

VISUALDSP⁺⁺™ 3.0

C Compiler and Library Manual for ADSP-218x DSPs

Revision 4.0, July 2002

Part Number:
82-000400-03

Analog Devices, Inc.
Digital Signal Processor Division
One Technology Way
Norwood, Mass. 02062-9106



Copyright Information

© 2002 Analog Devices, Inc., ALL RIGHTS RESERVED. This document may not be reproduced in any form without prior, express written consent from Analog Devices, Inc.

Printed in the USA.

Disclaimer

Analog Devices, Inc. reserves the right to change this product without prior notice. Information furnished by Analog Devices is believed to be accurate and reliable. However, no responsibility is assumed by Analog Devices for its use; nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under the patent rights of Analog Devices, Inc.

Trademark and Service Mark Notice

The Analog Devices logo, VisualDSP, the VisualDSP logo are registered trademarks; VisualDSP++, the VisualDSP++ logo, CROSSCORE, the CROSSCORE logo, EZ-KIT Lite, Apex-ICE, Mountain-ICE, Summit-ICE, Trek-ICE, and The DSP Collaborative are trademarks of Analog Devices, Inc.

All other brand and product names are trademarks or service marks of their respective owners.

CONTENTS

PREFACE

Purpose	xxi
Intended Audience	xxi
Manual Contents Description	xxii
What's New in this Manual	xxii
Technical or Customer Support	xxii
Supported Processors	xxiii
Product Information	xxiii
MyAnalog.com	xxiii
DSP Product Information	xxiv
Related Documents	xxv
Online Technical Documentation	xxv
From VisualDSP++	xxvi
From Windows	xxvi
Using Windows Explorer	xxvi
From the Web	xxvii
Printed Manuals	xxvii

CONTENTS

VisualDSP++ Documentation Set	xxvii
Hardware Manuals	xxvii
Datasheets	xxviii
Contacting DSP Publications	xxviii
Notation Conventions	xxix

COMPILER

C Compiler Overview	1-2
Standard Conformance	1-3
Compiler Command-Line Reference	1-6
Running the Compiler	1-7
Specifying Compiler Options in VisualDSP++	1-10
Compiler Command-Line Switches	1-12
-@ filename	1-18
-21{81 83 84 85 86 87 88 89}	1-18
-A name[tokens]	1-18
-alttok	1-19
-analog	1-20
-ansi	1-20
-build-lib	1-20
-C	1-20
-c	1-20
-Dmacro [=definition]	1-21
-debug-types	1-21
-default-linkage-{asm C}	1-22

-dollar	1-22
-dry	1-22
-dryrun	1-22
-E	1-23
-EE	1-23
-extra-keywords	1-23
-flags-{asm compiler lib link} switch [,switch2 [,...]	1-23
-g	1-24
-H	1-24
-HH	1-25
-h[elp]	1-25
-I-	1-25
-I directory [{, ;} directory...]	1-26
-include filename	1-26
-J	1-26
-jump-{dm pm}	1-26
-L directory [{, ;} directory...]	1-26
-l library	1-27
-M	1-27
-MM	1-27
-Mt filename	1-27
-MQ	1-28
-make-autostatic	1-28
-map filename	1-28

CONTENTS

-no-alttok	1-29
-no-builtin	1-29
-no-defs	1-29
-no-dir-warnings	1-29
-no-dollar	1-29
-no-extra-keywords	1-30
no-inline	1-30
-no-std-def	1-30
-no-std-inc	1-30
-no-std-lib	1-31
-no-widen-muls	1-31
-O	1-31
-Os	1-32
-o filename	1-32
-P	1-32
-path-{asm compiler def lib link} directory	1-32
-path-install directory	1-33
-path-output directory	1-33
-path-temp directory	1-33
-pch	1-33
-pchdir directory	1-33
-pedantic	1-34
-pedantic-errors	1-34
-pplist filename	1-34

-proc identifier	1-35
-R directory [{: ,}directory ...]	1-35
-R-	1-35
-reserve register[,register...]	1-36
-S	1-36
-s	1-36
-save-temps	1-36
-show	1-37
-signed-char	1-37
-syntax-only	1-37
-sysdefs	1-37
-T filename	1-38
-time	1-38
-traditional	1-39
-Umacro	1-39
-unsigned-char	1-39
-v	1-39
-val-global <name-list>	1-40
-verbose	1-40
-version	1-40
-Wdriver-limit number	1-40
-Werror-limit number	1-40
-W{error remark suppress warn} <num>[,num...]	1-41
-Wremarks	1-41

CONTENTS

-Wterse	1-41
-w	1-41
-warn-protos	1-42
-write-files	1-42
-write-opts	1-42
-xref <file>	1-42
Data Type Sizes	1-43
C Compiler Language Extensions	1-46
Inline Function Support Keyword (inline)	1-49
Inline Assembly Language Support Keyword (asm)	1-50
Assembly Construct Template	1-51
ASM() Construct Syntax:	1-51
ASM() Construct Syntax Rules	1-53
ASM() Construct Template Example	1-54
Assembly Construct Operand Description	1-55
Assembly Constructs With Multiple Instructions	1-57
Assembly Construct Reordering and Optimization	1-58
Assembly Constructs with Input and Output Operands	1-59
Assembly Constructs and Macros	1-60
Dual Memory Support Keywords (pm dm)	1-61
Memory Keywords and Assignments/Type Conversions	1-63
Memory Keywords and Function Declarations/Pointers	1-64
Memory Keywords and Function Arguments	1-65
Memory Keywords and Macros	1-65

Placement Support Keyword (section)	1-66
Boolean Type Support Keywords (bool, true, false)	1-67
Pointer Class Support Keyword (restrict)	1-67
Variable-Length Array Support	1-68
Non-Constant Aggregate Initializer Support	1-70
Indexed Initializer Support	1-70
Aggregate Constructor Expression Support	1-72
Preprocessor-Generated Warnings	1-73
C++-Style Comments	1-73
Compiler Built-in Functions	1-73
I/O Space for Read/Write	1-74
Read/Write of Non-Memory-Mapped Registers	1-75
Interrupt Control	1-76
ETSI Support	1-76
ETSI Support Overview	1-77
Calling ETSI Library Functions	1-79
Using the ETSI Built-In Functions	1-80
Linking ETSI Library Functions	1-80
Working with ETSI Library Source Code	1-80
ETSI Support for Data Types	1-81
ETSI Header File	1-81
Pragmas	1-84
Data Alignment Pragma - ALIGN NUM	1-84
Interrupt Handler Pragma	1-85

CONTENTS

Altregisters Pragma	1-86
Optimization Level Pragmas	1-86
Linkage Pragmas	1-87
linkage_name Identifier	1-87
weak_entry Pragma	1-87
Stack Usage Pragma	1-87
Preprocessor Features	1-89
Predefined Preprocessor Macros	1-89
__ADSP21XX__ and __ADSP218X__	1-89
__ADSP21{81 83 84 85 86 87 88 89}__	1-89
__ANALOG_EXTENSIONS__	1-90
__DATE__	1-90
__DOUBLES_ARE_FLOATS__	1-90
__ECC__	1-90
__EDG__	1-90
__EDG_VERSION__	1-90
__FILE__	1-90
__LANGUAGE_C	1-91
__LINE__	1-91
__NO_BUILTIN	1-91
__NO_LONG_LONG	1-91
__SIGNED_CHARS__	1-91
__STDC__	1-91
__STDC_VERSION__	1-91

__TIME__	1-92
Header Files	1-92
Writing Preprocessor Macros	1-92
Compound Statements as Macros	1-92
Preprocessing of .IDL Files	1-94
C Run-Time Model and Environment	1-96
Using the Run-Time Header	1-96
Interrupt Table and Interface	1-97
Autobuffering Support	1-98
Stack Frame	1-99
Stack Frame Description	1-100
General System-Wide Specifications	1-102
At a procedure call, the following must be true:	1-102
At an interrupt, the following must be true:	1-102
Return Values	1-103
Procedure Call and Return	1-103
On Entry:	1-103
To Return from a Procedure:	1-104
Miscellaneous Information	1-104
Register Classification	1-104
Callee Preserved Registers (“Preserved”)	1-104
Dedicated Registers	1-105
Caller Save Registers (“Scratch”)	1-105
Circular Buffer Length Registers	1-105

CONTENTS

Mode Status (MSTAT) Register	1-105
Complete List of Registers	1-106
C and Assembly Language Interface	1-109
Calling Assembly Subroutines from C Programs	1-109
Calling C Routines from Assembly Programs	1-112
Using Mixed C/Assembly Support Macros	1-113
function_entry	1-113
exit	1-114
leaf_entry	1-114
leaf_exit	1-114
alter(x)	1-114
save_reg	1-114
restore_reg	1-115
readsfirst(register)	1-115
register = readsnext	1-115
putsfirst = register	1-115
putsnext = register	1-115
getsfirst(register)	1-116
register = getsnext	1-116
Using Mixed C/Assembly Naming Conventions	1-116
Compatibility Call	1-117

LIBRARY

C Run-Time Library Guide	2-2
Calling Library Functions	2-2

Using the Compiler's Built-In Functions	2-2
Linking Library Functions	2-3
Working With Library Source Code	2-4
Working with Library Header Files	2-5
asm_sprt.h – Mixed C/Assembly Support	2-6
assert.h	2-7
ctype.h	2-7
def2181.h – Memory Map Definitions	2-7
errno.h	2-7
ffts.h – Fast Fourier Transforms	2-7
filters.h – DSP Filters	2-8
float.h – Floating Point	2-8
fract.h – ADSP-218x DSP Macro Fract Definitions	2-8
macros.h – Circular Buffers	2-8
math.h – Mathematics	2-9
misc.h – ADSP-218x DSP Timer Functions	2-9
signal.h – Signal Handling	2-9
sport.h – ADSP-218x DSP Serial Ports	2-9
stdio.h – Standard Input/Output	2-10
sysreg.h	2-10
Documented Library Functions	2-11
C Run-Time Library Reference	2-15
Notation Conventions	2-15
Reference Format	2-15

CONTENTS

abort	2-16
abs	2-17
acos	2-18
asin	2-19
atan	2-20
atan2	2-21
atexit	2-22
atof	2-23
atoi	2-24
atol	2-25
biquad	2-26
bsearch	2-29
calloc	2-31
ceil	2-32
clear_interrupt	2-33
copysign	2-35
cos	2-36
cosh	2-37
cot	2-38
demean_buffer	2-39
disable_interrupts	2-41
div	2-42
enable_interrupts	2-43
exit	2-44

exp	2-45
fabs	2-46
fftN	2-47
fir	2-50
floor	2-52
fmod	2-53
free	2-54
frexp	2-55
ifftN	2-56
iir	2-59
interrupt	2-63
io_space_read	2-66
io_space_write	2-67
isalnum	2-69
isalpha	2-70
iscntrl	2-71
isdigit	2-72
isgraph	2-73
isinf	2-74
islower	2-76
isnan	2-77
isprint	2-79
ispunct	2-80
isspace	2-81

CONTENTS

isupper	2-82
isxdigit	2-83
labs	2-84
ldexp	2-85
ldiv	2-86
log	2-87
log10	2-88
longjmp	2-89
malloc	2-91
memchr	2-92
memcmp	2-93
memcpy	2-94
memmove	2-95
memset	2-96
modf	2-97
pow	2-98
qsort	2-99
raise	2-101
rand	2-103
realloc	2-104
setjmp	2-105
signal	2-107
sin	2-110
sinh	2-111

<code>sqrt</code>	2-112
<code>srand</code>	2-113
<code>strcat</code>	2-114
<code>strchr</code>	2-115
<code>strcmp</code>	2-116
<code>strcoll</code>	2-117
<code>strcpy</code>	2-118
<code>strcspn</code>	2-119
<code>strerror</code>	2-120
<code>strlen</code>	2-121
<code>strncat</code>	2-122
<code>strncmp</code>	2-123
<code>strncpy</code>	2-124
<code>strpbrk</code>	2-125
<code>strrchr</code>	2-126
<code>trspns</code>	2-127
<code>strstr</code>	2-128
<code>strtod</code>	2-129
<code>strtodf</code>	2-131
<code>strtok</code>	2-133
<code>strtol</code>	2-135
<code>strtoul</code>	2-137
<code>strxfrm</code>	2-139
<code>sysreg_read</code>	2-141

CONTENTS

sysreg_write	2-143
tan	2-145
tanh	2-146
timer_off	2-147
timer_on	2-148
timer_set	2-149
tolower	2-150
toupper	2-151
va_arg	2-152
va_end	2-154
va_start	2-155

COMPILER LEGACY SUPPORT

Tools Differences	A-1
C Compiler and C Run-Time Library	A-3
Segment Placement Support Keyword Changed to Section ...	A-3
G21 Compatibility Call	A-3
Support for G21-Based Options and Extensions	A-3
Indexed Initializers	A-4
Compiler Diagnostics	A-4
ANSI C Extensions	A-4
Compiler Switch Modifications	A-5
Warnings	A-7
Run-Time Model	A-9

CONTENTS

INDEX

CONTENTS

PREFACE

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

Purpose

The *VisualDSP++ 3.0 C Compiler and Library Manual for ADSP-218x DSPs* contains information about the C compiler and run-time library program 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 DSP programmers who are familiar with Analog Devices DSPs. This manual assumes that the audience has a working knowledge of the ADSP-218x DSPs architecture and instruction set and the C instruction set.

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, “[Library](#)”

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

What's New in this Manual

This edition of the *VisualDSP++ 3.0 C Compiler and Library Manual for ADSP-218x DSPs* documents support for all ADSP-218x processors.

In addition to documenting all existing compiler features, this manual describes new features including: optimization pragmas, circular buffer intrinsics, source annotations, complex maths functions, .IDL file preprocessing, and other enhancements.

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.
DSP Division
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 DSP, ADSP-2192-12 DSP, ADSP-2195 DSP,
ADSP-2196 DSP, ADSP-21990 DSP, ADSP-21991 DSP, and
ADSP-21992 DSP

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

Product Information

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 serves as means for 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/dsp, 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](ftp://ftp.analog.com) or [ftp 137.71.23.21](ftp://137.71.23.21)
<ftp://ftp.analog.com>

Related Documents

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

VisualDSP++ 3.0 Getting Started Guide for ADSP-218x DSPs

VisualDSP++ 3.0 User's Guide for ADSP-218x DSPs

VisualDSP++ 3.0 C Compiler and Library Manual for ADSP-218x DSPs

VisualDSP++ 3.0 Assembler and Preprocessor Manual for ADSP-218x and ADSP-219x DSPs

VisualDSP++ 3.0 Linker and Utilities Manual for ADSP-218x and ADSP-219x DSPs

VisualDSP++ 3.0 Product Bulletin

VisualDSP++ Kernel (VDK) User's Guide

VisualDSP++ Component Software Engineering User's Guide

Quick Installation Reference Card

Online Technical Documentation

Online documentation comprises VisualDSP++ Help system and tools manuals, Dinkum Abridged C++ library 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.

