit

register a function to call at program termination

Synopsis

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

Description

The `atexit` function registers a function to be called at program termination. Functions are called once for each time they are registered, in the reverse order of registration. Up to 32 functions can be registered using `atexit`.

Error Conditions

The `atexit` function returns a non-zero value if the function cannot be registered.

Example

```
#include <stdlib.h>
extern void goodbye(void);

if (atexit(goodbye()))
   exit(1);
```

See Also

abort, exit

atof

convert string to a double

Synopsis

```
#include <stdlib.h>
double atof(const char *nptr);
```

Description

The `atof` function converts a character string into a floating-point value of type `double`, and returns its value. The character string is pointed to by the argument `nptr` and may contain any number of leading whitespace characters (as determined by the function `isspace`) followed by a floating-point number. The floating-point number may either be of the form of a decimal floating-point number or a hexadecimal floating-point number.

A decimal floating-point number has the form:

```
[sign] [digits] [.digits] [{e|E} [sign] [digits]]
```

The `sign` token is optional and is either plus (+) or minus (–); and `digits` are one or more decimal digits. The sequence of digits may contain a decimal point (.).

The decimal digits can be followed by an exponent, which consists of an introductory letter (`e` or `E`) and an optionally signed integer. If neither an exponent part nor a decimal point appears, a decimal point is assumed to follow the last digit in the string.

The form of a hexadecimal floating-point number is:

```
[sign] [{0x}|{0X}] [hexdigs] [.hexdigs] [{p|P} [sign] [digits]]
```

A hexadecimal floating-point number may start with an optional plus (+) or minus (-) followed by the hexadecimal prefix `0x` or `0X` . This character sequence must be followed by one or more hexadecimal characters that optionally contain a decimal point (.).

The hexadecimal digits are followed by a binary exponent that consists of the letter `p` or `P` , an optional sign, and a non-empty sequence of decimal digits. The exponent is interpreted as a power of two that is used to scale the fraction represented by the tokens `[hexdigs] [.hexdigs]`.

The first character that does not fit either form of number will stop the scan.

Error Conditions

The `atof` function returns a zero if no conversion could be made. If the correct value results in an overflow, a positive or negative (as appropriate) `HUGE_VAL` is returned. If the correct value results in an underflow, 0.0 is returned. The `ERANGE` value is stored in `errno` in the case of either an overflow or underflow.

Notes

The function reference `atof (pdata)` is functionally equivalent to:

```
strtod (pdata, (char *) NULL);
```

and therefore, if the function returns zero, it is not possible to determine whether the character string contained a (valid) representation of 0.0 or some invalid numerical string.

Example

```
#include <stdlib.h>
double x;
x = atof("5.5");      /* x == 5.5 */
```

See Also

atoi, atol, strtod

atoi

convert string to integer

Synopsis

```
#include <stdlib.h>
int atoi(const char *nptr);
```

Description

The `atoi` function converts a character string to an integer value. The character string to be converted is pointed to by the input pointer, `nptr`. The function clears any leading characters for which `isspace` would return true. Conversion begins at the first digit (with an optional preceding sign) and terminates at the first non-digit.

Error Conditions

The `atoi` function returns a zero if no conversion can be made.

Example

```
#include <stdlib.h>
int i;
i = atoi("5");       /* i == 5 */
```

See Also

atof, atol, strtod, strtol, strtoul

atol

convert string to long integer

Synopsis

```
#include <stdlib.h>
long atol(const char *nptr);
```

Description

The `atol` function converts a character string to a long integer value. The character string to be converted is pointed to by the input pointer, `nptr`. The function clears any leading characters for which `isspace` would return true. Conversion begins at the first digit (with an optional preceding sign) and terminates at the first non-digit.

There is no way to determine if a zero is a valid result or an indicator of an invalid string.

Error Conditions

The `atol` function returns a zero if no conversion can be made.

Example

```
#include <stdlib.h>
long int i;

i = atol("5");        /* i == 5 */
```

See Also

atof, atoi, strtod, strtol, strtoul

biquad

biquad filter section

Synopsis

```
#include <filters.h>
int biquad(int sample, int pm coeffs[],
           int dm state[], int sections);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `biquad` function implements a biquad filter. The function produces the filtered response of its input data. The parameter `sections` specifies the number of biquad sections. The `biquad` filter implemented in this function is based on the Oppenheim and Schafer Nth order Direct Form II cascaded scheme. The characteristics of the filter depend on the coefficient values supplied by the calling program.

The `coeffs` array must be five (5) times the number of sections in length and it also must be located in program memory (`pm`). The definition is:

```
int pm coeffs[5*sections];
```

The state array holds two delay elements per section. It also has one extra location that holds an internal pointer. The total length must be `2*sections + 1`. The definition is:

```
int dm state[2*sections + 1];
```

You cannot access the state array, except that the array should be initialized to all zeros before the first call to `biquad`. The first location of state is an address. Setting the address to zero tells the function that it is being called for the first time.

Error Conditions

The `biquad` function does not return an error condition.

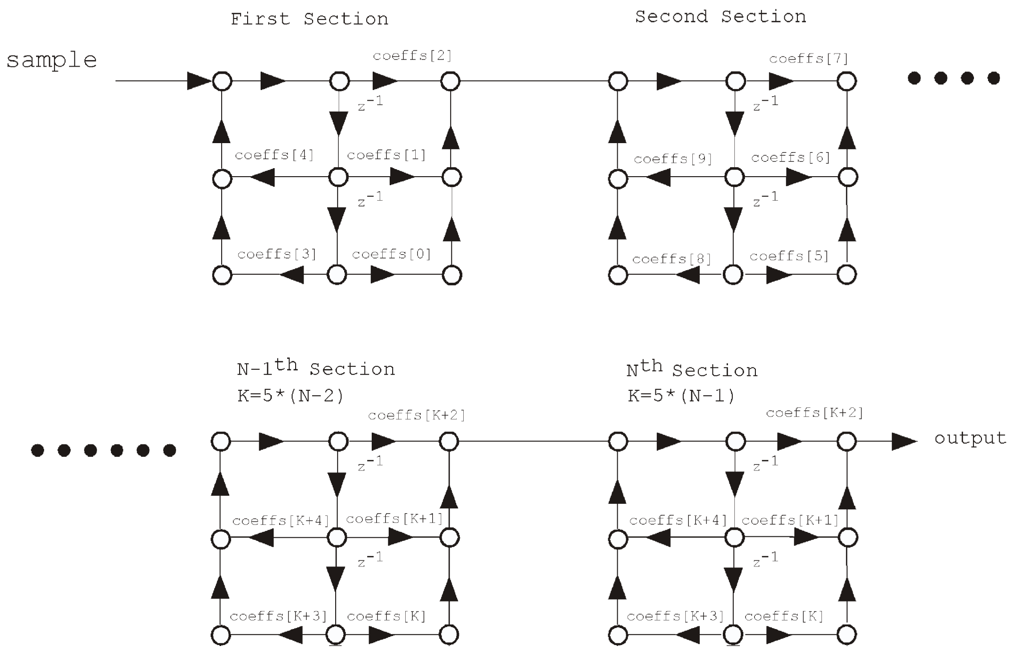
Example

```
#include <filters.h>
#define TAPS 9

float sample, output, state[2*TAPS+1];
float pm coeffs[5*TAPS];
int i;

for (i = 0; i < 2*TAPS+1; i++)
    state[i] = 0;/* initialize state array */

output = biquad (sample, coeffs, state, TAPS);
```



`N` = the number of biquad sections.

The algorithm shown here is adapted from Oppenheim, Alan V. and Ronald Schafer, *Digital Signal Processing*, Englewood Cliffs, New Jersey: Prentice Hall, 1975.

See Also

fir, iir

bsearch

perform binary search in a sorted array

Synopsis

```
#include <stdlib.h>
void *bsearch(const void *key, const void *base,
              size_t nelem, size_t size,
              int (*compare)(const void *, const void *));
```

Description

The `bsearch` function executes a binary search operation on a pre-sorted array, where:

- `key` is a pointer to the element to search for
- `base` points to the start of the array
- `nelem` is the number of elements in the array
- `size` is the size of each element of the array
- `*compare` points to the function used to compare two elements. It takes as parameters a pointer to the `key` and a pointer to an array element and should return a value less than, equal to, or greater than zero, according to whether the first parameter is less than, equal to, or greater than the second.

The `bsearch` function returns a pointer to the first occurrence of `key` in the array.

Error Conditions

The `bsearch` function returns a null pointer if the key is not found in the array.

Example

```
#include <stdlib.h>
char *answer;
char base[50][3];
answer = bsearch("g", base, 50, 3, strcmp);
```

See Also

qsort

calloc

allocate and initialize memory

Synopsis

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

Description

The `calloc` function returns a pointer to a range of dynamically allocated memory that has been initialized to zero. The number of elements (the first argument) multiplied by the size of each element (the second argument) is the total memory allocated. The memory may be deallocated with the `free` function.

Error Conditions

The `calloc` function returns a null pointer if unable to allocate the requested memory.

Example

```
#include <stdlib.h>
int *ptr;

ptr = (int *) calloc(10, sizeof(int));
       /* ptr points to a zeroed array of length 10 */
```

See Also

free, malloc, realloc

ceil

ceiling

Synopsis

```
#include <math.h>
double ceil (double x);
float ceilf (float x);
```

Description

The `ceil` functions return the smallest integral value, expressed as double, that is not less than its input.

Error Conditions

The `ceil` functions do not return an error condition.

Example

```
#include <math.h>
double y;
float x;

y = ceil (1.05);      /* y = 2.0  */
x = ceilf (-1.05);    /* y = -1.0 */
```

See Also

floor

clear_interrupt

clear a pending signal

Synopsis

```
#include <signal.h>
int clear_interrupt(int sig);
```

Description

The `clear_interrupt` function sets a bit in the `IFC` (Interrupt Force and Clear register) to clear a pending interrupt. The `sig` argument must be one of the processor signals shown below for the ADSP-218x DSPs.

Table 3-30. ADSP-218x DSP Signals

Sig Value	Definition
`SIGTIMER`	Timer interrupt
`SIGIRQ0`	Interrupt 0
`SIGSPORT1RECV`	Signal Sport 1 receive
`SIGIRQ1`	Interrupt 1
`SIGSPORT1XMIT`	Signal Sport 1 transmit
`SIGBDMA`	Byte DMA interrupt
`SIGIRQE`	Level sensitive
`SIGSPORT0RECV`	Signal Sport 0 receive
`SIGSPORT0XMIT`	Signal Sport 0 transmit
`SIGIRQ2`	Interrupt 2

Error Conditions

The `clear_interrupt` function returns a `-1` if the parameter is not a valid signal and a zero in all other cases.

Example

```
#include <signal.h>
clear_interrupt(SIGTIMER);
/* clears the timer clear bit in IFC */
```

See Also

interrupt, raise, signal

copysign

copy the sign

Synopsis

```
#include <math.h>
double copysign (double parm1, double parm2);
float copysignf (float parm1, float parm2);
fract16 copysign_fr16 (fract16 parm1, fract16 parm2);
```

Description

This function copies the sign of the second argument to the first argument.

Algorithm

```
return(|parm1| * copysignof(parm2))
```

Domain

Full Range for type of parameters used.

cos

cosine

Synopsis

```
#include <math.h>
double cos(double);
float cosf (float);
fract16 cos_fr16 (fract16);
```

Description

The `cos` function returns the cosine of the argument. The input is interpreted as radians; the output is in the range [-1, 1].

The `cos_fr16` function inputs a fractional value in the range [-1.0, 1.0) corresponding to [-π/2, π/2]. The domain represents half a cycle which can be used to derive a full cycle if required (see Notes below). The result, in radians, is in the range [-1.0, 1.0).

Error Conditions

The `cos` function does not return an error condition.

Example

```
#include <math.h>
double y;

y = cos(3.14159);        /* y = -1.0 */
```

Notes

The domain of the `cos_fr16` function is restricted to the fractional range [`0x8000`, `0x7fff`] which corresponds to half a period from –(π /2) to π/2. It is possible however to derive the full period using the following properties of the function.

```
cosine [0, π/2] = -cosine [π, 3/2 π]
cosine [-π/2, 0] = -cosine [π/2, π]
```

The function below uses these properties to calculate the full period (from 0 to 2π) of the `cosine` function using an input domain of `[0, 0x7fff]`.

