

# **VISUAL*****DSP++***<sup>™</sup> **3.0** **C/C++ Compiler and Library Manual** **for SHARC<sup>®</sup> DSPs**

Revision 4.0, January 2003

Part Number  
82-001963-01

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



## **Copyright Information**

©1996–2003 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, SHARC, the SHARC logo, TigerSHARC, and the TigerSHARC logo are registered trademarks of Analog Devices, Inc.

VisualDSP++, the VisualDSP++ logo, CROSSCORE, the CROSSCORE logo, Blackfin, the Blackfin logo, and EZ-KIT Lite are trademarks of Analog Devices, Inc.

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

# CONTENTS

## PREFACE

|                                     |         |
|-------------------------------------|---------|
| Purpose .....                       | xxxi    |
| Intended Audience .....             | xxxi    |
| Manual Contents Description .....   | xxxii   |
| What’s New in this Manual .....     | xxxii   |
| Technical or Customer Support ..... | xxxiii  |
| Supported Processors .....          | xxxiv   |
| Product Information .....           | xxxiv   |
| MyAnalog.com .....                  | xxxiv   |
| DSP Product Information .....       | xxxv    |
| Related Documents .....             | xxxv    |
| Online Documentation .....          | xxxvi   |
| From VisualDSP++ .....              | xxxvii  |
| From Windows .....                  | xxxvii  |
| From the Web .....                  | xxxvii  |
| Printed Manuals .....               | xxxviii |
| VisualDSP++ Documentation Set ..... | xxxviii |
| Hardware Manuals .....              | xxxviii |
| Data Sheets .....                   | xxxviii |

# CONTENTS

|                                   |       |
|-----------------------------------|-------|
| Contacting DSP Publications ..... | xxxix |
| Notation Conventions .....        | xl    |

## COMPILER

|                                                  |      |
|--------------------------------------------------|------|
| C/C++ Compiler Overview .....                    | 1-2  |
| Compiler Command-Line Interface .....            | 1-4  |
| Running the Compiler .....                       | 1-5  |
| Specifying Compiler Options in VisualDSP++ ..... | 1-8  |
| Compiler Command-Line Switches .....             | 1-10 |
| C/C++ Compiler Switch Summaries .....            | 1-10 |
| C/C++ Mode Selection Switch Descriptions .....   | 1-20 |
| -analog .....                                    | 1-20 |
| -c++ .....                                       | 1-20 |
| -traditional .....                               | 1-20 |
| C/C++ Compiler Common Switch Descriptions .....  | 1-21 |
| sourcefile .....                                 | 1-21 |
| -@ filename .....                                | 1-21 |
| -21020 .....                                     | 1-21 |
| -21060 .....                                     | 1-21 |
| -21061 .....                                     | 1-22 |
| -21062 .....                                     | 1-22 |
| -21065L .....                                    | 1-22 |
| -21160 .....                                     | 1-22 |
| -21160rev0 .....                                 | 1-23 |
| -21161 .....                                     | 1-23 |

|                                                                  |      |
|------------------------------------------------------------------|------|
| -A name[tokens] .....                                            | 1-23 |
| -aligned-stack .....                                             | 1-24 |
| -alttok .....                                                    | 1-24 |
| -auto-inline factor .....                                        | 1-24 |
| -build-lib .....                                                 | 1-25 |
| -C .....                                                         | 1-25 |
| -c .....                                                         | 1-25 |
| -circbuf .....                                                   | 1-25 |
| -Dmacro[=definition] .....                                       | 1-26 |
| -debug-types .....                                               | 1-26 |
| -default-linkage[-asm   -C   -C++] .....                         | 1-26 |
| -dollar .....                                                    | 1-27 |
| -double-size[-32 -64] .....                                      | 1-27 |
| -dry .....                                                       | 1-28 |
| -dryrun .....                                                    | 1-28 |
| -E .....                                                         | 1-28 |
| -EE .....                                                        | 1-28 |
| -expert-linker .....                                             | 1-29 |
| -extra-keywords .....                                            | 1-29 |
| -flags-{asm compiler lib link mem} switch [,switch2 [...]] ..... | 1-29 |
| -float_to_int .....                                              | 1-29 |
| -fp-associative .....                                            | 1-30 |
| -full-version .....                                              | 1-30 |
| -g .....                                                         | 1-30 |

## CONTENTS

|                                         |      |
|-----------------------------------------|------|
| -H .....                                | 1-30 |
| -HH .....                               | 1-31 |
| -h[elp] .....                           | 1-31 |
| -I directory [{, ;} directory...] ..... | 1-31 |
| -I- .....                               | 1-32 |
| -include filename .....                 | 1-32 |
| -ipa .....                              | 1-32 |
| -L directory [{, ;} directory...] ..... | 1-33 |
| -l library .....                        | 1-33 |
| -line-debug .....                       | 1-33 |
| -M .....                                | 1-34 |
| -MM .....                               | 1-34 |
| -Mt filename .....                      | 1-34 |
| -MQ .....                               | 1-34 |
| -map filename .....                     | 1-34 |
| -mem .....                              | 1-35 |
| -no-aligned-stack .....                 | 1-35 |
| -no-alttok .....                        | 1-35 |
| -no-builtin .....                       | 1-35 |
| -no-circbuf .....                       | 1-35 |
| -no-defs .....                          | 1-35 |
| -no-dir-warnings .....                  | 1-36 |
| -no-dollar .....                        | 1-36 |
| -no-extra-keywords .....                | 1-36 |

|                                                                   |      |
|-------------------------------------------------------------------|------|
| -no-fp-associative .....                                          | 1-36 |
| -no-inline .....                                                  | 1-36 |
| -no-mem .....                                                     | 1-36 |
| -no-restrict .....                                                | 1-37 |
| -no-simd .....                                                    | 1-37 |
| -no-std-def .....                                                 | 1-37 |
| -no-std-inc .....                                                 | 1-37 |
| -no-std-lib .....                                                 | 1-38 |
| -nothreads .....                                                  | 1-38 |
| -O[0 1] .....                                                     | 1-38 |
| -Os .....                                                         | 1-38 |
| -o filename .....                                                 | 1-38 |
| -P .....                                                          | 1-39 |
| -PP .....                                                         | 1-39 |
| -path-{ asm   compiler   def   lib   link   mem } directory ..... | 1-39 |
| -path-install directory .....                                     | 1-39 |
| -path-output directory .....                                      | 1-40 |
| -path-temp directory .....                                        | 1-40 |
| -pch .....                                                        | 1-40 |
| -pchdir directory .....                                           | 1-40 |
| -pedantic .....                                                   | 1-40 |
| -pedantic-errors .....                                            | 1-41 |
| -pplist filename .....                                            | 1-41 |
| -proc identifier .....                                            | 1-41 |

# CONTENTS

|                                                    |      |
|----------------------------------------------------|------|
| -R directory[{: ,}directory ...]                   | 1-42 |
| -R-                                                | 1-42 |
| -reserve register[, register ...]                  | 1-42 |
| -restrict                                          | 1-43 |
| -S                                                 | 1-43 |
| -s                                                 | 1-43 |
| -save-temps                                        | 1-43 |
| -show                                              | 1-43 |
| -swf_screening                                     | 1-44 |
| -switch-pm                                         | 1-44 |
| -syntax-only                                       | 1-44 |
| -sysdefs                                           | 1-44 |
| -T filename                                        | 1-45 |
| -threads                                           | 1-45 |
| -time                                              | 1-45 |
| -Umacro                                            | 1-45 |
| -v                                                 | 1-45 |
| -val-global <name-list>                            | 1-46 |
| -verbose                                           | 1-46 |
| -version                                           | 1-46 |
| -warn-protos                                       | 1-46 |
| -W[error remark suppress warn] number[,number ...] | 1-46 |
| -Wdriver-limit number                              | 1-47 |
| -Werror-limit number                               | 1-47 |



|                                             |      |
|---------------------------------------------|------|
| -Wremarks .....                             | 1-47 |
| -Wterse .....                               | 1-47 |
| -w .....                                    | 1-48 |
| -write-files .....                          | 1-48 |
| -write-opts .....                           | 1-48 |
| -xref <filename> .....                      | 1-48 |
| C++ Mode Compiler Switch Descriptions ..... | 1-49 |
| -explicit .....                             | 1-49 |
| -namespace .....                            | 1-49 |
| -newforinit .....                           | 1-49 |
| -newvec .....                               | 1-50 |
| -no-demangle .....                          | 1-50 |
| -no-explicit .....                          | 1-50 |
| -no-namespace .....                         | 1-50 |
| -no-newvec .....                            | 1-50 |
| -no-std .....                               | 1-50 |
| -notstrict .....                            | 1-51 |
| -no-wchar .....                             | 1-51 |
| -std .....                                  | 1-51 |
| -strict .....                               | 1-51 |
| -strictwarn .....                           | 1-51 |
| -tpautooff .....                            | 1-52 |
| -trdforinit .....                           | 1-52 |
| -typename .....                             | 1-52 |

# CONTENTS

|                                                          |      |
|----------------------------------------------------------|------|
| -wchar .....                                             | 1-52 |
| Data Type and Data Type Sizes .....                      | 1-53 |
| Integer Data Types .....                                 | 1-54 |
| Floating-Point Data Types .....                          | 1-54 |
| Optimization Control .....                               | 1-55 |
| Inlining Control .....                                   | 1-56 |
| Interprocedural Analysis .....                           | 1-57 |
| Interaction with Libraries .....                         | 1-58 |
| C/C++ Compiler Language Extensions .....                 | 1-60 |
| Inline Function Support Keyword (inline) .....           | 1-63 |
| Inline Assembly Language Support Keyword (asm) .....     | 1-64 |
| Assembly Construct Template .....                        | 1-65 |
| Assembly Construct Operand Description .....             | 1-68 |
| Assembly Constructs With Multiple Instructions .....     | 1-70 |
| Assembly Construct Reordering and Optimization .....     | 1-71 |
| Restrictions on the Use of the asm Construct .....       | 1-72 |
| Assembly Constructs with Input and Output Operands ..... | 1-72 |
| Assembly Constructs and Macros .....                     | 1-74 |
| Dual Memory Support Keywords (pm dm) .....               | 1-74 |
| Memory Keywords and Assignments/Type Conversions .....   | 1-76 |
| Memory Keywords and Function Declarations/Pointers ..... | 1-77 |
| Memory Keywords and Function Arguments .....             | 1-78 |
| Memory Keywords and Macros .....                         | 1-78 |
| Placement Support Keyword (section) .....                | 1-79 |

|                                                         |      |
|---------------------------------------------------------|------|
| Boolean Type Support Keywords (bool, true, false) ..... | 1-80 |
| Pointer Class Support Keyword (restrict) .....          | 1-80 |
| Variable-Length Array Support .....                     | 1-81 |
| Non-Constant Initializer Support .....                  | 1-83 |
| Indexed Initializer Support .....                       | 1-83 |
| Aggregate Constructor Expression Support .....          | 1-85 |
| Preprocessor Generated Warnings .....                   | 1-85 |
| C++ Style Comments .....                                | 1-86 |
| Built-In Functions .....                                | 1-86 |
| Access to System Registers .....                        | 1-87 |
| Circular Buffer Built-In Functions .....                | 1-90 |
| Circular Buffer Increment of an Index .....             | 1-90 |
| Circular Buffer Increment of a Pointer .....            | 1-90 |
| Supported Pragmas .....                                 | 1-91 |
| Data Alignment Pragmas .....                            | 1-92 |
| ALIGN NUM Pragma .....                                  | 1-92 |
| PACK (ALIGNOPT) Pragma .....                            | 1-93 |
| PAD (ALIGNOPT) Pragma .....                             | 1-93 |
| Interrupt Handler Pragma .....                          | 1-94 |
| SIMD_for Pragma .....                                   | 1-94 |
| Optimization Level Pragmas .....                        | 1-94 |
| External Linkage Pragmas .....                          | 1-95 |
| LINKAGE_NAME Identifier .....                           | 1-95 |
| RETAIN_NAME Pragma .....                                | 1-96 |

# CONTENTS

|                                                    |       |
|----------------------------------------------------|-------|
| C++ Fractional Type Support .....                  | 1-96  |
| Format of Fractional Literals .....                | 1-97  |
| Conversions Involving Fractional Values .....      | 1-97  |
| Fractional Arithmetic Operations .....             | 1-97  |
| Mixed Mode Operations .....                        | 1-98  |
| Saturated Arithmetic .....                         | 1-99  |
| SIMD Support .....                                 | 1-99  |
| Using SIMD Mode with Multichannel Data .....       | 1-101 |
| Using SIMD Mode with Single-Channel Data .....     | 1-102 |
| Restrictions to Using SIMD .....                   | 1-103 |
| SIMD_for Pragma Syntax .....                       | 1-105 |
| Compiler Constraints on Using SIMD C/C++ .....     | 1-106 |
| Impact of Anomaly #40 on SIMD .....                | 1-107 |
| Performance When Using SIMD C/C++ .....            | 1-108 |
| Examples Using SIMD C .....                        | 1-110 |
| Using SIMD C (Problem Cases—Data Increments) ..... | 1-110 |
| Using SIMD C (Problem Cases—Data Alignment) .....  | 1-112 |
| Preprocessor Features .....                        | 1-114 |
| Predefined Macros .....                            | 1-115 |
| __2106x__ .....                                    | 1-115 |
| __2116x__ .....                                    | 1-115 |
| __ADSP21000__ .....                                | 1-115 |
| ADSP21000 .....                                    | 1-115 |
| __ADSP21020__ .....                                | 1-115 |

|                              |       |
|------------------------------|-------|
| ADSP21020 .....              | 1-116 |
| __ADSP21060__ .....          | 1-116 |
| ADSP21060 .....              | 1-116 |
| __ADSP21061__ .....          | 1-116 |
| ADSP21061 .....              | 1-116 |
| __ADSP21062__ .....          | 1-116 |
| ADSP21062 .....              | 1-117 |
| __ADSP21065L__ .....         | 1-117 |
| ADSP21065L .....             | 1-117 |
| __ADSP21160__ .....          | 1-117 |
| ADSP21160 .....              | 1-117 |
| __ADSP21161__ .....          | 1-117 |
| ADSP21161 .....              | 1-118 |
| __ANALOG_EXTENSIONS__ .....  | 1-118 |
| __cplusplus .....            | 1-118 |
| __DATE__ .....               | 1-118 |
| __DOUBLES_ARE_FLOATS__ ..... | 1-118 |
| __ECC__ .....                | 1-118 |
| __EDG__ .....                | 1-118 |
| __EDG_VERSION__ .....        | 1-119 |
| __FILE__ .....               | 1-119 |
| __LINE__ .....               | 1-119 |
| _NO_LONGLONG .....           | 1-119 |
| __NO_BUILTIN .....           | 1-119 |

# CONTENTS

|                                                         |       |
|---------------------------------------------------------|-------|
| __SIGNED_CHARS__ .....                                  | 1-119 |
| __STDC__ .....                                          | 1-119 |
| __STDC_VERSION__ .....                                  | 1-120 |
| __TIME__ .....                                          | 1-120 |
| __VERSION_ .....                                        | 1-120 |
| Header Files .....                                      | 1-120 |
| Writing Macros .....                                    | 1-121 |
| C/C++ Run-Time Model and Environment .....              | 1-123 |
| C/C++ Run-Time Environment .....                        | 1-123 |
| Memory Usage .....                                      | 1-125 |
| Using Multiple Heaps .....                              | 1-130 |
| Declaring a Heap .....                                  | 1-131 |
| Heap Identifiers .....                                  | 1-134 |
| Using Alternate Heaps with the Standard Interface ..... | 1-134 |
| Using the Alternate Heap Interface .....                | 1-135 |
| Example C Programs .....                                | 1-136 |
| Compiler Registers .....                                | 1-137 |
| Miscellaneous Information about Registers .....         | 1-138 |
| User Registers .....                                    | 1-138 |
| Call Preserved Registers .....                          | 1-139 |
| Scratch Registers .....                                 | 1-140 |
| Stack Registers .....                                   | 1-141 |
| Alternate Registers .....                               | 1-141 |
| Managing the Stack .....                                | 1-142 |

|                                                        |       |
|--------------------------------------------------------|-------|
| Transferring Function Arguments and Return Value ..... | 1-147 |
| Using Data Storage Formats .....                       | 1-150 |
| Using the Run-Time Header .....                        | 1-153 |
| C/C++ and Assembly Interface .....                     | 1-154 |
| Calling Assembly Subroutines from C/C++ Programs ..... | 1-154 |
| Calling C/C++ Functions from Assembly Programs .....   | 1-156 |
| Using Mixed C/C++ and Assembly Support Macros .....    | 1-159 |
| Interface Support Macros, Defined .....                | 1-159 |
| entry .....                                            | 1-159 |
| exit .....                                             | 1-159 |
| leaf_entry .....                                       | 1-160 |
| leaf_exit .....                                        | 1-160 |
| ccall(x) .....                                         | 1-160 |
| reads(x) .....                                         | 1-160 |
| puts=x .....                                           | 1-160 |
| gets(x) .....                                          | 1-160 |
| alter(x) .....                                         | 1-161 |
| save_reg .....                                         | 1-161 |
| restore_reg .....                                      | 1-161 |
| Using Mixed C/C++ and Assembly Naming Conventions .    | 1-163 |
| Implementing C++ Member Functions in Assembly .....    | 1-165 |
| Writing C/C++ Callable SIMD Subroutines .....          | 1-167 |
| C++ Programming Examples .....                         | 1-168 |
| Using Fract Support .....                              | 1-168 |