Product Information

File	Description
.CHM	Help system files and VisualDSP++ tools manuals.
.HTM or .HTML	Dinkum Abridged C++ library and 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 Dinkum Abridged C++ library and 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.

From the Web

To download the tools manuals, point your browser at 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 get 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.

Product Information

Datasheets

All datasheets can be downloaded from the Analog Devices website. As a general rule, any datasheet 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. Datasheets without the suffix can be downloaded from the website only—no hard copies are available. You can ask for the datasheet by a part name or by product number.

If you want to have a datasheet faxed to you, the phone number for that service is **1-800-446-6212**. Follow the prompts and a list of datasheet 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 by:

- Emailing dsp.techpubs@analog.com
- Filling in and returning the attached Reader's Comments Card found in our manuals

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 <i>this</i> or <i>that</i> .
[this that]	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional <i>this</i> or <i>that</i> .
[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 <i>this</i> .
.SECTION	Commands, directives, keywords, and feature names are in text with letter gothic font.
<i>filename</i>	Non-keyword placeholders appear in text with italic style format.
	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.

Notation Conventions

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 1-46](#)
Describes the `ccADSP-218x` compiler’s extensions to the ISO/ANSI standard for the C languages.
- [“Preprocessor Features” on page 1-89](#)
Contains information on the preprocessor and ways to modify source compilation.
- [“C Run-Time Model and Environment” on page 1-96](#)
Contains reference information about implementation of C programs, data, and function calls in ADSP-ADSP-218x DSPs.
- [“C and Assembly Language Interface” on page 1-109](#) 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 1-46](#).

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++ 3.0 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 behaviour is disabled using the “`-no-widen-muls`” switch (on page 1-31).
- 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 using the `-no-inline` compiler switch (see on page 1-30).
- Dual memory support keywords (`pm/dm`). Disabled using the `-no-extra-keywords` compiler switch (see on page 1-30).
- Placement support keyword (`section`). Disabled using the `-no-extra-keywords` compiler switch (see 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 1-46](#)”.

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 1-12](#)
- [“Data Type Sizes” on page 1-43](#)

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 1-10](#) 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, `-0` 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 1-10](#) lists the allowed extensions.

For example, the following command line

```
cc218x -2189 -0 -Wremarks -o program.dxe source.c
```

runs `cc218x` with the following switches:

Compiler Command-Line Reference

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

The normal function of `cc218x` 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
<code>.c</code> <code>.C</code>	C source file is compiled, assembled, and linked
<code>.asm</code> , <code>.dsp</code> , or <code>.s</code>	Assembly language source file is assembled and linked
<code>.obj</code>	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++ IDE. 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 on page 1-10](#) 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.

For information on additional search directories, see the `-I` directory switch ([on page 1-26](#)) and `-L` directory switch ([on page 1-26](#)).

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.

Compiler Command-Line Reference

Table 1-1. Input and Output Files

Input File Extension	Input File Description
.c	C source file
.h	Header file (referenced by a <code>#include</code> statement)
.i	Preprocessed C source, created when preprocess only (<code>-E</code> compiler switch) is specified
.idl	Interface definition language files for VCSE.
.s,.dsp,.asm	Assembly language source file
.is	Preprocessed assembly language source (retained when <code>-save-temps</code> is specified)
.obj	Object file to be linked
.lib	Library of object files to be linked as needed
.map	DSP system memory map file output
.sym	DSP system symbol map file output

Specifying Compiler Options in VisualDSP++

When using the VisualDSP++ IDDE, use the **Compile** page from the **Project Options** dialog box to set compiler functional options..

Callouts in [Figure 1-1 on page 1-11](#) refer to the corresponding compiler command-line switches described in “[Compiler Command-Line Switches](#)”. The **Additional options** field is used to enter the appropriate file names and options that do not have corresponding controls on the **Compile** property page 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.

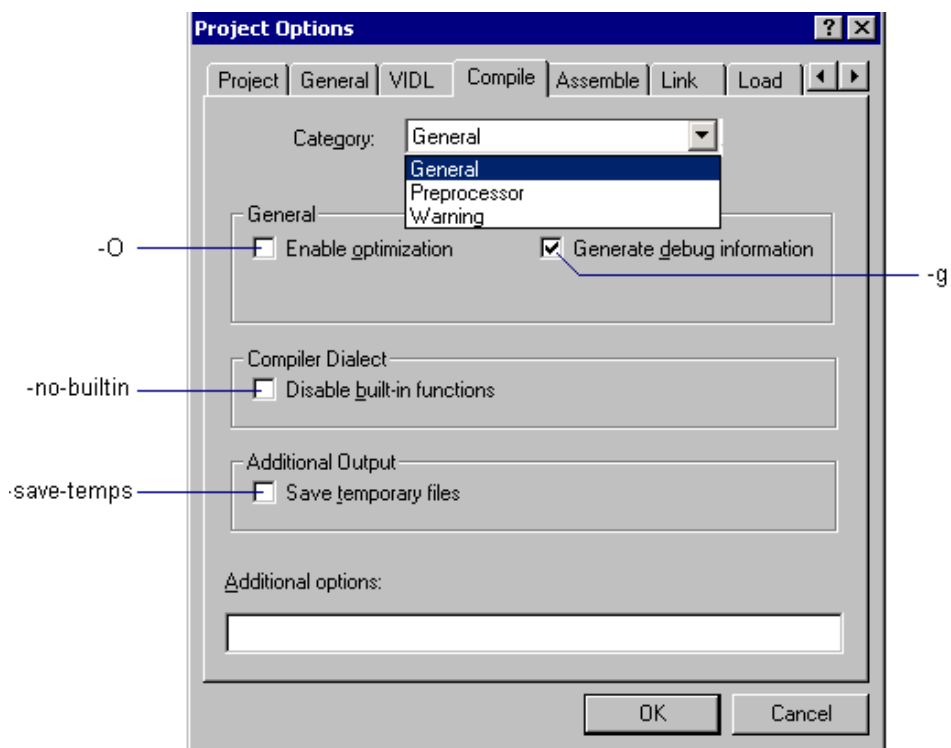


Figure 1-1. Project Options -- Compile Property Page

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.
-21{81 83 84 85 86 87 88 89} (on page 1-18)	Identifies the specific variant of the ADSP-218x DSPs.
-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.
-analog (on page 1-20)	Supports ANSI/ISO standard C with Analog Devices extensions. Default mode.
-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.
-Dmacro[=definition] (on page 1-21)	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 1-22)	Sets the default linkage type (asm, C).
-dollar (on page 1-22)	Accepts dollar signs in symbols.

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

Switch Name	Description
-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 1-23)	Preprocesses, but does not compile, the source file.
-EE (on page 1-23)	Preprocesses and compiles the source file.
-extra-keywords (on page 1-23)	Recognizes the Analog Devices extensions to ANSI/ISO standards for C. Default mode.
-flags-{asm compiler driver lib link} [, argument [...]] (on page 1-23)	Passes command-line switches through the compiler to other build tools.
-g (on page 1-24)	Generates DWARF-2 debug information.
-H (on page 1-24)	Outputs a list of header files, but does not compile.
-HH (on page 1-25)	Outputs a list of included header files and compiles.
-h[elp] (on page 1-25)	Outputs a list of command-line switches.
-I- (on page 1-25)	Establishes the point in the <code>include</code> directory list at which the search for header files enclosed in angle brackets should begin.
-I directory [; , directory...] (on page 1-26)	Appends the specified search directory to the standard search path.
-include filename (on page 1-26)	Includes named file prior to processing each source file.
-J (on page 1-26)	Performs "old-style" preprocessing.

Compiler Command-Line Reference

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

Switch Name	Description
<code>-jump-{dm pm}</code> (on page 1-26)	Specifies that the compiler should place jump tables in data memory (-jump-dm) or program memory (-jump-pm).
<code>-L directory [; , directory...]</code> (on page 1-26)	Appends the specified directory to the standard library search path when linking.
<code>-l library</code> (on page 1-27)	Searches the specified library for functions when linking.
<code>-M</code> (on page 1-27)	Generates make rules only; does not compile.
<code>-MM</code> (on page 1-27)	Generates make rules and compiles.
<code>-Mt filename</code> (on page 1-27)	Makes dependencies for the specified source file.
<code>-MQ</code> (on page 1-28)	Generates make rules only; does not compile.No notification when input files are missing.
<code>-make-autostatic</code> (on page 1-28)	Directs the compiler to place all automatic variables in static store.
<code>-map filename</code> (on page 1-28)	Directs the linker to generate a memory map of all symbols.
<code>-no-alttok</code> (on page 1-29)	Does not allow alternative keywords and sequences in sources.
<code>-no-builtin</code> (on page 1-29)	Disables recognition of <code>__builtin</code> functions.
<code>-no-defs</code> (on page 1-29)	Disables preprocessor definitions: macros, include directories, library directories, run-time headers, or keyword extensions.
<code>-no-dir-warnings</code> (on page 1-29)	Disables warnings about include and library switch directories that do not exist.
<code>-no-dollar</code> (on page 1-29)	Does not accept dollar signs in symbols.

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

Switch Name	Description
<code>-no-extra-keywords</code> (on page 1-30)	Does not define language extension keywords that could be valid C identifiers.
<code>-no-inline</code> (on page 1-30)	Does not perform high-level optimizations, and ignores the <code>inline</code> keyword.
<code>-no-std-def</code> (on page 1-30)	Disables normal macro definitions; also disables Analog Devices keyword extensions that do not have leading underscores (<code>__</code>).
<code>-no-std-inc</code> (on page 1-30)	Searches for preprocessor include header files only in the current directory and in directories specified with the <code>-I</code> switch.
<code>-no-std-lib</code> (on page 1-31)	Searches only for those linker libraries specified with the <code>-l</code> switch when linking.
<code>-no-widen-muls</code> (on page 1-31)	Disable widening multiplications optimization.
<code>-O</code> (on page 1-31)	Enables optimizations.
<code>-Os</code> (on page 1-32)	Optimize for code size.
<code>-o filename</code> (on page 1-32)	Specifies the output file name.
<code>-path-{asm compiler lib link} directory</code> (on page 1-32)	Uses the specified directory path as the location of the specified compilation tool (assembler, compiler, driver definitions file, library builder, or linker)
<code>-path-install directory</code> (on page 1-33)	Uses the specified directory as the location of all compilation tools.
<code>-path-output directory</code> (on page 1-33)	Specifies the location of non-temporary files.
<code>-path-temp directory</code> (on page 1-33)	Specifies the location of temporary files.
<code>-pch</code> (on page 1-33)	Generates and uses precompiled header files (*.pch)

Compiler Command-Line Reference

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

Switch Name	Description
<code>-pchdir <i>directory</i></code> (on page 1-33)	Specifies the location of PCHRepository.
<code>-pedantic</code> (on page 1-34)	Issues compiler warnings for any constructs that are not strictly ISO/ANSI standard C-compliant.
<code>-pedantic-errors</code> (on page 1-34)	Issues compiler errors for any constructs that are not strictly ISO/ANSI standard C-compliant.
<code>-pplist <i>filename</i></code> (on page 1-34)	Outputs a raw preprocessed listing to the specified file.
<code>-proc <i>identifier</i></code> (on page 1-35)	Specifies that the compiler should produce code suitable for the specified DSP.
<code>-R <i>directory</i></code> (on page 1-35)	Appends directory to the standard search path for source files.
<code>-R-</code> (on page 1-35)	Removes all directories from the standard search path for source files.
<code>-reserve <reg1>[,reg2...]</code> (on page 1-36)	Reserves specified registers for autobuffering.
<code>-S</code> (on page 1-36)	Stops compilation before running the assembler.
<code>-s</code> (on page 1-36)	Removes debugging information from the output executable file when linking.
<code>-save-temps</code> (on page 1-35)	Saves intermediate files.
<code>-show</code> (on page 1-37)	Displays the driver command-line information.
<code>-signed-char</code> (on page 1-37)	Makes the <code>char</code> data type signed.
<code>-syntax-only</code> (on page 1-37)	Checks the source code for compiler syntax errors, but does not write any output.
<code>-sysdefs</code> (on page 1-37)	Defines the system definition macros.

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

Switch Name	Description
<code>-T filename</code> (on page 1-38)	Specifies the Linker Description File.
<code>-time</code> (on page 1-38)	Displays the elapsed time as part of the output information on each part of the compilation process.
<code>-traditional</code> (on page 1-39)	Applies traditional C compiler rules (consistent with pre-ANSI K&R C compilers).
<code>-Umacro</code> (on page 1-39)	Undefines <code>macro</code> .
<code>-unsigned-char</code> (on page 1-39)	Makes the <code>char</code> data type unsigned.
<code>-v</code> (on page 1-39)	Displays both the version and command-line information.
<code>-val-global <name-list></code> (on page 1-40)	Adds global names.
<code>-verbose</code> (on page 1-40)	Displays command-line information.
<code>-version</code> (on page 1-40)	Displays version information.
<code>-Wdriver-limit number</code> (on page 1-40)	Halts the driver after reaching the specified number of errors.
<code>-Werror-limit number</code> (on page 1-40)	Stops compiling after reaching the specified number of errors.
<code>-W{error remark suppress warn} <num>[,num...]</code> (on page 1-41)	Overrides the severity of compilation diagnostic messages.
<code>-Wremarks</code> (on page 1-41)	Indicates that the compiler may issue remarks, which are diagnostic messages even milder than warnings.
<code>-Wterse</code> (on page 1-41)	Issues only the briefest form of compiler warnings, errors, and remarks.
<code>-w</code> (on page 1-41)	Does not display compiler warning messages.

Compiler Command-Line Reference

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

Switch Name	Description
<code>-warn-protos</code> (on page 1-42)	Produces a warning when a function is called without a prototype.
<code>-write-files</code> (on page 1-42)	Enables compiler I/O redirection.
<code>-write-opts</code> (on page 1-42)	Passes the user options (but not input filenames) via a temporary file.
<code>-xref <i>filename</i></code> (on page 1-42)	Outputs cross-reference information to the specified file.

-@ 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.

-21{81 | 83 | 84 | 85 | 86 | 87 | 88 | 89}

The `-21{81 | 83 | 84 | 85 | 86 | 87 | 88 | 89}` switch directs the build tools chain to generate code suitable for the specific variant of the ADSP-218x DSP. 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 defined). The compiler requires one of these switches when invoked from the command line and there is no default.