```
#include <math.h>

fract16 cos2pi_fr16 (fract16 x)
{
   if (x < 0x2000) {                  /* <0.25      */
      /* first quadrant [0..π/2):                   */
      /* cos_fr16([0x0..0x7fff]) = [0..0x7fff)      */
      return cos_fr16(x * 4);

   } else if (x < 0x6000) {           /* < 0.75     */
      /* if (x < 0x4000)                            */
      /* second quadrant [π/2..π):                  */
      /* -cos_fr16([0x8000..0x0)) = [0x7fff..0)     */
      /*                                            */
      /* if (x < 0x6000)                            */
      /* third quadrant [π..3/2π):                  */
      /* -cos_fr16([0x0..0x7fff]) = [0..0x8000)     */
      return -cos_fr16((0xc000 + x) * 4);

   } else {
      /* fourth quadrant [3/2π..π):                 */
      /* cos_fr16([0x8000..0x0)) = [0x8000..0)       */
      return cos_fr16((0x8000 + x) * 4);
   }
}
```

See Also

acos, sin

cosh

hyperbolic cosine

Synopsis

```
#include <math.h>
double cosh(double);
```

Description

The `cosh` function returns the hyperbolic cosine of its argument.

Error Conditions

The `cosh` function returns the IEEE constant `+Inf` if the argument is outside the domain.

Example

```
#include <math.h>
double x,y;

y = cosh(x);
```

See Also

sinh

cot

cotangent

Synopsis

```
#include <math.h>

float cotf (float a)
double cot (double a)
```

Description

This function calculates the cotangent of its argument a, which is measured in radians. If a is outside of the domain, the function returns 0.

Algorithm

```
 c = cot(a)
```

Domain

x = [–9099 ... 9099]

demean_buffer

remove the mean of a data buffer

Synopsis

```
#include <filters.h>
int demean_buffer (int *input_buffer, int old_mean, int length)
```

Description

The `demean_buffer()` routine removes a DC-bias from input signals and the mean from a buffer of data. It can also execute a notch filter on the input based on an adaptive filter. (See "*Adaptive Signal Processing*" from Prentice Hall (1985).)

`demean_buffer` returns the mean of the current buffer as a result. This value should be passed as a parameter to the function on the next call; the first call to `demean_buffer` should have a `0` for the `old_mean` value.

Error Conditions

The `demean_buffer` function does not return an error condition.

Example

```
#include <filters.h
#define BUFSIZE 1024

int data_buffer [BUFSIZE];
int data_mean = 0;
/* The buffer is filled with data, possibly from a converter. */

/* Remove the mean from the buffer with this demean function  */

data_demean = demean_buffer(data_buffer, data_mean, BUFSIZE);

/* or, as in this example:                                    */
```

```
{
int i, temp_mean;
temp_mean = 0;
for (i=0; i<BUFSIZE; i++)
    temp_mean += data_buffer[i];
temp_mean /= BUFSIZE;
for (i=0; i<BUFSIZE; i++)
    data_buffer[i] -= temp_mean;
}
```

See Also

No references to this function.

disable_interrupts

disable interrupts

Synopsis

```
include <sysreg.h>
void disable_interrupts(void)
```

Description

The `disable_interrupts` function causes the compiler to emit an instruction to disable hardware interrupts.

This function is implemented as a compiler built-in; the emitted instruction will be inline at the point of `disable_interrupts` use. The inclusion of the `sysreg.h` include file is mandatory when using `disable_interrupts`.

The `disable_interrupts` function does not return a value.

Error Conditions

The `disable_interrupts` function does not return, raise, or set any error conditions.

Example

```
#include <sysreg.h>
main(){
   disable_interrupts();  // emits "DIS INTS;" instruction inline
}
```

See Also

enable_interrupts, io_space_read, io_space_write, sysreg_read, sysreg_write

div

division

Synopsis

```
#include <stdlib.h>
div_t div(int numer, int denom);
```

Description

The `div` function divides `numer` by `denom`, both of type `int`, and returns a structure of type `div_t`. The type `div_t` is defined as

```
typedef struct {
   int quot;
   int rem;
} div_t
```

where `quot` is the quotient of the division and `rem` is the remainder, such that if `result` is of type `div_t`,

```
result.quot * denom + result.rem == numer
```

Error Conditions

If `denom` is zero, the behavior of the `div` function is undefined.

Example

```
#include <stdlib.h>
div_t result;

result = div(5, 2);   /* result.quot=2, result.rem=1 */
```

See Also

fmod, ldiv, modf

enable_interrupts

enable interrupts

Synopsis

```
include <sysreg.h>
void enable_interrupts(void)
```

Description

The `enable_interrupts` function causes the compiler to emit an instruction to enable hardware interrupts.

This function is implemented as a compiler built-in; the emitted instruction will be inline at the point of `enable_interrupts` use.

The inclusion of the `sysreg.h` include file is mandatory when using `enable_interrupts`.

The `enable_interrupts` function does not return a value.

Error Conditions

The enable_`interrupts` function does not return, raise, or set any error conditions.

Example

```
#include <sysreg.h>
main(){
   enable_interrupts();   // emits "ENA INTS;" instruction inline
}
```

See Also

disable_interrupts, io_space_read, io_space_write, sysreg_read, sysreg_write

exit

normal program termination

Synopsis

```
#include <stdlib.h>
void exit(int status);
```

Description

The `exit` function causes normal program termination. The functions registered by the `atexit` function are called in reverse order of their registration and the microprocessor is put into the `IDLE` state. The `status` argument is stored in register `AX1`, and control is passed to the label `___lib_prog_term`, which is defined in the run-time start-up.

Error Conditions

The `exit` function does not return an error condition.

Example

```
#include <stdlib.h>

exit(EXIT_SUCCESS);
```

See Also

abort, atexit

exp

exponential

Synopsis

```
#include <math.h>
double exp(double);
float expf(float);
```

Description

The `exp` function computes the exponential value e to the power of its argument. The argument must be in the range [-87.9 , 88.6].

Error Conditions

The `exp` function returns the value `HUGE_VAL` and stores the value `ERANGE` in `errno` when there is an overflow error. In the case of underflow, the `exp` function returns a zero.

Example

```
#include <math.h>
double y;
y = exp(1.0);   /* y = 2.71828...*/
```

See Also

log, pow

fabs

float absolute value

Synopsis

```
#include <math.h>
double fabs(double);
float fabsf(float);
```

Description

The `fabs` function returns the absolute value of the argument.

Error Conditions

The `fabs` function does not return an error condition.

Example

```
#include <math.h>
double y;

y = fabs(-2.3);      /* y = 2.3 */
y = fabs(2.3);       /* y = 2.3 */
```

See Also

abs, labs

fftN

N-point complex input fast Fourier transform (FFT)

Synopsis

```
#include <ffts.h>
int fft1024(int rl_in[],
    int im_in[],
    int rl_out[],
    int im_out[]);

int fft512(int rl_in[],
    int im_in[],
    int rl_out[],
    int im_out[]);

int fft256(int rl_in[],
    int im_in[],
    int rl_out[],
    int im_out[]);

int fft128(int rl_in[],
    int im_in[],
    int rl_out[],
    int im_out[]);

int fft64(int rl_in[],
    int im_in[],
    int rl_out[],
    int im_out[]);

int fft32(int rl_in[],
    int im_in[],
    int rl_out[],
    int im_out[]);

int fft16(int rl_in[],
    int im_in[],
    int rl_out[],
    int im_out[]);
int fft8(int rl_in[],
    int im_in[],
    int rl_out[],
    int im_out[]);
```

Description

These functions are Analog Devices extensions to the ANSI standard.

Each of these eight `fftN` functions computes the N-point radix-2 Fast Fourier transform (FFT) of its integer input (where N is `8`, `16`, `32`, `64`, `128`, `256`, `512`, or `1024`).

There are eight distinct functions in this set. They all perform the same function with the same type and number of arguments. The only difference between them is the size of the arrays on which they operate. To call a particular function, substitute the number of points for N; for example,

```
fft8(r_inp, i_inp, r_outp, i_outp);
```

not

```
fftN(r_inp, i_inp, r_outp, i_outp);
```

The input to `fftN` are two integer array of N points. The array `rl_in` contains the real components of the complex signal, and the array `im_in` contains the imaginary components. If there are fewer than N actual data points, you must pad the array with zeros to make N samples. Better results occur with less zero padding, however. The input data should be windowed (if necessary) before calling the function because no preprocessing is performed on the data. The functions return a block exponent. The input and output arrays must be different. The output arrays must be on circular boundaries.

Error Conditions

The `fftN` functions do not return error conditions.

Example

```
#include <ffts.h>
#define N 1024

int  real_input[N], imag_input[N];

#pragma align 1024
int real_output[N];
#pragma align 1024
int imag_output[N];

int block_exponent;

block_exponent = fft1024 (real_input, imag_input,
                          real_output, imag_output);
```

See Also

ifftN

fir

finite impulse response (FIR) filter

Synopsis

```
#include <filters.h>
int fir(int sample, int pm coeffs[],
                    int dm state[], int taps);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `fir` function implements a finite impulse response (FIR) filter defined by the coefficients and delay line that are supplied in the call of `fir`. The function produces the filtered response of its input data. This FIR filter is structured as a sum of products. The characteristics of the filter (passband, stop band, etc.) are dependent on the coefficient values and the number of taps supplied by the calling program.

The integer input to the filter is `sample`. The integer `taps` indicates the length of the filter, which is also the length of the array `coeffs`. The `coeffs` array holds one FIR filter coefficient per element. The coefficients are stored in reverse order; for example, `coeffs[0]` holds the `(taps-1)` coefficient. The `coeffs` array is located in program memory data space to use the single-cycle dual-memory fetch of the processor.

The `state` array contains a pointer to the delay line as its last element, preceded by the delay line values. The length of the `state` array is therefore one (1) greater than the number of taps. Each filter has its own `state` array, which should not be modified by the calling program, only by the `fir` function. The `state` array should be initialized to zeros before the `fir` function is called for the first time. The delay state array parameter must be located on a circular boundary, otherwise the function will not work.

The parameters `sample`, `coeffs[]`, and `state[]`, are all considered to be fractional numbers. The `fir` function executes fractional multiplications that preserve the format of the fractional input. If your application requires a true integer `fir()`, you should divide the output of the filter by two.

Error Conditions

The `fir` function does not return an error condition.

Example

```
#include <filters.h>
int y;
int pm coeffs[10]; /* coeffs array must be */
                   /* initialized and in   */
                   /* PM memory            */
#pragma align 16
int state[11];
int i;
for (i=0; i < 11; i++)
    state[i]=0;    /* initialize state array      */
y = fir(0x1234,coeffs, state, 10);
                   /* y holds the filtered output */
```

See Also

biquad, iir

floor

floor

Synopsis

```
#include <math.h>
double floor(double);
float floorf(float);
```

Description

The `floor` function returns the largest integral value that is not greater than its input.

Error Conditions

The `floor` function does not return an error condition.

Example

```
#include <math.h>
double y;

y = floor(1.25);       /* y = 1.0  */
y = floor(-1.25);      /* y = -2.0 */
```

See Also

ceil

fmod

floating-point modulus

Synopsis

```
#include <math.h>
double fmod(double numer, double denom);
float fmodf(float numer, float denom);
```

Description

The `fmod` function computes the floating-point remainder that results from dividing the first argument into the second argument. This value is less than the second argument and has the same sign as the first argument. If the second argument is equal to zero, `fmod` returns a zero.

Error Conditions

The `fmod` function does not return an error condition.

Example

```
#include <math.h>
double y;

y = fmod(5.0, 2.0);      /* y = 1.0 */
```

See Also

div, ldiv, modf

free

deallocate memory

Synopsis

```
#include <stdlib.h>
void free(void *ptr);
```

Description

The `free` function deallocates a pointer previously allocated to a range of memory (by `calloc` or `malloc`) to the free memory heap. If the pointer was not previously allocated by `calloc`, `malloc` or `realloc`, the behavior is undefined.

The `free` function returns the allocated memory to the heap from which it was allocated.

Error Conditions

The `free` function does not return an error condition.

Example

```
#include <stdlib.h>
char *ptr;

ptr = malloc(10);      /* Allocate 10 words from heap */
free(ptr);             /* Return space to free heap   */
```

See Also

calloc, malloc, realloc

frexp

separate fraction and exponent

Synopsis

```
#include <math.h>
double frexp(double f, int *expptr);
float frexpf(float f, int *expptr);
```

Description

The `frexp` function separates a floating-point input into a normalized fraction and a (base 2) exponent. The function returns the first argument as a fraction in the interval [½, 1), and stores a power of 2 in the integer pointed to by the second argument. If the input is zero, then the fraction and exponent will both be set to zero.

Error Conditions

The `frexp` function does not return an error condition.

Example

```
#include <math.h>
double y;
int exponent;

y = frexp(2.0, &exponent);      /* y=0.5, exponent=2 */
```

See Also

modf

ifftN

N-point inverse complex input fast Fourier transform (IFFT)

Synopsis

```
#include <ffts.h>
int  ifft1024(int dm real_input[],
     int dm imag_input[],
     int dm real_output[],
     int dm imag_output[]);

int  ifft512(int dm real_input[],
     int dm imag_input[],
     int dm real_output[],
     int dm imag_output[]);

int  ifft256(int dm real_input[],
     int dm imag_input[],
     int dm real_output[],
     int dm imag_output[]);

int  ifft128(int dm real_input[],
     int dm imag_input[],
     int dm real_output[],
     int dm imag_output[]);

int  ifft64(int dm real_input[],
     int dm imag_input[],
     int dm real_output[],
     int dm imag_output[]);

int  ifft32(int dm real_input[],
     int dm imag_input[],
     int dm real_output[],
     int dm imag_output[]);

int  ifft16(int dm real_input[],
     int dm imag_input[],
     int dm real_output[],
     int dm imag_output[]);
int  ifft8(int dm real_input[],
     int dm imag_input[],
     int dm real_output[],
     int dm imag_output[]);
```

Description

These functions are Analog Devices extensions to the ANSI standard.

Each of these eight `ifftN` functions computes the N-point radix-2 Inverse Fast Fourier transform (IFFT) of its integer input (where N is `8`, `16`, `32`, `64`, `128`, `256`, `512`, or `1024`).

There are eight distinct functions in this set. They all perform the same function with the same type and number of arguments. The only difference between them is the size of the arrays on which they operate. To call a particular function, substitute the number of points for N; for example,

```
ifft8(r_inp, i_inp, r_outp, i_outp);
```

not

```
ifftN(r_inp, i_inp, r_outp, i_outp);
```

The input to `ifftN` are two integer array of N points. The array `real_input` contains the real components of the inverse FFT input and the array `imag_input` contains the imaginary components. If there are fewer than N actual data points, you must pad the array with zeros to make N samples. Better results occur with less zero padding, however. The input data should be windowed (if necessary) before calling the function because no preprocessing is performed on the data. The functions return a block exponent. The input and output arrays must be different. The output arrays must be on circular boundaries.