# CONTENTS

|                                                        |       |
|--------------------------------------------------------|-------|
| Using Complex Support .....                            | 1-169 |
| Mixed C/C++/Assembly Programming Examples .....        | 1-171 |
| Using Inline Assembly (Add) .....                      | 1-172 |
| Using Macros to Manage the Stack .....                 | 1-173 |
| Using Scratch Registers (Dot Product) .....            | 1-174 |
| Using Void Functions (Delay) .....                     | 1-176 |
| Using the Stack for Arguments and Return (Add 5) ..... | 1-177 |
| Using Registers for Arguments and Return (Add 2) ..... | 1-178 |
| Using Non-Leaf Routines That Make Calls (RMS) .....    | 1-179 |
| Using Call Preserved Registers (Pass Array) .....      | 1-181 |

## C/C++ RUN-TIME LIBRARY

|                                                   |     |
|---------------------------------------------------|-----|
| C and C++ Run-Time Libraries Guide .....          | 2-3 |
| Calling Library Functions .....                   | 2-3 |
| Linking Library Functions .....                   | 2-4 |
| Working with Library Header Files .....           | 2-6 |
| Standard C Library Header Files .....             | 2-7 |
| Standard C Library Header File Descriptions ..... | 2-8 |
| assert.h .....                                    | 2-8 |
| ctype.h .....                                     | 2-8 |
| errno.h .....                                     | 2-8 |
| float.h .....                                     | 2-9 |
| limits.h .....                                    | 2-9 |
| locale.h .....                                    | 2-9 |
| math.h .....                                      | 2-9 |



|                                                     |      |
|-----------------------------------------------------|------|
| setjmp.h .....                                      | 2-10 |
| signal.h .....                                      | 2-10 |
| stdarg.h .....                                      | 2-13 |
| stddef.h .....                                      | 2-13 |
| stdio.h .....                                       | 2-13 |
| stdlib.h .....                                      | 2-18 |
| string.h .....                                      | 2-19 |
| Using Compiler's Built-In C Library Functions ..... | 2-19 |
| Abridged C++ Library Support .....                  | 2-21 |
| Embedded C++ Library Header Files .....             | 2-22 |
| complex .....                                       | 2-22 |
| exception .....                                     | 2-22 |
| fract .....                                         | 2-22 |
| fstream .....                                       | 2-22 |
| iomanip .....                                       | 2-23 |
| ios .....                                           | 2-23 |
| iosfwd .....                                        | 2-23 |
| iostream .....                                      | 2-23 |
| istream .....                                       | 2-23 |
| new .....                                           | 2-23 |
| ostream .....                                       | 2-23 |
| sstream .....                                       | 2-24 |
| stdexcept .....                                     | 2-24 |
| streambuf .....                                     | 2-24 |

# CONTENTS

|                                                       |      |
|-------------------------------------------------------|------|
| string .....                                          | 2-24 |
| stringstream .....                                    | 2-24 |
| C++ Header Files for C Library Facilities .....       | 2-24 |
| Embedded Standard Template Library Header Files ..... | 2-26 |
| algorithm .....                                       | 2-26 |
| deque .....                                           | 2-26 |
| functional .....                                      | 2-26 |
| hash_map .....                                        | 2-26 |
| hash_set .....                                        | 2-26 |
| iterator .....                                        | 2-26 |
| list .....                                            | 2-26 |
| map .....                                             | 2-27 |
| memory .....                                          | 2-27 |
| numeric .....                                         | 2-27 |
| queue .....                                           | 2-27 |
| set .....                                             | 2-27 |
| stack .....                                           | 2-27 |
| utility .....                                         | 2-27 |
| vector .....                                          | 2-27 |
| fstream.h .....                                       | 2-28 |
| iomanip.h .....                                       | 2-28 |
| iostream.h .....                                      | 2-28 |
| new.h .....                                           | 2-28 |
| C Run-Time Library Reference .....                    | 2-29 |

|                            |      |
|----------------------------|------|
| Notation Conventions ..... | 2-29 |
| Reference Format .....     | 2-29 |
| abort .....                | 2-30 |
| abs .....                  | 2-31 |
| acos, acosf .....          | 2-32 |
| asin, asinf .....          | 2-33 |
| atan, atanf .....          | 2-34 |
| atan2, atan2f .....        | 2-35 |
| atexit .....               | 2-36 |
| atof .....                 | 2-37 |
| atoi .....                 | 2-38 |
| atol .....                 | 2-39 |
| avg .....                  | 2-40 |
| bsearch .....              | 2-41 |
| calloc .....               | 2-43 |
| ceil, ceilf .....          | 2-44 |
| circindex .....            | 2-45 |
| circptr .....              | 2-47 |
| clear_interrupt .....      | 2-49 |
| clip .....                 | 2-54 |
| cos, cosf .....            | 2-55 |
| cosh, coshf .....          | 2-56 |
| div .....                  | 2-57 |
| exit .....                 | 2-58 |

# CONTENTS

|                        |      |
|------------------------|------|
| exp, expf .....        | 2-59 |
| fabs, fabsf .....      | 2-60 |
| floor, floorf .....    | 2-61 |
| fmod, fmodf .....      | 2-62 |
| free .....             | 2-63 |
| frexp, frexpf .....    | 2-64 |
| getenv .....           | 2-65 |
| heap_calloc .....      | 2-66 |
| heap_free .....        | 2-68 |
| heap_lookup_name ..... | 2-70 |
| heap_malloc .....      | 2-72 |
| heap_realloc .....     | 2-74 |
| interrupt .....        | 2-77 |
| isalnum .....          | 2-79 |
| isalpha .....          | 2-80 |
| iscntrl .....          | 2-81 |
| isdigit .....          | 2-82 |
| isgraph .....          | 2-83 |
| isinf .....            | 2-84 |
| islower .....          | 2-86 |
| isnan .....            | 2-87 |
| isprint .....          | 2-89 |
| ispunct .....          | 2-90 |
| isspace .....          | 2-91 |

|                     |       |
|---------------------|-------|
| isupper .....       | 2-92  |
| isxdigit .....      | 2-93  |
| labs .....          | 2-94  |
| lavg .....          | 2-95  |
| lclip .....         | 2-96  |
| ldexp, ldexpf ..... | 2-97  |
| ldiv .....          | 2-98  |
| lmax .....          | 2-99  |
| lmin .....          | 2-100 |
| localeconv .....    | 2-101 |
| log, logf .....     | 2-104 |
| log10, log10f ..... | 2-105 |
| longjmp .....       | 2-106 |
| malloc .....        | 2-108 |
| max .....           | 2-109 |
| memchr .....        | 2-110 |
| memcmp .....        | 2-111 |
| memcpy .....        | 2-112 |
| memmove .....       | 2-113 |
| memset .....        | 2-114 |
| min .....           | 2-115 |
| modf, modff .....   | 2-116 |
| pow, powf .....     | 2-117 |
| qsort .....         | 2-118 |

# CONTENTS

|                      |       |
|----------------------|-------|
| raise .....          | 2-120 |
| rand .....           | 2-121 |
| realloc .....        | 2-122 |
| setjmp .....         | 2-124 |
| setlocale .....      | 2-125 |
| set_alloc_type ..... | 2-126 |
| signal .....         | 2-128 |
| sin, sinf .....      | 2-130 |
| sinh, sinhlf .....   | 2-131 |
| sqrt, sqrtf .....    | 2-132 |
| srand .....          | 2-133 |
| strcat .....         | 2-134 |
| strchr .....         | 2-135 |
| strcmp .....         | 2-136 |
| strcoll .....        | 2-137 |
| strcpy .....         | 2-138 |
| strcspn .....        | 2-139 |
| strerror .....       | 2-140 |
| strlen .....         | 2-141 |
| strncat .....        | 2-142 |
| strncmp .....        | 2-143 |
| strncpy .....        | 2-144 |
| strpbrk .....        | 2-145 |
| strrchr .....        | 2-146 |

|                   |       |
|-------------------|-------|
| strspn .....      | 2-147 |
| strstr .....      | 2-148 |
| strtod .....      | 2-149 |
| strtok .....      | 2-151 |
| strtol .....      | 2-153 |
| strtoul .....     | 2-155 |
| strxfrm .....     | 2-157 |
| system .....      | 2-159 |
| tan, tanf .....   | 2-160 |
| tanh, tanhf ..... | 2-161 |
| tolower .....     | 2-162 |
| toupper .....     | 2-163 |
| va_arg .....      | 2-164 |
| va_end .....      | 2-166 |
| va_start .....    | 2-167 |

## DSP LIBRARY FOR ADSP-2106X AND ADSP-21020 PROCESSORS

|                                          |     |
|------------------------------------------|-----|
| DSP Run-Time Library Guide .....         | 3-2 |
| Calling DSP Library Functions .....      | 3-2 |
| Linking DSP Library Functions .....      | 3-3 |
| Working With Library Source Code .....   | 3-3 |
| DSP Header Files .....                   | 3-4 |
| 21020.h — ADSP-21020 DSP Functions ..... | 3-4 |
| 21060.h — ADSP-2106x DSP Functions ..... | 3-4 |

# CONTENTS

|                                                      |      |
|------------------------------------------------------|------|
| 21065l.h — ADSP-21065L DSP Functions .....           | 3-4  |
| asm_sprt.h — Mixed C/Assembly Support .....          | 3-5  |
| comm.h — A-law and $\mu$ -law Companders .....       | 3-5  |
| complex.h — Basic Complex Arithmetic Functions ..... | 3-5  |
| cvector.h — Complex Vector Functions .....           | 3-5  |
| def21020.h — ADSP-21020 DSP Bit Definitions .....    | 3-6  |
| def21060.h — ADSP-21060 DSP Bit Definitions .....    | 3-6  |
| def21061.h — ADSP-21061 DSP Bit Definitions .....    | 3-6  |
| def21062.h — ADSP-21062 DSP Bit Definitions .....    | 3-6  |
| def21065l.h — ADSP-21065L DSP Bit Definitions .....  | 3-6  |
| dma.h — DMA Support Functions .....                  | 3-7  |
| filters.h — DSP Filters .....                        | 3-7  |
| macros.h — Circular Buffers .....                    | 3-7  |
| math.h — Math Functions .....                        | 3-7  |
| matrix.h — Matrix Functions .....                    | 3-8  |
| saturate.h — Saturation Mode Arithmetic .....        | 3-8  |
| sport.h — Serial Port Support Functions .....        | 3-8  |
| stats.h — Statistical Functions .....                | 3-9  |
| sysreg.h — Register Access .....                     | 3-9  |
| trans.h — Fast Fourier Transforms .....              | 3-9  |
| vector.h — Vector Functions .....                    | 3-10 |
| window.h — Window Generators .....                   | 3-10 |
| Built-In DSP Functions .....                         | 3-11 |
| DSP Run-Time Library Reference .....                 | 3-13 |



|                           |      |
|---------------------------|------|
| a_compress .....          | 3-14 |
| a_expand .....            | 3-15 |
| autocoh .....             | 3-16 |
| autocorr .....            | 3-17 |
| biquad .....              | 3-18 |
| cabsf .....               | 3-21 |
| cexpf .....               | 3-22 |
| cfftN .....               | 3-23 |
| copysign, copysignf ..... | 3-26 |
| cot, cotf .....           | 3-27 |
| crosscoh .....            | 3-28 |
| crosscorr .....           | 3-29 |
| cvecdot .....             | 3-30 |
| cvecsadd .....            | 3-31 |
| cvecsmult .....           | 3-32 |
| cvecssub .....            | 3-33 |
| cvecvadd .....            | 3-34 |
| cvecvmlt .....            | 3-35 |
| cvecvsub .....            | 3-36 |
| favg, favgf .....         | 3-37 |
| fclip, fclipf .....       | 3-38 |
| fir .....                 | 3-39 |
| fmax, fmaxf .....         | 3-41 |
| fmin, fminf .....         | 3-42 |

# CONTENTS

|                       |      |
|-----------------------|------|
| gen_bartlett .....    | 3-43 |
| gen_blackman .....    | 3-45 |
| gen_gaussian .....    | 3-46 |
| gen_hamming .....     | 3-47 |
| gen_hanning .....     | 3-48 |
| gen_harris .....      | 3-49 |
| gen_kaiser .....      | 3-50 |
| gen_rectangular ..... | 3-52 |
| gen_triangle .....    | 3-53 |
| histogram .....       | 3-55 |
| idle .....            | 3-57 |
| ifftN .....           | 3-58 |
| iir .....             | 3-61 |
| matadd .....          | 3-64 |
| matinv .....          | 3-65 |
| matmul .....          | 3-66 |
| matscalmult .....     | 3-67 |
| matsub .....          | 3-68 |
| mean .....            | 3-69 |
| mu_compress .....     | 3-70 |
| mu_expand .....       | 3-71 |
| poll_flag_in .....    | 3-72 |
| rfftN .....           | 3-74 |
| rms .....             | 3-77 |

|                              |      |
|------------------------------|------|
| rsqrt, rsqrtf .....          | 3-78 |
| set_flag .....               | 3-79 |
| set_semaphore .....          | 3-81 |
| timer_off .....              | 3-82 |
| timer0_off, timer1_off ..... | 3-83 |
| timer_on .....               | 3-84 |
| timer0_on, timer1_on .....   | 3-85 |
| timer_set .....              | 3-86 |
| timer0_set, timer1_set ..... | 3-88 |
| transpm .....                | 3-90 |
| var .....                    | 3-91 |
| vecdot .....                 | 3-92 |
| vecsadd .....                | 3-93 |
| vecsmult .....               | 3-94 |
| vecssub .....                | 3-95 |
| vecvadd .....                | 3-96 |
| vecvmlt .....                | 3-97 |
| vecvsub .....                | 3-98 |
| zero_cross .....             | 3-99 |

## DSP LIBRARY FOR ADSP-2116X PROCESSORS

|                                        |     |
|----------------------------------------|-----|
| DSP Run-Time Library Guide .....       | 4-2 |
| Calling DSP Library Functions .....    | 4-2 |
| Linking DSP Library Functions .....    | 4-3 |
| Working With Library Source Code ..... | 4-3 |

# CONTENTS

|                                                      |      |
|------------------------------------------------------|------|
| DSP Header Files .....                               | 4-4  |
| 21160.h — ADSP-2116x DSP Functions .....             | 4-4  |
| asm_sprt.h — Mixed C/Assembly Support .....          | 4-4  |
| comm.h — A-law and $\mu$ -law Companders .....       | 4-4  |
| complex.h — Basic Complex Arithmetic Functions ..... | 4-5  |
| cvector.h — Complex Vector Functions .....           | 4-5  |
| def21160.h — ADSP-21160 DSP Bit Definitions .....    | 4-5  |
| def21161.h — ADSP-21161 DSP Bit Definitions .....    | 4-6  |
| dma.h — DMA Support Functions .....                  | 4-6  |
| filter.h — DSP Filters and Transformations .....     | 4-6  |
| filters.h — DSP Filters .....                        | 4-8  |
| macro.h — Circular Buffers .....                     | 4-8  |
| math.h — Math Functions .....                        | 4-8  |
| matrix.h — Matrix Functions .....                    | 4-9  |
| saturate.h — Saturation Mode Arithmetic .....        | 4-9  |
| sport.h — Serial Port Support Functions .....        | 4-10 |
| stats.h — Statistical Functions .....                | 4-10 |
| sysreg.h — Register Access .....                     | 4-10 |
| trans.h — Fast Fourier Transforms .....              | 4-10 |
| vector.h — Vector Functions .....                    | 4-10 |
| window.h — Window Generators .....                   | 4-11 |
| Built-In DSP Functions .....                         | 4-12 |
| Implications of Using SIMD Mode .....                | 4-13 |
| DSP Run-Time Library Reference .....                 | 4-15 |

|                           |      |
|---------------------------|------|
| a_compress .....          | 4-16 |
| a_expand .....            | 4-17 |
| autocoh .....             | 4-18 |
| autocorr .....            | 4-19 |
| biquad .....              | 4-21 |
| cabsf .....               | 4-24 |
| cexpf .....               | 4-25 |
| cfftN .....               | 4-26 |
| cfft .....                | 4-29 |
| copysign, copysignf ..... | 4-32 |
| cot, cotf .....           | 4-33 |
| crosscoh .....            | 4-34 |
| crosscorr .....           | 4-36 |
| cvecdot .....             | 4-38 |
| cvecsadd .....            | 4-39 |
| cvecsmult .....           | 4-40 |
| cvecssub .....            | 4-41 |
| cvecvadd .....            | 4-42 |
| cvecvmult .....           | 4-43 |
| cvecvsub .....            | 4-44 |
| favg, favgf .....         | 4-45 |
| fclip, fclipf .....       | 4-46 |
| fft_mag .....             | 4-47 |
| fir .....                 | 4-48 |

# CONTENTS

|                       |      |
|-----------------------|------|
| fmax, fmaxf .....     | 4-50 |
| fmin, fminf .....     | 4-51 |
| gen_bartlett .....    | 4-52 |
| gen_blackman .....    | 4-54 |
| gen_gaussian .....    | 4-55 |
| gen_hamming .....     | 4-56 |
| gen_hanning .....     | 4-57 |
| gen_harris .....      | 4-58 |
| gen_kaiser .....      | 4-59 |
| gen_rectangular ..... | 4-61 |
| gen_triangle .....    | 4-62 |
| histogram .....       | 4-64 |
| idle .....            | 4-66 |
| ifftN .....           | 4-67 |
| iir .....             | 4-70 |
| matadd .....          | 4-73 |
| matinv .....          | 4-74 |
| matmul .....          | 4-75 |
| matscaltmult .....    | 4-76 |
| matsub .....          | 4-77 |
| mean .....            | 4-78 |
| mu_compress .....     | 4-79 |
| mu_expand .....       | 4-80 |
| poll_flag_in .....    | 4-81 |

|                     |       |
|---------------------|-------|
| rfftN .....         | 4-83  |
| rms .....           | 4-86  |
| rsqrt, rsqrtf ..... | 4-87  |
| set_flag .....      | 4-88  |
| set_semaphore ..... | 4-90  |
| timer_off .....     | 4-91  |
| timer_on .....      | 4-92  |
| timer_set .....     | 4-93  |
| transpm .....       | 4-95  |
| twidfft .....       | 4-96  |
| var .....           | 4-98  |
| vecdot .....        | 4-99  |
| vecsadd .....       | 4-100 |
| vecsmult .....      | 4-101 |
| vecssub .....       | 4-102 |
| vecvadd .....       | 4-103 |
| vecvmlt .....       | 4-104 |
| vecvsub .....       | 4-105 |
| zero_cross .....    | 4-106 |

# CONTENTS



# PREFACE

Thank you for purchasing Analog Devices development software for digital signal processor (DSP) applications.