-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

-alttok

The `-alttok` (alternative tokens) switch directs the compiler to allow alternative operator keywords and digraph sequences in source files. This is the default mode. The “`-no-alttok`” switch ([on page 1-29](#)) can be used to disallow such support.

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

The following operator keywords are enabled when you use `#include <iso646.h>`:

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

-analog

The `-analog` (Analog C compilation) switch directs the compiler to support Analog Devices extensions to ISO/ANSI standard C. This is the default mode. For more information, see [“C Compiler Language Extensions” on page 1-46](#).

-ansi

The `-ansi` (ISO/ANSI standard C compilation) switch disables builtin functions and the definitions of `__ANALOG_EXTENSIONS__`, `inline`, `bool`, `false`, and `true`. This switch also enables the definition of `__STRICT_ANSI__`.



This switch can be invoked with the `Use ANSI standard` check box located in the IDDE's `Compile` dialog box, `General` selection. [“-no-extra-keywords” 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 (`.d1b`) as the output instead of using the linker to produce an executable file (`.dxe`). The `-o` option must be used to specify the name of the resulting library.

-C

The `-C` (comments) switch, which is only active in combination with the `-E` or `-P` switches, directs the C preprocessor to retain comments in its output file.

-c

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

-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, **Pre-processor** category.

-debug-types

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 need not be specified with the `-debug-types` switch because it is implied.

For example,

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

Until the introduction of `-debug-types`, the compiler would not accept a `*.h` file as a valid input file. 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.

-dollar

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

-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.



You can invoke it with the **Stop after: Preprocessor** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **General** category.

-EE

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

-extra-keywords

The `-extra-keywords` (enable short-form keywords) switch directs the compiler to recognize the Analog Devices keyword extensions to 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 1-47](#) provides a list and a brief description of keyword extensions.

-flags-{asm | compiler | lib | link} 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:

Compiler Command-Line Reference

Table 1-3. Build Tools' Options

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

-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.

-HH

The **-HH** (list headers and compile) switch directs the compiler to output to the standard out 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.

Compiler Command-Line Reference

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

The `-I directory` (include search directory) switch directs the C preprocessor to append the directory(directoryes) 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. Only one `-include` may be given.

-J

The `-J` switch directs the preprocessor to perform "old-style" preprocessing.

-jump-{*dm* | *pm*}

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`). 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 ...
```

-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.

-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

Compiler Command-Line Reference

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 1-87](#) 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.map`. The `.map` extension is added where necessary.

-no-alttok

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

-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 and also to disable the compiler’s C keyword extensions.

-no-dir-warnings

The `-no-dir-warnings` (disable directory warning) switch directs the compiler not to issue warnings when it encounters directories on the command line that do not exist. Such directories might be used as part of subsequent `-I` (include search directory) and `-L` (library search directory) switches.

-no-dollar

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

-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 1-23](#)) can be used to explicitly request support for the additional keywords.

no-inline

The `-no-inline` (disable inline keyword) switch directs the compiler not to perform any high-level optimizations and directs the compiler to ignore the `inline` keyword.

-no-std-def

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

 This switch also disables the Analog Devices keyword extensions which have no leading underscores, such as `pm` and `dm`.

-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.

 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 search only libraries specified with the `-l` switch.

-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.

Compiler Command-Line Reference

-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.

-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.

-path-{asm | compiler | def | lib | link} directory

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 and linker. Use this switch when you wish to override the normal version of one or more of the tools. The `-path-tool` switch also overrides the directory specified by the `-path-install` switch.

-path-install directory

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

 You can selectively override this switch with the `-path-tool` switch (see [on page 1-32](#)).

-path-output directory

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

-path-temp directory

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

-pch

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

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

-pchdir directory

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

-pedantic

The `-pedantic` (ANSI standard warnings) switch causes the compiler to issue 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 <code>#if</code> or <code>#ifdef</code>
L	Change in source position
R	Diagnostic message (remark)
W	Diagnostic message (warning)

Character	Meaning
E	Diagnostic message (error)
C	Diagnostic message (catastrophic error)

-proc identifier

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

-R directory [{: | ,}directory ...]

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

On Windows™ platforms, multiple source directories are given as a colon, 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 1-98](#).

-S

The `-S` (stop after compilation) switch directs the compiler to stop compilation before running the assembler. The compiler outputs an assembler file with an `.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 not to discard intermediate files. The compiler places the intermediate output (`*.i`, `*.is`, `*.s`) files in the current temp directory. See [Table 1-1 on page 1-10](#) for a list of intermediate files.

The location of the saved file is affected by the `-path-output` switch, if provided. That switch sets the path for all “permanent” outputs that do not otherwise have a path set, the object file included

-show

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

-signed-char

The `-signed-char` (make char signed) switch directs the compiler to make the default type for `char` signed. The compiler also defines the `__SIGNED_CHARS__` macro. This is the default mode when the `-unsigned-char` (make char unsigned) switch (see [on page 1-39](#)) is not used.

The compiler also defines the `__SIGNED_CHARS__` macro. This is the default mode when the `-unsigned-char` (make char unsigned) switch is not used.

-syntax-only

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

-sysdefs

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

Compiler Command-Line Reference

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.

-traditional

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

- All `extern` declarations (including implicit declarations of functions) take effect globally.
- The keywords `inline`, `signed`, `const`, and `volatile` are not recognized.
- Pointer/integer comparisons are always allowed.

-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-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.

Compiler Command-Line Reference

-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.

Compiler Command-Line Reference

-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
long	32 bits signed	2
unsigned long	32 bits unsigned	2
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

Compiler Command-Line Reference

Table 1-4. Data Type Sizes for the ADSP-218x DSPs (Cont'd)

Type	Bit Size	sizeof returns
function pointer	32 bits	2
float	32 bits float	2
double	32 bits float	2
fract16	16 bits fractional (defined in fract_typedef.h)	1
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 2 [“Library”](#).

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:

- [“Inline Function Support Keyword \(inline\)” on page 1-49](#)
- [“Inline Assembly Language Support Keyword \(asm\)” on page 1-50](#)
- [“Dual Memory Support Keywords \(pm dm\)” on page 1-61](#)
- [“Placement Support Keyword \(section\)” on page 1-66](#)
- [“Boolean Type Support Keywords \(bool, true, false\)” on page 1-67](#)
- [“Pointer Class Support Keyword \(restrict\)” on page 1-67](#)
- [“Variable-Length Array Support” on page 1-68](#)
- [“Non-Constant Aggregate Initializer Support” on page 1-70](#)
- [“Indexed Initializer Support” on page 1-70](#)
- [“Aggregate Constructor Expression Support” on page 1-72](#)
- [“Preprocessor-Generated Warnings” on page 1-73](#)
- [“C++-Style Comments” on page 1-73](#)
- [“ETSI Support” on page 1-76](#)
- [“Pragmas” on page 1-84](#)

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
<code>inline</code> (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 1-49.
<code>dm</code>	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 1-61.
<code>pm</code>	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 1-61.

C Compiler Language Extensions

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

Keyword extensions	Description
<code>section(string)</code>	Specifies the section in which an object or function is placed. For more information, see “Placement Support Keyword (section)” on page 1-66.
<code>bool, true, false</code>	A Boolean type. For more information, see “Boolean Type Support Keywords (bool, true, false)” on page 1-67.
<code>restrict</code> keyword	Specifies restricted pointer features. For more information, see “Pointer Class Support Keyword (restrict)” on page 1-67.

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 1-68.
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 1-70.
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 1-70.
Preprocessor generated warnings	Support for generating warning messages from the preprocessor. For more information, see “Preprocessor-Generated Warnings” on page 1-73.
C++-style comments	Support for C++-style comments in C programs. For more information, see “C++-Style Comments” on page 1-73.

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. 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 single_addition (int *a)
{
    (*a)++;
}
```

A function declared `inline` must be defined (its body must be included) in every file in which the function is used. The normal way to do this is to place the inline definition in a header file. 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 `-no-inline` and `-traditional` switches disable function inlining. [For more information, see “Compiler Command-Line Switches” on page 1-12.](#)
-  If creating a debug target (or using the `-g` switch on the command line) and optimizations are not enabled, the compiler ignores the `inline` keyword and disables function inlining. This is to allow users maximum debug capabilities.

Inline Assembly Language Support Keyword (`asm`)

The `cc218x asm()` construct lets you code ADSP-218x processor's assembly language instructions within a C function. 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 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.

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

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

The complete assembly language instruction, enclosed in quotes, is the argument to `asm()`.

 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 1-51](#)
- [“Assembly Construct Operand Description” on page 1-55](#)
- [“Assembly Constructs With Multiple Instructions” on page 1-57](#)
- [“Assembly Construct Reordering and Optimization” on page 1-58](#)
- [“Assembly Constructs with Input and Output Operands” on page 1-59](#)
- [“Assembly Constructs and Macros” on page 1-60](#)

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]]]
);
```

C Compiler Language Extensions

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 appearance from left to right, starting at 0. Separate multiple instructions with a semicolon, and enclose the entire string within double quotes. For more information on templates containing multiple instructions, see [“Assembly Constructs With Multiple Instructions” on page 1-57](#).

- *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 1-55](#).

- *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 over-written by the assembly instructions. Use lowercase characters to name clobbered registers. Enclose each register name within double quotes, and separate the quoted register names with commas.

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, and the last input operand from the clobbered registers. If there are no output operands and there are input operands, there must be two consecutive colons separating the assembly template from the input operands.
- A comma separates operands and registers within arguments.
- The number of operands in arguments must match the number of operands in your template.
- The maximum permissible number of operands is ten (`%0`, `%1`, `%2`, `%3`, `%4`, `%5`, `%6`, `%7`, `%8`, and `%9`).



The compiler cannot check whether the operands have data types that are reasonable for the instruction being executed. The compiler does not parse the assembler instruction template, interpret the template, 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 x, result;

asm("%0=abs %1;":"=c"(result):"c"(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 1-56](#) 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.

C Compiler Language Extensions

- 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 constraints.

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

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

Constraint ¹	Description	Registers
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
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	

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.

Assembly Constructs With Multiple Instructions

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

C Compiler Language Extensions

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

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

```
#define IOWrite(val,addr) \  
asm volatile ("si="#val";I0("#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

C Compiler Language Extensions

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);
}
```

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.

C Compiler Language Extensions

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 * PFL0ATP;
/* PFL0ATP defines a type which is a pointer to a */
/* float which resides in pm. */

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

Memory Keywords and Assignments/Type Conversions

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

- 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 */
```

C Compiler Language Extensions

```
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
    an int residing in pm as its argument and which
    returns an int */

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

foo(&x); /* call function foo() */
    /* using pm pointer (location of x) as the    */
    /* argument. 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.

C Compiler Language Extensions

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);
```

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 to determine 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 fi (short *in, short *c, short *restrict out, int n)
```

C Compiler Language Extensions

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 than 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 cc218x C compiler includes support for the ISO/ANSI standard definition of the C language and also includes extended support for initializers. The compiler does not require the elements of an aggregate initializer for an automatic variable to be constant expressions.

The following example shows an initializer with elements that vary at run time:

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

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
};
```

C Compiler Language Extensions

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

C Compiler Language Extensions

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 2-2](#).

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 1-74](#)
- [“Read/Write of Non-Memory-Mapped Registers” on page 1-75](#)
- [“Interrupt Control” on page 1-76](#)

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 2-66](#) and [“`io\_space\_write`” on page 2-67](#).

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. This enumeration includes:

```
enum SysReg {
    sysreg_ASTAT = 0x0, // ASTAT register - arithmetic status
    sysreg_SSTAT = 0x1, // SSTAT register - shifter status
    sysreg_MSTAT = 0x2, // MSTAT register - multiplier status
    sysreg_ICNTL = 0x3, // ICNTL register - interrupt control
    sysreg_IMASK = 0x4, // IMASK register - interrupts enabled mask
    sysreg_IFC = 0x5, // IFC register - Interrupt force and clear
};
```

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 2-141](#) and [“sysreg_write” on page 2-143](#).

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 2-43](#) and [“disable_interrupts” on page 2-41](#).

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 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 1-77](#)
- [“Calling ETSI Library Functions” on page 1-79](#)
- [“Using the ETSI Built-In Functions” on page 1-80](#)
- [“Linking ETSI Library Functions” on page 1-80](#)
- [“Working with ETSI Library Source Code” on page 1-80](#)

- [“ETSI Support for Data Types” on page 1-81](#)
- [“ETSI Header File” on page 1-81](#)

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 1-81](#).

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 1-109](#).

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 1-73](#). These built-in implementations will be automatically defined when header file `ETSI_fract_arith.h` is included.

Linking ETSI Library Functions

When your C/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 requires 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 processor 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 `-lets` 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 1-109](#).

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 table provides a summary of the functions provided by the ETSI library, as defined in the header file `ETSI_fract_arith.h`.

C Compiler Language Extensions

Table 1-8. ETSI Library Functions

Description	Prototype
Short absolute	fract16 abs_s (fract16)
Short add	fract16 add (fract16, fract16)
Short division	fract16 div_s (fract16, fract16)
Long division	fract16 div_l (fract32, fract16)
Extract high (most significant 16 bits)	fract16 extract_h (fract32)
Extract low (least significant 16 bits)	fract16 extract_l (fract32)
Multiply and accumulate with rounding	fract16 mac_r (fract32, fract16, fract16)
Multiply and subtract with rounding	fract16 msu_r (fract32, fract16, fract16)
Short multiply	fract16 mult (fract16, fract16)
Multiply with rounding	fract16 mult_r (fract16, fract16)
Short negate	fract16 negate (fract16)
Long normalize	fract16 norm_l (fract16)
Short normalize	fract16 norm_s (fract16)
Round	fract16 round (fract16)
Saturate	fract16 saturate (fract32)
Short shift left	fract16 shl (fract16, fract16)
Short shift right	fract16 shr (fract16, fract16)
Shift right with rounding	fract16 shr_r (fract16, fract16)
Short subtract	fract16 sub (fract16, fract16)
Long absolute	fract32 L_abs (fract32)
Long add	fract32 L_add (fract32, fract32)
Long add with carry	fract32 L_add_c (fract32, fract32)
16 bit variable -> most significant bits(least significant bits zeroed)	fract32 L_deposit_h (fract16)

Table 1-8. ETSI Library Functions (Cont'd)

Description	Prototype
16 bit variable -> least significant bits(sign extended)	fract32 L_deposit_l (fract16)
Multiply and accumulate	fract32 L_mac (fract32, fract16, fract16)
Multiply and accumulate without saturation	fract32 L_macNs (fract32, fract16, fract16)
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)
Multiply and subtract without saturation	fract32 L_msuNs (fract32, fract16, fract16)
Long multiply	fract32 L_mult (fract16, fract16)
Long negate	fract32 L_negate (fract32)
Long saturation	fract32 L_sat (fract32)
Long shift left	fract32 L_shl (fract32, fract16)
Long shift right	fract32 L_shr (fract32, fract16)
Long shift right with rounding	fract32 L_shr_r (fract32, fract16)
Long subtract	fract32 L_sub (fract32, fract32)
Long subtract with carry	fract32 L_sub_c (fract32, fract32)
Compose long	fract32 L_Comp (fract16, fract16)
Multiply two longs	fract32 Mpy_32 (fract16, fract16, fract16, fract16)
Multiply short by a long	fract32 Mpy_32_16 (fract16, fract16, fract16)
Extract a long from two shorts	void L_Extract (fract32, fract16*, fract16*)
Fract integer division of two longs	fract32 Div_32 (fract32, fract16, fract16)

Pragmas

The C compiler for ADSP-218x DSPs supports a number of pragmas. Pragmas are implementation-specific directives that modify the compiler's behavior. 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

The following sections describe the pragmas that support these features.