Error Conditions

The `ifftN` functions do not return error conditions.

Example

```
#include <ffts.h>
#define N 1024

int real_input[N], imag_input[N];

#pragma align 1024
int imag_output[N];
#pragma align 1024
int real_output[N];

int block exponent;

block exponent = ifft1024 (real_input, imag_input,
                           real_output, imag_output);
```

See Also

fftN

iir

infinite impulse response (IIR) filter

Synopsis

```
#include <filters.h>
int iir (int sample,
         int pm a_coeffs[],
         int float pm b_coeffs[],
         int dm state[],
         int taps);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `iir` function implements an infinite impulse response (IIR) filter based on the Oppenheim and Schafer direct form II. The function returns the filtered response of the input data `sample`. The characteristics of the filter are dependent upon a set of coefficients, a delay line, and the length of the filter. The length of filter is specified by the argument `taps`.

The set of IIR filter coefficients is composed of a-coefficients and b-coefficients. The a_0 coefficient is assumed to be `1.0`, and the remaining a-coefficients should be scaled accordingly and stored in the array `a_coeffs` in reverse order. The length of the `a_coeffs` array is `taps` and therefore `a_coeffs[0]` should contain a_{taps}, and `a_coeffs[taps-1]` should contain a_1.

The b-coefficients are stored in the array `b_coeffs`, also in reverse order. The length of the `b_coeffs` is `taps+1`, and so `b_coeffs[0]` will contain b_{taps} and `b_coeffs[taps]` will contain b_0

Both the a_coeffs and b_coeffs arrays must be located in program memory (PM) so that the single-cycle dual-memory fetch of the processor can be used.

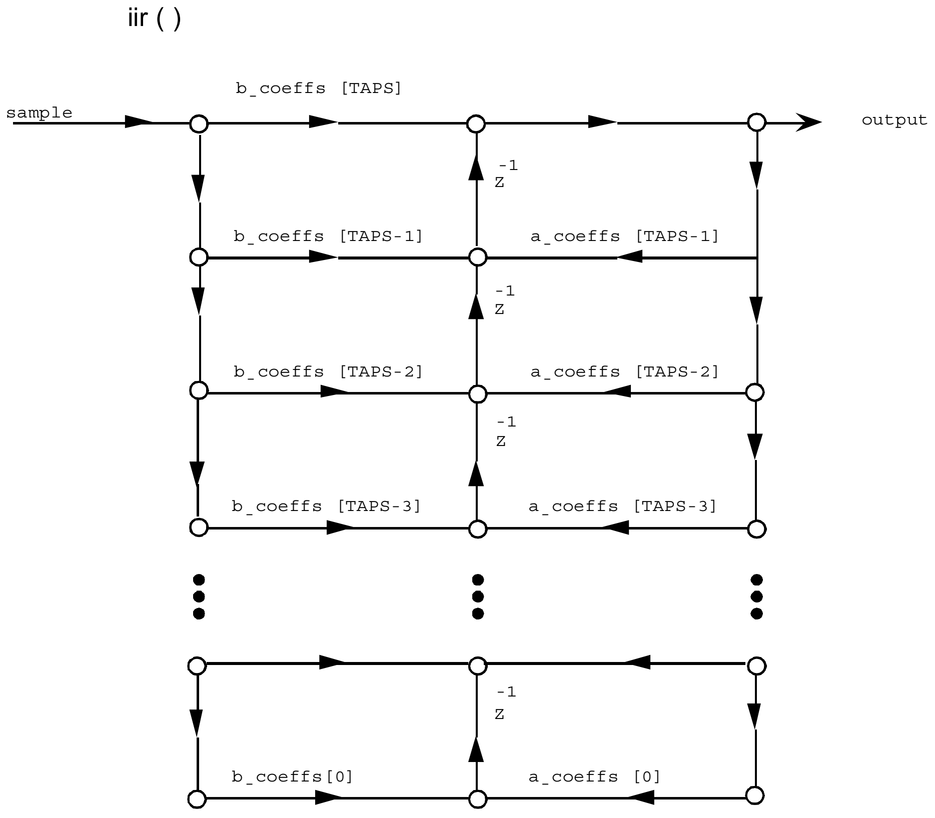
When importing coefficients from a filter design tool that employs a transposed direct form II, the a_1 and $a_2$coefficients have to be negated. For example, if a filter design tool returns A = [1.0, 0.2, -0.9], then the a-coefficients will first have to be inverted to A = [1.0, -0.2, 0.9].

The state array contains a pointer to the delay line as its last element, preceded by the delay line values. The length of the state array is therefore one (1) greater than the number of taps. Each filter has its own state array, which should not be modified by the calling program, only by the iir function. The state array should be initialized to zeros (0) before the iir function is called for the first time. The state array needs to be declared on a circular boundary.

The parameters sample, a_coeffs[], b_coeffs[] and state[], are all considered to be fractional numbers. The iir function executes fractional multiplications that preserve the format of the fractional input.

The following flow graph corresponds to the iir() routine as part of the C Run-Time Library (adapted from the Oppenheim and Schafer text "*Digital Signal Processing*" from Prentice Hall (1975)):

- The b_coeffs and state arrays should equal length TAPS+1
- The a_coeffs array should equal length TAPS



Algorithm

```
d(n) = x(n) - a(1) * d(n-1) - a(2) * d(n-2) ...
y(n) = b(0) * d(n) + b(1) * d(n-1) + b(2) * d(n-2) ...

x(n) is sample and y(n) is output
```

Error Conditions

The `iir` function does not return an error condition.

Example

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

#define FRACT int
#define TAPS 4

static FRACT output[TAPS];
       /* verified with matlab */
static FRACT e_output[TAPS]={0x225,0xe10,0x545,0x16a0};
static int fail=0;
       /* state array needs to be declared on circular boundary */
#pragma align 8static FRACT state[TAPS+1];

main(){
int i;  /* temporary loop index */

       /* example random input/coeffs */
static FRACT pm a_coeffs[TAPS]=
{0x4321, 0x8765, 0x1234, 0x5678};

static FRACT pm b_coeffs[TAPS+1]=
{0x1234, 0x5678, 0x9876, 0x5432, 0x1012};

int sample[TAPS] = {0x1111, 0x2222, 0x3333, 0x4444};

extern FRACT state[TAPS+1];
       /* state array needs to be intialised to zero */
for (i=0; i<TAPS+1; i++) state[i]=0;

       /* iir loop */
for (i=0; i<TAPS; i++)
   output[i] = iir(sample[i], a_coeffs, b_coeffs, state, TAPS);

       /* check output with expected */
for (i=0; i<TAPS; i++)
    if ( output[i] != e_output[i] ) fail++;

       /* print out pass/fail message */
if ( fail==0 )
    printf("Test passed\n");
else
    printf("Test failed\n");
}
```

See Also

biquad, fir

interrupt

define interrupt handling

Synopsis

```
#include <signal.h>
void (*interrupt (int sig, void(*func)(int))) (int);
void (*interruptf(int sig, void (*func)(int))) (int);
void (*interrupts(int sig, void (*func)())) ();
```

Description

These functions are Analog Devices extensions to the ANSI standard.

The `interrupt` function determines how a signal received during program execution is handled. The `interrupt` function executes the function pointed to by `func` at every interrupt `sig`; the `signal` function executes the function only once.

The different variants of the `interrupt` functions differentiate between handler dispatching functions. The variants will be appropriate for some applications and provide improved efficiency. The default `interrupt` function dispatcher saves and restores all scratch registers and modes on the data stack around a call to the handler (`func`) when servicing an interrupt. This dispatcher will pass the interrupt ID (for example, `SIG_PWRDWN`) to the handler as its parameter.

`The interruptf` interrupt dispatcher does the same as `interrupt`, except it switches between primary and secondary register sets to save and restore registers instead of using the data stack. The `interruptf` function cannot be used in applications where nested interrupts are enabled. This interrupt dispatcher will pass the interrupt ID to the handler as its parameter.

The `interrupts` interrupt dispatcher saves and restores only the smallest number of registers and modes required to determine if a handler has been registered and to call that handler. The handler passed as input to `interrupts` must be declared using the `#pragma interrupt` directive (on page 1-102). The `#pragma altregisters` directive (on page 1-103) may be used in conjunction with the `interrupt pragma` in the definition of the handler. This dispatcher will *not* pass the interrupt ID to the handler.

The `sig` argument must be one of the signals listed in priority order in Table 3-31.

Table 3-31. Interrupt Function Signals - Values and Meanings

Sig Value	**Definition**
`SIGPWRDWN`	Power down interrupt
`SIGIRQ2`	Interrupt 2
`SIGIRQL1`	Interrupt 1 (level sensitive)
`SIGIRQL0`	Interrupt 0 (level sensitive)
`SIGSPORT0XMIT`	Signal Sport 0 transmit
`SIGSPORT0RECV`	Signal Sport 0 receive
`SIGIRQE`	Level sensitive
`SIGBDMA`	Byte DMA interrupt
`SIGSPORT1XMIT`	Signal Sport 1 transmit
`SIGIRQ1`	Interrupt 1
`SIGSPORT1RECV`	Signal Sport 1 receive
`SIGIRQ0`	Interrupt 0
`SIGTIMER`	Timer interrupt
`SIGABRT`	Software abort signal
`SIGILL`	Software illegal signal
`SIGINT`	Software segmentation signal
`SIGTERM`	Software termination signal
`SIGFPE`	Software floating point exception signal

The `interrupt` function causes the receipt of the signal number `sig` to be handled in one of the following ways:

- `SIG_DFL`—The default action is taken.
- `SIG_IGN`—The signal is ignored.
- Function Address—The function pointed to by `func` is executed.

The function pointed to by `func` is executed each time the interrupt is received. The `interrupt` function must be called with the `SIG_IGN` argument to disable interrupt handling.

Interrupts are *not* nested by the default start-up file.

Error Conditions

The `interrupt` function returns `SIG_ERR` and sets `errno` equal to `SIG_ERR` if the requested interrupt is not recognized.

Example

```
include <signal.h>

void handler (int sig) {  /* Interrupt Service Routine (ISR) */
}

main () {
   /* enable power down interrupt and register ISR */
   interrupt(SIG_PWRDWN, handler);

   /* disable power down interrupt */
   interrupt(SIG_PWRDWN, SIG_IGN);

   /* enable power down interrupt and register ISR */
   interruptf(SIG_PWRDWN, handler);

   /* disable power down interrupt */
   interruptf(SIG_PWRDWN, SIG_IGN);
}
```

See Also

raise, signal

io_space_read

read I/O space

Synopsis

```
#include <sysreg.h>
int io_space_read(const int)
```

Description

The `io_space_read` function returns the value read from I/O memory space at the address specified by the parameter.

The function is implemented as a compiler built-in. If the input argument is a constant literal value, the compiler will emit a Type 29 instruction that will be inlined at the point of `io_space_read` use. For non-literal inputs, the compiler will call a library compiler support routine to perform the required read.

Error Conditions

The `io_space_read` function does not return, raise, or set any error conditions.

Example

```
#include <sysreg.h>
int addr = 0xA;

main(){
   int v1 = io_space_read(0xA);    /* inline instruction
                                      will be generated       */
   int v2 = io_space_read(addr);   /* library support routine
                                      will be called          */
}
```

See Also

disable_interrupts, enable_interrupts, io_space_write, sysreg_read, sysreg_write

io_space_write

write I/O space

Synopsis

```
#include <sysreg.h>
void io_space_write(const int address, const unsigned int value)
```

Description

The `io_space_write` function stores the value passed as the second parameter to I/O memory space at the address passed as the first parameter.

This function is implemented as a compiler built-in. If the address parameter is a constant literal value the compiler will emit a Type 29 instruction that will be inlined at the point of `io_space_write` use. For non-literal addresses, the compiler will call a library compiler support routine to perform the required write.

The inclusion of the `sysreg.h` include file is mandatory when using `io_space_write`.

Error Conditions

The `io_space_write` function does not return, raise or set any error conditions.

Example

```
#include <sysreg.h>
int addr = 0xA;
int val = 0xA;

main(){
int v1 = io_space_write(0xA, val);     /* inline instruction
                                          will be generated   */
int v2 = io_space_write(addr, 0xFF);   /* support routine
                                          will be called      */
}
```

See Also

disable_interrupts, enable_interrupts, io_space_read, sysreg_read, sysreg_write

isalnum

detect alphanumeric character

Synopsis

```
#include <ctype.h>
int isalnum(int c);
```

Description

The `isalnum` function determines if the argument is an alphanumeric character (`A-Z`, `a-z`, or `0-9`). If the argument is not alphanumeric, `isalnum` returns a zero. If the argument is alphanumeric, `isalnum` returns a non-zero value.

Error Conditions

The `isalnum` function does not return any error conditions.

Example

```
#include <ctype.h>
int ch;

for (ch=0; ch<=0x7f; ch++) {
   printf("%#04x", ch);
   printf("%3s", isalnum(ch) ? "alphanumeric" : "");
   putchar('\n');
}
```

See Also

isalpha, isdigit

isalpha

detect alphabetic character

Synopsis

```
#include <ctype.h>
int isalpha(int c);
```

Description

The `isalpha` function determines if the input is an alphabetic character (`A-Z` or `a-z`). If the input is not alphabetic, `isalpha` returns a zero. If the input is alphabetic, `isalpha` returns a non-zero value.

Error Conditions

The `isalpha` function does not return any error conditions.

Example

```
#include <ctype.h>
int ch;

for (ch=0; ch<=0x7f; ch++) {
   printf("%#04x", ch);
   printf("%2s", isalpha(ch) ? "alphabetic" : "");
   putchar('\n');
}
```

See Also

isalnum, isdigit

iscntrl

detect control character

Synopsis

```
#include <ctype.h>
int iscntrl(int c);
```

Description

The `iscntrl` function determines if the argument is a control character (`0x00-0x1F` or `0x7F`). If the argument is not a control character, `iscntrl` returns a zero. If the argument is a control character, `iscntrl` returns a non-zero value.