## Purpose

The *VisualDSP++ 3.0 C/C++ Compiler and Library Manual for SHARC DSPs* contains information about the C/C++ compiler and run-time library for SHARC® (ADSP-21xxx) 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/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 SHARC DSPs architecture and instruction set and the C/C++ programming languages.

Programmers who are unfamiliar with SHARC DSPs can use this manual, but they should supplement it with other texts (such as the appropriate *Hardware Reference* and *Programming Reference* manuals) that describe your target architecture.

# Manual Contents Description

This manual contains:

- Chapter 1, “[Compiler](#)”  
Provides information on compiler options, language extensions and C/C++/assembly interfacing
- Chapter 2, “[C/C++ Run-Time Library](#)”  
Shows how to use library functions and provides a complete C/C++ library function reference (for functions covered in the current compiler release)
- Chapter 3, “[DSP Library for ADSP-2106x and ADSP-21020 Processors](#)”  
Shows how to use DSP library functions and provides a complete library function reference used with **ADSP-2106x** and **ADSP-21020 DSPs** (for functions covered in the current compiler release)
- Chapter 4, “[DSP Library for ADSP-2116x Processors](#)”  
Shows how to use DSP library functions and provides a complete library function reference used with **ADSP-2116x DSPs** (for functions covered in the current compiler release)

## What’s New in this Manual

This edition of the *VisualDSP++ 3.0 C/C++ Compiler and Library Manual for SHARC DSPs* documents support for all SHARC processors.

In addition to updating the documentation on all existing compiler features, this manual describes new features, including ADSP-21020 DSP compiler functions,, pragmas, circular buffer intrinsics, source annotations, `.IDL` file preprocessing, and other enhancements.

New command-line switches that support new compiler features include `fp_associative`, `-expert-linker`, `-nothreads`, `-pat-def`, `-swf-screeing`, `-proc ID`, and more. New library functions include `circindex`, `circptr`, `cfft`, `heap_lookup_name`, `matinv`, `rfft`, `set_alloc_type`, `transpm`, and `twidfft`.

Refer to the *VisualDSP++ 3.0 Product Bulletin for SHARC DSPs* for a complete list of new compiler features and 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](http://www.analog.com/technology/dsp/developmentTools/index.html)
- Email questions to [dsptools.support@analog.com](mailto: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 “SHARC” refers to a family of Analog Devices, Inc. high-performance 32-bit floating-point digital signal processor for speech, sound, graphics, and imaging applications. VisualDSP++ currently supports the following SHARC processors:

ADSP-21020, ADSP-21060/60L, ADSP-21061/61L, ADSP-21062/62L,  
ADSP-21065L, ADSP-21160M/60N, and ADSP-21161N

## 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](http://www.analog.com). Our website provides information about a broad range of products—analog integrated circuits, amplifiers, converters, and digital signal processors.

## MyAnalog.com

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

### Registration:

Visit [www.myanalog.com](http://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](http://www.analog.com/dsp), which provides access to technical publications, data sheets, 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](mailto:dsp.support@analog.com)
- Fax questions or requests for information to  
**1-781-461-3010** (North America)  
**089/76 903-157** (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://ftp.analog.com>

## Related Documents

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

*VisualDSP++ 3.0 Getting Started Guide for SHARC DSPs*

*VisualDSP++ 3.0 User's Guide for SHARC DSPs*

*VisualDSP++ 3.0 C/C++ Compiler and Library Manual for SHARC DSPs*

*VisualDSP++ 3.0 Linker and Utilities Manual for SHARC DSPs*

*VisualDSP++ 3.0 Product Bulletin for SHARC DSPs*

*VisualDSP++ Kernel (VDK) User's Guide*

*VisualDSP++ Component Software Engineering User's Guide*

*Quick Installation Reference Card*

## Product Information

For hardware information, refer to your DSP's *Hardware Reference*, *Programming Reference*, and data sheet. All documentation is available online. Most documentation is available in printed form.

## Online Documentation

Online documentation comprises Microsoft HTML Help (.CHM), Adobe Portable Documentation Format (.PDF), and HTML (.HTM and .HTML) files. A description of each file type is as follows.

| File                | Description                                                                                                                                                                                                                                                                                                                                                                 |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| .CHM                | VisualDSP++ online Help system files and VisualDSP++ manuals are provided in Microsoft HTML Help format. Installing VisualDSP++ automatically copies these files to the VisualDSP\Help folder. Online Help is ideal for searching the entire tools manual set. Invoke Help from the VisualDSP++ Help menu or via the Windows Start button.                                  |
| .PDF                | Manuals and data sheets in Portable Documentation Format are located in the installation CD's Docs folder. Viewing and printing a .PDF file requires a PDF reader, such as Adobe Acrobat Reader (4.0 or higher). Running setup.exe on the installation CD provides easy access to these documents. You can also copy .PDF files from the installation CD onto another disk. |
| .HTM<br>or<br>.HTML | Dinkum Abridged C++ library and FlexLM network license manager software documentation is located on the installation CD in the Docs\Reference folder. Viewing or printing these files requires a browser, such as Internet Explorer 4.0 (or higher). You can copy these files from the installation CD onto another disk.                                                   |

Access the online documentation from the VisualDSP++ environment, Windows<sup>®</sup> Explorer, or Analog Devices Web site.

### From VisualDSP++

VisualDSP++ provides access to online Help. It does not provide access to .PDF files or the supplemental reference documentation (Dinkum Abridged C++ library and FlexLM network licence). Access Help by:

- Choosing **Contents**, **Search**, or **Index** from the VisualDSP++ **Help** menu
- Invoking context-sensitive Help on a user interface item (toolbar button, menu command, or window)

### From Windows

In addition to shortcuts you may construct, Windows provides many ways to open VisualDSP++ online Help or the supplementary documentation.

Help system files (.CHM) are located in the `VisualDSP\Help` folder. Manuals and data sheets in PDF format are located in the `Docs` folder of the installation CD-ROM. The installation CD also contains the Dinkum Abridged C++ library and FlexLM network license manager software documentation in the `\Reference` folder.

#### Using Windows Explorer

- Double-click any file that is part of the VisualDSP++ documentation set.
- Double-click `vdsp-help.chm`, 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](http://www.analog.com/technology/dsp/developmentTools/gen_purpose.html)

## Product Information

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

Printed copies of hardware manuals can be ordered through the Literature Center or downloaded from the Analog Devices Web site. 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.

## Data Sheets

All data sheets can be downloaded from the Analog Devices Web site. As a general rule, printed copies of data sheet with a letter suffix (L, M, N, S) can be obtained from the Literature Center at 1-800-ANALOGD (1-800-262-5643) or downloaded from the Web site. Data sheets without



the suffix can be downloaded from the Web site only—no hard copies are available. You can ask for the data sheet by a part name or by product number.

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

## Contacting DSP Publications

Please send your comments and recommendations on how to improve our manuals and online Help. You can contact us by:

- E-mailing [dsp.techpubs@analog.com](mailto: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.

 Code has been formatted to fit this manual's page width.

| Example                                                                                             | Description                                                                                                                                                                                                    |
|-----------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Close</b> command<br>( <b>File</b> menu)                                                         | Text in <b>bold</b> style indicates the location of an item within the VisualDSP++ environment's menu system and user interface items.                                                                         |
| {this   that}                                                                                       | Alternative required items in syntax descriptions appear within curly brackets 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.                                                                                                                                              |
|  <b>Note:</b>    | A note providing information of special interest or identifying a related topic. In the online version of this book, the word <b>Note</b> appears instead of this symbol.                                      |
|  <b>Caution:</b> | 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 <b>Caution</b> appears instead of this symbol. |

# 1 COMPILER

The C/C++ compiler (`cc21k`) is part of Analog Devices development software for SHARC (ADSP-21xxx) DSPs.

This chapter contains:

- [“C/C++ Compiler Overview” on page 1-2](#)  
Provides an overview of C/C++ compiler for SHARC DSPs.
- [“Compiler Command-Line Interface” on page 1-4](#)  
Describes the operation of the compiler as it processes programs, including input and output files, and command-line switches.
- [“C/C++ Compiler Language Extensions” on page 1-60](#)  
Describes the `cc21k` compiler’s extensions to the ISO/ANSI standard for the C and C++ languages.
- [“Preprocessor Features” on page 1-114](#)  
Contains information on the preprocessor and ways to modify source compilation.
- [“C/C++ Run-Time Model and Environment” on page 1-123](#)  
Contains reference information about implementation of C/C++ programs, data, and function calls in ADSP-21xxx DSPs.
- [“C/C++ and Assembly Interface” on page 1-154](#)  
Describes how to call an assembly language subroutine from a C or C++ program, and how to call a C or C++ function from within an assembly language program.

# C/C++ Compiler Overview

The C/C++ compiler (`cc21k`) is designed to aid your DSP project development efforts by:

- Processing C and 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/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/C++ compiler (`cc21k`) compiles ISO/ANSI standard C and C++ code for the SHARC DSPs. Additionally, Analog Devices includes within the compiler a number of C language extensions designed to assist in DSP development. The `cc21k` compiler runs from the VisualDSP++ environment or from an operating system command line.

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

Your source files contain the C/C++ program to be processed by the compiler. The `cc21k` compiler supports the ANSI/ISO standard definitions of the C and C++ languages. For information on the C language standard,

see any of the many reference texts on the C language. Analog Devices recommends the Bjarne Stroustrup text “*The C++ Programming Language*” from Addison Wesley Longman Publishing Co (ISBN: 0201889544) (1997) as a reference text for the C++ programming language.

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

The `cc21k` compiler supports a set of C/C++ language extensions. These extensions support hardware features of the ADSP-21xxx DSPs. For information on these extensions, see “[C/C++ Compiler Language Extensions](#)” on page 1-60.

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 1-8). 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 SHARC DSPs* and online Help.

# Compiler Command-Line Interface

This section describes how the `ADSP-21xxx` 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 1-5](#)
- [“Specifying Compiler Options in VisualDSP++” on page 1-8](#)
- [“Compiler Command-Line Switches” on page 1-10](#)
- [“Data Type and Data Type Sizes” on page 1-53](#)

By default, the 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 1-6](#) lists valid extensions. By default, the compiler processes the input file through the listed stages to produce a `.DXE` file (see file names in [Table 1-2 on page 1-8](#)). [Table 1-3 on page 1-10](#) lists the switches that select the language dialect.

Although many switches are generic between C and C++, some of them are valid in C++ mode only. A summary of the generic C/C++ compiler switches appears in [Table 1-4 on page 1-11](#). A summary of the C++-specific compiler switches appears in [Table 1-5 on page 1-19](#). The summaries are followed by descriptions of each switch.



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 1-8](#) for more information.

## Running the Compiler

Use the following syntax for the `cc21k` command line:

```
cc21k [-switch [-switch ...] sourcefile [sourcefile ...]]
```

where:

- *-switch* is the name of the switch to be processed. The compiler has many switches. These switches select the operations and modes for the compiler and other tools. Command-line switches are case sensitive. For example, `-O` is not the same as `-o`.
- *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 compiler uses the file extension to determine what the file contains and what operations to perform upon it. [Table 1-2 on page 1-8](#) lists the allowed extensions.

For example, the following command line

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

runs `cc21k` with:

|                             |                                                                            |
|-----------------------------|----------------------------------------------------------------------------|
| <code>-O</code>             | Specifies optimization for the compiler                                    |
| <code>-21062</code>         | Identifies the target processor type in your DSP system                    |
| <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                        |

# Compiler Command-Line Interface

The following example command line

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

runs cc21k with:

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

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

In normal operation the compiler uses the files listed in [Table 1-1](#) to perform the specified action.

Table 1-1. File Extensions

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

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

You can stop this sequence at various points by using appropriate compiler switches, or by selecting options with the VisualDSP++ environment. 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, `cc21k` names the output for you. [Table 1-2 on page 1-8](#) lists the type of files, names, and extensions `cc21k` 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 1-2](#) indicates the searches that the preprocessor, compiler, assembler, and linker support. The compiler supports relative and absolute directory names to define file search paths. For information on additional search directories, see the `-I` directory switch ([on page 1-31](#)) and `-L` directory switch ([on page 1-33](#)).

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 `cc21k` compiler uses the file extension conventions listed in [Table 1-2](#) to determine the input file type.

- Verify that the compiler is using the correct file. If you do not provide the complete file path as part of the parameter or add additional search directories, `cc21k` looks for input in the current directory.



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

# Compiler Command-Line Interface

Table 1-2. Input and Output Files

| Input File Extension | File Extension Description                                                                                              |
|----------------------|-------------------------------------------------------------------------------------------------------------------------|
| .c                   | C/C++ source file.                                                                                                      |
| .cpp, .cxx           | C++ source file.                                                                                                        |
| .h                   | Header file (referenced by a <code>#include</code> directive).                                                          |
| .pch                 | C++ pre-compiled header file.                                                                                           |
| .ii, .ti             | Template instantiation files — used internally by the compiler when instantiating templates.                            |
| .ipa, .opa           | Interprocedural analysis files — used internally by the compiler when performing interprocedural analysis.              |
| .i                   | Preprocessed C source, created when preprocess only ( <code>-E</code> or <code>-P</code> compiler switch) is specified. |
| .s, .asm             | Assembler source file.                                                                                                  |
| .is                  | Preprocessed assembly source (retained when <code>-save_temps</code> is specified).                                     |
| .ldf                 | Linker Description File.                                                                                                |
| .doj                 | Object file to be linked.                                                                                               |
| .dlb                 | Library of object files to be linked as needed.                                                                         |
| .map                 | DSP system memory map file output.                                                                                      |
| .sym                 | DSP system symbol map file output.                                                                                      |

## Specifying Compiler Options in VisualDSP++

When using the VisualDSP++ Integrated Development and Debug Environment (IDDE), use the **Compile** page from the **Project Options** dialog box to set compiler functional options as shown on [Figure 1-1](#).

Callouts refer to the corresponding compiler command-line switches described in [“Compiler Command-Line Switches” on page 1-10](#). The **Additional options** field is used to enter the appropriate file names and options that do not have corresponding controls on the **Compile** 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.

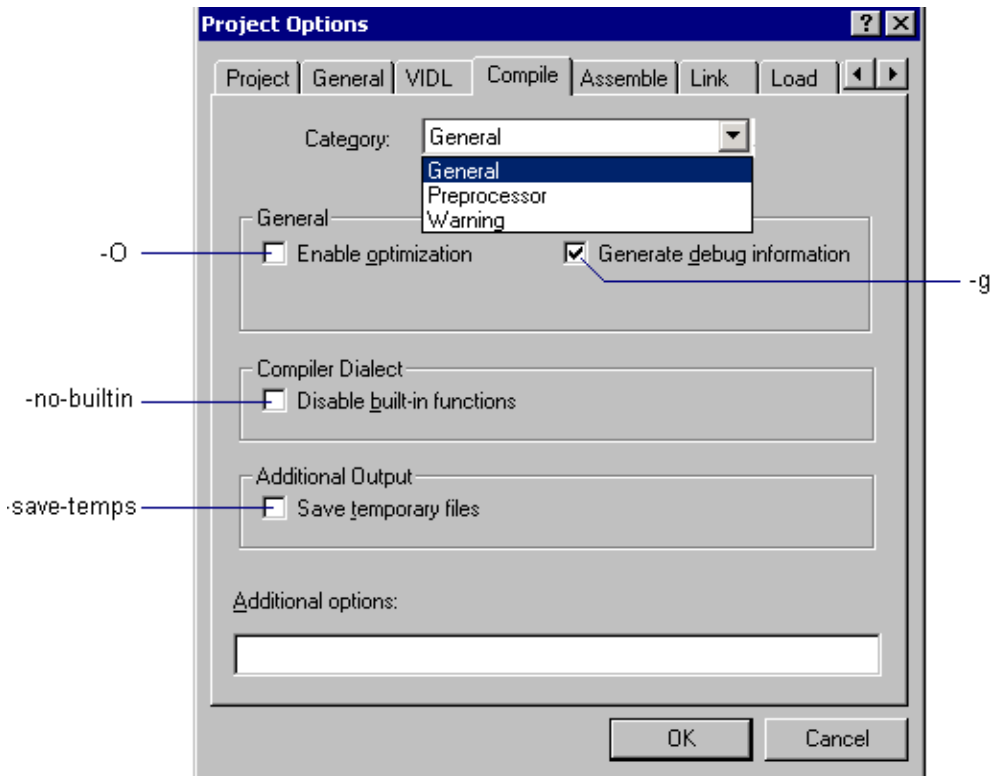


Figure 1-1. Project Options – Compile Property Page

## Compiler Command-Line Switches

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

### C/C++ Compiler Switch Summaries

This section contains a set of tables that summarize generic and specific switches (options).

- “C or C++ Mode Selection Switches”, Table 1-3
- “C/C++ Compiler Common Switches”, Table 1-4 on page 1-11
- “C++ Mode Compiler Switches”, Table 1-5 on page 1-19

A brief description of each switch follows the tables, beginning on page 1-20.