- [“Data Alignment Pragma - ALIGN NUM” on page 1-84](#)
- [“Interrupt Handler Pragma” on page 1-85](#)
- [“Optimization Level Pragmas” on page 1-86](#)
- [“Linkage Pragmas” on page 1-87](#)
- [“Stack Usage Pragma” on page 1-87](#)

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

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

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

Compile using the `-flags-link -ip` switch options (described on [page 1-23](#)) if the `#pragma align` directives cause insufficient memory errors at link time due to fragmentation. See the `-ip` switch description in Chapter 1 of the *VisualDSP++ 3.0 Linker and Utilities Manual for ADSP-218x and ADSP-219x DSPs* for more information.

Interrupt Handler Pragma

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 */
}
```

Altregisters Pragma

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 */
}
```

Optimization Level Pragmas

The `optimize_off`, `optimize_for_space`, and `optimize_for_speed` pragmas change the optimization level while a given module is being compiled. The `optimize_off` pragma turns off the optimizer, if it was enabled. The `optimize_for_space` and `optimize_for_speed` pragmas turn the optimizer back on, if it was disabled, or switch focus between reducing code size and increasing performance, if it was already enabled.

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

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

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

Linkage Pragmas

Linkage pragmas—`linkage_name` and `weak_entry`—change how a given global function or variable is viewed during the linking stage.

`linkage_name` Identifier

The `linkage_name` pragma 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.

```
_Pragma("linkage_name __realfuncname")
void funcname ();
```

`weak_entry` Pragma

The `weak_entry` pragma may be used before a function or 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.

```
#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`.

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

C Compiler Language Extensions

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` (make automatic variables static) switch (see [on page 1-28](#)) 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.

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 1-73](#) and [“C++-Style Comments” on page 1-73](#).

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 `-21{81|83|84|85|86|87|88|89}` command-line switch.

Preprocessor Features

__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`

cc219x always defines `_LANGUAGE_C` as 1 when compiling C (or 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 and 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 unless you compile with the `-traditional` (ANSI standard) command-line switch.

`__STDC_VERSION__`

cc218x defines `__STDC__` as 199409L unless you compile with the `-traditional` command-line switch (ANSI standard).

__TIME__

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

Header Files

A header file contains C 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, it can be useful to define a macro that expands into a compound statement. You can define such a macro so that it can be invoked in the same way you would call a function, making your source code easier to read and maintain.

The following two code segments define two versions of the macro SKIP_SPACES:

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

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

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

Preprocessor Features

With the `do {...} while (0)` construct, you can pretend that the macro is a function and always put the semicolon after it. For example,

```
/* SKIP_SPACES, enclosed macro, ends without ';' */
if (*p != 0)
    SKIP_SPACES (p, lim);
else ...
```

This expands to

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

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

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

This is not legal C syntax. For more information on macros, see the *VisualDSP++ 3.0 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](#).

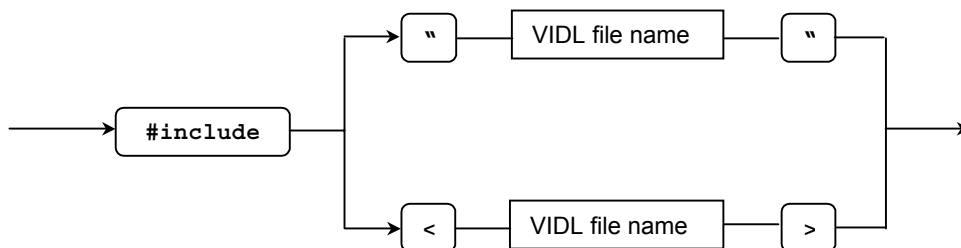


Figure 1-2. #INCLUDE Syntax Diagram

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.

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,

<code>_lib_int_table</code>	Static table holding interrupt information defined in the C run-time library
<code>__lib_int_determiner</code>	An interrupt dispatcher defined in the C run-time library
<code>____system_start</code>	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.

C Run-Time Model and Environment

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 Pragma” on page 1-85](#)).



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 (see [on page 1-36](#)). This support makes it possible 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 `libioab.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 across 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.

C Run-Time Model and Environment

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

Figure 1-3. ADSP-218x DSP Stack

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).

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.

C Run-Time Model and Environment

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.

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 1-36](#) 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/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.

C Run-Time Model and Environment

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	
AY1	scratch	Argument 2 for compatibility call
AR	scratch;	Argument 1 for compatibility call
AF	scratch	
MX0	scratch	
MX1	scratch	

Table 1-10. Data Register File Registers (Cont'd)

Register	Descriptuon	Notes
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
M0	preserved
M1	dedicated: +1
M2	dedicated: 0
M3	scratch
L0-3	not used, must be zero

C Run-Time Model and Environment

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 1-96](#).

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.

C and Assembly Language Interface

- 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 1-36](#)). 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, 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 Assembly 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 1-109](#), 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` command-line switch). 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

<code>function_entry</code>	<code>exit</code>	<code>leaf_entry</code>	<code>leaf_exit</code>
<code>alter(x)</code>			
<code>save_reg</code>	<code>restore_reg</code>	<code>readsfirst(x)</code>	<code>readsnext</code>
<code>putsfirst</code>	<code>putsnext</code>	<code>getsfirst(x)</code>	<code>getsnext</code>

`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.

C and Assembly Language Interface

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.

readfirst(register)

The preprocessor expands the `readfirst` 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 `readsnxt` macro. The `readfirst` macro references the stack-pointer (I4) and might be used to read values that were placed on the stack using the `putfirst` and `putsnxt` macros.

register = readsnxt

The preprocessor expands the `readsnxt` macro into a series of assembly language commands. These commands continue the read process set up by the `readfirst` macro by reading the next value off the top of the stack and writing it to `register`.

putfirst = register

The preprocessor expands the `putfirst` 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 `putsnxt` macro.

putsnxt = register

The preprocessor expands the `putsnxt` macro into a series of assembly language commands. These commands continue the write process set up by the `putfirst` macro by writing the next `register` to the top of the stack.

C and Assembly Language Interface

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
<code>int c_var;</code> <code>/* declared global */</code>	<code>.extern _c_var;</code>
<code>void c_func();</code>	<code>.extern _c_func;</code>
<code>extern int asm_var;</code>	<code>.global _asm_var;</code>
<code>extern void asm_func();</code>	<code>.global _asm_func;</code> <code>_asm_func:</code>
<code>extern "asm" void asm_func();</code>	<code>.global asm_func;</code> <code>asm_func:</code>

Compatibility Call

The `cc218x` compiler in VisualDSP++ 3.0 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

C and Assembly Language Interface

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 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.

For more information on the algorithms on which many of the library's math functions are based, see the Cody and Waite text "*Software Manual For The Elementary Functions*" from Prentice Hall (1980).

This chapter contains:

- [“C Run-Time Library Guide” on page 2-2](#)
- [“Documented Library Functions” on page 2-11](#)
- [“C Run-Time Library Reference” on page 2-15](#)

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 2-15](#).

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 2-5](#).

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 1-109](#).

 You can use the archiver, described in the *VisualDSP++ 3.0 Linker and Utilities Manual for ADSP-218x and ADSP-219x DSPs*, 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 2-15 on page 2-4](#). 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 -lets` 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 1-76](#). For more information on the `.LDF` file, see the *VisualDSP++ 3.0 Linker and Utilities Manual for ADSP-218x and ADSP-219x DSPs*.

C Run-Time Library Guide

Table 2-15. C Run-Time Library Files

218x\lib	Directory Description
218x_hdr.doj	Default C run-time initialization code . Calls <code>main()</code> .
218x_ezkit_hdr.doj	C run-time initialization code with EZ-Kit monitor program support. Calls <code>main()</code> .
218x_int_tab.doj	Default interrupt vector code
218x_ezkit_int_tab.doj	Default interrupt vector code with EZ-Kit monitor program support
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 extension `.asm`, `.dsp`, or `.c`. 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 1-96](#). 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 2-16](#). You should use a C standard text to augment the information supplied in this chapter.

Table 2-16. C Run-Time Library Header Files

Header	Purpose	Standard
asm_sprt.h	Mixed C/assembly support	C Extension
assert.h	Diagnostics	ANSI
circ.h	Support for circular buffers	C Extension
ctype.h	Character handling	ANSI
def2181.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 2-16. 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
macros.h	Circular buffers	C Extension
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.

asm_sprt.h – Mixed C/Assembly Support

This header consists of ADSP-218x DSP assembly language macros, not C functions. They are used in assembly routines that interface with C functions. For more information on this header, see [“C and Assembly Language Interface” on page 1-109](#).

assert.h

The `assert.h` header contains the `assert` macro.

ctype.h

The `ctype.h` header 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. These memory addresses are used as system registers.

errno.h

The `errno.h` header provides access to `errno`. This facility is not, in general, supported by the rest of the library. Be sure to include the header file.

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.



`cfftN` stands for the entire family of Fast Fourier Transform functions `cfft1024`, `cfft512`, etc.

The `cfftN` 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 float 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.

C Run-Time Library Guide

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 format of floating point data types are defined in the header file `float.h`. The `FLT_ROUNDS` macro is defined as 4 in the `float.h` header file. The 4 indicates an implementation-defined rounding mode.

The C run-time environment defines the rounding mode for `float` variables as round-to-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++ 3.0 (see [“ETSI Support” on page 1-76](#)).

macros.h – Circular Buffers

This category consists of ADSP-218x assembly language macros, not C functions. Some are used to manipulate the circular buffer features of the ADSP-218x processors.

math.h – Mathematics

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.

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.

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`.

signal.h – Signal Handling

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()`. It also includes ANSI-standard signal handling functions of the C library.

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.

stdio.h – Standard Input/Output

The implementation of the `stdio.h` routines provides for a simple interface with a host environment. The simulator provides internal host support for these routines. Calls to `stdio.h` routines are trapped by the simulator, and a host implementation provides the correct functionality. To make use of the `stdio.h` routines, it is necessary to link with `libio.dlb` in the same way as linking to `libc.dlb`.

Refer to [Table 2-25 on page 2-12](#) for the list of all supported `stdio.h` library functions.

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 1-73](#) 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 2-17. Library Functions in the `ctype.h` Header File

isalnum	isalpha	isctrl
isdigit	isgraph	isinf
islower	isnan	isprint
ispunct	isspace	isupper
isxdigit	tolower	toupper

Table 2-18. 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 2-19. Library Functions in the `filters.h` Header File

biquad	demean_buffer	fir
iir		

Table 2-20. Library Functions in the `math.h` Header File

acos	asin	atan
atan2	ceil	cos
cosh	cot	exp
fabs	floor	fmod

Documented Library Functions

Table 2-20. Library Functions in the `math.h` Header File (Cont'd)

frexp	isinf	isnan
ldexp	log	log10
modf	pow	sin
sinh	strcat	tan
tanh		

Table 2-21. Library Functions in the `misc.h` Header File

timer_off	timer_on	timer_set
---------------------------	--------------------------	---------------------------

Table 2-22. Library Functions in the `setjmp.h` Header File

longjmp	setjmp
-------------------------	------------------------

Table 2-23. Library Functions in the `signal.h` Header File

clear_interruptx	interrupt	raise
signal		

Table 2-24. Library Functions in the `stdarg.h` Header File

va_arg	va_end	va_start
------------------------	------------------------	--------------------------

Table 2-25. Supported Library Functions in the `stdio.h` Header File

clearerr	fclose	feof
ferror	fflush	fgetc
fgets	fprintf	fputc
fputs	fopen	fread
freopen	fscanf	fwrite
getc	getchar	gets

Table 2-25. Supported Library Functions in the `stdio.h` Header File

<code>perror</code>	<code>qsort</code>	<code>putc</code>
<code>putchar</code>	<code>puts</code>	<code>scanf</code>
<code>setbuf</code>	<code>setvbuf</code>	<code>sprintf</code>
<code>sscanf</code>	<code>ungetc</code>	<code>vfprintf</code>
<code>vprintf</code>	<code>vsprintf</code>	

Table 2-26. Library Functions in the `stdlib.h` Header File

<code>abort</code>	<code>abs</code>	<code>atexit</code>
<code>atof</code>	<code>atoi</code>	<code>atol</code>
<code>bsearch</code>	<code>calloc</code>	<code>div</code>
<code>exit</code>	<code>free</code>	<code>labs</code>
<code>ldiv</code>	<code>malloc</code>	<code>qsort</code>
<code>rand</code>	<code>realloc</code>	<code>srand</code>
<code>strtod</code>	<code>strtodf</code>	<code>strtol</code>
<code>strtoul</code>		

Table 2-27. Library Functions in the `string.h` Header File

<code>memchr</code>	<code>memcmp</code>	<code>memcpy</code>
<code>memmove</code>	<code>memset</code>	<code>strcat</code>
<code>strchr</code>	<code>strcmp</code>	<code>strcoll</code>
<code>strcpy</code>	<code>strcspn</code>	<code>strerror</code>
<code>strlen</code>	<code>strncat</code>	<code>strncmp</code>
<code>strncpy</code>	<code>strpbrk</code>	<code>strrchr</code>
<code>trspns</code>	<code>strstr</code>	<code>strtok</code>
<code>strxfrm</code>		

Documented Library Functions

Table 2-28. Library Functions in the `sysreg.h` Header File

<code>disable_interrupts</code>	<code>enable_interrupts</code>
<code>io_space_read</code>	<code>io_space_write</code>
<code>sysreg_read</code>	<code>sysreg_write</code>

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

The reference pages for the library functions use 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](#)

C Run-Time Library Reference

acos

arc cosine

Synopsis

```
#include <math.h>
double acos(double);
fract16 acos_fr16 (fract16);
```

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 $[0, \pi]$.

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 $\text{acos}(0) \times 2/\pi$ ($=0x7FFF$) to $\text{acos}(0.9) \times 2/\pi$ ($=0x24C1$).

Error Conditions

The `acos` function indicates a domain error (by setting `errno` to `EDOM`) and returns a zero if the input is not in the range $[-1, 1]$.

Example

```
#include <math.h>
double y;
y = acos(0.0); /* y =  $\pi/2$  */
```

See Also

[cos](#)

asin

arc sine

Synopsis

```
#include <math.h>
double asin(double);
fract16 asin_fr16 (fract16);
```

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 $-\pi/2$ to $\pi/2$.

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/ $\pi$` (=0xA4C1) to `asin(0.9)*2/ $\pi$` (=0x5B3F).

Error Conditions

The `asin` function indicates a domain error (set `errno` to `EDOM`) and returns a zero if the input is not in the range $[-1, 1]$.

Example

```
#include <math.h>
double y;
y = asin(1.0); /* y =  $\pi/2$  */
```

See Also

[sin](#)

atan

arc tangent

Synopsis

```
#include <math.h>
double atan(double);
fract16 atan_fr16 (fract16);
```

Description

The `atan` function returns the arc tangent of the argument. The output, in radians, is in the range $-\pi/2$ to $\pi/2$.

The `atan_fr16` function covers the output range from $-\pi/4$ (input value 0x8000, output value 0x9B78) to $\pi/4$ (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);
fract16 atan2_fr16 (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 $[-\pi, \pi]$.

The `atan2_fr16` function uses the full range from $-\pi/4$ to $\pi/4$ (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 is equal to zero and y less or greater than zero.

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 =  $\pi/2$  /
```

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

[exit](#)

atof

convert string to a double

Synopsis

```
#include <stdlib.h>
double atof(const char *nptr);
```

Description

The `atof` function converts a character string to a double 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). Conversion terminates at the first non-digit (exceptions are “.”, “e”, “E”, and exponents, including the sign).