Error Conditions

The `iscntrl` function does not return any error conditions.

Example

```
#include <ctype.h>
int ch;

for (ch=0; ch<=0x7f; ch++) {
   printf("%#04x", ch);
   printf("%2s", iscntrl(ch) ? "control" : "");
   putchar('\n');
}
```

See Also

isalnum, isgraph

isdigit

detect decimal digit

Synopsis

```
#include <ctype.h>
int isdigit(int c);
```

Description

The `isdigit` function determines if the input character is a decimal digit (`0-9`). If the input is not a digit, `isdigit` returns a zero. If the input is a digit, `isdigit` returns a non-zero value.

Error Conditions

The `isdigit` function does not return an error condition.

Example

```
#include <ctype.h>
int ch;

for (ch=0; ch<=0x7f; ch++) {
     printf("%#04x", ch);
     printf("%2s", isdigit(ch) ? "digit" : "");
     putchar('\n');
}
```

See Also

isalnum, isalpha, isxdigit

isgraph

detect printable character, not including white space

Synopsis

```
#include <ctype.h>
int isgraph(int c);
```

Description

The `isgraph` function determines if the argument is a printable character, not including white space (`0x21-0x7e`). If the argument is not a printable character, `isgraph` returns a zero. If the argument is a printable character, `isgraph` returns a non-zero value.

Error Conditions

The `isgraph` function does not return any error conditions.

Example

```
#include <ctype.h>
int ch;

for (ch=0; ch<=0x7f; ch++) {
     printf("%#04x", ch);
     printf("%2s", isgraph(ch) ? "graph" : "");
     putchar('\n');
}
```

See Also

isalnum, iscntrl, isprint

isinf

test for infinity

Synopsis

```
#include <math.h>
int isinff(float x);
int isinf(double x);
```

Description

The `isinf` function returns a zero if the argument is not set to the IEEE constant for `+Infnity` or `-Infinity`; otherwise, the function will return a non-zero value.

Error Conditions

The `isinf` function does not return or set any error conditions.

Example

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

static int fail=0;

main(){
   /* test int isinf(double) */
   union {
      double d; float f; unsigned long l;
   } u;

   #ifdef __DOUBLES_ARE_FLOATS__
      u.l=0xFF800000L; if ( isinf(u.d)==0 ) fail++;
      u.l=0xFF800001L; if ( isinf(u.d)!=0 ) fail++;
      u.l=0x7F800000L; if ( isinf(u.d)==0 ) fail++;
      u.l=0x7F800001L; if ( isinf(u.d)!=0 ) fail++;
   #endif
```

```
      /* test int isinff(float) */
      u.l=0xFF800000L; if ( isinff(u.f)==0 ) fail++;
      u.l=0xFF800001L; if ( isinff(u.f)!=0 ) fail++;
      u.l=0x7F800000L; if ( isinff(u.f)==0 ) fail++;
      u.l=0x7F800001L; if ( isinff(u.f)!=0 ) fail++;

   /* print pass/fail message */
   if ( fail==0 )
      printf("Test passed\n");
   else
      printf("Test failed: %d\n", fail);
}
```

See Also

isnan

islower

detect lowercase character

Synopsis

```
#include <ctype.h>
int islower(int c);
```

Description

The `islower` function determines if the argument is a lowercase character (`a-z`). If the argument is not lowercase, `islower` returns a zero. If the argument is lowercase, `islower` returns a non-zero value.

Error Conditions

The `islower` function does not return any error conditions.

Example

```
#include <ctype.h>
int ch;

for (ch=0; ch<=0x7f; ch++) {
     printf("%#04x", ch);
     printf("%2s", islower(ch) ? "lowercase" : "");
     putchar('\n');
}
```

See Also

isalpha, isupper

isnan

test for not-a-number (NAN)

Synopsis

```
#include <math.h>
int isnanf(float x);
int isnan(double x);
```

Description

The `isnan` function returns a zero if the argument is not set to an IEEE `NaN` (Not a Number); otherwise, the function will return a non-zero value.

Error Conditions

The `isnan` function does not return or set any error conditions.

Example

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

static int fail=0;

main(){
   /* test int isnan(double) */
   union {
      double d; float f; unsigned long l;
   } u;

   #ifdef __DOUBLES_ARE_FLOATS__
      u.l=0xFF800000L; if ( isnan(u.d)!=0 ) fail++;
      u.l=0xFF800001L; if ( isnan(u.d)==0 ) fail++;
      u.l=0x7F800000L; if ( isnan(u.d)!=0 ) fail++;
      u.l=0x7F800001L; if ( isnan(u.d)==0 ) fail++;
   #endif
```

```
    /* test int isnanf(float) */
      u.l=0xFF800000L; if ( isnanf(u.f)!=0 ) fail++;
      u.l=0xFF800001L; if ( isnanf(u.f)==0 ) fail++;
      u.l=0x7F800000L; if ( isnanf(u.f)!=0 ) fail++;
      u.l=0x7F800001L; if ( isnanf(u.f)==0 ) fail++;

    /* print pass/fail message */
    if ( fail==0 )
      printf("Test passed\n");
    else
      printf("Test failed: %d\n", fail);
  }
```

See Also

isinf

isprint

detect printable character

Synopsis

```
#include <ctype.h>
int isprint(int c);
```

Description

The `isprint` function determines if the argument is a printable character (`0x20-0x7E`). If the argument is not a printable character, `isprint` returns a zero. If the argument is a printable character, `isprint` returns a non-zero value.

Error Conditions

The `isprint` function does not return any error conditions.

Example

```
#include <ctype.h>
int ch;

for (ch=0; ch<=0x7f; ch++) {
    printf("%#04x", ch);
    printf("%3s", isprint(ch) ? "printable" : "");
    putchar('\n');
}
```

See Also

isgraph, isspace

ispunct

detect punctuation character

Synopsis

```
#include <ctype.h>
int ispunct(int c);
```

Description

The `ispunct` function determines if the argument is a punctuation character. If the argument is not a punctuation character, `ispunct` returns a zero. If the argument is a punctuation character, `ispunct` returns a non-zero value.

Error Conditions

The `ispunct` function does not return any error conditions.

Example

```
#include <ctype.h>
int ch;

for (ch=0; ch<=0x7f; ch++) {
    printf("%#04x", ch);
    printf("%3s", ispunct(ch) ? "punctuation" : "");
    putchar('\n');
}
```

See Also

isalnum

isspace

detect whitespace character

Synopsis

```
#include <ctype.h>
int isspace(int c);
```

Description

The `isspace` function determines if the argument is a blank whitespace character (`0x09-0x0D` or `0x20`). This includes space, form feed (`\f`), new line (`\n`), carriage return (`\r`), horizontal tab (`\t`) and vertical tab (`\v`).

If the argument is not a blank space character, `isspace` returns a zero. If the argument is a blank space character, `isspace` returns a non-zero value.

Error Conditions

The `isspace` function does not return any error conditions.

Example

```
#include <ctype.h>
int ch;

for (ch=0; ch<=0x7f; ch++) {
    printf("%#04x", ch);
    printf("%2s", isspace(ch) ? "space" : "");
    putchar('\n');
}
```

See Also

iscntrl, isgraph

isupper

detect uppercase character

Synopsis

```
#include <ctype.h>
int isupper(int c);
```

Description

The `isupper` function determines if the argument is an uppercase character (`A-Z`). If the argument is not an uppercase character, `isupper` returns a zero. If the argument is an uppercase character, `isupper` returns a non-zero value.

Error Conditions

The `isupper` function does not return any error conditions.

Example

```
#include <ctype.h>
int ch;

for (ch=0; ch<=0x7f; ch++) {
    printf("%#04x", ch);
    printf("%2s", isupper(ch) ? "uppercase" : "");
    putchar('\n');
}
```

See Also

isalpha, islower

isxdigit

detect hexadecimal digit

Synopsis

```
#include <ctype.h>
int isxdigit(int c);
```

Description

The `isxdigit` function determines if the argument character is a hexadecimal digit character (`A-F`, `a-f`, or `0-9`). If the argument is not a hexadecimal digit, `isxdigit` returns a zero. If the argument is a hexadecimal digit, `isxdigit` returns a non-zero value.

Error Conditions

The `isxdigit` function does not return any error conditions.

Example

```
#include <ctype.h>
int ch;

for (ch=0; ch<=0x7f; ch++) {
    printf("%#04x", ch);
    printf("%2s", isxdigit(ch) ? "hexadecimal" : "");
    putchar('\n');
}
```

See Also

isalnum, isdigit

labs

long integer absolute value

Synopsis

```
#include <stdlib.h>
long int labs(long int);
```

Description

The `labs` function returns the absolute value of its long integer input.

Error Conditions

The `labs` function does not return an error condition.

Example

```
#include <stdlib.h>
long int j;
j = labs(-285128);    /* j = 285128 */
```

See Also

abs, fabs

ldexp

multiply by power of 2

Synopsis

```
#include <math.h>
double ldexp(double x, int n);
float ldexpf(float x, int n);
```

Description

The `ldexp` function returns the value of the floating-point input multiplied by 2 raised to the power of `n`. It adds the value of the second argument `n` to the exponent of the first argument `x`.

Error Conditions

If the result overflows, `ldexp` returns a `NaN`. If the result underflows, `ldexp` returns a zero.

Example

```
#include <math.h>
double y;

y = ldexp(0.5, 2);    /* y = 2.0 */
```

See Also

exp, pow

ldiv

long division

Synopsis

```
#include <stdlib.h>
ldiv_t ldiv(long int numer, long int denom);
```

Description

The `ldiv` function divides `numer` by `denom`, and returns a structure of type `ldiv_t`. The type `ldiv_t` is defined as:

```
typedef struct {
   long int quot;
   long int rem;
} ldiv_t
```

where `quot` is the quotient of the division and `rem` is the remainder, such that if result is of type `ldiv_t`:

```
result.quot * denom + result.rem = numer
```

Error Conditions

If `denom` is zero, the behavior of the `ldiv` function is undefined.

Example

```
#include <stdlib.h>
ldiv_t result;

result = ldiv(7, 2);      /* result.quot=3, result.rem=1 */
```

See Also

fmod, div

log

natural logarithm

Synopsis

```
#include <math.h>
double log(double);
float logf(float);
```

Description

The `log` function computes the natural (`base e`) logarithm of its input.

Error Conditions

The `log` function returns a zero and sets `errno` to `EDOM` if the input value is negative.

Example

```
#include <math.h>
double y;

y = log(1.0);       /* y = 0.0 */
```

See Also

exp, log10

log10

base 10 logarithm

Synopsis

```
#include <math.h>
double log10(double);
float log10f(float);
```

Description

The `log10` function returns the base 10 logarithm of its input.

Error Conditions

The `log10` function indicates a domain error (sets `errno` to `EDOM`) and returns a zero if the input is negative.

Example

```
#include <math.h>
double y;

y = log10(100.0);      /* y = 2.0 */
```

See Also

log, pow

longjmp

second return from setjmp

Synopsis

```
#include <setjmp.h>
void longjmp(jmp_buf env, int return_val);
```

Description

The `longjmp` function causes the program to execute a second return from the place where `setjmp (env)` was called (with the same `jmp_buf` argument).

The `longjmp` function takes as its arguments a jump buffer that contains the context at the time of the original call to `setjmp`. It also takes an integer, `return_val`, which `setjmp` returns if `return_val` is non-zero. Otherwise, `setjmp` returns a 1.

If `env` was not initialized through a previous call to `setjmp` or the function that called `setjmp` has since returned, the behavior is undefined. Also, automatic variables that are local to the original function calling `setjmp`, that do not have `volatile`-qualified type, and that have changed their value prior to the `longjmp` call, have indeterminate value.

Error Conditions

The `longjmp` function does not return an error condition.

Example

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

```
jmp_buf env;
int res;

if ((res == setjmp(env)) != 0) {
   printf ("Problem %d reported by func ()", res);
   exit (EXIT_FAILURE);
}
func ();

void func (void)
{
   if (errno != 0) {
      longjmp (env, errno);
   }
}
```

See Also

setjmp

malloc

allocate memory

Synopsis

```
#include <stdlib.h>
void *malloc(size_t size);
```

Description

The `malloc` function returns a pointer to a block of memory of length `size`. The block of memory is uninitialized.

Error Conditions

The `malloc` function returns a null pointer if it is unable to allocate the requested memory.

Example

```
#include <stdlib.h>
int *ptr;

ptr = (int *)malloc(10);      /* ptr points to an   */
                              /* array of length 10 */
```

See Also

calloc, free, realloc

memchr

find first occurrence of character

Synopsis

```
#include <string.h>
void *memchr(const void *s1, int c, size_t n);
```

Description

The `memchr` function compares the range of memory pointed to by `s1` with the input character `c` and returns a pointer to the first occurrence of `c`. A null pointer is returned if `c` does not occur in the first `n` characters.

Error Conditions

The `memchr` function does not return an error condition.

Example

```
#include <string.h>
char *ptr;

ptr= memchr("TESTING", ‘E’, 7);
     /* ptr points to the E in TESTING */
```

See Also

strchr, strrchr

memcmp

compare objects

Synopsis

```
#include <string.h>
int memcmp(const void *s1, const void *s2, size_t n);
```

Description

The `memcmp` function compares the first `n` characters of the objects pointed to by `s1` and `s2`. It returns a positive value if the `s1` object is lexically greater than the `s2` object, a negative value if the `s2` object is lexically greater than the `s1` object, and a zero if the objects are the same.