Table 1-3. C or C++ Mode Selection Switches

| Switch Name                    | Description                                                                                                          |
|--------------------------------|----------------------------------------------------------------------------------------------------------------------|
| -analog<br>(on page 1-20)      | Supports ANSI/ISO standard C with Analog Devices extensions. Default mode.                                           |
| -c++<br>(on page 1-20)         | Supports ANSI/ISO standard C++ with Analog Devices extensions. Note that C++ is not supported on the ADSP-21020 DSP. |
| -traditional<br>(on page 1-20) | Supports pre-ANSI K&R C.                                                                                             |

Table 1-4. C/C++ Compiler Common Switches

| Switch Name                                        | Description                                                       |
|----------------------------------------------------|-------------------------------------------------------------------|
| <code>sourcefile</code><br>(on page 1-21)          | Specifies file to be compiled.                                    |
| <code>-@ filename</code><br>(on page 1-21)         | Reads command-line input from the file.                           |
| <code>-21020</code><br>(on page 1-21)              | Generates code for ADSP-21020 DSPs.                               |
| <code>-21060</code><br>(on page 1-21)              | Generates code for ADSP-21060 DSPs.                               |
| <code>-21061</code><br>(on page 1-22)              | Generates code for ADSP-21061 DSPs.                               |
| <code>-21062</code><br>(on page 1-22)              | Generates code for ADSP-21062 DSPs.                               |
| <code>-21065L</code><br>(on page 1-22)             | Generates code for ADSP-21065L DSPs.                              |
| <code>-21160</code><br>(on page 1-23)              | Generates code for ADSP-21160 DSPs.                               |
| <code>-21160rev0</code><br>(on page 1-23)          | Generates code for ADSP-21160 DSPs and avoids the RFRAME anomaly. |
| <code>-21161</code><br>(on page 1-23)              | Generates code for ADSP-21161 DSPs.                               |
| <code>-A name[tokens]</code><br>(on page 1-23)     | Asserts the specified name as a predicate.                        |
| <code>-aligned-stack</code><br>(on page 1-24)      | Aligns the program stack on a double-word boundary.               |
| <code>-alttok</code><br>(on page 1-24)             | Allows alternative keywords and sequences in sources.             |
| <code>-auto-inline factor</code><br>(on page 1-24) | Controls how much the compiler automatically inlines functions.   |
| <code>-build-lib</code><br>(on page 1-25)          | Directs the librarian to build a library file.                    |

# Compiler Command-Line Interface

Table 1-4. C/C++ Compiler Common Switches (Cont'd)

| Switch Name                                                 | Description                                                                                                         |
|-------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------|
| <code>-C</code><br>(on page 1-25)                           | Retains preprocessor comments in the output file; must run with the <code>-E</code> or <code>-P</code> switch.      |
| <code>-c</code><br>(on page 1-25)                           | Compiles and/or assembles only, but does not link.                                                                  |
| <code>-circbuf</code><br>(on page 1-25)                     | Causes the compiler to treat certain array references as circular buffer references, and generate code accordingly. |
| <code>-Dmacro[=definition]</code><br>(on page 1-26)         | Defines a macro.                                                                                                    |
| <code>-debug-types</code><br>(on page 1-26)                 | Supports building a *.h file directly and writing a complete set of debugging information for the header file.      |
| <code>-default-linkage-{asm C C++}</code><br>(on page 1-26) | Sets the default linkage type (C, C++, asm).                                                                        |
| <code>-dollar</code><br>(on page 1-27)                      | Accepts dollar signs in identifiers.                                                                                |
| <code>-double-size [-32 -64]</code><br>(on page 1-27)       | Selects 32- or 64-bit IEEE format for double.                                                                       |
| <code>-dry</code><br>(on page 1-28)                         | Displays, but does not perform, main driver actions (verbose dry-run).                                              |
| <code>-dryrun</code><br>(on page 1-28)                      | Displays, but does not perform, top-level driver actions (terse dry-run).                                           |
| <code>-E</code><br>(on page 1-28)                           | Preprocesses, but does not compile, the source file.                                                                |
| <code>-EE</code><br>(on page 1-28)                          | Preprocesses and compiles the source file.                                                                          |
| <code>-expert-linker</code><br>(on page 1-29)               | Provides the initial Expert Linker link line.                                                                       |
| <code>-extra-keywords</code><br>(on page 1-29)              | Recognizes ADI extensions to ANSI/ISO standards for C and C++. Default mode.                                        |

Table 1-4. C/C++ Compiler Common Switches (Cont'd)

| Switch Name                                                           | Description                                                                                                                                    |
|-----------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>-flags-{tools} &lt;arg1&gt; [,arg2...]</code><br>(on page 1-29) | Passes command-line switches through the compiler to other build tools.                                                                        |
| <code>-float_to_int</code><br>(on page 1-29)                          | Uses a support library function to convert a <code>float</code> to an integer.                                                                 |
| <code>-fp-associative</code><br>(on page 1-30)                        | Treats floating point multiply and addition as an associative.                                                                                 |
| <code>-full-version</code><br>(on page 1-30)                          | Displays the version number of the driver and any processes invoked by the driver.                                                             |
| <code>-g</code><br>(on page 1-30)                                     | Generates DWARF-2 debug information.                                                                                                           |
| <code>-H</code><br>(on page 1-30)                                     | Outputs a list of included header files, but does not compile.                                                                                 |
| <code>-HH</code><br>(on page 1-31)                                    | Outputs a list of included header files and compiles.                                                                                          |
| <code>-h[elp]</code><br>(on page 1-31)                                | Outputs a list of command-line switches.                                                                                                       |
| <code>-I directory</code><br>(on page 1-31)                           | Appends directory to the standard search path.                                                                                                 |
| <code>-I-</code><br>(on page 1-32)                                    | Establishes the point in the <code>include</code> directory list at which the search for header files enclosed in angle brackets should begin. |
| <code>-include filename</code><br>(on page 1-32)                      | Includes named file prior to preprocessing each source file.                                                                                   |
| <code>-ipa</code><br>(on page 1-32)                                   | Enables interprocedural analysis.                                                                                                              |
| <code>-L directory</code><br>(on page 1-33)                           | Appends directory to the standard library search path.                                                                                         |
| <code>-l library</code><br>(on page 1-33)                             | Searches library for functions when linking.                                                                                                   |

# Compiler Command-Line Interface

Table 1-4. C/C++ Compiler Common Switches (Cont'd)

| Switch Name                            | Description                                                                                                                   |
|----------------------------------------|-------------------------------------------------------------------------------------------------------------------------------|
| -line-debug<br>(on page 1-33)          | Instructs the compiler to generate limited debugging information.                                                             |
| -M<br>(on page 1-34)                   | Generates make rules only, but does not compile.                                                                              |
| -MM<br>(on page 1-34)                  | Generates make rules and compiles.                                                                                            |
| -Mt <i>filename</i><br>(on page 1-34)  | Specifies an alternative output file name for the make dependency target.                                                     |
| -MQ<br>(on page 1-34)                  | Generates make rules only; does not compile. No notification when input files are missing.                                    |
| -map <i>filename</i><br>(on page 1-34) | Directs the linker to generate a memory map of all symbols.                                                                   |
| -mem<br>(on page 1-35)                 | Enables memory initialization.                                                                                                |
| -no-aligned-stack<br>(on page 1-35)    | Does not double-word align the program stack.                                                                                 |
| -no-allowtok<br>(on page 1-35)         | Does not allow alternative keywords and sequences in sources.                                                                 |
| -no-builtin<br>(on page 1-35)          | Recognizes only built-in functions that begin with two underscores(__).                                                       |
| -no-circbuf<br>(on page 1-35)          | Disables the automatic generation of circular buffer code by the compiler.                                                    |
| -no-defs<br>(on page 1-35)             | Disables preprocessor definitions: macros, include directories, library directories, run-time headers, or keyword extensions. |
| -no-dir-warnings<br>(on page 1-36)     | Removes the directory non-existent warning from the rest of the command line.                                                 |
| -no-dollar<br>(on page 1-36)           | Does not accept dollar signs in symbols.                                                                                      |



Table 1-4. C/C++ Compiler Common Switches (Cont'd)

| Switch Name                                       | Description                                                                                                                                |
|---------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| <code>-no-extra-keywords</code><br>(on page 1-36) | Does not accept ADI keyword extensions that might affect ISO/ANSI standards for C and C++.                                                 |
| <code>-no-fp-associative</code><br>(on page 1-36) | Does not treat floating point multiply and addition as an associative.                                                                     |
| <code>-no-inline</code><br>(on page 1-36)         | Ignores the <code>inline</code> keyword.                                                                                                   |
| <code>-no-mem</code><br>(on page 1-36)            | Disables memory initialization.                                                                                                            |
| <code>-no-restrict</code> (on page 1-37)          | Disables the <code>restrict</code> keyword.                                                                                                |
| <code>-no-simd</code><br>(on page 1-37)           | Disables automatic SIMD mode when compiling for ADSP-2116x DSPs.                                                                           |
| <code>-no-std-def</code><br>(on page 1-37)        | Disables preprocessor definitions and ADI keyword extensions that do not have leading underscores(__).                                     |
| <code>-no-std-inc</code><br>(on page 1-37)        | 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><br>(on page 1-38)        | Searches for only those library files specified with the <code>-l</code> switch.                                                           |
| <code>-nothreads</code><br>(on page 1-38)         | Specifies that all compiled code need not be thread-safe.                                                                                  |
| <code>-O [0 1]</code><br>(on page 1-38)           | Enables code optimizations.                                                                                                                |
| <code>-Os</code><br>(on page 1-38)                | Optimizes for code size.                                                                                                                   |
| <code>-o filename</code><br>(on page 1-38)        | Specifies the output file name.                                                                                                            |
| <code>-P</code><br>(on page 1-39)                 | Preprocesses, but does not compile, the source file. Omits line numbers in the preprocessor output.                                        |

# Compiler Command-Line Interface

Table 1-4. C/C++ Compiler Common Switches (Cont'd)

| Switch Name                                                                    | Description                                                                                                                                                               |
|--------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>-PP</code><br>(on page 1-39)                                             | Similar to <code>-P</code> , but does not halt compilation after preprocessing.                                                                                           |
| <code>-path-{asm compiler def lib link mem} directory</code><br>(on page 1-39) | Uses the specified directory as the location of the specified compilation tool (assembler, compiler, driver, librarian, linker, or and memory initializer, respectively). |
| <code>-path-install directory</code><br>(on page 1-39)                         | Uses the specified directory as the location of all compilation tools.                                                                                                    |
| <code>-path-output directory</code><br>(on page 1-40)                          | Specifies the location of non-temporary files.                                                                                                                            |
| <code>-path-temp directory</code><br>(on page 1-40)                            | Specifies the location of temporary files.                                                                                                                                |
| <code>-pch</code><br>(on page 1-40)                                            | Generates and uses precompiled header files (*.pch)                                                                                                                       |
| <code>-pchdir directory</code><br>(on page 1-40)                               | Specifies the location of PCHRepository.                                                                                                                                  |
| <code>-pedantic</code><br>(on page 1-40)                                       | Issues compiler warnings for any constructs that are not ISO/ANSI standard C/C++-compliant.                                                                               |
| <code>-pedantic-errors</code><br>(on page 1-41)                                | Issues compiler errors for any constructs that are not ISO/ANSI standard C/C++-compliant.                                                                                 |
| <code>-pplist filename</code><br>(on page 1-41)                                | Outputs a raw preprocessed listing to the specified file.                                                                                                                 |
| <code>-proc identifier</code><br>(on page 1-41)                                | Specifies that the compiler should produce code suitable for the specified DSP.                                                                                           |
| <code>-R directory</code><br>(on page 1-42)                                    | Appends directory to the standard search path for source files.                                                                                                           |
| <code>-R-</code><br>(on page 1-42)                                             | Removes all directories from the standard search path for source files.                                                                                                   |
| <code>-reserve &lt;reg1&gt;[,reg2...]</code><br>(on page 1-42)                 | Reserves certain registers from compiler use.<br><b>Note:</b> Reserving registers can have a detrimental effect on the compiler's optimization capabilities.              |

Table 1-4. C/C++ Compiler Common Switches (Cont'd)

| Switch Name                                                  | Description                                                                                          |
|--------------------------------------------------------------|------------------------------------------------------------------------------------------------------|
| <code>-restrict</code><br>(on page 1-43)                     | Enables the <code>restrict</code> keyword. This is the default mode.                                 |
| <code>-S</code><br>(on page 1-43)                            | Stops compilation before running the assembler.                                                      |
| <code>-s</code><br>(on page 1-43)                            | Removes debug info from the output executable file.                                                  |
| <code>-save-temps</code><br>(on page 1-43)                   | Saves intermediate files.                                                                            |
| <code>-show</code><br>(on page 1-43)                         | Displays the driver command-line information.                                                        |
| <code>-swf_screening</code><br>(on page 1-44)                | Ensures that the code compiled with this option will avoid the FIFO anomaly in ADSP-2116x DSPs.      |
| <code>-switch-pm</code><br>(on page 1-44)                    | Specifies that the compiler should place switch tables in program memory.                            |
| <code>-syntax-only</code><br>(on page 1-44)                  | Checks the source code for compiler syntax errors, but does not write any output.                    |
| <code>-sysdefs</code><br>(on page 1-44)                      | Defines the system definition macros.                                                                |
| <code>-T filename</code><br>(on page 1-45)                   | Specifies the Linker Description File.                                                               |
| <code>-threads</code><br>(on page 1-45)                      | Specifies that support for multi-threaded applications is to be enabled.                             |
| <code>-time</code><br>(on page 1-45)                         | Displays the elapsed time as part of the output information on each part of the compilation process. |
| <code>-Umacro</code><br>(on page 1-45)                       | Undefines macro(s).                                                                                  |
| <code>-v</code><br>(on page 1-45)                            | Displays both the version and command-line information.                                              |
| <code>-val-global &lt;name-list&gt;</code><br>(on page 1-46) | Specifies that the names given by the name-list are present in globally defined variables.           |

# Compiler Command-Line Interface

Table 1-4. C/C++ Compiler Common Switches (Cont'd)

| Switch Name                                                        | Description                                                                                             |
|--------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------|
| -verbose<br>(on page 1-46)                                         | Displays command-line information.                                                                      |
| -version<br>(on page 1-46)                                         | Displays version information.                                                                           |
| -warn-protos<br>(on page 1-46)                                     | Produces a warnings when a function is called without a prototype.                                      |
| -W{error remark <br>suppress warn} <i>number</i><br>(on page 1-46) | Overrides the default severity of the specified error message.                                          |
| -Wdriver-limit <i>number</i><br>(on page 1-47)                     | Aborts the driver after reaching the specified number of errors.                                        |
| -Werror-limit <i>number</i><br>(on page 1-47)                      | Stops compiling after reaching the specified number of errors.                                          |
| -Wremarks<br>(on page 1-47)                                        | Indicates that the compiler may issue remarks, which are diagnostic messages even milder than warnings. |
| -Wterse<br>(on page 1-47)                                          | Issues only the briefest form of compiler warning, errors, and remarks.                                 |
| -w<br>(on page 1-48)                                               | Does not display compiler warning messages.                                                             |
| -write-files<br>(on page 1-48)                                     | Enables compiler I/O redirection.                                                                       |
| -write-opts<br>(on page 1-48)                                      | Passes the user options (but not input filenames) via a temporary file.                                 |
| -xref <i>filename</i><br>(on page 1-48)                            | Outputs cross-reference information to the specified file.                                              |

Table 1-5. C++ Mode Compiler Switches

| Switch Name                                  | Description                                                                                         |
|----------------------------------------------|-----------------------------------------------------------------------------------------------------|
| <code>-explicit</code><br>(on page 1-49)     | Supports the <code>explicit</code> specifier on constructor declarations. This is the default mode. |
| <code>-namespace</code><br>(on page 1-49)    | Supports namespaces. This is the default mode.                                                      |
| <code>-newforinit</code><br>(on page 1-49)   | Limits the scope of any symbol declared within a “for” statement.                                   |
| <code>-newvec</code><br>(on page 1-50)       | Allows the overloading of <code>new[]</code> and <code>delete[]</code> .                            |
| <code>-no-demangle</code><br>(on page 1-50)  | Prevents filtering of any linker errors through the demangler.                                      |
| <code>-no-explicit</code><br>(on page 1-50)  | Does not support the <code>explicit</code> specifier on constructor declarations.                   |
| <code>-no-namespace</code><br>(on page 1-50) | Does not support namespaces.                                                                        |
| <code>-no-newvec</code><br>(on page 1-50)    | Does not allow the overloading of <code>new[]</code> and <code>delete[]</code> .                    |
| <code>-no-std</code><br>(on page 1-50)       | Disables the implicit use of the <code>std</code> namespace.                                        |
| <code>-no-wchar</code><br>(on page 1-51)     | Disables <code>new wchar_t</code> .                                                                 |
| <code>-std</code><br>(on page 1-51)          | Enables the implicit use of the <code>std</code> namespace.                                         |
| <code>-strict</code><br>(on page 1-51)       | Generates error messages for non-ANSI constructs.                                                   |
| <code>-strictwarn</code><br>(on page 1-51)   | Generates warning messages for non-ANSI constructs.                                                 |
| <code>-tpautooff</code><br>(on page 1-52)    | Disables automatic instantiation of templates.                                                      |
| <code>-trdforinit</code><br>(on page 1-52)   | Limits the scope of any symbol declared within a “for” statement.                                   |

# Compiler Command-Line Interface

Table 1-5. C++ Mode Compiler Switches (Cont'd)

| Switch Name                              | Description                                                             |
|------------------------------------------|-------------------------------------------------------------------------|
| <code>-typename</code><br>(on page 1-52) | Recognizes the <code>typename</code> keyword. This is the default mode. |
| <code>-wchar</code><br>(on page 1-52)    | Enables new <code>wchar_t</code> .                                      |

## C/C++ Mode Selection Switch Descriptions

### -analog

The `-analog` (Analog C compilation) switch directs the compiler to support Analog Devices extensions to ANSI/ISO standard C. This is the default mode. For more information about these extensions, see “[C/C++ Compiler Language Extensions](#)” on page 1-60.

### -c++

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

 C++ is not supported on the ADSP-21020 DSP.