There is no way to determine if a zero is a valid result or an indicator of an invalid string.

Error Conditions

The `atof` function returns a zero if no conversion can be made.

Example

```
#include <stdlib.h>
double x;
x = atof("5.5");    /* x == 5.5 */
```

See Also

[atoi](#), [atol](#), [strtol](#), [strtoul](#)

C Run-Time Library Reference

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

[atol](#), [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](#), [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.

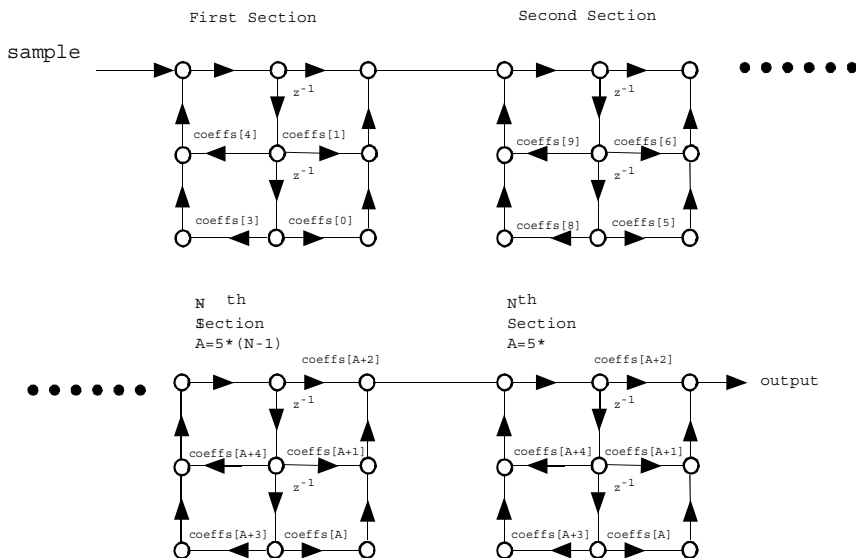
Example

```
#include <filter.h>
#define TAPS 9

int i;
int input, output, state[TAPS+ 1];
int pm coeffs;

for (i=0; i < TAPS+1; ++i)
state[i]=0; /*initialize state array*/

output = fir(sample, coeffs, state, TAPS)
```



C Run-Time Library Reference

N = the number of biquad sections.

The `coeffs` array holds 5 coefficients for each section and therefore should be $5 * N$ in length. The `delay` array holds 2 delayed elements for each section, and should be $2 * N + 1$ in length.

The algorithm here is adapted from the Oppenheim and Schaffer text “*Digital Signal Processing*” from Prentice Hall (1975).

See Also

[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

`bsearch` 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 size of 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.

`bsearch` returns a pointer to the first occurrence of `key` in the array.

Error Conditions

The `bsearch` returns `NULL` if the key is not found in the array.

C Run-Time Library Reference

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);
```

Description

The `ceil` function returns the smallest integral value, expressed as double, that is not less than its input.

Error Conditions

The `ceil` function does not return an error condition.

Example

```
#include <math.h>
double y;
y = ceil(1.05); /* y = 2.0 */
```

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.

Table 2-29. ADSP-218x 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.

C Run-Time Library Reference

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);
fract16 copysign_fr16 (fract16 parm1, fract16 parm2);
```

Description

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);
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 is defined by the domain of `0x8000` to `0x7FFF` corresponding to $[-\pi/2, \pi/2]$. The domain represents half a cycle which can be used to derive a full cycle if required. 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 */
```

See Also

[acos](#), [malloc](#), [realloc](#)

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

```
#define BUSIZE 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, like this example: */
{
    int i, temp_mean;
```

C Run-Time Library Reference

```
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`,

$$\text{result.quot} * \text{denom} + \text{result.rem} == \text{numer}$$

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 "DIS INTS;" instruction inline
}
```

See Also

[disable_interrupts](#), [io_space_read](#), [io_space_write](#), [sysreg_read](#), [sysreg_write](#)

C Run-Time Library Reference

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 header.

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);
```

Description

The `exp` function computes the exponential value e to the power of its argument.

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);
```

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[],
```

C Run-Time Library Reference

```
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` is a integer array of N points. 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 i;
int real_input[N];
imag_input[N];
#pragma align 1024
int imag_output[N];
#pragma align 1024
int real_output[N];
fft1024 (real_input, imag_input, real_output, imag_output);
/* Arrays are filled with FFT data */
```

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

[iir](#)

floor

floor

Synopsis

```
#include <math.h>
double floor(double);
```

Description

The `floor` function produces 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, double);
```

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

[modf](#)

C Run-Time Library Reference

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, int*);
```

Description

The `frexp` function separates a floating-point input into a normalized fraction and a (base 2) exponent. The function returns the first argument, a fraction in the interval (1/2, 1), and stores a power of 2 in the integer pointed to by the second argument. If the input is zero, then zeros are stored in both arguments.

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` is a integer array of N points. 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

C Run-Time Library Reference

preprocessing is performed on the data. The functions return a pointer to the `real_output` array. 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 <fft.h>
#define N 1024
int i;
int real_input[N];
imag_input[N];
#pragma align 1024
int imag_output[N];
#pragma align 1024
int real_output[N];
    /* Real input array is filled from a previous */
    /* fft1024() or other source                  */
ifft1024 (real_input, imag_input, real_output, imag_output);
    /* Arrays are filled with FFT data            */
```

See Also

[fftN](#)

iir

infinite impulse response (IIR) filter

Synopsis

```
#include <filters.h>
int iir(int sample, int pm a_coeffs[],
int 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 defined by the coefficients and delay line that are supplied in the call of `iir`. The function produces the filtered response of its input data. The IIR filter implemented in this function is based on the Oppenheim and Schaffer Direct Form II. The characteristics of the filter depend on the coefficient values supplied by the calling program.

The integer input to the filter is `sample`. The integer `taps` indicates the length of the filter, which is the length of the input array `a_coeffs`, while input arrays `b_coeffs` and delay are length `taps+1`. The `a_coeffs` and `b_coeffs` arrays hold one IIR coefficient per element. The coefficients are stored in reverse order; for example, `a_coeffs[0]` holds the $(taps-1)$ coefficient. The `a_coeffs` and `b_coeffs` arrays are located in program memory data space to allow the single-cycle dual-memory fetch of the processor to be used.

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` func-

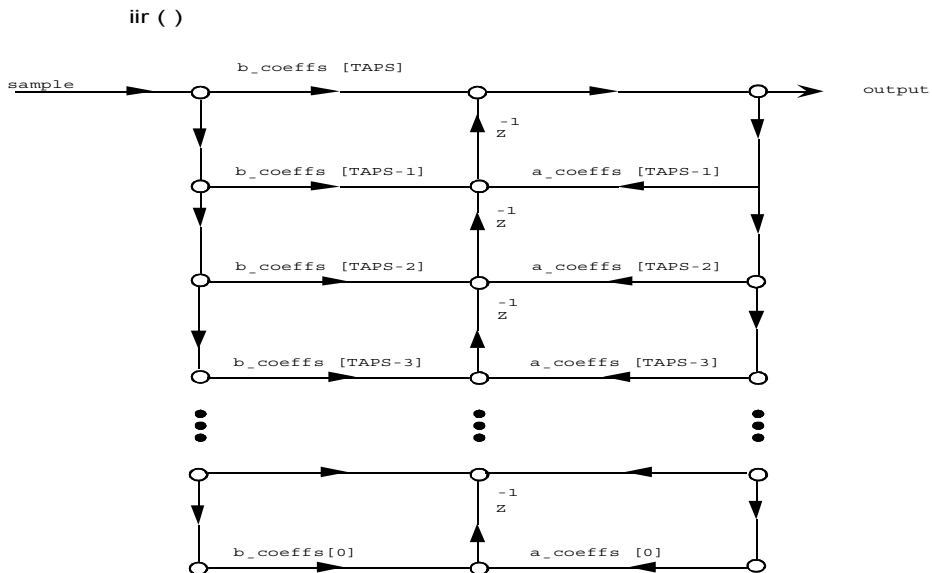
C Run-Time Library Reference

tion. 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 Schaffer 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

$$d(n) = x(n) - a(1) * d(n-1) - a(2) * d(n-2) \dots$$

$$y(n) = b(0) * d(n) + b(1) * d(n-1) + b(2) * d(n-2) \dots$$

$x(n)$ is sample and $y(n)$ is output

Error Conditions

The `irr` 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]=
```

C Run-Time Library Reference

```
{0x1234, 0x5678, 0x9876, 0x5432, 0x1012};

int sample[TAPS] = {0x1111, 0x2222, 0x3333, 0x4444};

extern FRACT state[TAPS+1];
    /* state array needs to be initialised 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, void (*func)(int))) (int);
void (*interrupts(int, 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; 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

C Run-Time Library Reference

interrupts must be declared using the `#pragma interrupt` directive (on page 1-85). The `#pragma altregisters` directive (on page 1-86) 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 2-30.

Table 2-30. 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 default.

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) { /* empty function for this example */ };
main(){
/* enable power down interrupt and register service routine handler */
interrupt(SIG_PWRDWN, handler);
interrupt(SIG_PWRDWN, SIG_IGN); /* disable power down interrupt */
/* enable power down interrupt and register service routine handler */
interruptf(SIG_PWRDWN, handler);
interruptf(SIG_PWRDWN, SIG_IGN); /* disable power down interrupt */
```

See Also

[raise](#), [signal](#)

io_space_read

read I/O space

Synopsis

```
#include <sysreg.h>
int io_space_read(const unsigned int)
```

Description

The `io_space_read` function causes the compiler to emit an instruction to read from I/O space at an address, which is passed as a parameter, and to set the value read from the address as a return value.

The `io_space_read` function is implemented as a compiler built-in; the emitted instruction is inline at the point of `io_space_read` use. The inclusion of the `sysreg.h` include file is mandatory when using `io_space_read`.

Error Conditions

The `io_space_read` function does not return, raise, or set any error conditions.

Example

```
#include <sysreg.h>
main(){
    int value = io_space_read(0xA);
}
```

See Also

[disable_interrupts](#), [enable_interrupts](#), [io_space_read](#), [io_space_write](#), [sysreg_read](#), [sysreg_write](#)

io_space_write

write I/O space

Synopsis

```
#include <sysreg.h>
void io_space_write(const unsigned int, const int)
```

Description

The `io_space_write` function causes the compiler to emit an instruction to write to I/O space at an address, which is passed as the first parameter, the value, which is passed as the second parameter.

The `io_space_write` function is implemented as a compiler built-in; the emitted instruction will be inline at the point of `io_space_write` use. The inclusion of the `sysreg.h` include file is mandatory when using the `io_space_write` function.

The `io_space_write` function does not return a value.

Error Conditions

The `io_space_write` function does not return, raise, or set any error conditions.

Example

```
#include <sysreg.h>
main(){
    io_space_write(0xA, 0xF);
}
```

C Run-Time Library Reference

See Also

[disable_interrupts](#), [enable_interrupts](#), [io_space_read](#), [io_space_write](#),
[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](#)

isctrl

detect control character

Synopsis

```
#include <ctype.h>
int isctrl(int c);
```

Description

The `isctrl` function determines if the argument is a control character (0x00-0x1F or 0x7F). If the argument is not a control character, `isctrl` returns a zero. If the argument is a control character, `isctrl` returns a non-zero value.

Error Conditions

The `isctrl` function does not return any error conditions.

Example

```
#include <ctype.h>
int ch;

for (ch=0; ch<=0x7f; ch++) {
    printf("%#04x", ch);
    printf("%2s", isctrl(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 `+Infinity` or `-Infinity`; otherwise, the function will return a non-zero value.

Error Conditions

`isinf` 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

`isnan` 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++;
```

C Run-Time Library Reference

```
        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 space 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

[isctrl](#)

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 `n` (first parameter) to the exponent of `x` (second parameter).