Error Conditions

The `memcmp` function does not return an error condition.

Example

```
#include <string.h>
char string1 = “ABC”;
char string2 = “BCD”;
int result;

result = memcmp (string1, string2, 3);      /* result < 0 */
```

See Also

strcmp, strcoll, strncmp

memcpy

copy characters from one object to another

Synopsis

```
#include <string.h>
void *memcpy(void *s1, const void *s2, size_t n);
```

Description

The `memcpy` function copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. The behavior of `memcpy` is undefined if the two objects overlap.

The `memcpy` function returns the address of `s1`.

Error Conditions

The `memcpy` function does not return an error condition.

Example

```
#include <string.h>
char *a = "SRC";
char *b = "DEST";

result=memcpy (b, a, 3);      /* *b="SRCT" */
```

See Also

memmove, strcpy, strncpy

memmove

copy characters between overlapping objects

Synopsis

```
#include <string.h>
void *memmove(void *s1, const void *s2, size_t n);
```

Description

The `memmove` function copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. The entire object is copied correctly even if the objects overlap.

The `memmove` function returns a pointer to `s1`.

Error Conditions

The `memmove` function does not return an error condition.

Example

```
#include <string.h>
char *ptr, *str = "ABCDE";

ptr = str + 2;
memmove(str, str, 3);      /*  *ptr = "ABC",  *str = "ABABC" */
```

See Also

memcpy, strcpy, strncpy

memset

set range of memory to a character

Synopsis

```
#include <string.h>
void *memset(void *s1, int c, size_t n);
```

Description

The `memset` function sets a range of memory to the input character `c`. The first `n` characters of `s1` are set to `c`.

The `memset` function returns a pointer to `s1`.

Error Conditions

The `memset` function does not return an error condition.

Example

```
#include <string.h>
char string1[50];
memset(string1, ‘\0’, 50);      /* set string1 to 0 */
```

See Also

memcpy

modf

separate integral and fractional parts

Synopsis

```
#include <math.h>
double modf(double f, double *fraction);
float modff (float f, float *fraction);
```

Description

The `modf` function separates the first argument into integral and fractional portions. The fractional portion is returned and the integral portion is stored in the object pointed to by the second argument. The integral and fractional portions have the same sign as the input.

Error Conditions

The `modf` function does not return an error condition.

Example

```
#include <math.h>
double y, n;

y = modf(-12.345, &n);      /* y = -0.345, n = -12.0 */
```

See Also

frexp

pow

raise to a power

Synopsis

```
#include <math.h>
double pow(double, double);
```

Description

The `pow` function computes the value of the first argument raised to the power of the second argument.

Error Conditions

A domain error occurs if the first argument is negative and the second argument cannot be represented as an integer. If the first argument is zero, the second argument is less than or equal to zero, and the result cannot be represented, `EDOM` is stored in `errno`.

Example

```
#include <math.h>
double z;
z = pow(4.0, 2.0);      /* z = 16.0 */
```

See Also

exp, ldexp

qsort

quicksort

Synopsis

```
#include <stdlib.h>
void qsort(void *base, size_t nelem, size_t size,
           int (*compare) (const void *, const void *));
```

Description

The `qsort` function sorts an array of `nelem` objects, pointed to by `base`. The size of each object is specified by `size`.

The contents of the array are sorted into ascending order according to a comparison function pointed to by `compare`, which is called with two arguments that point to the objects being compared. The function shall return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

If two elements compare as equal, their order in the sorted array is unspecified. The `qsort` function executes a binary-search operation on a pre-sorted array. Note that:

- `base` points to the start of the array
- `nelem` is the number of elements in the array
- `size` is the size of each element of the array.
- `compare` is a pointer to a function that is called by `qsort` to compare two elements of the array. The function should return a value less than, equal to, or greater than zero, according to whether the first argument is less than, equal to, or greater than the second.

Error Conditions

The `qsort` function does not return an error condition.

Example

```
#include <stdlib.h>
float a[10];

int compare_float (const void *a, const void *b)
{
   float aval = *(float *)a;
   float bval = *(float *)b;
   if (aval < bval)
      return -1;
   else if (aval == bval)
      return 0;
   else
      return 1;
}

qsort (a, sizeof (a)/sizeof (a[0]), sizeof (a[0]), compare_float);
```

See Also

bsearch

raise

force a signal

Synopsis

```
#include <signal.h>
int raise(int sig);
```

Description

The `raise` function sends the signal `sig` to the executing program. The `raise` function forces interrupts wherever possible and simulates an interrupt otherwise.

Edge sensitive hardware interrupts are raised by setting the correct bit in the Interrupt Force and Clear (`IFC`) Register. Setting this bit forces a full interrupt and causes the program execution to change to the interrupt vector address for `sig`. All other interrupts and signals are raised by calling the handler (if set) for `sig`, directly from `raise`.

The `sig` argument must be one of the signals listed in priority order in Table 3-32.

Table 3-32. Raise Function Signals - Values and Meanings

Sig Value	Definition
`SIGPWRDWN`	Power down interrupt
`SIGIRQ2`	Interrupt 2
`SIGIRQL1`	Interrupt 1 (level sensitive)
`SIGIRQL0`	Interrupt 0 (level sensitive)
`SIGSPORT0XMIT`	Signal Sport 0 transmit
`SIGSPORT0RECV`	Signal Sport 0 receive
`SIGIRQE`	Level sensitive

Table 3-32. Raise Function Signals - Values and Meanings (Cont'd)

Sig Value	Definition
`SIGBDMA`	Byte DMA interrupt
`SIGSPORT1XMIT`	Signal Sport 1 transmit
`SIGIRQ1`	Interrupt 1
`SIGSPORT1RECV`	Signal Sport 1 receive
`SIGIRQ0`	Interrupt 0
`SIGTIMER`	Timer interrupt
`SIGABRT`	Software abort signal
`SIGILL`	Software illegal signal
`SIGINT`	Software segmentation signal
`SIGTERM`	Software termination signal
`SIGFPE`	Software floating point exception signal

Interrupts are *not* nested by the default start-up file.

Error Conditions

The `raise` function returns a zero if successful, a non-zero value if it fails.

Example

```
#include <signal.h>
raise(SIGABRT);
```

See Also

interrupt, signal

rand

random number generator

Synopsis

```
#include <stdlib.h>
int rand(void);
```

Description

The `rand` function returns a pseudo-random integer value in the range $[0, 2^{15} - 1]$.

For this function, the measure of randomness is its periodicity, the number of values it is likely to generate before repeating a pattern. The output of the pseudo-random number generator has a period in the order of $2^{15} - 1$.

Error Conditions

The `rand` function does not return an error condition.

Example

```
#include <stdlib.h>
int i;

i = rand();
```

See Also

srand

realloc

change memory allocation

Synopsis

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

Description

The realloc function changes the memory allocation of the object pointed to by ptr to size. Initial values for the new object are taken from those in the object pointed to by ptr. If the size of the new object is greater than the size of the object pointed to by ptr, then the values in the newly allocated section are undefined.

If ptr is a non-null pointer that was not allocated with malloc or calloc, the behavior is undefined. If ptr is a null pointer, realloc imitates malloc. If size is zero and ptr is not a null pointer, realloc imitates free.

Error Conditions

If memory cannot be allocated, ptr remains unchanged and realloc returns a null pointer.

Example

```
#include <stdlib.h>
int *ptr;

ptr = (int *)malloc(10);            /* intervening code   */
ptr = (int *)realloc(ptr, 20);      /* the size is now 20 */
```

See Also

calloc, free, malloc

setjmp

define a run-time label

Synopsis

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

Description

The `setjmp` function saves the calling environment in the `jmp_buf` argument. The effect of the call is to declare a run-time label that can be jumped to via a subsequent call to `longjmp`.

When `setjmp` is called, it immediately returns with a result of zero to indicate that the environment has been saved in the `jmp_buf` argument. If, at some later point, `longjmp` is called with the same `jmp_buf` argument, `longjmp` will restore the environment from the argument. The execution will then resume at the statement immediately following the corresponding call to `setjmp`. The effect is as if the call to `setjmp` has returned for a second time but this time the function returns a non-zero result.

The effect of calling `longjmp` will be undefined if the function that called `setjmp` has returned in the interim.

Error Conditions

The `setjmp` function does not return an error condition.

Example

See code example for "longjmp" on page 3-99.

See Also

longjmp

signal

define signal handling

Synopsis

```
#include <signal.h>
void (*signal(int sig, void (*func)(int))) (int);
void (*signalf(int sig, void (*func)(int))) (int);
void (*signals(int sig, void (*func)())) ();
```

Description

These functions are Analog Devices extensions to the ANSI standard.

The `signal` function determines how a signal received during program execution is handled. The `signal` functions cause a single execution the function pointed to by `func`; the `interrupt` functions cause the function to be executed for every interrupt.

The different variants of the `signal` functions differentiate between handler dispatching functions. The variants will be appropriate for some applications and provide improved efficiency. The default `signal` function dispatcher saves and restores all scratch registers and modes on the data stack around a call to the handler (`func`) when servicing an interrupt. This dispatcher will pass the interrupt ID (for example, `SIG_PWRDWN`) to the handler as its parameter.

The `signalf` interrupt dispatcher does the same as `interrupt`, except it switches between primary and secondary register sets to save and restore registers instead of using the data stack. The `signalf` function cannot be used in applications where nested interrupts are enabled. This interrupt dispatcher will pass the interrupt ID to the handler as its parameter.

The `signals` interrupt dispatcher saves and restores only the smallest number of registers and modes required to determine if a handler has been registered and to call that handler. The handler passed as input to `signals` must be declared using the `#pragma interrupt` directive (see on page 1-102). The `altregisters` directive (see on page 1-103) may be used in conjunction with the `interrupt` pragma in the definition of the handler. This dispatcher will *not* pass the interrupt ID to the handler.

The `sig` argument must be one of the signals listed in highest to lowest priority of interrupts in Table 3-33.

Table 3-33. Signal Function Signals - Values and Meanings

Sig Value	Definition
`SIGPWRDWN`	Power down interrupt
`SIGIRQ2`	Interrupt 2
`SIGIRQL1`	Interrupt 1 (level sensitive)
`SIGIRQL0`	Interrupt 0 (level sensitive)
`SIGSPORT0XMIT`	Signal Sport 0 transmit
`SIGSPORT0RECV`	Signal Sport 0 receive
`SIGIRQE`	Level sensitive
`SIGBDMA`	Byte DMA interrupt
`SIGSPORT1XMIT`	Signal Sport 1 transmit
`SIGIRQ1`	Interrupt 1
`SIGSPORT1RECV`	Signal Sport 1 receive
`SIGIRQ0`	Interrupt 0
`SIGTIMER`	Timer interrupt
`SIGABRT`	Software abort signal
`SIGILL`	Software illegal signal
`SIGINT`	Software segmentation signal
`SIGTERM`	Software termination signal
`SIGFPE`	Software floating point exception signal

The `signal` function causes the receipt of the signal number `sig` to be handled in one of the following ways:

- `SIG_DFL`—The default action is taken.
- `SIG_IGN`—The signal is ignored.
- Function address—The function pointed to by `func` is executed. The function pointed to by `func` is executed once when the signal is received. Handling is then returned to the default state.

Interrupts are *not* nested by the default start-up file.

Error Conditions

The `signal` function returns `SIG_ERR` and sets `errno` to `SIG_ERR` if it does not recognize the requested signal.

Example

```
#include <signal.h>

void handler (int sig) {    /* Interrupt Service Routine (ISR) */
}

main () {

    /* enable power down interrupt and register ISR */
    signal(SIG_PWRDWN, handler);

    /* disable power down interrupt */
    signal(SIG_PWRDWN, SIG_IGN);

    /* enable power down interrupt and register ISR */
    signalf(SIG_PWRDWN, handler);

    /* disable power down interrupt */
    signalf(SIG_PWRDWN, SIG_IGN);
}
```

See Also

interrupt, raise

sin

sine

Synopsis

```
#include <math.h>
double sin(double x);
float sinf (float x);
fract16 sin_fr16 (fract16 x);
```

Description

The `sin` function returns the sine of the argument `x`. The input is interpreted as a radian; the output is in the range [-1, 1].

The `sin_fr16` function inputs a fractional value in the range [-1.0, 1.0) corresponding to [-π/2, π/2]. The domain represents half a cycle which can be used to derive a full cycle if required (see Notes below). The result, in radians, is in the range [-1.0, 1.0).

Error Conditions

The `sin` function does not return an error condition.

Example

```
#include <math.h>
double y;
y = sin(3.14159);       /* y = 0.0 */
```

Notes

The domain of the `sin_fr16` function is restricted to the fractional range [`0x8000`, `0x7fff`] which corresponds to half a period from –(π/2) to π/2. It is possible however to derive the full period using the following properties of the function.

```
sine [0, π/2] = -sine [π, 3/2 π]

sine [-π/2, 0] = -sine [π/2, π]
```

The function below uses these properties to calculate the full period (from 0 to 2π) of the `sine` function using an input domain of `[0, 0x7fff]`.

```
#include <math.h>

fract16 sin2pi_fr16 (fract16 x)
{
   if (x < 0x2000) {                   /* <0.25   */
      /* first quadrant [0..π/2):              */
      /* sin_fr16([0x0..0x7fff]) = [0..0x7fff)    */
      return sin_fr16(x * 4);

   } else if (x < 0x6000) {            /* < 0.75   */
      /* if (x < 0x4000)                        */
      /* second quadrant [π/2..π):              */
      /* -sin_fr16([0x8000..0x0)) = [0x7fff..0)   */
      /*                                        */
      /* if (x < 0x6000)                        */
      /* third quadrant [π..3/2π):              */
      /* -sin_fr16([0x0..0x7fff]) = [0..0x8000)   */
      return -sin_fr16((0xc000 + x) * 4);

   } else {
      /* fourth quadrant [3/2π..π):             */
      /* sin_fr16([0x8000..0x0)) = [0x8000..0)    */
      return sin_fr16((0x8000 + x) * 4);
   }
}
```

See Also

asin, cos

sinh

hyperbolic sine

Synopsis

```
#include <math.h>
double sinh(double);
float sinhf(float);
```

Description

The `sinh` function returns the hyperbolic sine of the input parameter.