### -traditional

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

- All `extern` declarations (including implicit declarations of functions) take effect globally
- The keywords `inline`, `signed`, `const`, and `volatile` are not recognized

- Analog Devices C/C++ language extensions are disabled except for the forms of the extra keywords that begin with a double underscore (\_\_).
- Pointer/integer comparisons are always allowed

## C/C++ Compiler Common Switch Descriptions

### sourcefile

The *sourcefile* parameter (or parameters) specifies the name of the file (or files) to be preprocessed, compiled, assembled, and/or linked. A file name can include the drive, directory, file name, and file extension. `cc21k` uses the file extension to determine the operations to perform. [Table 1-2 on page 1-8](#) lists the permitted extensions and matching compiler operations.

### -@ filename

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

### -21020

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

### -21060

The `-21060` (compile for ADSP-21060) switch directs the compiler to generate code suitable for the ADSP-21060 DSPs. When compiling for ADSP-21060 DSP, the C/C++ preprocessor defines the `__2106x__`,

## Compiler Command-Line Interface

`__ADSP21000__`, and `__ADSP21060__` macros. The `ADSP21000` and `ADSP21060` macros are also defined unless the `-pedantic-errors` switch is used.

### -21061

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

### -21062

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

### -21065L

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

### -21160

The `-21160` (compile for ADSP-21160) switch directs the compiler to generate code suitable for the ADSP-21160 DSPs. When compiling for the ADSP-21160 DSP, the C/C++ preprocessor defines `__2116x__`,



`__ADSP21000__`, and `__ADSP21160__` macros. The `ADSP21000` and `ADSP21160` macros are also defined unless the `-pedantic-errors` switch is used.

## **-21160rev0**

The `-21160rev0` (compile for ADSP-21160) switch directs the compiler to generate code suitable for the ADSP-21160 DSPs and work around the `RFRAME` anomaly. The `-21160rev0` switch ensures that the `rframe` instruction is not generated by the compiler. When compiling for the ADSP-21160 DSP, the C/C++ preprocessor defines `__2116x__`, `__ADSP21000__`, and `__ADSP21160__` macros. The `ADSP21000` and `ADSP21160` macros are also defined unless the `-pedantic-errors` switch is used.

## **-21161**

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

## **-A name[tokens]**

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

|                 |                        |
|-----------------|------------------------|
| <b>System</b>   | <code>embedded</code>  |
| <b>Machine</b>  | <code>adsp21xxx</code> |
| <b>CPU</b>      | <code>adsp21xxx</code> |
| <b>Compiler</b> | <code>cc21k</code>     |

# Compiler Command-Line Interface

## **-aligned-stack**

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

## **-alttok**

The `-alttok` (alternative tokens) switch directs the compiler to allow alternative operator keywords and digraph sequences in source files. This is the default mode. ANSI C trigraphs sequences are always expanded (even with the `-no-alttok` option), and only digraph sequences are expanded in C source files.

The following operator keywords are enabled by default:

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

## **-auto-inline factor**

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

The amount of inlining effected is shown by the following examples.

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

## **-build-lib**

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

## **-C**

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

## **-c**

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

## **-cirbuf**

The `-cirbuf` (circular buffer) option causes the compiler to treat all array references of the form `"array_var[i%n]"` as circular buffer references, and generate code accordingly, if possible.

# Compiler Command-Line Interface

## **-Dmacro[=definition]**

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

## **-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,

```
cc21k -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 | -C++]**

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



This switch can be 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.

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

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

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

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

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

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

## Compiler Command-Line Interface

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

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

### **-dry**

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

### **-dryrun**

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

### **-E**

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

## **-expert-linker**

The `-expert-linker` switch provides the link used initially by the Expert Linker, VisualDSP++ graphic memory mapping and linking tool.

## **-extra-keywords**

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

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

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

Table 1-6. Build Tools' Options

| Option                       | Tool                     |
|------------------------------|--------------------------|
| <code>-flags-asm</code>      | easm21k assembler driver |
| <code>-flags-compiler</code> | Compiler executable      |
| <code>-flags-lib</code>      | elfar library builder    |
| <code>-flags-link</code>     | Linker                   |
| <code>-flags-mem</code>      | Memory initializer       |

## **-float\_to\_int**

The `-float_to_int` switch instructs the compiler to use a support library function to convert a float to an integer. The library support routine performs extra checking to avoid a floating-point underflow occurring.

# Compiler Command-Line Interface

## **-fp-associative**

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

## **-full-version**

The `-full-version` (display versions) switch directs the compiler to display version information for build tools used in a compilation.

## **-g**

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

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

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

## **-H**

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



**-HH**

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

**-h[elp]**

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

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

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

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

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

`<VDSP++ install dir>/.../include`



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

# Compiler Command-Line Interface

## **-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.

## **-include filename**

The **-include** (include file) switch directs the preprocessor to process the specified file before processing the regular input file. Any **-D** and **-U** options on the command line are always processed before an **-include** file. Only one **-include** may be given.

## **-ipa**

The **-ipa** (interprocedural analysis) switch turns on Interprocedural Analysis (IPA) in the compiler. This option enables optimization across the entire program, including between source files that were compiled separately. The **-ipa** option should be applied to all C and C++ files in the

program. For more information, see “Interprocedural Analysis” on page 1-57. Specifying `-ipa` also implies setting the `-O` switch (on page 1-38).



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

## **-L directory[{|,}directory...]**

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

## **-l library**

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

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

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

## **-line-debug**

The `-line-debug` switch directs the compiler to generate limited debugging information. Line number information and global variable information will be generated, but local variable information will not. This option may be used with the `-O` switch, but the line number information may not always be correct.

# Compiler Command-Line Interface

## -M

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

```
object-file: include-file ...
```

## -MM

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

## -Mt filename

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

## -MQ

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

## -map filename

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

## **-mem**

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

## **-no-aligned-stack**

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

## **-no-alttok**

The `-no-alttok` (disable alternative tokens) switch directs the compiler not to accept alternative operator keywords and digraph sequences in the source files. For more information, see [“-alttok” on page 1-24](#).

## **-no-builtin**

The `-no-builtin` (no built-in functions) switch directs the compiler to ignore built-in functions that begin with two underscores (`__`). Note that this switch influences many functions. For more information on built-in functions, see [“C++ Fractional Type Support” on page 1-96](#).

## **-no-circbuf**

The `-no-circbuf` (no circular buffer) switch disables the automatic generation of circular buffer code by the compiler. Uses of the `circindex()` and `circptr()` functions (that is, explicit circular buffer operations) will not be affected.

## **-no-defs**

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

# Compiler Command-Line Interface

## **-no-dir-warnings**

The `-no-dir-warnings` switch directs the compiler to remove the directory non-existent warning from the rest of the command line.

## **-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 C and C++ languages. These include keywords such as `pm` and `dm`, which may be used as identifiers in standard conforming programs. Alternate keywords, which are prefixed with two leading underscores such as `__pm` and `__dm`, continue to work.

## **-no-fp-associative**

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

## **-no-inline**

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

## **-no-mem**

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

**-no-restrict**

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

**-no-simd**

The `-no-simd` (disable SIMD mode) switch directs the compiler to disable automatic SIMD code generation when compiling for ADSP-2116x DSPs. Note that SIMD code will still be generated for a loop if it is preceded with the “`SIMD_for`” pragma. The pragma is treated as an explicit user request to generate SIMD code and will always be obeyed, if possible. See [“SIMD Support” on page 1-99](#) for more information.

**-no-std-def**

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



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

**-no-std-inc**

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



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.

# Compiler Command-Line Interface

## **-no-std-lib**

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

## **-nothreads**

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

## **-O[0|1]**

The `-O` (enable optimizations) switch directs the compiler to produce code that is optimized for performance. Optimizations are not enabled by default for the `cc21k` compiler. The switch setting `-O` or `-O1` turns optimization on, while setting `-O0` turns off all optimizations except inlining.



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

## **-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, some delay slot filling, and jump avoidance.

## **-o filename**

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



## **-P**

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

## **-PP**

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

## **-path-{ asm | compiler | def | lib | link | mem } directory**

The `-path-{asm|compiler|def|lib|link|mem}` directory (tool location) switch directs the compiler to use the specified component in place of the default-installed version of the compilation tool. The component should comprise a relative or absolute path to its location. Respectively, the tools are the assembler, compiler, driver definitions file, librarian, linker or memory initializer. Use this switch when you wish to override the normal version of one or more of the tools. The `-path-{}`  switch also overrides the directory specified by the `-path-install` switch.

## **-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.

# Compiler Command-Line Interface

## **-path-output directory**

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

## **-path-temp directory**

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

## **-pch**

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



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

## **-pchdir directory**

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

## **-pedantic**

The `-pedantic` (ANSI standard warnings) switch causes the compiler to issue a warning for each construct found in your program that does not strictly conform to ANSI/ISO standard C or C++.



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

**-pedantic-errors**

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

**-pplist filename**

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

| 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)                                      |
| E         | Diagnostic message (error)                                        |
| C         | Diagnostic message (catastrophic error)                           |

**-proc identifier**

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

## Compiler Command-Line Interface

### **-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. The **-R directory** 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 registers. This guarantees that a known set of registers are available for inline assembly code or linked assembly modules. Separate each register name with a comma on the compiler command line.

You can reserve the following registers: **b0**, **l0**, **m0**, **i0**, **b1**, **l1**, **m1**, **i1**, **b8**, **l8**, **m8**, **i8**, **b9**, **l9**, **m9**, **i9**, **ustat1**, and **ustat2** (as well as **ustat3** and **ustat4** on ADSP-2116x DSPs). When reserving an **L** (length) register, you must reserve the corresponding **I** (index) register; reserving an **L** register without reserving the corresponding **I** register may result in execution problems.

## **-restrict**

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

## **-S**

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



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

## **-save-temps**

The `-save-temps` (save intermediate files) switch directs the compiler not to discard intermediate files. The compiler places the intermediate output (`*.i`, `*.is`, `*.s`, `*.doj`) files in the `temp` subdirectory of the current project directory. See [Table 1-2 on page 1-8](#) for a list of intermediate files.

## **-show**

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

# Compiler Command-Line Interface

## **-swf\_screening**

The `-swf_screening` switch ensures that code compiled with this option will avoid the FIFO anomaly in ADSP-2116x DSPs.

## **-switch-pm**

The `-switch-pm` (switch tables) switch directs the compiler to place switch tables in program memory.

## **-syntax-only**

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

## **-sysdefs**

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

| Macro         | Description                     |
|---------------|---------------------------------|
| __HOSTNAME__  | The name of the host machine    |
| __MACHINE__   | The 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 the linker to use the specified Linker Description File (`.LDF`) as control input for linking. If `-T` is not specified, a default `.LDF` file is selected based on the processor variant.

## **-threads**

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



This switch is only likely to be used by applications involving the VisualDSP++ Kernel (VDK).

## **-time**

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

## **-Umacro**

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



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

## **-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 Interface

## **-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++, these names are encoded as enclosing namespace or classes. 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 version) switch directs the compiler to display version information for all the compilation tools as they process each file.

## **-warn-protos**

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

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

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

## **-Wdriver-limit number**

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

## **-Werror-limit number**

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

## **-Wremarks**

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

# Compiler Command-Line Interface

## **-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.

## **-write-files**

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

## **-write-opts**

The `-write-opts` (user options) switch directs the compiler to pass the user options (but not the input *filenames*) to the main driver via a temporary file which can help if the resulting main driver command line is too long.

## **-xref <filename>**

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

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

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

is written to the named file.

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

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

## C++ Mode Compiler Switch Descriptions

The following switches apply only to C++.

### **-explicit**

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

### **-namespace**

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

### **-newforinit**

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

# Compiler Command-Line Interface

## **-newvec**

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

## **-no-demangle**

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

## **-no-explicit**

The `-no-explicit` (disable explicit specifier) switch directs the compiler to disable support for the explicit specifier on constructor declarations. For more information, see [“-explicit” on page 1-49](#).

## **-no-namespace**

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

## **-no-newvec**

The `-no-newvec` (disallow a new vector) switch directs the compiler to disallow the overloading of the `new[]` and `delete[]` operators. For more information, see [“-newvec” on page 1-50](#).

## **-no-std**

The `-no-std` (disable std namespace) switch directs the compiler to disable the implicit use of the `std` namespace when the standard header files are included. For more information, see [“-std” on page 1-51](#).

## **-notstrict**

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

## **-no-wchar**

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

## **-std**

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

## **-strict**

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

## **-strictwarn**

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

## Compiler Command-Line Interface

### **-tpautooff**

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

### **-trdforinit**

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

### **-typename**

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

### **-wchar**

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

## Data Type and Data Type Sizes

The sizes of intrinsic C/C++ data types are selected by Analog Devices so that normal C/C++ programs execute with hardware-native data types and therefore at high speed. [Table 1-7](#) shows the size used for each of the intrinsic C/C++ data types.

Table 1-7. Data Type Sizes for the ADSP-21xxx DSPs

| Type           | Bit Size                                | Result of sizeof operator |
|----------------|-----------------------------------------|---------------------------|
| int            | 32 bits signed                          | 1                         |
| unsigned int   | 32 bits unsigned                        | 1                         |
| long           | 32 bits signed                          | 1                         |
| unsigned long  | 32 bits unsigned                        | 1                         |
| char           | 32 bits signed                          | 1                         |
| unsigned char  | 32 bits unsigned                        | 1                         |
| short          | 32 bits signed                          | 1                         |
| unsigned short | 32 bits unsigned                        | 1                         |
| pointer        | 32 bits                                 | 1                         |
| float          | 32 bits float                           | 1                         |
| double         | either 32 or 64 bits float (default 32) | either 1 or 2 (default 1) |
| long double    | 64 bits float                           | 2                         |
| fract          | 32 bits fixed-point                     | 1                         |

Analog Devices does not support data sizes smaller than the addressable unit size on the processor. For the ADSP-21xxx processors, this means that both `short` and `char` have the same size as `int`. Although 32-bit `chars` are unusual, they do conform to the standard. For information about the `fract` data type, refer to [“C++ Fractional Type Support” on page 1-96](#).

## Integer Data Types

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

## Floating-Point Data Types

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

Type `double` poses a special problem. Without some special handling, many programs would inadvertently end up using slow-speed, emulated, 64-bit floating-point arithmetic, even when variables are declared consistently as `float`. In order to avoid this problem, Analog provides the “[-double-size\[-32|-64\]](#)” switch (on [page 1-27](#)) that allows you to set the size of `double` to either 32 bits (default) or 64 bits. The 32-bit setting gives good performance and should be acceptable for most DSP programming. However, it does not conform fully to the ANSI C standard.

For a larger floating-point type, the `long double` data type provides 64-bit floating-point arithmetic.

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

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



For full descriptions of these functions and their implementation, see Chapter 2, “C/C++ Run-Time Library”.

## Optimization Control

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

- **Debugging.** The compiler produces debug information to ensure that the object code can be matched to the appropriate source code line. See “`-g`” on page 1-30 for more information.
- **Default.** The compiler does basic high-level optimization, such as inlining functions that are explicitly marked for inlining.
- **Procedural Optimization.** The compiler does advanced, aggressive optimization on each procedure in the file being compiled. If debugging is also requested, the optimization is given priority so that debugging functionality may be limited. See “`-O[0|1]`” on page 1-38 for more information.
- **Interprocedural Optimization.** The compiler does advanced, aggressive optimization over the whole program in addition to the per-file optimizations in procedural optimization. See on page 1-32 for more information.

Interprocedural analysis (see “Interprocedural Analysis” on page 1-57) allows the compiler to see all of the source files that are used and to use that information to enable the other optimizations to be exploited as fully as possible.

When no optimization switches are specified, `cc21k` effects only basic high level optimizations, such as inlining functions, which have been explicitly marked for inlining. When `-g` is specified, however, all inlining is suppressed to provide as comprehensive debugging information as possible.

## Compiler Command-Line Interface

When `-inline` is specified with `-g`, the system provides the explicitly specified inlining, which reduces the amount of source line debug information that is available. Therefore, the use of `-g` by itself effectively disables almost all optimizations.

Normally, a program is optimized to process the data as quickly as possible, but in some circumstances, the speed of the program may be less important than reducing the size of the generated code. When the `-Os` switch is specified, the compiler will only perform standard optimizations that do not significantly increase the size of the generated code. (See “[-Os](#)” on page 1-38 for more information.)

The `-O` switch (on page 1-38) directs the compiler to effect all normally safe optimizations. It also directs the compiler to generate the fastest possible executing code while conforming to standard language interpretations and a conservative view of any possible interactions between variables. The use of interprocedural analysis can be very useful in enabling the compiler to be more aggressive in optimizing the program since it has much greater knowledge of the overall structure of the program and the data being manipulated by the program.

For the ADSP-2116x DSPs, the optimizer always attempts to vectorize loops when it is safe to do so and uses information from the Interprocedural Analyzer to identify more opportunities to do so. In addition, there may be other loops that you know are safe candidates for the vectorizer; you can use pragmas to inform the optimizer of such loops (see “[SIMD Support](#)” on page 1-99).

### Inlining Control

By default, `cc21k` inlines class members and those functions that are explicitly marked to be inlined. When the `-no-inline` switch (see “[-no-inline](#)” on page 1-36) is specified, then any explicit request for inlining is ignored.

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

## Interprocedural Analysis

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

Interprocedural analysis is enabled by selecting the **Interprocedural Analysis** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **General** category, or by specifying the `-ipa` command-line switch (see [“-ipa” on page 1-32](#)).

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

Use of the `-ipa` switch causes additional files to be generated along with the object file produced by the compiler. These files have `.ipa` and `.opa` filename extensions and should not be deleted manually unless the associated object file is also deleted.

All of the `-ipa` optimizations are invoked after the initial link, whereupon a special program called the prelinker reinvokes the compiler to perform the new optimizations.

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

## Interaction with Libraries

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

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

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

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

```
int delete_me(int x) {  
    return x-2;  
}  
  
#pragma retain_name("keep_me")  
int keep_me(int y) {
```

```
        return y+2;
    }

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

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

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

# C/C++ Compiler Language Extensions

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

This section contains:

- [“Inline Function Support Keyword \(inline\)” on page 1-63](#)
- [“Inline Assembly Language Support Keyword \(asm\)” on page 1-64](#)
- [“Dual Memory Support Keywords \(pm dm\)” on page 1-74](#)
- [“Placement Support Keyword \(section\)” on page 1-79](#)
- [“Boolean Type Support Keywords \(bool, true, false\)” on page 1-80](#)
- [“Pointer Class Support Keyword \(restrict\)” on page 1-80](#)
- [“Variable-Length Array Support” on page 1-81](#)
- [“Non-Constant Initializer Support” on page 1-83](#)
- [“Indexed Initializer Support” on page 1-83](#)
- [“Aggregate Constructor Expression Support” on page 1-85](#)
- [“Preprocessor Generated Warnings” on page 1-85](#)
- [“C++ Style Comments” on page 1-86](#)
- [“Built-In Functions” on page 1-86](#)
- [“Supported Pragmas” on page 1-91](#)
- [“C++ Fractional Type Support” on page 1-96](#)
- [“Saturated Arithmetic” on page 1-99](#)
- [“SIMD Support” on page 1-99](#)

The additional keywords that are part of the C/C++ extensions do not conflict with any ANSI C/C++ keywords. The formal definitions of these extension keywords are prefixed with a leading double underscore (`__`). Unless the `-no-extra-keywords` command-line switch is used, the compiler defines the shorter form of the keyword extension that omits the leading underscores. [For more information, see “C++ Mode Compiler Switch Descriptions” on page 1-49.](#)

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

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

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

[Table 1-8 on page 1-62](#) provides a brief description of each keyword extension and directs you to sections of this chapter that document the extensions in more detail. [Table 1-9 on page 1-62](#) provides a brief description of each operational extension and directs you to sections that document these extensions in more detail.