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](#)

C Run-Time Library Reference

ldiv

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`:

$$\text{result.quot} * \text{denom} + \text{result.rem} = \text{numer}$$

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](#), [ldiv](#)

log

natural logarithm

Synopsis

```
#include <math.h>
double log(double);
```

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);
```

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 `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>
jmp_buf env;

if (setjmp(env) != 0) {
    printf("Unexpected return from subroutine\n");
```

C Run-Time Library Reference

```
        exit(EXIT_FAILURE);
    }
    ...    /* intervening code */
    if(unexpected_error)
        longjmp(env,1);
```

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 lexical value if the `s1` object is greater than the `s2` object. If the `s2` object is greater than the `s1` object, a negative value is returned. A zero is returned 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](#), [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";
int result;

result=memcpy (b, a, 3);  /* *b="SRC" */
```

See Also

[strcpy](#), [strncpy](#)

memmove

move characters from one object to another

Synopsis

```
#include <string.h>
void *memmove(void *s1, const void *s2, size_t n);
```

Description

The `memmove` function moves `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. The entire object is moved 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, 5);    /* *ptr = "ABCDE" */
```

See Also

[memcpy](#), [strcpy](#), [strncpy](#)

C Run-Time Library Reference

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, double *);
```

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](#)

C Run-Time Library Reference

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

[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` 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.

C Run-Time Library Reference

It 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.

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 2-31](#).

Table 2-31. Interrupt Function Signals - Values and Meanings

Sig Value	Definition
SIGPWRDWN	Power down interrupt
SIGIRQ2	Interrupt 2
SIGIRQ1L1	Interrupt 1 (level sensitive)
SIGIRQ1L0	Interrupt 0 (level sensitive)
SIGSPORT0XMIT	Signal Sport 0 transmit
SIGSPORT0RECV	Signal Sport 0 receive
SIGIRQE	Level sensitive

C Run-Time Library Reference

Table 2-31. Interrupt 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 nested by default.

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 on 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](#)

C Run-Time Library Reference

realloc

change memory allocation

Synopsis

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

Description

The function `realloc` 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 can not 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 to 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

```
#include <setjmp.h>
jmp_buf env;

if (setjmp(env) != 0)
{
```

C Run-Time Library Reference

```
        printf("Unexpected return from subroutine\n");
        exit(EXIT_FAILURE);
    }
    /* intervening code */
    if(unexpected_error)
        longjmp(env,1);
```

See Also

[longjmp](#)

signal

define signal handling

Synopsis

```
#include <signal.h>
void (*signal(int sig, void (*func)(int))) (int);
void (*signalf(int, void (*func)(int))) (int);
void (*signals(int, 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`

C Run-Time Library Reference

must be declared using the `#pragma interrupt` directive (see [on page 1-85](#)). The `altregisters` directive (see [on page 1-86](#)) 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 2-32](#).

Table 2-32. Signal Function Signals - Values and Meanings

Sig Value	Definition
SIGPWRDWN	Power down interrupt
SIGIRQ2	Interrupt 2
SIGIRQL1	Interrupt 1 (level sensitive)
SIGIRQLO	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 default.

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 service routine handler */
    signal(SIG_PWRDWN, handler);
    signal(SIG_PWRDWN, SIG_IGN); /* disable power down interrupt */
    /* enable power down interrupt and register service routine handler */
    signal(SIG_PWRDWN, handler);
    signal(SIG_PWRDWN, SIG_IGN); /* disable power down interrupt */
}
```

See Also

[interrupt](#), [raise](#),

sin

sine

Synopsis

```
#include <math.h>
double sin(double);
fract16 sin_fr16 (fract16);
```

Description

The `sin` function returns the sine of the input parameter. The input is interpreted as a radian; the output is in the range $[-1, 1]$.

The `sin_fr16` function is defined by the domain of `0x8000` to `0x7FFF` corresponding to $[-\pi/2, \pi/2]$. This domain only covers half a cycle. Because of the periodic nature of the function, this is deemed sufficient to derive a full period if required. The input argument is in radians.

Error Conditions

The `sin` function does not return an error condition.

Example

```
#include <math.h>
double y;
y = sin(3.14159);    /* y = 0.0 */
```

See Also

[asin](#), [cos](#)

sinh

hyperbolic sine

Synopsis

```
#include <math.h>
double sinh(double);
```

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](#)

C Run-Time Library Reference

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 null 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` to 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](#), [strncpy](#)

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 null terminating 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

[strncat](#)

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](#), [strncmp](#)

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 will not end with a null.

The behavior of `strncpy` is undefined if 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 nulls 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](#), [strncpy](#)

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

[strcspn](#), [strpbrk](#)

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](#)

trspns

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 portion of string to double representation

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 of the form:

`[whitespace] [sign] [digits] [.digits] [{d|D|e|E}[sign]digits]`

The `whitespace` token may consist of space and tab characters, which are ignored; `sign` is either plus (+) or minus (–); and `digits` are one or more decimal digits. If no digits appear before the radix character (.), at least one digit must appear after the radix character.

The decimal digits can be followed by an exponent, which consists of an introductory letter (d, D, e, or E) and an optionally signed integer. If neither an exponent part nor a radix character appears, a radix character is assumed to follow the last digit in the string. The first character that does not fit this form stops 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 appropri-

C Run-Time Library Reference

ate) 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" */
```

See Also

[atof](#), [strtoul](#), [strtoul](#)

strtodf

convert portion of string to float representation

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 of the form:

`[whitespace] [sign] [digits] [.digits] [{d|D|e|E}[sign]digits]`

The `whitespace` token may consist of space and tab characters, which are ignored; `sign` is either plus (+) or minus (–); and `digits` are one or more decimal digits. If no digits appear before the radix character (`.`), at least one digit must appear after the radix character.

The decimal digits can be followed by an exponent, which consists of an introductory letter (`d`, `D`, `e`, or `E`) and an optionally signed integer. If neither an exponent part nor a radix character appears, a radix character is assumed to follow the last digit in the string. The first character that does not fit this form stops 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 appropri-

C Run-Time Library Reference

ate) `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" */
```

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"        */
```

C Run-Time Library Reference

```
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 the 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.

C Run-Time Library Reference

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.

C Run-Time Library Reference

Error Conditions

The `strtoul` 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, `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](#), [strtod](#) [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 */
```

C Run-Time Library Reference

See Also

[strcmp](#), [strcoll](#), [strncpy](#)

sysreg_read

read from non-memory-mapped register

Synopsis

```
#include <sysreg.h>
int sysreg_read(const unsigned 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 first parameter for `sysreg_read` should be a member of the `SysReg` enumeration defined in `sysreg.h`. This enumerations is used to map the non-memory-mapped actual registers to a small constant defined as a user friendly name. This enumerations is:

```
enum SysReg {
    sysreg_ASTAT = 0x0, // ASTAT register - arithmetic status
    sysreg_SSTAT = 0x1  // SSTAT register - shifter status
    sysreg_MSTAT = 0x2, // MSTAT register - multiplier status
    sysreg_ICNTL = 0x3, // ICNTL register - interrupt control
    sysreg_IMASK = 0x4, // IMASK register - interrupts enabled mask
    sysreg_IFC = 0x5    // IFC register - interrupt force and clear
};
```

The `sysreg_read` function is implemented as a compiler built-in; the emitted instructions will be inline at the point of `sysreg_read` use.

The inclusion of the `sysreg.h` include file is mandatory when using `sysreg_read`.

C Run-Time Library Reference

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_read](#), [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, which is passed as the second parameter.

The first parameter for `sysreg_write` should be a member of the `SysReg` enumeration defined in `sysreg.h`. This enumerations is used to map the non-memory-mapped actual registers to a small constant, which is defined as a user friendly name. This enumerations is:

```
enum SysReg {
    sysreg_ASTAT = 0x0, // ASTAT register - arithmetic status
    sysreg_SSTAT = 0x1, // SSTAT register - shifter status
    sysreg_MSTAT = 0x2, // MSTAT register - multiplier status
    sysreg_ICNTL = 0x3, // ICNTL register - interrupt control
    sysreg_IMASK = 0x4, // IMASK register - interrupts enabled mask
    sysreg_IFC = 0x5    // IFC register - interrupt force and clear
};
```

The `sysreg_write` function is implemented as a compiler built-in; the emitted instructions will be inline at the point of `sysreg_write` use.

The inclusion of the `sysreg.h` include file is mandatory when using `sysreg_write`.

C Run-Time Library Reference

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](#), [sysreg_write](#)

tan

tangent

Synopsis

```
#include <math.h>
double tan(double);
fract16 tan_fr16 (fract16);
```

Description

The `tan` function returns the tangent of the input argument.

The `tan_fr16` function is only defined for input values between $-\pi/4$ ($=0\times9B78$) and $\pi/4$ ($=0\times6488$). The input argument is in radians. Output values range from 0×8000 to $0\times7FFF$. The library function returns 0 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);
```

Description

The `tanh` function returns the hyperbolic tangent of the `input` argument.

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](#)

C Run-Time Library Reference

va_arg

get next argument in variable list

Synopsis

```
#include <stdarg.h>
void va_arg(va_list ap, type);
```

Description

The `va_arg` macro can only be used after the `va_start` macro has been invoked. The `va_arg` macro uses the pointer initialized by `va_start` to return the value and type of the next argument in the list of optional arguments. It then increments the pointer. The header file `stdarg.h` defines a pointer type called `va_list` that is used to access the list of variable arguments.

The type parameter is a type name to which a `*` may be appended to obtain a pointer to an object of type `type`. If there is no next variable, then there is no defined behavior for `va_arg`.

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

reset variable list pointer

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 2-152.

See Also

[va_arg](#), [va_start](#)

va_start

set variable list pointer

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. The macro initializes a pointer of type `va_list` which is used by `va_arg` to walk through the list of variable 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 2-152](#).

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++ 3.0 Linker and Utilities Manual for ADSP-218x and ADSP-219x DSPs* and *VisualDSP++ 3.0 C Assembler and Preprocessor Manual for ADSP-218x and ADSP-219x DSPs*.

Tools Differences

VisualDSP++ 3.0 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++ 3.0 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 (.DXX), and the debugger requires DSP executables to be in the ELF/DWARF-2 format. As a result, AEXE-formatted files must be

Tools Differences

recompiled or reassembled in order to be debugged under VisualDSP++ 3.0. An ELF/DWARF-to-AEXE conversion utility is available in VisualDSP++ 3.0 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++ 3.0 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++ 3.0. 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++ 3.0 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 1-66.

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++/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 1-70](#).

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++ 3.0
-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-bitfields	Negative form of the -fsigned-bitfield and -funsigned-bitfield.	Removed. Bitfield is signed or unsigned based on the sign of the type definition declaring the bitfield.
-fsigned-char -funsigned-char	Specify whether to default to signed or unsigned char type.	Replaced with -signed-char and -unsigned-char.

Tools Differences

Table A-1. Obsolete and Replaced Switches (Cont'd)

Switch Name	Operation under Release 6.1	Change for VisualDSP++ 3.0
-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 <code>seg_data1</code> section. Definition in the <code>.LDF</code> can place this section in RAM or ROM.
-I-	Modifies directory search order.	Removed.
-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 processor does not support paging.
-mdmdata= -mpmdata= -mdcode=	Specify architecture file segments.	Removed. You can control placement of object file segments using the <code>.SECTION</code> directive in the Linker Description 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 <code>pm</code> memory.	Replaced with -jump-{dm pm}.
-mreserved=	Instructs the compiler not to use specified registers.	Removed.

Table A-1. Obsolete and Replaced Switches (Cont'd)

Switch Name	Operation under Release 6.1	Change for VisualDSP++ 3.0
<code>-mrom</code>	Make the module a ROM module.	Removed.
<code>-msmall-code</code>	Optimize for size, not for speed.	Removed.
<code>-mstatic-spill</code>	Use dm memory when all register are used.	Removed.
<code>-nostdinc</code>	Do not search standard system directories for header files.	Replaced with <code>-no-std-inc</code> .
<code>-nostdlib</code>	Do not use standard system libraries and startup files when linking.	Replaced with <code>-no-std-lib</code> .
<code>-runhdr</code>	Specify a particular run-time header.	Removed. This feature can now be defined in the <code>.LDF</code> .
<code>-traditional-cpp</code>	Support some preprocessing features.	Removed.

Warnings

The VisualDSP++ 3.0 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 ++ 3.0 Warning Switch	Description
<code>-warn-protos</code>	Produce a warning when a function is called without a full prototype.
<code>-Wdriver-limit <i>number</i></code>	Set a maximum <i>number</i> of driver errors.
<code>-Werror-limit <i>number</i></code>	Set a maximum <i>number</i> of compiler errors.

Table A-2. Compiler -- New Warning Switches (Cont'd)

VisualDSP ++ 3.0 Warning Switch	Description
-Wremarks	Indicates that the compiler may issue remarks, which are diagnostic messages even milder than warnings.
-Wterse	Enable terse warnings.
-Wpedantic	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 <i>num</i> 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++ 3.0 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++ 3.0 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++ 3.0 run-time model, see [“C Run-Time Model and Environment” on page 1-96..](#)

I INDEX

Symbols

- + $\dagger$ debug-types compiler switch [1-21](#)
- .IDL file [1-94](#)
- @ filename (command file) compiler switch [1-18](#)
- __GROUPNAME__ macro [1-38](#)
- __HOSTNAME__ macro [1-38](#)
- __MACHINE__ macro [1-38](#)
- __REALNAME__ macro [1-38](#)
- __SIGNED_CHARS__ macro [1-37](#)
- __SYSTEM__ macro [1-38](#)
- __USERNAME__ macro [1-38](#)

A

- A (assert) compiler switch [1-18](#)
- abend (see abort function)
- abort (abnormal program end) function [2-16](#)
- abs (absolute value) function [2-17](#)
- acos (arc cosine) function [2-18](#)
- acos_fr16 function [2-18](#)
- activation record [1-99](#)
- aggregate assignment support (compiler) [1-72](#)
- aggregate constructor expression support [1-72](#)
- align num pragma [1-84](#)

- allocate memory (see calloc, free, malloc, realloc functions)
- alphanumeric character test (see isalnum function)
- alter(x) macro [1-114](#)
- altregisters pragma [1-86](#)
- alttok (alternative tokens) compiler switch [1-19](#)
- analog (Analog C compilation) compiler switch [1-20](#)
- ansi (ANSI standard C compilation) compiler switch [1-20](#)
- ANSI C extensions [A-4](#)
- ANSI standard compiler [1-20](#), [1-23](#)
- array length [1-68](#)
- array search, binary (see bsearch function)
- ASCII string (see atof, atoi, atol functions)
- asin (arc sine) function [2-19](#)
- asin_fr16 function [2-19](#)
- asm compiler keyword [1-47](#)
- asm compiler keyword (see inline assembly language support keyword (asm))
- asm_sprt.h – mixed C/assembly support [1-113](#), [2-6](#)

INDEX

assembler 1-2
assembly constructs 1-51
assembly language subroutines
 1-109
assignments 1-63
atan (arc tangent) function 2-20
atan_fr16 function 2-20
atan2 (arc tangent of quotient)
 function 2-21
atan2_fr16 function 2-21
atexit (select exit function) function
 2-22
atof (convert string to double)
 function 2-23
atoi (convert string to integer)
 function 2-24
atol (convert string to long integer)
 function 2-25
autobuffering 1-36, 1-98
automatic variables 1-61