Error Conditions

The `sinh` function returns the IEEE constant `+Inf` if the argument is outside the domain.

Example

```
#include <math.h>
double x,y;
y = sinh(x);
```

See Also

cosh

sqrt

square root

Synopsis

```
#include <math.h>
double sqrt(double);
fract16 sqrt_fr16 (fract16);
```

Description

The `sqrt` function returns the positive square root of the input parameter.

Error Conditions

The `sqrt` function returns a zero for a negative input.

Example

```
#include <math.h>
double y;
y = sqrt(2.0);      /* y = 1.414..... */
```

See Also

No references to this function.

srand

random number seed

Synopsis

```
#include <stdlib.h>
void srand(unsigned int seed);
```

Description

The `srand` function is used to set the seed value for the `rand` function. A particular seed value always produces the same sequence of pseudo-random numbers.

Error Conditions

The `srand` function does not return an error condition.

Example

```
#include <stdlib.h>

srand(22);
```

See Also

rand

strcat

concatenate strings

Synopsis

```
#include <string.h>
char *strcat(char *s1, const char *s2);
```

Description

The `strcat` function appends a copy of the null-terminated string pointed to by `s2` to the end of the null-terminated string pointed to by `s1`. It returns a pointer to the new `s1` string, which is null-terminated. The behavior of `strcat` is undefined if the two strings overlap.

Error Conditions

The `strcat` function does not return an error condition.

Example

```
#include <string.h>
char string1[50];

string1[0] = ‘A’;
string1[1] = ‘B’;
string1[2] = ‘\0’;
strcat(string1, "CD");      /* new string is "ABCD" */
```

See Also

strncat

strchr

find first occurrence of character in string

Synopsis

```
#include <string.h>
char *strchr(const char *s1, int c);
```

Description

The `strchr` function returns a pointer to the first location in `s1`, a null-terminated string that contains the character `c`.

Error Conditions

The `strchr` function returns a null pointer if `c` is not part of the string.

Example

```
#include <string.h>
char *ptr1, *ptr2;

ptr1 = "TESTING";
ptr2 = strchr(ptr1, ‘E’);
       /* ptr2 points to the E in TESTING */
```

See Also

memchr, strrchr

strcmp

compare strings

Synopsis

```
#include <string.h>
int strcmp(const char *s1, const char *s2);
```

Description

The `strcmp` function lexicographically compares the null-terminated strings pointed to by `s1` and `s2`. It returns a positive value if the `s1` string is greater than the `s2` string, a negative value if the `s2` string is greater than the `s1` string, and a zero if the strings are the same.

Error Conditions

The `strcmp` function does not return an error condition.

Example

```
#include <string.h>
char string1[50], string2[50];

if (strcmp(string1, string2))
    printf("%s is different than %s \n", string1, string2);
```

See Also

memcmp, strncmp

strcoll

compare strings

Synopsis

```
#include <string.h>
int strcoll(const char *s1, const char *s2);
```

Description

The `strcoll` function compares the string pointed to by `s1` with the string pointed to by `s2`. The comparison is based on the locale macro, `LC_COLLATE`. Because only the C locale is defined in the ADSP-218x DSP environment, the `strcoll` function is identical to the `strcmp` function. The function returns a positive value if the `s1` string is greater than the `s2` string, a negative value if the `s2` string is greater than the `s1` string, and a zero if the strings are the same.

Error Conditions

The `strcoll` function does not return an error condition.

Example

```
#include <string.h>
char string1[50], string2[50];

if (strcoll(string1, string2))
    printf("%s is different than %s \n", string1, string2);
```

See Also

strcmp, strncmp

strcpy

copy from one string to another

Synopsis

```
#include <string.h>
void *strcpy(char *, const char *);
```

Description

The `strcpy` function copies the null-terminated string pointed to by `s2` into the space pointed to by `s1`. Memory allocated for `s1` must be large enough to hold `s2`, plus one space for the null character (‘`\0`’). The behavior of `strcpy` is undefined if the two objects overlap or if `s1` is not large enough. The `strcpy` function returns the new `s1`.

Error Conditions

The `strcpy` function does not return an error condition.

Example

```
#include <string.h>
char string1[50];

strcpy(string1, "SOMEFUN");
      /* SOMEFUN is copied into string1 */
```

See Also

memcpy, memmove, strncpy

strcspn

length of character segment in one string but not the other

Synopsis

```
#include <string.h>
size_t strcspn(const char *s1, const char *s2);
```

Description

The `strcspn` function returns the length of the initial segment of `s1` which consists entirely of characters not in the string pointed to by `s2`. The string pointed to by `s2` is treated as a set of characters. The order of the characters in the string is not significant.

Error Conditions

The `strcspn` function does not return an error condition.

Example

```
#include <string.h>
char *ptr1, *ptr2;
size_t len;

ptr1 = "Tried and Tested";
ptr2 = "aeiou";
len = strcspn (ptr1, ptr2);      /* len = 2 */
```

See Also

strlen, strspn

strerror

get string containing error message

Synopsis

```
#include <string.h>
char *strerror(int errnum);
```

Description

The `strerror` function returns a pointer to a string containing an error message by mapping the number in `errnum` to that string.

Error Conditions

The `strerror` function does not return an error condition.

Example

```
#include <string.h>
char *ptr1;

ptr1 = strerror(1);
```

See Also

No references to this function.

strlen

string length

Synopsis

```
#include <string.h>
size_t strlen(const char *s1);
```

Description

The `strlen` function returns the length of the null-terminated string pointed to by `s` (not including the terminating null character).

Error Conditions

The `strlen` function does not return an error condition.

Example

```
#include <string.h>
size_t len;

len = strlen("SOMEFUN");      /* len = 7 */
```

See Also

No references to this function.

strncat

concatenate characters from one string to another

Synopsis

```
#include <string.h>
char *strncat(char *s1, const char *s2, size_t n);
```

Description

The `strncat` function appends a copy of up to `n` characters in the null-terminated string pointed to by `s2` to the end of the null-terminated string pointed to by `s1`. It returns a pointer to the new `s1` string.

The behavior of `strncat` is undefined if the two strings overlap. The new `s1` string is terminated with a null (‘`\0`’).

Error Conditions

The `strncat` function does not return an error condition.

Example

```
#include <string.h>
char string1[50], *ptr;

string1[0]='\0';
ptr = strncat(string1, "MOREFUN", 4);
      /* string1 equals "MORE" */
```

See Also

strcat

strncmp

compare characters in strings

Synopsis

```
#include <string.h>
int strncmp(const char *s1, const char *s2, size_t n);
```

Description

The `strncmp` function lexicographically compares up to `n` characters of the null-terminated strings pointed to by `s1` and `s2`. It returns a positive value if the `s1` string is greater than the `s2` string, a negative value if the s2 string is greater than the `s1` string, and a zero if the strings are the same.

Error Conditions

The `strncmp` function does not return an error condition.

Example

```
#include <string.h>
char *ptr1;

ptr1 = "TEST1";
if (strncmp(ptr1, "TEST", 4) == 0)
    printf("%s starts with TEST \n", ptr1);
```

See Also

memcmp, strcmp

strncpy

copy characters from one string to another

Synopsis

```
#include <string.h>
char *strncpy(char *s1, const char *s2, size_t n);
```

Description

The `strncpy` function copies up to `n` characters of the null-terminated string pointed to by `s2` into the space pointed to by `s1`. If the last character copied from `s2` is not a null, the result does not end with a null. The behavior of `strncpy` is undefined when the two objects overlap. The `strncpy` function returns the new `s1`.

If the `s2` string contains fewer than `n` characters, the `s1` string is padded with the null character until all `n` characters have been written.

Error Conditions

The `strncpy` function does not return an error condition.

Example

```
#include <string.h>
char string1[50];

strncpy(string1, “MOREFUN”, 4);
                        /* MORE is copied into string1 */
string1[4] = ‘\0’;      /* must null-terminate string1 */
```

See Also

memcpy, memmove, strcpy

strpbrk

find character match in two strings

Synopsis

```
#include <string.h>
char *strpbrk(const char *s1, const char *s2);
```

Description

The `strpbrk` function returns a pointer to the first character in `s1` that is also found in `s2`. The string pointed to by `s2` is treated as a set of characters. The order of the characters in the string is not significant.

Error Conditions

In the event that no character in `s1` matches any in `s2`, a null pointer is returned.

Example

```
#include <string.h>
char *ptr1, *ptr2, *ptr3;

ptr1 = "TESTING";
ptr2 = "SHOP"
ptr3 = strpbrk(ptr1, ptr2);
       /* ptr3 points to the S in TESTING */
```

See Also

strspn

strrchr

find last occurrence of character in string

Synopsis

```
#include <string.h>
char *strrchr(const char *s1, int c);
```

Description

The `strrchr` function returns a pointer to the last occurrence of character `c` in the null-terminated input string `s1`.

Error Conditions

The `strrchr` function returns a null pointer if `c` is not found.

Example

```
#include <string.h>
char *ptr1, *ptr2;

ptr1 = "TESTING”;
ptr2 = strrchr(ptr1, ‘T’);
       /* ptr2 points to the second T of TESTING */
```

See Also

memchr, strchr

strspn

length of segment of characters in both strings

Synopsis

```
#include <string.h>
size_t strspn(const char *s1, const char *s2);
```

Description

The `strspn` function returns the length of the initial segment of `s1` which consists entirely of characters in the string pointed to by `s2`. The string pointed to by `s2` is treated as a set of characters. The order of the characters in the string is not significant.

Error Conditions

The `strspn` function does not return an error condition.

Example

```
#include <string.h>
size_t len;
char *ptr1, *ptr2;

ptr1 = "TESTING";
ptr2 = "ERST";
len = strspn(ptr1, ptr2);      /* len = 4 */
```

See Also

strcspn, strlen

strstr

find string within string

Synopsis

```
#include <string.h>
char *strstr(const char *s1, const char *s2);
```

Description

The `strstr` function returns a pointer to the first occurrence in the string pointed to by `s1` of the characters in the string pointed to by `s2`. This excludes the terminating null character in `s1`.

Error Conditions

If the string is not found, `strstr` returns a null pointer. If `s2` points to a string of zero length, `s1` is returned.

Example

```
#include <string.h>
char *ptr1, *ptr2;

ptr1 = "TESTING";
ptr2 = strstr(ptr1, "E");
       /* ptr2 points to the E in TESTING */
```

See Also

strchr

strtod

convert string to double

Synopsis

```
#include <stdlib.h>
double strtod(const char *nptr, char **endptr)
```

Description

The `strtod` function extracts a value from the string pointed to by `nptr`, and returns the value as a `double`. The `strtod` function expects `nptr` to point to a string that represents either a decimal floating-point number or a hexadecimal floating-point number. Either form of number may be preceded by a sequence of whitespace characters (as determined by the `isspace` function) that the function ignores.

A decimal floating-point number has the form:

```
[sign] [digits] [.digits] [{e|E} [sign] [digits]]
```

The `sign` token is optional and is either plus (+) or minus (–); and digits are one or more decimal digits. The sequence of digits may contain a decimal point (.).

The decimal digits can be followed by an exponent, which consists of an introductory letter (`e` or `E`) and an optionally signed integer. If neither an exponent part nor a decimal point appears, a decimal point is assumed to follow the last digit in the string.

The form of a hexadecimal floating-point number is:

```
[sign] [{0x}|{0X}] [hexdigs] [.hexdigs] [{p|P} [sign] [digits]]
```

A hexadecimal floating-point number may start with an optional plus (+) or minus (–) followed by the hexadecimal prefix 0x or 0X. This character sequence must be followed by one or more hexadecimal characters that optionally contain a decimal point (.).

The hexadecimal digits are followed by a binary exponent that consists of the letter p or P, an optional sign, and a non-empty sequence of decimal digits. The exponent is interpreted as a power of two that is used to scale the fraction represented by the tokens [hexdigs] [.hexdigs].

The first character that does not fit either form of number will stop the scan. If endptr is not NULL, a pointer to the character that stopped the scan is stored at the location pointed to by endptr. If no conversion can be performed, the value of nptr is stored at the location pointed to by endptr.

Error Conditions

The strtod function returns a zero if no conversion can be made and a pointer to the invalid string is stored in the object pointed to by endptr. If the correct value results in an overflow, a positive or negative (as appropriate) HUGE_VAL is returned. If the correct value results in an underflow, 0 is returned. The ERANGE value is stored in errno in the case of either an overflow or underflow.

Example

```
#include <stdlib.h>
char *rem;
double dd;

dd = strtod ("2345.5E4 abc",&rem);
     /* dd = 2.3455E+7, rem = "abc" */

dd = strtod ("-0x1.800p+9,123",&rem);
     /* dd = -768.0, rem = ",123"   */
```

See Also

atof, strtol, strtoul

strtodf

convert string to float

Synopsis

```
#include <stdlib.h>
float strtodf(const char *nptr, char **endptr)
```

Description

The `strtodf` function extracts a value from the string pointed to by `nptr`, and returns the value as a `float`. The `strtodf` function expects `nptr` to point to a string that represents either a decimal floating-point number or a hexadecimal floating-point number. Either form of number may be preceded by a sequence of whitespace characters (as determined by the `isspace` function) that the function ignores.