Table 1-8. Keyword Extensions

| Keyword extensions             | Description                                                                                                                                                                                                                                                                                     |
|--------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>inline(function)</code>  | Directs the compiler to integrate the function code into the code of the calling function(s). <a href="#">For more information, see “Inline Function Support Keyword (inline)” on page 1-63.</a>                                                                                                |
| <code>asm()</code>             | Places ADSP-21xxx family assembly language instructions directly in your C/C++ program. <a href="#">For more information, see “Inline Assembly Language Support Keyword (asm)” on page 1-64.</a>                                                                                                |
| <code>dm</code>                | Specifies the location of a static or global variable or qualifies a pointer declaration “*” as referring to Data Memory (DM). <a href="#">For more information, see “Dual Memory Support Keywords (pm dm)” on page 1-74.</a>                                                                   |
| <code>pm</code>                | Specifies the location of a static or global variable or qualifies a pointer declaration “*” as referring to Program Memory (PM). <a href="#">For more information, see “Dual Memory Support Keywords (pm dm)” on page 1-74.</a>                                                                |
| <code>section("string")</code> | Specifies the section in which an object or function is placed. The <code>section</code> keyword has replaced the <code>segment</code> keyword of the previous releases of the compiler software. <a href="#">For more information, see “Placement Support Keyword (section)” on page 1-79.</a> |
| <code>bool, true, false</code> | A boolean type. <a href="#">For more information, see “Boolean Type Support Keywords (bool, true, false)” on page 1-80.</a>                                                                                                                                                                     |
| <code>restrict keyword</code>  | Specifies restricted pointer features. <a href="#">For more information, see “Pointer Class Support Keyword (restrict)” on page 1-80.</a>                                                                                                                                                       |

Table 1-9. Operational Extensions

| Operation extensions      | Description                                                                                                                                                                                                                |
|---------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Variable-length arrays    | Support for variable-length arrays lets you use arrays whose length is not known until run time. <a href="#">For more information, see “Variable-Length Array Support” on page 1-81.</a>                                   |
| Non-constant initializers | Support for non-constant initializers lets you use non-constants as elements of aggregate initializers for automatic variables. <a href="#">For more information, see “Non-Constant Initializer Support” on page 1-83.</a> |



Table 1-9. Operational Extensions (Cont'd)

| Operation extensions                    | Description                                                                                                                                                                                                                            |
|-----------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Indexed initializers                    | Support for indexed initializers lets you specify elements of an aggregate initializer in an arbitrary order. <a href="#">For more information, see “Indexed Initializer Support” on page 1-83.</a>                                    |
| Aggregate constructor expressions       | Support for aggregate assignments lets you create an aggregate array or structure value from component values within an expression. <a href="#">For more information, see “Aggregate Constructor Expression Support” on page 1-85.</a> |
| <code>fract</code> data type (C++ mode) | Support for the fractional data type, fractional and saturated arithmetic. <a href="#">For more information, see “C++ Fractional Type Support” on page 1-96.</a>                                                                       |
| Preprocessor generated warnings         | Lets you generate warning messages from the preprocessor. <a href="#">For more information, see “Preprocessor Generated Warnings” on page 1-85.</a>                                                                                    |
| C++ style comments                      | Allows for “//” C++ style comments in C programs. <a href="#">For more information, see “C++ Style Comments” on page 1-86.</a>                                                                                                         |

## Inline Function Support Keyword (`inline`)

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

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

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

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

 The `-no-inline` and `-traditional` switches disable function inlining. For more information, see “C/C++ Compiler Common Switch Descriptions” on page 1-21.

### Inline Assembly Language Support Keyword (`asm`)

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

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

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

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

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

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

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

- [“Assembly Construct Template” on page 1-65](#)
- [“Assembly Construct Operand Description” on page 1-68](#)
- [“Assembly Constructs With Multiple Instructions” on page 1-70](#)
- [“Assembly Construct Reordering and Optimization” on page 1-71](#)
- [“Assembly Constructs with Input and Output Operands” on page 1-72](#)
- [“Assembly Constructs and Macros” on page 1-74](#)

## Assembly Construct Template

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

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

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 semico-

lon, and enclose the entire string within double quotes. For more information on templates containing multiple instructions, see [“Assembly Constructs With Multiple Instructions” on page 1-70](#).

- **constraint**

The constraint string directs cc21k to use certain groups of registers for the input and output operands. Enclose the constraint string within double quotes. For more information on operand constraints, see [“Assembly Construct Operand Description” on page 1-68](#).

- **output operand**

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

- **input operand**

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

- **clobber**

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

The following rules apply to assembly construct template syntax.

- The template is the only mandatory argument to `asm()`. All other arguments are optional.
- An operand constraint string followed by a C or C++ expression in parentheses describes each operand. For output operands, it must be possible to assign to the expression, that is, the expression must be legal on the left side of an assignment statement.

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



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

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

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

In the above example, note:

- The template is `"%0=clip %1 by %2;"`. The %0 is replaced with operand zero (`result`), the first operand, %1, is replaced with operand one (`x`), and %2 is replaced with operand two (`y`).

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

### Assembly Construct Operand Description

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

[Table 1-9 on page 1-62](#) describes the correspondence between constraint letters and register classes. Note that the use of any letter not listed in [Table 1-9](#) results in unspecified behavior. The compiler does not check the validity of the code by using the constraint letter.

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

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

- The output operand list appears as the first argument after the assembly language template. The list is separated from the assembly language template with a colon. The input operands are separated from the output operands with a colon and always follow the output operands.
- The operand constraints (see [Table 1-10](#)) describe which registers are modified by an assembly language instruction. The `= in =con-straint` indicates that the operand is an output; all output operand constraints must use `=`.
- The compiler may allocate an output operand in the same register as an unrelated input operand, unless the output operand has the `&=` constraint modifier. This is because `cc21k` assumes that the inputs are consumed before the outputs are produced. This assumption may be false if the assembler code actually consists of more than one instruction. In such a case, use `&=` for each output operand that must not overlap an input.

Table 1-10. ASM() Operand Constraints

| Constraint <sup>1</sup> | Register type            | Registers |
|-------------------------|--------------------------|-----------|
| a                       | DAG2 B registers         | b8 — b15  |
| b                       | Q2 R registers           | r4 — r7   |
| c                       | Q3 R registers           | r8 — r11  |
| d                       | All R registers          | r0 — r15  |
| e                       | DAG2 L registers         | l8 — l15  |
| F                       | Floating-point registers | F0 — F15  |

Table 1-10. ASM() Operand Constraints (Cont'd)

| Constraint <sup>1</sup> | Register type                                                                                       | Registers                                                           |
|-------------------------|-----------------------------------------------------------------------------------------------------|---------------------------------------------------------------------|
| f                       | Accumulator register                                                                                | mrf, mrb                                                            |
| h                       | DAG1 B registers                                                                                    | b0 — b7                                                             |
| j                       | DAG1 L registers                                                                                    | l0 — l7                                                             |
| k                       | Q1 R registers                                                                                      | r0 - r3                                                             |
| l                       | Q4 R registers                                                                                      | r12 - r15                                                           |
| r                       | All general registers                                                                               | r0 — r15, i0 — i15, l0 — l15,<br>m0 — m15, b0 — b15, ustat1, ustat2 |
| u                       | User registers                                                                                      | ustat1, ustat2 (also ustat3, ustat4 on<br>ADSP-2116x DSPs)          |
| w                       | DAG1 I registers                                                                                    | I0 — I7                                                             |
| x                       | DAG1 M registers                                                                                    | M0 — M7                                                             |
| y                       | DAG2 I registers                                                                                    | I8 — I15                                                            |
| z                       | DAG2 M registers                                                                                    | M8 — M15                                                            |
| =& constraint           | Indicates that the constraint is applied to an output operand that may not overlap an input operand |                                                                     |
| =constraint             | Indicates that the constraint is applied to an output operand                                       |                                                                     |

1 The use of any letter not listed in [Table 1-10](#) results in unspecified behavior. The compiler does not check the validity of the code by using the constraint letter.

### Assembly Constructs With Multiple Instructions

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



The following listing is an example of multiple instructions in a template.

```

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

```

## Assembly Construct Reordering and Optimization

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

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

```

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

```

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

### Restrictions on the Use of the `asm` Construct

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

- Control flow through a procedure must not be changed using instructions contained in an `asm()` construct. Control flow should only be specified through use of the C/C++ source constructs.
- C variables should not be referenced explicitly in the `asm()` construct template code. Such references should be through the input and output operands of the construct.
- All registers changed in the `asm()` construct template must be listed in the clobber section.
- Preprocessor macros defined in the C/C++ source, if used in the `asm()` construct template, must be expanded in the C/C++ source or defined in an prior `asm()` construct to achieve the correct definition.

### Assembly Constructs with Input and Output Operands

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

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

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

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

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

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

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

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

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

### Assembly Constructs and Macros

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

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

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

### Dual Memory Support Keywords (pm dm)

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

The following rules apply to dual memory support keywords:

- The memory space keyword (`dm` or `pm`) refers to the expression to the right of the keyword.
- You can specify a memory space for each level of pointer. This corresponds to one memory space for each `*` in the declaration.

- The compiler uses Data Memory (DM) as the default memory space for all variables. All undeclared spaces for data are Data Memory spaces.
- The compiler always uses Program Memory (PM) as the memory space for functions. Function pointers always point to Program Memory.
- You cannot assign memory spaces to automatic variables. All automatic variables reside on the stack, which is always in Data Memory.
- Literal character strings always reside in Data Memory.

The following listing shows examples of dual memory keyword syntax.

```
int pm buf[100];
/* declares an array buf with 100 elements in Program Memory */
int dm samples[100];
/* declares an array samples with 100 elements in Data Memory */
int points[100];
/* declares an array points with 100 elements in Data Memory */
int pm * pm xy;
/* declares xy to be a pointer which resides in Program
   Memory and points to a Program Memory integer */
int dm * dm xy;
/* declares xy to be a pointer which resides in Data Memory and
   points to a Data Memory integer */
int *xy;
/* declares xy to be a pointer which resides in Data Memory
   and points to a Data Memory integer */
int pm * dm datp;
/* declares datp to be a pointer which resides in Data Memory
   and points to a Program Memory integer */
int pm * datp;
/* declares datp to be a pointer which resides in Data Memory
   and points to a Program Memory integer */
int dm * pm progd;
/* declares progd to be a pointer which resides in Program
```

## C/C++ Compiler Language Extensions

```
Memory and points to a Data Memory integer */
int * pm progd;
/* declares progd to be a pointer which resides in Program
Memory and points to a Data Memory integer */
float pm * dm * pm xp;
/* The first *xp is in Program Memory,
the following *xp in Data Memory, and xp itself is
in Program Memory */
```

Memory space specification keywords cannot qualify type names and structure tags, but you can use them in pointer declarations. The following 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, such as:

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

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

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

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

## Memory Keywords and Function Declarations/Pointers

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

```
/* Dual Memory Support Keyword Function Declaration (With Pointers) Syntax Examples */
int * y();          /* function y resides in */
                   /* pm and returns a      */
                   /* pointer to an integer */
                   /* which resides in dm   */
int pm * y();       /* function y resides in */
                   /* pm and returns a      */
                   /* pointer to an integer */
                   /* which resides in pm   */

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

int * pm * y();     /* function y resides in */
```

```
/* pm and returns a          */
/* pointer to a pointer      */
/* residing in pm that       */
/* points to an integer      */
/* which resides in dm       */
```

### Memory Keywords and Function Arguments

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

```
/* Illegal Dual Memory Support Keywords & Calls To Prototyped
Functions */
extern int foo(int pm*);
/* declare function foo() which expects a pointer to an int
residing in pm as its argument and which returns an int */

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

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

### Memory Keywords and Macros

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



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

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

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

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

## Placement Support Keyword (section)

The `section` keyword directs the compiler to place an object or function in an assembly `.SECTION` of the compiler's intermediate output file. You name the assembly `.SECTION` directive with the `section()`'s string literal parameter. If you do not specify a `section()` for an object or function declaration, the compiler uses a default section. For information on the default sections, see [“Memory Usage” on page 1-125](#).



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

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

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



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

### Boolean Type Support Keywords (`bool`, `true`, `false`)

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

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

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

### Pointer Class Support Keyword (`restrict`)

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

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

For example,

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

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

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

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

## Variable-Length Array Support

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

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

You can use variable-length arrays as function arguments, such as:

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

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

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

As an example, if you were to implement a routine for computation of a product of three matrices, you need to allocate a temporary matrix of the same size as the input matrices. Declaring an automatic variable size matrix is much easier 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 case the `sizeof()` operator is used to determine the size of the array. For multidimensional arrays, the boundaries are also saved for address computation. After leaving the block, all the space allocated for the array is deallocated. For example, the following program prints 10, not 50.

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

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

## Non-Constant Initializer Support

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

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

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

## Indexed Initializer Support

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

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

```
/* Example 1 Standard & cc21k C/C++ Array Initializer */
/* Standard array initializer */
int a[6] = { 0, 0, 15, 0, 29, 0 };

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

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

## C/C++ Compiler Language Extensions

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

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

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

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

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

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

```
/* Example 3 Array Initializer With enum Type Indices */
/* cc21k C/C++ array initializer */

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

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

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

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

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

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

## Aggregate Constructor Expression Support

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

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

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

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

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

## Preprocessor Generated Warnings

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

### C++ Style Comments

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

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

which a standard C compiler processes as:

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

### Built-In Functions

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

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.

This section describes:

- [“Access to System Registers” on page 1-87](#)
- [“Circular Buffer Built-In Functions” on page 1-90](#)

The `cc21k` compiler provides built-in versions of some of the C library functions as described in [“Using Compiler’s Built-In C Library Functions” on page 2-19](#).



## Access to System Registers

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.

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

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

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

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

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

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

`sysreg_write_nop` toggles all the bits of the nominated system register that are set in the supplied bit mask, but also places a 'NOP;' after the instruction.

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

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

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

`sysreg_bit_clr_nop` toggles all the bits of the nominated system register that are set in the supplied bit mask, but also places 'NOP;' after the instruction.

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

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

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

`sysreg_bit_set_nop` toggles all the bits of the nominated system register that are set in the supplied bit mask, but also places ‘NOP;’ after the instruction.

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

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

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

`sysreg_bit_tgl_nop` toggles all the bits of the nominated system register that are set in the supplied bit mask, but also places ‘NOP;’ after the instruction.

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

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

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

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



The register names are defined in `sysreg.h` and must be a compile-time literal. The effect of using the incorrect function for the size of the register or using an undefined register number is undefined.

On all ADSP-21xxx DSPs, the system registers are:

```
sysreg_IMASK  
sysreg_IMASKP  
sysreg_ASTAT  
sysreg_STKY  
sysreg_USTAT1  
sysreg_USTAT2
```

In addition, ADSP-2116x DSPs have the following system registers:

```
sysreg_LIRPTL  
sysreg_MMASK  
sysreg_ASTATY  
sysreg_FLAGS  
sysreg_STKYY  
sysreg_USTAT3  
sysreg_USTAT4
```



Due to hardware characteristics of the SHARC processor, the bit values for the `sysreg_bit_*` interrogation and manipulation functions must be compile-time constants.

For the ADSP-2106x and ADSP-21020 DSPs, the header files `def21060.h`, `def21061.h`, `def210651.h` and `def21020.h` provide symbolic names for the individual bits in the system registers.

For the ADSP-2116x DSPs, the header files `def21160.h` and `def21161.h` provide symbolic names for the individual bits in the system registers.



The compiler considers `USTAT1` and `USTAT2` to be “preserved” registers. If one is modified by a `sysreg` operation, it will be saved during the prolog of the containing function and restored to its old value on exit.

### Circular Buffer Built-In Functions

The C/C++ compiler provides the following two built-in functions for using the SHARC DSP’s circular buffer mechanisms.

#### Circular Buffer Increment of an Index

The following operation performs a circular buffer increment of an index.