B

binary array search (see bsearch
 function)
biquad (biquad filter) function 2-26
bool (see Boolean type support
 keywords (bool, true, false))
boolean type support keywords 1-67
Boolean type support keywords
 (bool, true, false) 1-47
bsearch (binary search in sorted
 array) function 2-29
-build-lib (build library) compiler
 switch 1-20

built-in functions 1-74
builtin functions 1-73

C

-C (comments) compiler switch
 1-20
-c (compile only) compiler switch
 1-20
C compiler and C run-time library
 A-3
C language extensions 1-47
 aggregate assignments 1-72
 asm keyword 1-50
 bool keyword 1-67
 C++-style comments 1-48
 compound statements 1-69
 false keyword 1-47
 indexed initializers 1-71
 inline keyword 1-49
 non-constant initializers 1-70
 segment keyword 1-47
 true keyword 1-47
C run-time environment
 see also C Run-time library
C run-time library
 guide 2-2
 header files 2-5--??
C/assembly interface, *see* mixed
 C/assembly programming)
calling assembly language
 subroutine 1-109
calling library functions 2-2
calloc (allocate and initialize
 memory) function 2-31

- ceil (ceiling) function [2-32](#)
- character string search (see `strchr` function)
- character string search, recursive (see `strrchr` function)
- circular buffer length registers [1-105](#)
- `clear_interrupt` (clear a pending signal) function [2-33](#)
- `clobber` [1-52](#)
- `clock` (count clock ticks) function [2-36](#)
- command-line interface [1-6–??](#)
- compatibility call [1-117](#), [1-118](#)
- compiler
 - C extensions [1-47](#)
 - intrinsic functions [1-73](#)
 - legacy support [A-1–A-9](#)
 - reference [1-96](#)
- compiler command-line switches
 - `-@ filename` (command file) [1-18](#)
 - `-21{part number}` [1-18](#)
 - `-a` (assert) [1-18](#)
 - `-alttok` (alternative tokens) [1-19](#)
 - `-analog` (analog c compilation) [1-20](#)
 - `-ansi` (ISO/ANSI standard C compilation) [1-20](#)
 - `-build-lib` (build library) [1-20](#)
 - `-C` (comments) [1-20](#)
 - `-c` (compile only) [1-20](#)
 - `-debug-types` [1-21](#)
 - `-default-linkage-` (assembler linkage) [1-22](#)
 - `-Dmacro` (define macro) [1-21](#)
 - `-dollar` (dollar format) [1-22](#)
 - `-dry` (verbose dry-run) [1-22](#)
 - `-dryrun` (terse dry-run) [1-22](#)
 - `-E` (stop after preprocessing) [1-23](#)
 - `-EE` (run after preprocessing) [1-23](#)
 - `-extra-keywords` [1-23](#)
 - `-flags` (command-line input) [1-23](#)
 - `-g` (generate debug information) [1-24](#)
 - `-H` (list headers) [1-24](#)
 - `-h[elp]` (command-line help) [1-25](#)
 - `-HH` (list headers and compile) [1-25](#)
 - `-I-` (start include directory) [1-25](#)
 - `-I` directory (include search directory) [1-26](#)
 - `-include filename` (include file) [1-26](#)
 - `-J` (old-style preprocessing) [1-26](#)
 - `-jump-{dm|pm}` (select jump table memory type) [1-26](#)
 - `-L` (library search directory) [1-26](#)
 - `-l` (link library) [1-27](#)
 - `-M` (generate make rules only) [1-27](#)
 - `-make-autostatic` (make automatic variables static) [1-28](#)
 - `-map` (generate a memory map) [1-28](#)
 - `-MM` (generate make rules and compile) [1-27](#)
 - `-MQ` [1-28](#)
 - `-Mt` (output make rules) [1-28](#)
 - `-no-alttok` (disable tokens) [1-29](#)

INDEX

- no-builtin (no built-in functions) 1-29
- no-defs (disable defaults) 1-29
- no-dir-warnings 1-29
- no-dollar (disable dollar format) 1-29
- no-extra-keywords (disable short-form keywords) 1-30
- no-inline (disable inline keyword) 1-30
- no-std-def (disable standard macro definitions) 1-30
- nostd-inc (disable standard include search) 1-30
- no-std-lib (disable standard library search) 1-31
- no-widen-muls (disable widening multiplications) 1-31
- O (enable optimizations) 1-31
- o (output file) 1-32
- Os (optimize for size) 1-32
- P (omit line numbers) 1-32
- path-install directory (installation location) 1-33
- path-output (non-temporary files location) 1-33
- path-temp (temporary files location) 1-33
- pch (precompiled header) 1-33
- pchdir (locate PCHRepository) 1-33
- pedantic (ANSI standard warnings) 1-34
- pedantic-errors (ANSI C errors) 1-34
- pplist (preprocessor listing) 1-34
- proc identifier 1-35
- R (add source directory) 1-35
- R- (disable source path) 1-35
- reserve (reserve register) 1-36
- S (stop after compilation) 1-36
- s (strip debugging information) 1-36
- save-temps (save intermediate files) 1-36
- show (display command line) 1-37
- signed-char (make char signed) 1-37
- syntax-only (just check syntax) 1-37
- sysdefs (system definitions) 1-37
- T (linker description file) 1-38
- time (time the compiler) 1-38
- traditional (traditional C compilation) 1-39
- Umacro (undefine macro) 1-39
- unsigned-char (make char unsigned) 1-39
- v (version and verbose) 1-39
- val-global (add global names) 1-40
- verbose (display command line) 1-40
- version (display version) 1-40
- w (display all warnings) 1-41
- W (override error messages) 1-41
- warns-protos (prototypes)

- warning) 1-42
- Wdriver-limit (maximum process errors) 1-40
- Werror-limit (maximum compiler errors) 1-40
- Wremarks (enable diagnostic warnings) 1-41
- write-files (enable driver I/O redirection) 1-42
- Wterse (terse warnings) 1-41
- xref file (cross-reference list) 1-42
- compiler diagnostics A-4
- compiler obsolete/modified switches A-5
- compiler optimization disabling 1-31
- compound statements with in expressions support (compiler) 1-69
- const keyword 1-39
- control character test (see iscntrl function)
- controlling interrupts 1-76
- convert
 - characters (see tolower, toupper functions)
 - strings (see atof, atoi, atol, strtok, strtol, strtoul, functions)
- converting
 - portion of string to double representation 2-129
 - portion of string to float representation 2-131
- copysign 2-35

- cos (cosine) function 2-36
- cos_fr16 function 2-36
- cosh (hyperbolic cosine) function 2-37
- cotangent 2-38
- C-type functions
 - isalnum 2-69
 - iscntrl 2-71
 - isdigit 2-72
 - isgraph 2-73
 - islower 2-74, 2-76
 - isprint 2-79
 - ispunct 2-80
 - isspace 2-81
 - isupper 2-82
 - isxdigit 2-83
 - tolower 2-150
 - toupper 2-151

D

- D (define macro) compiler switch 1-21, 1-39
- DAG1 registers 1-107
- data register 1-106
- deallocate memory (see free function)
- debugging information 1-36
- dedicated registers 1-105
- default-linkage- (assembler linkage) compiler switch 1-22
- demean_buffer (remove mean of data buffer) function 2-39
- differences between releases A-1

INDEX

- disable_interrupts (disable interrupts) function [2-41](#)
- disabling compiler optimization [1-31](#)
- div (division) function [2-42](#)
- division (see div, ldiv functions)
- dm (see dual memory support keywords (pm dm))
- dollar (dollar format) compiler switch [1-22](#)
- double data type [1-44](#)
- double representation [2-129](#)
- dry (verbose dry-run) compiler switch [1-22](#)
- dryrun (terse dry-run) compiler switch [1-22](#)
- dual memory support keywords (pm dm) [1-43](#), [1-47](#), [1-61](#)
- dual-memory space [1-61](#)
- duplicate sign bit elimination [1-78](#)

E

- E stop after preprocessing) compiler switch [1-23](#)
- EE (run after preprocessing) compiler switch [1-23](#)
- enable_interrupts (enable interrupts) function [2-43](#)
- end (see atexit, exit functions)
- ETSI library [1-80](#), [1-81](#)
- ETSI support [1-76](#)
 - for fract16 and fract32 data types [1-81](#)

- exit (normal program termination) function [2-44](#)
- exit macro [1-114](#)
- exp (exponential) function [2-45](#)
- extensions
 - using compiler language [1-3](#)
- extra-keywords (not quite -analog) compiler switch [1-23](#)

F

- fabs (float absolute value) function [2-46](#)
- false (see Boolean type support keywords (bool, true, false))
- far jump return (see longjmp, setjmp functions)
- fftN (N-point complex input FFT) function [2-47](#)
- ffts.h – Fast Fourier Transforms [2-7](#)
- file
 - searches [1-9](#)
- file extension [1-7](#), [1-9](#), [1-10](#)
- file searching
 - <filename> [1-95](#)
- filenames [1-18](#)
- filter functions
 - biquad [2-26](#)
- filters.h – DSP filters [2-8](#)
- fir (finite impulse response filter) function [2-50](#)
- flags (command line input) compiler switch [1-23](#)
- float representation [2-131](#)
- float.h – floating point [2-8](#)

fmod (floating-point modulus)
 function [2-53](#)
 frame pointer [1-100](#)
 free (deallocate memory) function
[2-54](#)
 frexp (separate fraction and
 exponent) function [2-55](#)
 function_entry macro [1-113](#)
 functions
 primitive I/O [2-10](#)

G

-g (generate debug information)
 compiler switch [1-24](#)
 g21-based options and extensions
[A-3](#)
 getsfirst macro [1-116](#)
 getsnext macro [1-116](#)
 graphical character test (see isgraph
 function)

H

-H (list *.h) compiler switch [1-24](#)
 header files [1-30](#), [1-92](#)
 asm_sprt.h [2-6](#)
 C run-time library [2-5](#)–??
 def2181.h [2-7](#)
 defESP202.h [2-7](#)
 ffts.h [2-7](#)
 filters.h [2-8](#)
 fract.h [2-8](#)
 macros.h [2-8](#)
 misc.h [2-9](#)
 signal.h [2-9](#)

sport.h [2-9](#)
 stdio.h [2-10](#)
 sysreg.h [1-74](#), [2-10](#)
 -help (command line help) compiler
 switch [1-25](#)
 hexadecimal digit test (see isxdigit
 function)
 -HH (list *.h and compile) compiler
 switch [1-25](#)

I

-I (include search directory)
 compiler switch [1-26](#), [1-30](#)
 -I- (start include directory) compiler
 switch [1-25](#)
 I/O space [1-74](#)
 ifftN (inverse complex FFT)
 function [2-56](#)
 iir (infinite impulse response) filter
 function [2-59](#)
 -include (include file) compiler
 switch [1-26](#)
 include directives [1-94](#)
 indexed initializer support [1-70](#)
 indexed initializers [A-4](#)
 inline assembler [1-50](#)
 inline assembly code [1-96](#)
 inline assembly language support
 keyword (asm) [1-50](#)
 construct
 template [1-51](#)
 construct I/O operands [1-59](#)
 construct optimization [1-58](#)
 construct template [1-51](#)

INDEX

- construct template operands 1-55
- constructs with multiple
 - instructions 1-58
- macros containing asm 1-60
- inline function support keyword
 - (inline) 1-39, 1-43, 1-47, 1-49
- interface support macros 1-113
- interfacing C/assembly (see mixed C/assembly programming)
- interrupt
 - RTI instruction 1-97
- interrupt (define interrupt handling) function 2-63
- interrupt control 1-76
- interrupt pragma 1-85, 1-86
- interrupt table
 - 218x_int_tab.asm 1-97
- interrupt table symbols
 - ____system_start 1-97
 - __lib_int_determiner 1-97
 - _lib_int_table 1-97
- interruptf function 2-63
- interrupts (see clear_interrupt, interruptf, interrupts, signal, raise functions)
- interrupts interrupt dispatcher 2-63
- intrinsic functions 1-73
- io_space_read (read I/O space)
 - function 2-66
- io_space_write (write I/O space)
 - function 2-67
- isalnum (detect alphanumeric character) function 2-69

- isalpha (detect alphabetic character)
 - function 2-70
- isctrl (detect control character)
 - function 2-71
- isdigit (detect decimal digit)
 - function 2-72
- isgraph (detect printable character)
 - function 2-73
- islower (detect lowercase character)
 - function 2-76
- isprint (detect printable character)
 - function 2-79
- ispunct (detect punctuation character) function 2-80
- isspace (detect whitespace character)
 - function 2-81
- isupper (detect uppercase character)
 - function 2-82
- isxdigit (detect hexadecimal digit)
 - function 2-83

J

- J compiler switch 1-26
- jump-{dm|pm} (select jump table memory type) compiler switch 1-26

K

- keywords (compiler) (see compiler C extensions)

L

- L (library search directory)
 - compiler switch 1-26

- l (link library) compiler switch
1-27, 1-31
- labs (long integer absolute value)
function 2-84
- language extensions (compiler) (see
compiler C extensions)
- ldexp (multiple by power of 2)
function 2-85
- ldiv (division) function 2-86
- ldiv (division, long) function 2-86
- leaf procedure 1-99
- leaf_entry macro 1-114
- leaf_exit macro 1-114
- library
 - header files, working with 2-5
 - linking functions 2-3
 - source code, working with 2-4
- linkage pragmas 1-87
- linkage_name pragma 1-87
- linking library functions 2-3
- localtime (return address of encoded
calendar time) function 2-87
- log (natural logarithm) function
2-87
- log, logf (log base e) function 2-91
- log10 (base 10 logarithm) function
2-88
- log10, log10f (log base 10) function
2-91
- long jump (see longjmp, setjmp
functions)
- longjmp (far jump return) function
2-91

- longjmp (second return from
setjmp) function 2-89
- lower case (see islower, tolower
functions)

M

- M (generate make rules only)
compiler switch 1-27
- macros
 - __NO_LONG_LONG 1-91
 - _ADSP21_ 1-89
 - _ADSP21XX_and_ADSP218X_
1-89
 - _ANALOG_EXTENSIONS_
1-90
 - _DATE_ 1-90
 - _DOUBLES_ARE_FLOATS_
1-90
 - _ECC_ 1-90
 - _EDG_ 1-90
 - _EDG_VERSION_ 1-90
 - _FILE_ 1-90
 - _LINE_ 1-91
 - _NO_BUILTIN 1-91
 - _SIGNED_CHARS_ 1-91
 - _STDC_ 1-91
 - _STDC_VERSION_ 1-91
 - _TIME_ 1-92
- circular buffer 2-8
- EDOM 2-9
- ERANGE 2-9
- for C/Assembly interface support
1-113
- HUGE_VAL 2-9