A decimal floating-point number has the form:

```
[sign] [digits] [.digits] [{e|E} [sign] [digits]]
```

The `sign` token is optional and is either plus (+) or minus (–); and digits are one or more decimal digits. The sequence of digits may contain a decimal point (.).

The decimal digits can be followed by an exponent, which consists of an introductory letter (`e` or `E`) and an optionally signed integer. If neither an exponent part nor a decimal point appears, a decimal point is assumed to follow the last digit in the string.

The form of a hexadecimal floating-point number is:

```
[sign] [{0x}|{0X}] [hexdigs] [.hexdigs] [{p|P} [sign] [digits]]
```

A hexadecimal floating-point number may start with an optional plus (+) or minus (–) followed by the hexadecimal prefix `0x` or `0X`. This character sequence must be followed by one or more hexadecimal characters that optionally contain a decimal point (.).

The hexadecimal digits are followed by a binary exponent that consists of the letter `p` or `P`, an optional sign, and a non-empty sequence of decimal digits. The exponent is interpreted as a power of two that is used to scale the fraction represented by the tokens `[hexdigs] [.hexdigs]`.

The first character that does not fit either form of number will stop the scan. If `endptr` is not NULL, a pointer to the character that stopped the scan is stored at the location pointed to by `endptr`. If no conversion can be performed, the value of `nptr` is stored at the location pointed to by `endptr`.

Error Conditions

The `strtodf` function returns a zero if no conversion can be made and a pointer to the invalid string is stored in the object pointed to by `endptr`. If the correct value results in an overflow, a positive or negative (as appropriate) `HUGE_VAL` is returned. If the correct value results in an underflow, 0 is returned. The `ERANGE` value is stored in `errno` in the case of either an overflow or underflow.

Example

```
#include <stdlib.h>
char *rem;
float f;

f = strtodf ("2345.5E4 abc",&rem);
       /* f = 2.3455E+7, rem = "abc" */
f = strtodf ("-0x1.800p+9,123",&rem);
       /* f = -768.0, rem = ",123    */
```

See Also

atof, strtol, strtoul

strtok

convert string to tokens

Synopsis

```
#include <string.h>
char *strtok(char *s1, const char *s2);
```

Description

The `strtok` function returns successive tokens from the string `s1`, where each token is delimited by characters from `s2`.

A call to `strtok` with `s1` not NULL returns a pointer to the first token in `s1`, where a token is a consecutive sequence of characters not in `s2`. `s1` is modified in place to insert a null character at the end of the token returned. If `s1` consists entirely of characters from `s2`, NULL is returned.

Subsequent calls to `strtok` with `s1` equal to NULL will return successive tokens from the same string. When the string contains no further tokens, NULL is returned. Each new call to `strtok` may use a new delimiter string, even if `s1` is NULL, in which case the remainder of the string is tokenized using the new delimiter characters.

Error Conditions

The `strtok` function returns a null pointer if there are no tokens remaining in the string.

Example

```
#include <string.h>
static char str[] = "a phrase to be tested, today";
char *t;

t = strtok(str, " ");      /* t points to "a"        */
t = strtok(NULL, " ");     /* t points to "phrase"   */
```

```
t = strtok(NULL, ",");    /* t points to "to be tested"  */
t = strtok(NULL, ".");    /* t points to " today"        */
t = strtok(NULL, ".");    /* t = NULL                    */
```

See Also

No references to this function.

strtol

convert string to long integer

Synopsis

```
#include <stdlib.h>
long int strtol(const char *nptr, char **endptr, int base);
```

Description

The `strtol` function returns as a `long int` the value that was represented by the string `nptr`. If `endptr` is not a null pointer, `strtol` stores a pointer to the unconverted remainder in `*endptr`.

The `strtol` function breaks down the input into three sections: white space (as determined by `isspace`), the initial characters, and unrecognized characters, including a terminating null character. The initial characters may be composed of an optional sign character, `0x` or `0X` if `base` is 16, and those letters and digits which represent an integer with a radix of `base`. The letters (`a-z` or `A-Z`) are assigned the values 10 to 35, and their use is permitted only when those values are less than the value of `base`.

If `base` is zero, then the base is taken from the initial characters. A leading `0x` indicates base 16; a leading 0 indicates base 8. For any other leading characters, base 10 is used. If `base` is between 2 and 36, it is used as a base for conversion.

Error Conditions

The `strtol` function returns a zero if no conversion can be made and the invalid string is stored in the object pointed to by `endptr`. If the correct value results in an overflow, positive or negative (as appropriate) `LONG_MAX` is returned. If the correct value results in an underflow, `LONG_MIN` is returned. `ERANGE` is stored in `errno` in the case of either overflow or underflow.

Example

```
#include <stdlib.h>
#define base 10
char *rem;
long int i;

i = strtol("2345.5", &rem, base);
    /* i=2345, rem=".5" */
```

See Also

atoi, atol, strtoul

strtoul

convert string to unsigned long integer

Synopsis

```
#include <stdlib.h>
unsigned long int strtoul(const char *nptr, char **endptr, int base);
```

Description

The `strtoul` function returns as an `unsigned long int` the value represented by the string `nptr`. If `endptr` is not a null pointer, `strtoul` stores a pointer to the unconverted remainder in `*endptr`.

The `strtoul` function breaks down the input into three sections:

- white space (as determined by `isspace`)
- initial characters
- unrecognized characters including a terminating null character.

The initial characters may be composed of an optional sign character, `0x` or `0X` if `base` is 16, and those letters and digits which represent an integer with a radix of `base`. The letters (`a-z` or `A-Z`) are assigned the values 10 to 35, and their use is permitted only when those values are less than the value of `base`.

If `base` is zero, then the base is taken from the initial characters. A leading `0x` indicates base 16; a leading 0 indicates base 8. For any other leading characters, base 10 is used. If `base` is between 2 and 36, it is used as a base for conversion.

Error Conditions

The `strtoul` function returns a zero if no conversion can be made and a pointer to the invalid string is stored in the object pointed to by `endptr`. If the correct value results in an overflow, `ULONG_MAX` is returned. `ERANGE` is stored in `errno` in the case of overflow.

Example

```
#include <stdlib.h>
#define base 10

char *rem;
unsigned long int i;

i = strtoul("2345.5", &rem, base);
    /* i = 2345, rem = ".5" */
```

See Also

atoi, atol, strtoul

strxfrm

transform string using `LC_COLLATE`

Synopsis

```
#include <string.h>
size_t strxfrm(char *s1, const char *s2, size_t n);
```

Description

The `strxfrm` function transforms the string pointed to by `s2` using the locale specific category `LC_COLLATE`. It places the result in the array pointed to by `s1`.

The function returns the length of the transformed string (not including the terminating null character). If `n` is zero and `s1` is set to the null pointer, then `strxfrm` will return the number of characters required for the transformed string. Overlapping strings are not supported

The transformation is such that `strcmp` will return the same result for two transformed strings as `strcoll` would for the same original strings. However, because only the C locale is defined in the ADSP-218x DSP environment, the `strxfrm` function is similar to the `strncpy` function except that the null character is always appended at the end of the output string.

Error Conditions

The `strxfrm` function does not return an error condition.

Example

```
#include <string.h>
char string1[50];
strxfrm(string1, "SOMEFUN", 49);
      /* SOMEFUN is copied into string1 */
```

See Also

strcmp, strcoll, strncpy

sysreg_read

read from non-memory-mapped register

Synopsis

```
#include <sysreg.h>
int sysreg_read(const int)
```

Description

The `sysreg_read` function causes the compiler to emit instructions to read the non-memory-mapped register, which is passed as a parameter, and to set the value read from that register as a return value.

The input parameter for `sysreg_read` should be a member of the `SysReg` enumeration defined in `sysreg.h`. This enumeration is used to map the non-memory-mapped actual registers to a small constant defined as a user friendly name. The enumerated variables defined are:

`sysreg_ASTAT`	arithmetic status
`sysreg_SSTAT`	shifter status
`sysreg_MSTAT`	multiplier status
`sysreg_ICNTL`	interrupt control
`sysreg_IMASK`	interrupts enabled mask
`sysreg_IFC`	interrupt force and clear

The `sysreg_read` function is implemented as a compiler built-in; the emitted instructions will be inlined at the point of `sysreg_read` use.

The inclusion of the `sysreg.h` include file is mandatory when using `sysreg_read`.

Error Conditions

The `sysreg_read` function does not return, raise, or set any error conditions.

Example

```
#include <sysreg.h>

main(){
    int value = sysreg_read(sysreg_IMASK);
}
```

See Also

disable_interrupts, enable_interrupts, io_space_read, io_space_write, sysreg_write

sysreg_write

write to non-memory-mapped register

Synopsis

```
#include <sysreg.h>
void sysreg_write (const int, const unsigned int);
```

Description

The `sysreg_write` function causes the compiler to emit instructions to write the non-memory-mapped register, which is passed as the first parameter, with the value, passed as the second parameter.

The first parameter for `sysreg_write` should be a member of the `SysReg` enumeration defined in `sysreg.h`. This enumeration is used to map the non-memory-mapped actual registers to a small constant, which is defined as a user friendly name. The enumerated variables defined are:

`sysreg_ASTAT`	arithmetic status
`sysreg_SSTAT`	shifter status
`sysreg_MSTAT`	multiplier status
`sysreg_ICNTL`	interrupt control
`sysreg_IMASK`	interrupts enabled mask
`sysreg_IFC`	interrupt force and clear

The `sysreg_write` function is implemented as a compiler built-in; the emitted instructions will be inlined at the point of `sysreg_write` use.

The inclusion of the `sysreg.h` include file is mandatory when using `sysreg_write`.

Error Conditions

The `sysreg_write` function does not return, raise, or set any error conditions.

Example

```
#include <sysreg.h>

main(){
    sysreg_write(sysreg_IMASK, 0x1);
}
```

See Also

disable_interrupts, enable_interrupts, io_space_read, io_space_write, sysreg_read

tan

tangent

Synopsis

```
#include <math.h>
double tan(double x);
float tanf(float x);
fract16 tan_fr16 (fract16 x);
```

Description

The tan function returns the tangent of the argument x. The input, in radians, must be in the range [-9099, 9099].

The tan_fr16 function is only defined for input values between -π/4 (=0x9B78) and π/4 (=0x6488). The input argument is in radians. Output values range from 0x8000 to 0x7FFF. The library function returns zero for any input argument that is outside the defined domain.

Error Conditions

The tan function returns zero if the input argument is outside the defined domain.

Example

```
#include <math.h>
double y;
y = tan(3.14159/4.0);      /* y = 1.0 */
```

See Also

atan, atan2

tanh

hyperbolic tangent

Synopsis

```
#include <math.h>
double tanh(double x);
float tanhf(float x);
```

Description

The `tanh` function returns the hyperbolic tangent of the argument `x`.

Error Conditions

The `tanh` function does not return an error condition.

Example

```
#include <math.h>
double x,y;
y = tanh(x);
```

See Also

cosh, sinh

timer_off

disable ADSP-218x DSP timer

Synopsis

```
#include <misc.h>
unsigned int timer_off(void);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `timer_off` function disables the ADSP-218x DSP timer and returns the current value of the `TCOUNT` register.

Error Conditions

The `timer_off` function does not return an error condition.

Example

```
#include <misc.h>
unsigned int hold_tcount;

hold_tcount = timer_off();
    /* hold_tcount contains value of TCOUNT */
    /* register AFTER timer has stopped     */
```

See Also

timer_on, timer_set

timer_on

enable ADSP-218x DSP timer

Synopsis

```
#include <misc.h>
unsigned int timer_on(void);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `timer_on` function enables the ADSP-218x DSP timer and returns the current value of the `TCOUNT` register.

Error Conditions

The `timer_on` function does not return an error condition.