```
int circindex(int index, int incr, int num_items)
```

The equivalent operation is:

```
index +=incr;
if (index <0)index +=nitems;
else if (index >=nitems)index -=nitems;
```

#### Circular Buffer Increment of a Pointer

The following operation provides a circular buffer increment of an pointer.

```
int circptr(void * ptr, size_t incr, void * base, size_t buflen)
```

The equivalent operation is:

```
ptr +=incr;
if (ptr <base)ptr +=buflen;
else if (ptr >=(base+buflen))ptr -=buflen;
```

For more information, see [“circindex” on page 2-45](#) and [“circptr” on page 2-47](#).

The compiler also attempts to generate circular buffer increments for modulus array references, such as `array[ index %nitems ]`. For this to happen, the compiler must be able to determine that the starting value for `index` will be within the range `0..(nitems-1)`. When the “`-circbuf`” switch (see [on page 1-25](#)) is specified, the compiler will always treat array references of the form `"[ i%n ]"` as a circular buffer operation on the array.

## Supported Pragmas

The SHARC C/C++ compiler supports a number of pragmas. Pragmas are implementation-specific directives that modify the compiler’s behavior. The C/C++ compiler supports pragmas for:

- Arranging special alignment for data
- Defining functions that act as interrupt or exception handlers
- Giving additional information about loop usage to improve optimization
- Changing the optimization level, midway through a module
- Changing how an externally visible function is linked

The following sections describe the pragmas that support the features listed above.

- [“Data Alignment Pragmas”](#)
- [“Interrupt Handler Pragma” on page 1-94](#)
- [“SIMD\\_for Pragma” on page 1-94](#)
- [“Optimization Level Pragmas” on page 1-94](#)
- [“External Linkage Pragmas” on page 1-95](#)

### Data Alignment Pragmas

The data alignment pragmas are used to modify how the compiler arranges data within memory. When structs are defined, the struct's overall alignment is the same as the field which has the largest alignment. The struct's size may need padding to ensure all fields are properly aligned, and that the struct's overall size is a multiple of its alignment.

Sometimes, it is useful to change these alignments. A struct may have its alignment reduced, so that a large array occupies less space. A struct may have its alignment increased to improve the compiler's opportunities in vectorizing access to the data.

The data alignment pragmas include `ALIGN NUM`, `PACK (ALIGNOPT)`, and `PAD (ALIGNOPT)` pragmas. Alignments specified using these pragmas must be a power of 2.

#### ALIGN NUM Pragma

The `align num` pragma may be used before variable declarations and field declarations. It applies to the variable or field declaration that immediately follows the pragma.

The pragma's effect is that the next variable or field declaration should be forced to be aligned on a boundary specified by *num*.

- If *num* is greater than the alignment normally required by the following variable or field declaration, then the variable or field declaration's alignment is changed to be *num*.
- If *num* is less than alignment normally required, then the variable or field declaration's alignment is changed to be *num*, and a warning is given that the alignment has been reduced.

If the `pack` or `pad` pragmas (see below) are currently active then `align` will override for the immediately following field declaration.

## PACK (ALIGNOPT) Pragma

The `pack()` pragma may be applied to struct definitions. It applies to all struct definitions that follow, until the default alignment is restored by omitting `alignopt`.

The pragma is used to reduce the default alignment of the struct to be aligned. If there are fields within the struct that have a default alignment greater than `align`, their alignment is reduced to be `alignopt`. If there are fields within the struct that have alignment less than `align`, their alignment is unchanged.

If `alignopt` is specified, it is illegal to invoke `#pragma pad`, until default alignment is restored.

## PAD (ALIGNOPT) Pragma

The `pad()` pragma may be applied to struct definitions. It applies to struct definitions that follow, until the default alignment is restored by omitting `alignopt`.

This pragma is effectively shorthand for placing `#pragma align` before every field within the struct definition. Like `pragma pack`, it reduces the alignment of fields which default to an alignment greater than `alignopt`. However, unlike `pragma pack`, it also increases the alignment of fields which default to an alignment less than `alignopt`.



While `pragma alignopt` generates a warning if a field alignment is reduced, `pragma pad alignopt` does not.

If `alignopt` is specified, it is illegal to invoke `#pragma pack`, until default alignment is restored.

### 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. This pragma indicates that the compiler should ensure that all used registers (including scratch registers) are restored at the end of the function. The compiler will also ensure that, if an I register is used in the function, the corresponding L register will be set to zero, so that circular buffering is not inadvertently invoked.

The `#pragma interrupt` should be used for interrupt handlers which are enabled with the `interruptss()` or `signalss()` family of interrupt functions as these functions ensure that the correct dispatcher is called. For maximum benefit, the `#pragma interrupt` should be used for leaf routines only (that is, functions which do not call any other functions). The reason for this is that the interrupt handler will ensure that L registers are zeroed for the I registers which are used in the function; if a function call is present, the handler must ensure that all appropriate L registers are set to zero, and this will add considerably to the execution time of the handler.

### SIMD\_for Pragma

The `SIMD_for` pragma is used with ADSP-2116x DSPs. It enables the compiler to take advantage of Single-Instruction Multiple-Data (SIMD) operations. See [“SIMD Support” on page 1-99](#) for more details.

### 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_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. The `#pragma optimize_for_space` has the same effect as setting the `-O` switch and the `#pragma optimize_for_speed` has the same effect as setting the `-Os` switch. The `optimize_off` pragma turns off the optimizer, if it was enabled. The `#pragma optimize_off` has the same effect as compiling with no optimization 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 */ }
```

Note the certain high-level optimizations (for example, function inlining) are performed on the entire module, not on a function-by-function basis. As a result, it is not possible to control these using the pragmas.

## External Linkage Pragmas

External linkage pragmas (`linkage_name` and `retain_name`) 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 the identifier is used as the external reference, instead of following the compiler's usual conventions.

For example,

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

### RETAIN\_NAME Pragma

The `retain_name` pragma indicates that the following external function or variable declaration is not removed even though inter-procedural analysis sees that is not used. Use this pragma before C/C++ function declarations that are only called from assembly routines.

```
#pragma retain_name  
int w_var = 0;  
#pragma retain_name  
void w_func(){}  

```

### C++ Fractional Type Support

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

Fractional values are declared with the `fract` data type. Ensure that your program includes the `<fract>` header file. The `fract` data type is a C++ class that supports a set of standard arithmetic operators used in arithmetic expressions. Fractional values are represented as signed values in a range of  $[-1 \dots 1)$  with a binary point immediately after the sign bit.

Other value ranges are obtained by scaling or shifting. In addition to the arithmetic, assignment, and shift operations, `fract` provides several type-conversion operations.

For more information about supported fractional arithmetic operators, see [“Fractional Arithmetic Operations” on page 1-97](#). For sample programs demonstrating the use of the `fract` type, see [Listing 1-3 on page 1-169](#).



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

## Format of Fractional Literals

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

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

## Conversions Involving Fractional Values

The following notes apply to type-conversion operations:

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

## Fractional Arithmetic Operations

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

- Standard arithmetic operations on two `fract` items include addition, subtraction, and multiplication.
- Assignment operations include `+=`, `-=`, and `*=`.

- Shift operations include left and right shifts. A left shift is implemented as a logical shift and a right shift is an arithmetic shift. Shifting left by a negative amount is not recommended.
- Comparison operations are supported between two `fract` items.
- Mixed-mode arithmetic has a preference for `fract`. For more information about the mixed-mode arithmetic, see [“Mixed Mode Operations”](#).
- Multiplication of a fractional and an integer produces an integer result or a fractional result. The program context determines which type of result is generated following the conversion algorithm of C++. When the compiler does not have enough context, it generates an ambiguous operator message, for example:

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

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

### Mixed Mode Operations

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

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

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

## Saturated Arithmetic

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

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

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

## SIMD Support

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

This section contains:

- [“Using SIMD Mode with Multichannel Data” on page 1-101](#)
- [“Using SIMD Mode with Single-Channel Data” on page 1-102](#)
- [“Restrictions to Using SIMD” on page 1-103](#)
- [“SIMD\\_for Pragma Syntax” on page 1-105](#)
- [“Compiler Constraints on Using SIMD C/C++” on page 1-106](#)
- [“Impact of Anomaly #40 on SIMD” on page 1-107](#)

- [“Examples Using SIMD C” on page 1-110](#)
- [“Performance When Using SIMD C/C++” on page 1-108](#)

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

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

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

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

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

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

## Using SIMD Mode with Multichannel Data

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

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

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

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



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

### Using SIMD Mode with Single-Channel Data

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

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

$$c[j] = a[j] + b[j] \quad \text{for each } j.$$

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

$$c[j] = a[j] + b[j] \quad \text{in one processing element}$$
$$c[j+1] = a[j+1] + b[j+1] \quad \text{in the other element}$$


Note that the loop index now increments by 2.

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

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

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

Note that a reduction loop has to compute a single result. To use SIMD processing, the SISD algorithm would be transformed slightly. Two partial results are accumulated with all the even elements contributing to one



result and the odds to the other. At the end of the loop, the program must combine the partial results into a final one. The reduction approach has two effects for which you must account:

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

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



The compiler provides extended C/C++ support for SIMD mode and single channel data.

## Restrictions to Using SIMD

Be aware that there are various implications when a program is transformed to process single-channel data in SIMD mode.

- The data and the access pattern must be arranged for SIMD fetches, which are always at immediately adjoining locations. For example, a program that sums every third element of an array or the first column of a multi-dimensional array would not be a good candidate for SIMD, because the second fetch does not pick up the element for the next iteration.
- The data must always be aligned on double-word boundaries when in SIMD mode. The compiler attempts to align arrays properly in memory, so that operations on whole arrays work. Under certain circumstances, it is possible that some arrays which are placed in named sections may be allocated on an odd word boundary, and may therefore be accessed incorrectly in SIMD mode. These circumstances are usually associated with use of the `RESOLVE` directive

in an `.LDF` file or with data elimination by the linker. Therefore, it is suggested that the declaration of any data that is placed in a named section and could be accessed in SIMD mode is preceded by a `#pragma align 2` directive.

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

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

Some ADSP-2116x architectures only have a 32-bit external bus. Because of a shorter bus length, the effect of SIMD accesses to external memory on such architectures will not be the same as SIMD accesses to internal memory. If data is placed in external memory, then the “`-no-simd`” switch (see [on page 1-37](#)) should be used to disable SIMD access to this data.

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

Looking at the example,

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

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

## SIMD\_for Pragma Syntax

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

```
#pragma SIMD_for
```

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

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

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

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

The compiler transforms this loop as:

```
    // declare SIMD temporaries
float t_sum[2], t_c[2];
    // initialize both partial sums
t_sum[0] = t_sum[1] = 0;
    // initialize both parts of scalar constant
t_c[0] = t_c[1] = c;
```

## C/C++ Compiler Language Extensions

```
// ENTER_SIMD_MODE -- set machine mode
for (j=0; j<N; j+=2) {
    t_sum[0] += t_c[0] * A[j];
    // -- implicit SIMD processing performs:
    // t_sum[1] += t_c[1] * A[j+1];
}
// LEAVE_SIMD_MODE
// combine partial sums
sum = t_sum[0] + t_sum[1];
```

### Compiler Constraints on Using SIMD C/C++

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

The compiler usually avoids changing a program when the compiler is not certain that the transformed program produces the same results as the original.

The SIMD transformations are handled a bit differently for two reasons.

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

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

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

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

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

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

## Impact of Anomaly #40 on SIMD

The SIMD read from internal memory with a Shadow Write FIFO hit does not always function correctly. This anomaly has been identified in the Shadow Write FIFOs that exist between the internal memory array of the ADSP-21160M and core /IOP busses that access the memory. (See *ADSP-21160 SHARC DSP Hardware Reference* for more details on shadow register operation).

If performing SIMD reads which cross Long Word Address boundaries (for example, odd Normal Word addresses or non-Long Word boundary aligned Short Word addresses) and the data for the read is in the Shadow Write FIFO, the read will result in revision 0.0 behavior for the read.

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

- Allocates all static arrays on an appropriate double-word boundary
- Ensures that arrays on the stack are double-word aligned by ensuring the stack is aligned on an even word boundary.
- Generates SIMD operations when it knows it is operating with double-word-aligned elements. It will be able to do this when arrays are defined statically or locally, or the arrays are arguments

whose properties can be determined by Inter Procedural Analysis. A further requirement is that the initial index and increment are both explicit.

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

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

### Performance When Using SIMD C/C++

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

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

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

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

Specific items include:

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

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

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

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

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

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

## Examples Using SIMD C

The following are two examples on how to use SIMD.

### Using SIMD C (Problem Cases—Data Increments)

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

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

The following matrix multiplication function demonstrates some stride issues.

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

```
float *matmul(void *x_input,
              void *y_input,
              void *output,
              int r,
              int s,
              int t) {
    float *ipx, *ipy, *output_new;
    float tmp = 0;
    int i = 0, j = 0, k = 0;
    ipx = (float *) x_input;
    ipy = (float *) y_input;
    output_new = (float *) output;
    for (i = 0; i < r; i++)
        for (k = 0; k < t; k++) {
            tmp = 0;
            #pragma SIMD_for
```



```

    for (j = 0; j < s; j++)
    // The next two lines are wrong in SIMD mode
    tmp += ipx[j + (i * s)] * ipy[k + (j * t)];
    output_new[k + (i * r)] = tmp;
}
printf("SIMD\n");
return output_new;
}

```

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

The SIMD and non-SIMD versions of this code address the following memory locations on each iteration of the innermost loop:

| non-SIMD                    | SIMD                                      |
|-----------------------------|-------------------------------------------|
| <code>ipy[k]</code>         | <code>ipy[k], ipy[k+1]</code>             |
| <code>ipy[k + t]</code>     | <code>ipy[k+(2*t)], ipy[k+(2*t)+1]</code> |
| <code>ipy[k + (2*t)]</code> | <code>ipy[k+(4*t)], ipy[k+(4*t)+1]</code> |
| <code>ipy[k + (3*t)]</code> | <code>ipy[k+(6*t)], ipy[k+(6*t)+1]</code> |
| <code>ipy[k + (4*t)]</code> | <code>ipy[k+(8*t)], ipy[k+(8*t)+1]</code> |

Each version addresses different memory locations, giving different results.

### SIMD C Loop Counter Rules:

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

### Using SIMD C (Problem Cases—Data Alignment)

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

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

#### Using Two-Dimensional Arrays

The following array,

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

would be double-word aligned by the compiler.

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

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

#### SIMD C/C++ Data Alignment Rule:

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

#### Adding To Array Offsets

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

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

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

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

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

# Preprocessor Features

The compiler includes a preprocessor that lets you use preprocessor commands within your C or C++ source. [Table 1-11](#) lists these commands and provides a brief description of each. The preprocessor automatically runs before the compiler.

Table 1-11. Preprocessor Commands

| Command               | Description                                                                       |
|-----------------------|-----------------------------------------------------------------------------------|
| <code>#define</code>  | Defines a macro or constant.                                                      |
| <code>#elif</code>    | Sub-divides an <code>#if ... #endif</code> pair.                                  |
| <code>#else</code>    | Identifies alternative instructions within an <code>#if ... #endif</code> pair.   |
| <code>#endif</code>   | Ends an <code>#if ... #endif</code> pair.                                         |
| <code>#error</code>   | Reports an error message.                                                         |
| <code>#if</code>      | Begins an <code>#if ... #endif</code> pair.                                       |
| <code>#ifdef</code>   | Begins an <code>#ifdef ... #endif</code> pair and tests if macro is defined.      |
| <code>#ifndef</code>  | Begins an <code>#ifndef ... #endif</code> pair and tests if macro is not defined. |
| <code>#include</code> | Includes source code from another file.                                           |
| <code>#line</code>    | Outputs specified line number before preprocessing.                               |
| <code>#undef</code>   | Removes macro definition.                                                         |
| <code>#warning</code> | Reports a warning message.                                                        |
| <code>#</code>        | Converts a macro argument into a string constant.                                 |
| <code>##</code>       | Concatenates two strings.                                                         |

Preprocessor commands are also useful for modifying the compilation. Using the `#include` command, you can include header files (`.h`) that contain code and/or data. A macro, which you declare with the `#define` preprocessor command, can specify simple text substitutions or complex

substitutions with parameters. The preprocessor replaces each occurrence of the macro reference found throughout the program with the specified value.

The preprocessor is separate from the compiler and has some features that may not be used within your C or C++ source file. For more information, see the *VisualDSP++ 3.0 Assembler and Preprocessor Manual for SHARC DSPs*.

## Predefined Macros

The predefined macros that `cc21k` provides are listed below.

### **\_\_2106x\_\_**

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

### **\_\_2116x\_\_**

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

### **\_\_ADSP21000\_\_**

`cc21k` always defines `__ADSP21000__` as 1.

### **ADSP21000**

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

### **\_\_ADSP21020\_\_**

`cc21k` defines `__ADSP21020__` as 1 when you compile with the `-21020` command-line switch.

### ADSP21020

`cc21k` defines `ADSP21020` as 1 when you compile with the `-21020` command-line switch, but the compiler undefines this macro if you compile with `-pedantic` or `-pedantic-errors`.

### `__ADSP21060__`

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

### ADSP21060

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

### `__ADSP21061__`

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

### ADSP21061

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

### `__ADSP21062__`

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

## ADSP21062

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

## \_\_ADSP21065L\_\_

cc21k defines \_\_ADSP21065L\_\_ as 1 when you compile with the -21065L command-line switch.

## ADSP21065L

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

## \_\_ADSP21160\_\_

cc21k defines \_\_ADSP21160\_\_ as 1 when you compile with the -21160 command-line switch.

## ADSP21160

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

## \_\_ADSP21161\_\_

cc21k defines \_\_ADSP21161\_\_ as 1 when you compile with the -21161 command-line switch.