INDEX

- macros and asm() C program
 - constructs [1-60](#)
 - make_auto_static pragma [1-87](#)
 - make-autostatic (make automatic variables static) compiler switch [1-28](#), [1-88](#)
 - malloc (allocate memory) function [2-91](#)
 - map (generate a memory map) compiler switch [1-28](#)
 - map file [1-28](#)
 - math.h – Mathematics [2-9](#)
 - max (find larger, int) function [2-105](#)
 - mean (mean, array) function [2-105](#)
 - memchr (find first occurrence of character) function [2-92](#)
 - memcmp (compare objects) function [2-93](#)
 - memcpy (copy characters from one object to another) function [2-94](#)
 - memmove (move characters from one object to another) function [2-95](#)
 - memory (see calloc, free, malloc, memcmp, memcpy, memset, memmove, memchar, realloc functions)
 - memset (set range of memory to a character) function [2-96](#)
 - mixed C/assembly
 - C run-time model [1-96](#)
 - calling C from assembler [1-112](#)
 - naming conventions [1-116](#)
 - programming [1-96](#)
 - asm() constructs [1-50](#), [1-51](#), [1-55](#), [1-58](#), [1-59](#), [1-60](#)
 - support macros [1-113](#)
 - mixed C/assembly programming
 - asm() constructs [1-51](#)
 - MM (generate make rules and compile) compiler switch [1-27](#)
 - modf (separate integral and fractional parts) function [2-97](#)
 - move memory range (see memmove function)
 - MQ compiler switch [1-28](#)
 - MSB extraction [1-77](#)
 - MSTAT (mode status) register [1-105](#)
 - Mt filename (output make rule) compiler switch [1-27](#)
 - multidimensional arrays [1-69](#)
 - multi-line asm() C program
 - constructs [1-58](#)
-
- N
 - naming conventions
 - C and assembly [1-116](#)
 - no-alttok (disable alternative tokens) compiler switch [1-29](#)
 - no-builtin (no builtin functions) compiler switch [1-29](#)
 - no-def (disable defaults) compiler switch [1-29](#)
 - no-dir-warnings (disable directory warning) compiler switch [1-29](#)

- no-dollar (disable dollar format)
C++ mode compiler switch
[1-29](#)
- no-extra-keywords (disable
short-form keywords) compiler
switch [1-30](#)
- no-inline (disable inline keyword)
compiler switch [1-30](#)
- non-constant initializer support
(compiler) [1-70](#)
- non-memory-mapped registers [1-75](#)
- no-std-def (disable standard macro
definitions) compiler switch
[1-30](#)
- no-std-inc (disable standard
include search) compiler switch
[1-30](#)
- no-std-lib (disable standard library
search) compiler switch [1-31](#)
- no-widen-muls (disable widening
multiplications) compiler
switch [1-31](#)

O

- O (enable optimization) compiler
switch [1-31](#)
- o (output file) compiler switch
[1-32](#)
- obsolete warning switches [A-8](#)
- obsolete/modified switches [A-5](#)
- OldAsmCall declaration [1-118](#)
- optimization pragmas [1-86](#)
- optimizing asm() C program
constructs [1-58](#)

- Os (optimize for size) compiler
switch [1-32](#)

P

- P (omit line numbers) compiler
switch [1-32](#)
- path-install (installation location)
compiler switch [1-33](#)
- path-output (non-temporary files
location) compiler switch [1-33](#)
- path-temp (temporary files
location) compiler switch [1-33](#)
- path-tool (tool location) compiler
switch [1-32](#)
- pch (precompiled header) compiler
switch [1-33](#)
- pchdir (locate PCHRepository)
compiler switch [1-33](#)
- PCHRepository directory [1-33](#)
- pedantic (ANSI standard warnings)
compiler switch [1-34](#)
- pedantic-errors (ANSI C errors)
compiler switch [1-34](#)
- placement support keyword
(segment) [1-47](#), [1-66](#)
- pm (see dual memory support
keywords (pm dm))
- pointer class support keyword
(restrict) [1-47](#), [1-67](#)
- pow (raise to a power) function [2-98](#)
- pplist (preprocessor listing)
compiler switch [1-34](#)
- pragmas [1-84](#)
align num [1-84](#), [1-85](#)

INDEX

- altregisters [1-86](#)
- interrupt handler [1-85](#)
- linking [1-87](#)
- make_auto_static [1-87](#)
- optimization level [1-86](#)
- weak_entry [1-87](#)
- precompiled header [1-33](#)
- predefined macros [1-89](#)
 - __NO_LONG_LONG [1-91](#)
 - __ADSP21{81|83|84|85|86|87|88|89}__ [1-89](#)
 - __ADSP21XX__ ,
__ADSP218X__ [1-89](#)
 - __ANALOG_EXTENSIONS__
[1-90](#)
 - __DATE__ [1-90](#)
 - __DOUBLES_ARE_FLOATS__
[1-90](#)
 - __ECC__ [1-90](#)
 - __EDG__ [1-90](#)
 - __EDG_VERSION__ [1-90](#)
 - __FILE__ [1-90](#)
 - __LINE__ [1-91](#)
 - __NO_BUILTIN [1-91](#)
 - __SIGNED_CHARS__ [1-91](#)
 - __STDC__ [1-91](#)
 - __STDC_VERSION__ [1-91](#)
 - __TIME__ [1-92](#)
 - __LANGUAGE_C [1-91](#)
- preprocessing
 - IDL files [1-95](#)
- preprocessing a program [1-89](#)
- preprocessor features [1-89](#)
- preprocessor macros [1-89](#)

- primitive I/O functions [2-10](#)
- printable character test (see isprint
function)
- proc (target processor) compiler
switch [1-35](#)
- procedure call [1-102](#), [1-103](#)
- procedure call/return [1-103](#)
- program control functions
 - calloc [2-31](#)
 - malloc [2-91](#)
 - realloc [2-104](#)
- punctuation character test (ispunct)
function [2-80](#)
- putsfirst macro [1-115](#)
- putsnext macro [1-115](#)
- PX register [1-61](#)

Q

- qsort (quicksort) function [2-99](#)

R

- R- (disable source path) compiler
switch [1-35](#)
- R (search for source files) compiler
switch [1-35](#)
- raise (raise a signal) function [2-101](#)
- rand (random number generator)
function [2-103](#)
- random number (see rand, srand
functions) [2-103](#)
- readsfirst macro [1-115](#)
- readsnext macro [1-115](#)
- realloc (allocate used memory)
function [2-105](#)

- realloc (change memory allocation)
 - function [2-104](#)
- real-time signals (see
 - clear_interrupt, interruptf,
 - interrupts, poll_flag_in, raise,
 - signal functions)
- register classification [1-104](#)
 - circular buffer [1-105](#)
 - dedicated [1-105](#)
 - mode status (MSTAT) [1-105](#)
 - preserved [1-104](#)
 - save/scratch [1-105](#)
- register usage (see mixed C/assembly programming)
- registers
 - DAG1 [1-107](#)
 - DAG2 [1-108](#)
 - data [1-106](#)
- registers for asm() constructs [1-55](#)
- reordering asm() C program
 - constructs [1-58](#)
- reserve compiler switch [1-36](#)
- restore_reg macro [1-115](#)
- restrict (see pointer class support
 - keyword (restrict))
- restrict operator keyword [1-67](#)
- return from interrupt (RTI)
 - instruction [1-97](#)
- return value [1-103](#)
- return values [1-103](#)
- run-time environment (see mixed C/assembly programming)

- run-time environment
 - programming (see mixed C/assembly programming)
- run-time header [1-96](#)
 - 218x_hdr.asm [1-96](#)
- run-time model [A-9](#)

S

- S (stop after compilation) compiler switch [1-36](#)
- s (strip debugging information)
 - compiler switch [1-36](#)
- saturation [1-78](#)
- save/scratch registers [1-105](#)
- save_reg macro [1-114](#)
- save-temps (save intermediate files)
 - compiler switch [1-36](#)
- search character string (see strchr, strchr functions)
- search memory, character (see memchar function)
- search path
 - for include files [1-26](#)
 - for library files [1-26](#)
- searching for #included files [1-95](#)
- segment (see placement support
 - keyword (segment))
- segment placement [A-3](#)
- set jump (see longjmp, setjmp functions)
- setjmp (label for external linkage)
 - function [2-105](#)
- setjmp (long jump) function [2-105](#)
- setting compiler options [1-96](#)

INDEX

- show (display command line)
 - compiler switch [1-37](#)
- signal (define signal handling)
 - function [2-107](#)
- signalf function [2-107](#)
- signals (see `clear_interrupt`,
`interruptf`, `interrupts`,
`poll_flag_in`, `raise`, `signal`
functions)
- signed keyword [1-39](#)
- signed-char (make char signed)
 - compiler switch [1-37](#)
- sin (sine) function [2-110](#)
- sin_fr16 function [2-110](#)
- sinh (hyperbolic sine) function
[2-111](#)
- SPORT [1-98](#)
- sqrt (square root) function [2-114](#)
- srand (random number seed)
 - function [2-113](#)
- stack frame [1-99](#)
 - free space [1-101](#)
 - incoming arguments [1-100](#)
 - linkage information [1-101](#)
 - local variables/temporaries [1-101](#)
 - outgoing arguments [1-101](#)
 - system-wide specifications [1-102](#)
- stack pointer [1-100](#)
- stack usage pragma [1-87](#)
- standard conformances [1-3](#)
- standard header files
 - `assert.h` [2-7](#)
 - `ctype.h` [2-7](#)
 - `errno.h` [2-7](#)
 - `float.h` [2-8](#)
 - `math.h` [2-9](#)
 - `signal.h` [2-9](#)
 - `stdio.h` [2-10](#)
- standard library functions
 - `abort` [2-16](#)
 - `atexit` [2-22](#)
 - `atof` [2-23](#)
 - `atol` [2-25](#)
 - `bsearch` [2-29](#)
 - `div` [2-42](#)
 - `exit` [2-44](#)
 - `free` [2-54](#)
 - `ldiv` [2-86](#)
 - `malloc` [2-91](#)
 - `rand` [2-103](#)
 - `srand` [2-113](#)
 - `strtol` [2-135](#)
 - `strtoul` [2-137](#)
- stop (see `atexit`, `exit` functions)
- strcat (concatenate strings) function
[2-114](#)
- strchr (find first occurrence of
character in string) function
[2-115](#)
- strcmp (compare strings) function
[2-116](#), [2-119](#)
- strcoll (compare strings) function
[2-117](#)
- strcpy (copy from one string to
another) function [2-118](#)
- strcspn (length of character segment
in one string) function [2-119](#)

- strerror (get string containing error message) function [2-120](#)
- string conversion (see atof, atoi, atol, strtok, strtol, strxfrm functions)
- string functions
 - compare characters in [2-123](#)
 - copy characters [2-124](#)
 - memchar [2-92](#)
 - memmove [2-95](#)
 - strchr [2-115](#)
 - strcoll [2-117](#)
 - strcspn [2-119](#)
 - strerror [2-120](#)
 - string length [2-121](#)
 - strpbrk [2-125](#)
 - strrchr [2-126](#), [2-127](#)
 - strspn [2-127](#)
 - strstr [2-128](#)
 - strtod [2-129](#)
 - strtodf [2-131](#)
 - strtok [2-133](#)
 - strxfrm [2-139](#)
- strlen (string length) function [2-121](#)
- strncat (concatenate characters from one string to another) function [2-122](#)
- strncmp (compare characters in strings) function [2-123](#)
- strncpy (copy characters from one string to another) function [2-124](#)
- strpbrk (find character match in two strings) function [2-125](#)
- strrchr (find last occurrence of character in string) function [2-126](#)
- strspn (length of segment of characters in both strings) function [2-127](#)
- strstr (find string within string) function [2-128](#)
- strtod (convert portion of string to double) function [2-129](#)
- strtodf (convert portion of string to float) function [2-131](#)
- strtodf (convert to float representation) function [2-131](#)
- strtok (convert string to tokens) function [2-133](#)
- strtol (convert string to long integer) function [2-135](#)
- strxfrm (transform string using LC_COLLATE) function [2-139](#)
- switches
 - compiler common
 - path (tool location) [1-32](#)
 - write-opts [1-42](#)
 - syntax-only (just check syntax)
 - compiler switch [1-37](#)
 - sysdef (system definitions)
 - compiler switch [1-37](#)
 - sysreg_read (read from non-memory-mapped register) function [2-141](#)

INDEX

sysreg_write (write to
non-memory-mapped register)
function [2-143](#)
system header file [1-113](#)
system header files [1-92](#)

T

-T (linker description file) compiler
switch [1-38](#)
tan (tangent) function [2-145](#)
tan_fr16 function [2-145](#)
tanh (hyperbolic tangent) function
[2-146](#)
template
for asm() in C programs [1-51](#)
template for asm() in C programs
[1-51](#)
terminate (see atexit, exit functions)
time (store and return current
calendar time) function [2-147](#)
-time (time the compiler) compiler
switch [1-38](#)
timer_off (disable ADSP-218x
timer) function [2-147](#)
timer_on (enable ADSP-218x
timer) function [2-148](#)
timer_set (initialize ADSP-218x
timer) function [2-149](#)
tokens, string convert (see strtok
function)
tolower (convert from uppercase to
lowercase) function [2-150](#)
toupper (convert characters to upper
case) function [2-151](#)

-traditional (traditional C
compilation) compiler switch
[1-39](#)
true (see Boolean type support
keywords (bool, true, false))

U

-U (undefine macro) compiler
switch [1-21](#), [1-39](#)
-unsigned-char (make char
unsigned) compiler switch [1-39](#)
upper case (see isupper, toupper
functions)
user header files [1-92](#)

V

-v (version & verbose) compiler
switch [1-39](#)
va_arg (get next argument in
variable list) function [2-152](#)
va_arg macro [2-152](#)
va_end (reset variable list pointer)
function [2-154](#)
va_end macro [2-154](#)
va_start (set variable list pointer)
function [2-155](#)
va_start macro [2-155](#)
-val-global (add global names)
compiler switch [1-40](#)
variable-length array support [1-68](#)
-verbose (display command line)
compiler switch [1-40](#)
-version (display version) switch
[1-40](#)

VIDL source text [1-94](#)

VisualDSP++

compiler C language extensions
[1-47](#)

mixed C/assembly reference [1-96](#)

volatile keyword [1-39](#)

volatile and asm() C program
constructs [1-58](#)

W

-w (disable all warnings) switch [1-41](#)

-W (override error message)
compiler switch [1-41](#)

warnings, new in this release [A-7](#)

-warn-protos (prototypes warning)
compiler switch [1-42](#)

-Wdriver-limit (maximum process
errors) compiler switch [1-40](#)

weak_entry pragma [1-87](#)

-Werror-limit (maximum compiler
errors) compiler switch [1-40](#)

white space character test (see
isspace function)

-Wremarks (enable diagnostic
warnings) compiler switch [1-41](#)

-write-files (enable driver I/O
redirection) compiler switch
[1-42](#)

-write-opts compiler switch [1-42](#)

writing macros [1-92](#)

-Wterse (enable terse warnings)
compiler switch [1-41](#)

X

-xref (cross-reference list) compiler
switch [1-42](#)