Example

```
#include <misc.h>
unsigned int hold_tcount;

hold_tcount = timer_on();
     /* hold_tcount contains value of TCOUNT */
     /* register when timer starts           */
```

See Also

timer_off, timer_set

timer_set

initialize ADSP-218x DSP timer

Synopsis

```
#include <misc.h>
int timer_set(unsigned int tperiod,
              unsigned int tcount, int tscale);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `timer_set` function sets the ADSP-218x DSP timer registers `TPERIOD` and `TCOUNT`. The function returns a `1` if the timer is enabled, a `0` if the timer is disabled.

The `TSCALE` value is used to set the `TSCALE` register. For a complete description of the ADSP-218x DSP timer, refer to the *ADSP-218x DSP Hardware Reference.*

Each interrupt call takes approximately 50 cycles on entrance and 50 cycles on return. If `tperiod` and `tcount` are set too low, you may incur initializing overhead that could create an infinite loop.

Error Conditions

The `timer_set` function does not return an error condition.

Example

```
#include <misc.h>

if(timer_set(1000, 1000,1) != 1)
   timer_on();     /* enable timer */
```

See Also

timer_off, timer_on

tolower

convert from uppercase to lowercase

Synopsis

```
#include <ctype.h>
int tolower(int c);
```

Description

The `tolower` function converts the input character to lowercase if it is uppercase; otherwise, it returns the character.

Error Conditions

The `tolower` function does not return an error condition.

Example

```
#include <ctype.h>
int ch;

for (ch=0; ch<=0x7f; ch++) {
    printf("%#04x", ch);
    if(isupper(ch))
       printf("tolower=%#04x", tolower(ch));
    putchar('\n');
}
```

See Also

islower, isupper, toupper

toupper

convert from lowercase to uppercase

Synopsis

```
#include <ctype.h>
int toupper(int c);
```

Description

The `toupper` function converts the input character to uppercase if it is in lowercase; otherwise, it returns the character.

Error Conditions

The `toupper` function does not return an error condition.

Example

```
#include <ctype.h>
int ch;

for (ch=0; ch<=0x7f; ch++) {
    printf("%#04x", ch);
    if(islower(ch))
       printf("toupper=%#04x", toupper(ch));
    putchar('\n');
}
```

See Also

islower, isupper, tolower

va_arg

get next argument in variable-length list of arguments

Synopsis

```
#include <stdarg.h>
void va_arg(va_list ap, type);
```

Description

The `va_arg` macro is used to walk through the variable length list of arguments to a function.

After starting to process a variable-length list of arguments with `va_start`, call `va_arg` with the same `va_list` variable to extract arguments from the list. Each call to `va_arg` returns a new argument from the list.

Substitute a `type` name corresponding to the type of the next argument for the `type` parameter in each call to `va_arg`. After processing the list, call `va_end`.

The header file `stdarg.h` defines a pointer type called `va_list` that is used to access the list of variable arguments.

The function calling `va_arg` is responsible for determining the number and types of arguments in the list. It needs this information to determine how many times to call `va_arg` and what to pass for the type parameter each time. There are several common ways for a function to determine this type of information. The standard C `printf` function reads its first argument looking for %-sequences to determine the number and types of its extra arguments. In the example below, all of the arguments are of the same type (`char*`), and a termination value (NULL) is used to indicate the end of the argument list. Other methods are also possible.

If a call to va_arg is made after all arguments have been processed, or if va_arg is called with a type parameter that is different from the type of the next argument in the list, the behavior of va_arg is undefined.

Error Conditions

The va_arg macro does not return an error condition.

Example

```
#include <stdarg.h>
#include <string.h>
#include <stdlib.h>

char *concat(char *s1,...)
{
   int len = 0;
   char *result;
   char *s;
   va_list ap;

   va_start (ap,s1);
   s = s1;
   while (s){
      len += strlen (s);
      s = va_arg (ap,char *);
   }
   va_end (ap);

   result = malloc (len +7);
   if (!result)
      return result;
   *result = '';
   va_start (ap,s1);
   s = s1;
   while (s){
      strcat (result,s);
      s = va_arg (ap,char *);
```

```
    }
    va_end (ap);
    return result;

}
```

See Also

va_end, va_start

va_end

finish variable-length argument list processing

Synopsis

```
#include <stdarg.h>
void va_end(va_list ap);
```

Description

The `va_end` macro can only be used after the `va_start` macro has been invoked. A call to `va_end` concludes the processing of a variable-length list of arguments that was begun by `va_start`.

Error Conditions

The `va_end` macro does not return an error condition.

Example

See "va_arg" on page 3-165.

See Also

va_arg, va_start

va_start

initialize the variable-length argument list processing

Synopsis

```
#include <stdarg.h>
void va_start(va_list ap, parmN);
```

Description

The `va_start` macro is used in a function declared to take a variable number of arguments to start processing those variable arguments. The first argument to `va_start` should be a variable of type `va_list`, which is used by `va_arg` to walk through the arguments.

The second argument is the name of the last *named* parameter in the function's parameter list; the list of variable arguments immediately follows this parameter. The `va_start` macro must be invoked before either the `va_arg` or `va_end` macro can be invoked.

Error Conditions

The `va_start` macro does not return an error condition.

Example

See "va_arg" on page 3-165.

See Also

va_arg, va_end

A COMPILER LEGACY SUPPORT

The VisualDSP++ environment and tools provide several types of support for legacy code that was developed with previous releases of the development tools. For more information on legacy code support, see the *VisualDSP++3.5 Linker and Utilities Manual for 16-Bit Processors* and V*isualDSP++ 3.5 C Assembler and Preprocessor Manual for ADSP-218x and ADSP-219x DSPs*.

Tools Differences

VisualDSP++ 3.5 includes an updated C compiler, linker, and debugger, and a binary file format, ELF. Due to use of the VisualDSP++ Integrated Development and Debugging Environment (IDDE) and other enhancements, VisualDSP++ 3.5 has significant differences from Release 6.1 that you will need to be aware of. In some cases you will need to modify your sources to use the new tools.

Of the new features and enhancements, the following have the most impact on your existing projects:

- Some tools' switches have changed. If you use any of the modified or obsolete switches, you must revise your command-line scripts or batch files in order to rebuild your project.
- The code generation tools no longer support AEXE-format DSP executables (`.EXE`). They now generate ELF-format DSP executables (`.DXE`), and the debugger requires DSP executables to be in the ELF/DWARF-2 format. As a result, AEXE-formatted files must be

recompiled or reassembled in order to be debugged under VisualDSP++ 3.5. An ELF/DWARF-to-AEXE conversion utility is available in VisualDSP++ 3.5 to perform back-conversion. An AEXE-to-ELF conversion utility performs forward conversion.

- Some assembly instructions and directives have changed from the VisualDSP 6.1 syntax, but a `-legacy` assembler switch has been provided to assemble files in the old syntax. You may need to review diagnostic messages and revise your source code in order to reassemble your source. Legacy syntax and the new syntax under VisualDSP++ 3.5 cannot be used together in the same source file. They can be mixed together within the same project, as long as they are assembled in different source files.

- Some C compiler extensions to the ISO/ANSI standard have changed. If you use any of the modified or removed extensions, you must revise your code in order to rebuild your project.

- The run-time model has changed. If you call a Release 6.1 assembly-language subroutine from your C program, you must revise the assembly code to comply with the new rules for the C run-time environment.

- The Architecture File (`.ACH`) is no longer supported. If you re-link using your Release 6.1 object files or object libraries, you must create a Linker Description File for each object or object library before using the new linker.

The remainder of this section describes these and other known differences between VisualDSP Releases 6.1 and VisualDSP++ 3.5. It also provides assistance when possible to make transformation to the new software easier.

C Compiler and C Run-Time Library

The new `cc218x` compiler provided in VisualDSP++ 3.5 does not support some switches and extensions that were available in the g21 compiler. As a result, the compiler supports a set of new rules for the run-time environment. This section lists the extensions and switches that have been removed, replaced, or whose function works differently than in Release 6.1. For further details about the `cc218x` compiler, see Chapter 1, "Compiler".

Segment Placement Support Keyword Changed to Section

The `segment()` placement keyword has changed to `section()`. The `section()` construct now precedes the variable declaration, and its argument is a string; for example:

```
section("my_sec") int myvar;
```

For more information about the section() construct, see "Placement Support Keyword (section)" on page 1-76.

G21 Compatibility Call

The C compiler provides a special `g21` compatibility call that enables use of existing libraries with the new compiler. The extern `OldAsmCall` declaration can be added to the prototype(s) of the functions developed under Release 6.1. Your programs will be faster, smaller, and more reliable after the C code is upgraded to use the new compiler.

ⓘ This convention is similar to the C linkage specification.

Support for G21-Based Options and Extensions

The C compiler supports most of the switches and extensions of the previous GNU-based compiler release. For a list of absolute or modified options, see "Compiler Switch Modifications" on page A-5.

Indexed Initializers

The syntax for indexed initializers may change in a future release. This change may be incompatible with the existing syntax. For more information about indexed initializers, see"Indexed Initializer Support" on page 1-80.

Compiler Diagnostics

Compiler diagnostics now go to `stderr` on the `PC`, rather than to `stdout`.

ANSI C Extensions

The following extensions are no longer supported:

- `typeof` — This extension was used to obtain the type of an expression.
- complex types: `complex`, `creal`, `cimag`, and `conj` — These extensions were used to define complex numbers. Although you cannot write complex number literals, you can have a complex type defined with real and imaginary components. These types need to be managed by the programmer.
- compound statements within expressions — This extension was used to declare variables within an expression. You can achieve similar results using in-line functions.
- iterator types: `iter`, `sum` — These extensions created loop expressions that were used as a shorthand for working with arrays.
- assigning variables to specific registers: `asm` — This extension was used to declare a variable and specify a machine register in which to store it.

If you used any of these extensions in your C source code, you must remove them if you modify or re-compile that source.

Compiler Switch Modifications

Compiler switches (listed in Table A-1) have been removed or their action has been modified. If you used any of these switches to compile your C code, you should remove or replace the switch before recompiling the code with the new C compiler.

Table A-1. Obsolete and Replaced Switches

Switch Name	Operation under Release 6.1	Change for VisualDSP++ 3.5
`-a`	Specify Architecture File.	Replaced with `-T` and used with a Linker Description File.
`-dD`, `-dM`, and `-dN`	Output the results of preprocessing and/or list of # defines.	Removed.
`-deps`	Generate dependencies list.	Removed.
`-fcond-mismatch`	Allow conditional expression mismatch.	Removed.
`-finline-functions`	Force function inlining.	Removed.
`-fkeep-inline-functions`	Force keeping of inlined functions.	Removed.
`-fno-asm`	Don't recognize asm as a keyword.	Replaced with `-no-extra-keywords.`
`-fno-builtin`	Don't recognize builtin functions.	Replaced with `-no-builtin.`
`-fsigned-bitfields` `-funsigned-bitfields`	Control whether bit field is signed or unsigned.	Removed. Bitfield is signed or unsigned based on the sign of the type definition declaring the bitfield.

`-fno-signed-bitfields`
`-fno-unsigned- bitields`
	Negative form of the `-fsigned-bitfield` and `-funsigned-bitfield.`	Replaced with `-signed-bitfield` and `-unsigned-bitfield`..

`-fsigned-char` `-funsigned-char`	Specify whether to default to signed or unsigned char type.	Replaced with `-signed-char` and `-unsigned-char.`

Table A-1. Obsolete and Replaced Switches (Cont'd)

Switch Name	Operation under Release 6.1	Change for VisualDSP++ 3.5
`-fsyntax_only`	Check syntax only; no output.	Replaced with `-syntax-only`.
`-fwritable-strings`	Store string constants in the writable data segment.	Removed. String constants are placed in the `seg_data1` section. Definition in the `.LDF` file can place this section in RAM or ROM.
`-imacros`	Process macro file.	Removed.
`-MD, -MM, and -MMD`	Output rules for the make utility; used with -E.	Removed.
`-mboot-page=`	Specify boot page.	Removed. The ADSP-218x DSP does not support paging.

`-mdmdata=`
`-mpmdata=`
`-mdcode=`
	Specify architecture file segments.	Removed.
		You can control placement of object file segments using the `.SECTION` directive in the `.LDF` file.

`-mlistm`	Merge C code with assembler-generated code.	Removed.
`-mno-doloops`	Don't generate loop structures in assembled code.	Removed.
`-mno-inits`	Don't initialize variables in assembled code.	Removed.
`-mpjump`	Place the jump table in `pm` memory.	Replaced with `-jump-{dm|pm}`.
`-mreserved=`	Instructs the compiler not to use specified registers.	Replaced with `-reserve`.
`-mrom`	Make the module a ROM module.	Removed.
`-msmall-code`	Optimize for size, not for speed.	Replaced with `-Os`.

Table A-1. Obsolete and Replaced Switches (Cont'd)

Switch Name	Operation under Release 6.1	Change for VisualDSP++ 3.5
`-mstatic-spill`	Use `dm` memory when all register are used.	Removed.
`-nostdinc`	Do not search standard system directories for header files.	Replaced with `-no-std-inc`.
`-nostdlib`	Do not use standard system libraries and startup files when linking.	Replaced with `-no-std-lib`.
`-runhdr`	Specify a particular run-time header.	Removed. This feature can now be defined in the `.LDF` file.
`-traditional-cpp`	Support some preprocessing features.	Removed.

Warnings

The VisualDSP++ 3.5 compiler includes several new warning switches. These switches control the number and type of messages reported during a given compilation. They are described in Table A-2.

Table A-2. Compiler -- New Warning Switches

VisualDSP ++ 3.5 Warning Switch	Description
`-warn-protos`	Produce a warning when a function is called without a full prototype.
`-Wdriver-limit` *`number`*	Set a maximum *`number`* of driver errors.
`-Werror-limit` *`number`*	Set a maximum *`number`* of compiler errors.
`-Wremarks`	Indicates that the compiler may issue remarks, which are diagnostic messages even milder than warnings.
`-Wterse`	Enable terse warnings.

Table A-2. Compiler -- New Warning Switches (Cont'd)

VisualDSP ++ 3.5 Warning Switch	Description
`-pedantic`	Causes the compiler to generate warnings for any constructs in a C source file that does not conform to the ANSI standard.
`-W{error|remark|suppress|warn}` `<num> [, num ...]`	Overrides the severity of specific compilation diagnostic messages, where *num* is the number representing the message to override.

The Release 6.1 warning switches that are no longer supported are listed in Table A-3.

Table A-3. Obsolete Warning Switches

`-W`	`-Wformat`	`-Wtrigraphs`
`-Wall`	`-Wimplicit`	`-Wuninitialized`
`-Wchar-subscripts`	`-Wparentheses`	`-Wunused`
`-Wcomment`	`-Wreturn-type`	
`-Werror`	`-Wswitch`	

See "Compiler Command-Line Reference" on page 1-6 for complete descriptions of current C compiler switches.

Run-Time Model

The `cc218x` compiler in VisualDSP++ 3.5 produces code that is not fully compatible with the Release 6.1 run-time model. However, the new compiler is superior in many ways to the old one, and your programs will be faster, smaller, and more reliable after the C code is converted to the new system.

VisualDSP++ 3.5 includes significant changes to the registers, stack usage, and other features of the run-time model; these changes are especially important if you wish to call assembly language subroutines from C programs, or C functions from assembly language programs. For complete details about the VisualDSP++ 3.5 run-time model, see "C Run-Time Model and Environment" on page 1-132.

I INDEX