### ADSP21161

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

### `__ANALOG_EXTENSIONS__`

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

### `__cplusplus`

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

### `__DATE__`

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

### `__DOUBLES_ARE_FLOATS__`

cc21k defines `__DOUBLES_ARE_FLOATS__` as 1 when you compile with the `-double-size-32` command-line switch ([on page 1-27](#)). Note that `-double-size-32` is the default switch. The compiler undefines this macro if the `-double-size-64` switch is used.

### `__ECC__`

cc21k always defines `__ECC__` as 1.

### `__EDG__`

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



## **\_\_EDG\_VERSION\_\_**

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

## **\_\_FILE\_\_**

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

## **\_\_LINE\_\_**

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

## **\_\_NO\_LONGLONG**

cc21k always defines `__NO_LONGLONG` as 1.

## **\_\_NO\_BUILTIN**

cc21k defines `__NO_BUILTIN` as 1 when you compile with the `-no-builtin` command-line switch ([on page 1-35](#)).

## **\_\_SIGNED\_CHARS\_\_**

cc21k defines `__SIGNED_CHARS__` as 1. The macro is defined by default,

## **\_\_STDC\_\_**

cc21k always defines `__STDC__` as 1, but the compiler undefines this macro if you compile with `-traditional` (ANSI standard) ([on page 1-20](#)).

## Preprocessor Features

### `__STDC_VERSION__`

cc21k always defines `__STDC_VERSION__` as 199409L, but the compiler undefines this macro if you compile with `-traditional` (ANSI standard) (on page 1-20).

### `__TIME__`

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

### `__VERSION__`

The preprocessor expands this macro into a string constant containing the current compiler version.

## Header Files

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

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

## Writing Macros

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

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

The following two code segments define two versions of the macro

`SKIP_SPACES:`

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

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

## Preprocessor Features

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

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

This expands to:

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

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

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

For more information on macros, see the *VisualDSP++ 3.0 Assembler and Preprocessor Manual for SHARC DSPs*.

## C/C++ Run-Time Model and Environment

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

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

This section describes:

- [“C/C++ Run-Time Environment”](#)
- [“Using Multiple Heaps”](#)
- [“Compiler Registers”](#)

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

## C/C++ Run-Time Environment

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

[Figure 1-2 on page 1-124](#) shows an overview of the run-time environment issues that you must consider as you write assembly routines that link with C/C++ routines. These issues include::

## C/C++ Run-Time Model and Environment

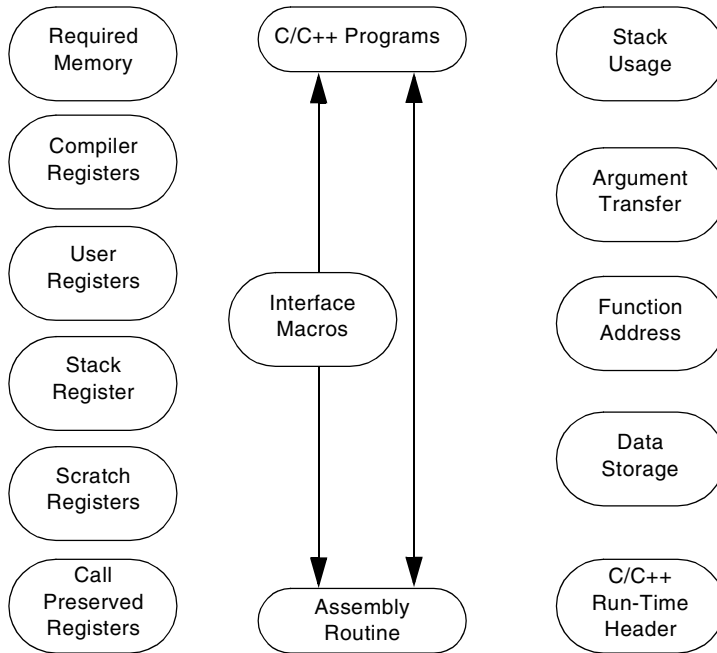


Figure 1-2. Assembly Language Interfacing Overview

- Register usage conventions (see the following sections)
  - “Compiler Registers” on page 1-137
  - “Miscellaneous Information about Registers” on page 1-138
  - “Call Preserved Registers” on page 1-139
  - “Scratch Registers” on page 1-140
  - “Stack Registers” on page 1-141

- Memory usage conventions (see the following sections)
  - [“Memory Usage” on page 1-125](#)
  - [“Using Data Storage Formats” on page 1-150](#)
- Program control conventions (see the following sections)
  - [“Managing the Stack” on page 1-142](#)
  - [“Transferring Function Arguments and Return Value” on page 1-147](#)
  - [“Using Data Storage Formats” on page 1-150](#)

## Memory Usage

The cc21k C/C++ run-time environment requires that a specific set of memory section names be used for placing code in memory. In assembly language files, these names are used as labels for the `.SECTION` directive. In the linker description file, these names are used as labels for the output section names within the `SECTIONS{ }` command.

For information on syntax for the Linker Description File and other information on the linker, see the *VisualDSP++ 3.0 Linker and Utilities Manual for SHARC DSPs*. [Table 1-12 on page 1-126](#) lists the memory section and output section names.

Because the compiler and linker must know the processor type to create code for the correct memory model, you must specify the processor for which you are developing. If you are using the VisualDSP++ IDDE, you specify the processor in the Project Options dialog box. If you are running the compiler from the command line, you specify the processor with a compiler switch. For more information on processor selection switches, see [“C/C++ Compiler Common Switch Descriptions” on page 1-21](#).

Table 1-12. Memory .SECTION and SECTION{} Names

| Names    | Usage Description                                                                                                                                                                                                                                      |
|----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| seg_pmco | This section must be in Program Memory, holds code, and is required by some functions in the C/C++ run-time library. For more information, see “Program Memory Code Storage” on page 1-126.                                                            |
| seg_dmda | This section must be in Data Memory, is the default location for global and static variables and string literals, and is required by some functions in the C/C++ run-time library. For more information, see “Data Memory Data Storage” on page 1-126. |
| seg_pmda | This section must be in PM, holds PM data variables, and is required by some functions in the C/C++ run-time library. For more information, see “Program Memory Data Storage” on page 1-127.                                                           |
| seg_stak | This section must be in DM, holds the run-time stack, and is required by the C/C++ run-time environment. For more information, see “Run-Time Stack Storage” on page 1-127.                                                                             |
| seg_heap | This section must be in DM, holds the default run-time heap, and is required by the C/C++ run-time environment. For more information, see “Run-Time Heap Storage” on page 1-128.                                                                       |
| seg_init | This section must be in PM, holds system initialization data, and is required for system initialization. For more information, see “Initialization Data Storage” on page 1-128.                                                                        |
| seg_rth  | This section must be in the interrupt table area of PM, holds system initialization code and interrupt service routines, and is required for system initialization. For more information, see “Run-Time Header Storage” on page 1-129.                 |

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

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



By default, the compiler stores static variables in the Data Memory data section. The compiler's `dm` and `pm` keywords (memory type qualifiers) let you override this default. If a memory type qualifier is not specified, the compiler places static and global variables in Data Memory. For more information on type qualifiers, see [“Dual Memory Support Keywords \(pm dm\)” on page 1-74](#). The following example allocates an array of 10 integers in the Data Memory data section.

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

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

By default, the compiler stores static variables in the Data Memory data section. The compiler's `pm` keyword (memory type qualifier) lets you override this default and place variables in the Program Memory data section. If a memory type qualifier is not specified, the compiler places static and global variables in Data Memory. For more information on type qualifiers, see [“Dual Memory Support Keywords \(pm dm\)” on page 1-74](#). The following example allocates an array of 10 integers in the Program Memory data section.

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

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

The run-time stack is a 32-bit wide structure, growing from high memory to low memory. The compiler uses the run-time stack as the storage area for local variables and return addresses.

During a function call, the calling function pushes the return address onto the stack. [For more information, see “Managing the Stack” on page 1-142.](#)

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

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

The run-time library also provides support for multiple heaps, which allow dynamically allocated memory to be located in different blocks. See [“Using Multiple Heaps” on page 1-130](#) for more information on the use of multiple heaps.



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

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

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

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

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

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

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

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

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

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

## C/C++ Run-Time Model and Environment

If you do not specify a run-time header file, the compiler uses a default run-time header from the `21k\lib` or `211xx\lib` directory. For ADSP-21020 DSPs, use `020_hdr.doj`; for ADSP-21060 and ADSP-21062 DSPs, use `060_hdr.doj`; for ADSP-21061 and ADSP-21065L DSPs, use `065_hdr.doj`; for ADSP-21260 DSPs, use `160_hdr.doj`; and for ADSP-21161 DSPs, use `161_hdr.doj`.

Note that if the compiler finds a copy of `xxx_hdr.obj` in the current directory, the compiler uses the copy instead of the default from the `21k\lib` or `211xx\lib` directory.

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

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

## Using Multiple Heaps

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

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

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

## Declaring a Heap

Each heap must be declared with a `.VAR` directive in the `seg_init.asm` file and the `.LDF` file must declare memory and section placement for the heaps. The default `seg_init.asm` file declares one heap, `seg_heap`. The following customized `seg_init.asm` file shows how to declare two heaps: `seg_heap` and `seg_heaq`.

To use a custom `seg_init.asm`, assemble it and use it to replace the default `seg_init.doj` that is a member of the `libc.dlb` archive. For information on how to modify an archive file, see the *VisualDSP++ 3.0 Linker and Utilities Manual for SHARC DSPs*.

For example,

```
.segment/pm seg_init;

.extern ldf_heap_space;    /* The base of a primary DM heap
"seg_heap" */
.extern ldf_heap_length;   /* Length of heap "seg_heap" */
.extern ldf_heaq_space;    /* Base of a primary DM heap
"seg_heaq" */
.extern ldf_heaq_length;   /* Length of heap "seg_heaq" */

/* The first two 48-bit words represent the heap name and the
heap location */
/*  FFFFFFFF DM location    00000001 PM location */
/* The heap name must be exactly 8 characters long */
/* The next 3 words set the heap's initialization value, size and
length. */
/* The size and length are set with macros whose values are cal-
culated according the information in the project's .ldf file. */
```

## C/C++ Run-Time Model and Environment

```
.global ____lib_heap_space;
.var    ____lib_heap_space[5] =
        0x7365675F6865, /* 'seg_he' */
        0x6170FFFFFFFF, /* 'ap'      */
        0,
        ldf_heap_space,
        ldf_heap_length;
.____lib_heap_space.end:

/* Add more heap descriptions here */
.global ____lib_heaq_space;
.var    ____lib_heaq_space[5] =
        0x7365675F6865, /* 'seg_he' */
        0x6171FFFFFFFF, /* 'aq'      */
        0,
        ldf_heaq_space,
        ldf_heaq_length;
.____lib_heaq_space.end:

.var    ____lib_end_of_heap_descriptions = 0;
        /* Zero for end of list */

.____lib_end_of_heap_descriptions.end:

.endseg;
```

As noted above, the calculation for a heap's size and length occur in the project's Linker Description File. When linking, the linker handles substitution of values to resolve the heap's definition (the `.VAR` directive in the `seg_init.asm` file). The following LDF supports the customized `seg_init.asm` file from the previous example.

```
ARCHITECTURE(ADSP-21062)
SEARCH_DIR( $ADI_DSP\21k\lib )
$LIBRARIES = lib060.dlb, libc.dlb ;
$OBJECTS = $COMMAND_LINE_OBJECTS, adi_dsp\21k\lib\060_hdr.obj, seg_init.doj;

MEMORY {
    seg_rth {TYPE(PM RAM) START(0x20000) END(0x20fff) WIDTH(48)}
    seg_init{TYPE(PM RAM) START(0x21000) END(0x2100f) WIDTH(48)}
```

```

seg_pmco{TYPE(PM RAM) START(0x21010) END(0x24fff) WIDTH(48)}
seg_pmda{TYPE(DM RAM) START(0x28000) END(0x28fff) WIDTH(32)}
seg_dmda{TYPE(DM RAM) START(0x29000) END(0x29fff) WIDTH(32)}
seg_stak{TYPE(DM RAM) START(0x2e000) END(0x2ffff) WIDTH(32)}
/* memory declarations for default heap */
seg_heap{TYPE(DM RAM) START(0x2a000) END(0x2bfff) WIDTH(32)}
/* memory declarations for custom heap */
seg_heaq{TYPE(DM RAM) START(0x2c000) END(0x2dfff) WIDTH(32)}
} // End MEMORY

PROCESSOR p0 {
  LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST)
  OUTPUT( $COMMAND_LINE_OUTPUT_FILE )

  SECTIONS {
    .seg_rth {
      INPUT_SECTIONS( $OBJECTS(seg_rth) $LIBRARIES(seg_rth))
    } > seg_rth
    .seg_init {
      INPUT_SECTIONS( $OBJECTS(seg_init) $LIBRARIES(seg_init))
    } > seg_init
    .seg_pmco {
      INPUT_SECTIONS( $OBJECTS(seg_pmco) $LIBRARIES(seg_pmco))
    } > seg_pmco
    .seg_pmda {
      INPUT_SECTIONS( $OBJECTS(seg_pmda) $LIBRARIES(seg_pmda))
    } > seg_pmda
    .seg_dmda {
      INPUT_SECTIONS( $OBJECTS(seg_dmda) $LIBRARIES(seg_dmda))
    } > seg_dmda
    .stackseg {
      ldf_stack_space = .;
      ldf_stack_length = 0x2000;
    } > seg_stak

    /* section placement for default heap */
    .heap {
      ldf_heap_space = .;
      ldf_heap_end = ldf_heap_space + 0x2000;
      ldf_heap_length = ldf_heap_end - ldf_heap_space;
    }
  }
}

```

## C/C++ Run-Time Model and Environment

```
    } > seg_heap

/* section placement for additional custom heap */
.heap {
    ldf_heap_space = .;
    ldf_heap_end = ldf_heap_space + 0x2000;
    ldf_heap_length = ldf_heap_end - ldf_heap_space;
} > seg_heap
} // End SECTIONS
} // End P0
```

### Heap Identifiers

All heaps have two identifiers:

- Primary heap ID is the index of the descriptor for that heap in the heap descriptor table (in `seg_init.asm`). The primary heap ID of the default heap is always 0, and the primary IDs of user defined heaps will be 1, 2, 3, and so on.
- Each heap also has a unique 8-letter name associated with it. The heap ID can be obtained by calling the function `heap_lookup_name` with this name as its parameter.

### Using Alternate Heaps with the Standard Interface

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

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



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



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

## Using the Alternate Heap Interface

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

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

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

## Example C Programs

The C programs below show how to allocate and initialize heaps.

### Standard Heap Interface

```
// Example program using the standard heap interface
// Assumes that the user has created an additional heap, "seg_heap",
// which is located in PM memory

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

void func(int * a, int pm * b);

main()
{
    int * x;
    int pm * y;
    int loop;

    x = malloc(1000);           // get 1K words of DM heap space
    set_alloc_type("seg_heap"); // Set the current heap to "seg_heap"
    y = (int pm *)malloc(1000); // get 1K words of PM heap space
    set_alloc_type("seg_heap"); // Reset the current heap to
                                // "seg_heap" in case it is referred
                                // to elsewhere
    for (loop = 0; loop < 1000; loop++)
        x[loop] = y[loop] = loop;
    func(x, y); // Do something with x and y
}
```

### Alternate Heap Interface

```
// Example function using the alternate heap interface
// Assumes that the user has created an additional heap, "seg_heap",
// which is located in PM memory
#include <stdlib.h>

void func(int * a, int pm * b);
```

```
main()

{
    int * x;
    int pm * y;
    int loop, pm_heapID;
    pm_heapID = heap_lookup_name("seg_heap");
    x = heap_malloc(0, 1000); // get 1K words of DM heap space
    y = (int pm *)heap_malloc(pm_heapID, 1000);
                                // get 1K words of PM heap space
    for (loop = 0; loop < 1000; loop++)
        x[loop] = y[loop] = loop;
    func(x, y); // Do something with x and y
}
```

## Compiler Registers

The cc21k C/C++ run-time environment reserves a set of registers for its own use. [Table 1-13](#) lists these registers and the values the C/C++ run-time environment expects to be in them. Do not modify these registers, except as noted in the table.

Table 1-13. Compiler Registers

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

### Miscellaneous Information about Registers

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

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

### User Registers

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

You must reserve the same list of registers in all linked files; the whole project must use the same `-reserve` option. [Table 1-14](#) lists these registers. Note that the C run-time library does not use these registers.

Table 1-14. User Registers

| Register                                                                                                        | Value        | Modification Rule                                                                                 |
|-----------------------------------------------------------------------------------------------------------------|--------------|---------------------------------------------------------------------------------------------------|
| i0, b0, l0, m0,<br>i1, b1, l1, m1,<br>i8, b8, l8, m8,<br>i9, b9, l9, m9,<br>mrb, ustat1, ustat2, ustat3, ustat4 | user defined | If not reserved, modify for temporary use, restore when done<br>If reserved, usage is not limited |



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

## Call Preserved Registers

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

Table 1-15 lists these registers.

Table 1-15. Call Preserved Registers<sup>1</sup>

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

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

## C/C++ Run-Time Model and Environment

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

The C/C++ run-time environment:

- Uses default bit order for `DAG` operations (no bit reversal)
- Uses the primary register set (not background set)
- Uses `.PRECISION=32` (32-bit floating-point) and `.ROUND_NEAREST` (round-to-nearest value)
- Disables ALU saturation (`mode1` register, `ALUSAT` bit = 0)
- Uses default `FIX` instruction rounding to nearest (`MODE1` register, `TRUNCATE=0`)
- Enables circular buffering on ADSP-2116x DSPs by setting `CBUFEN` on `MODE1`

### Scratch Registers

The cc21k C/C++ run-time environment specifies a set of registers whose contents do not need to be saved and restored. Note that the contents of these registers are not preserved across function calls. [Table 1-16](#) lists these registers.

Table 1-16. Scratch Registers

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

In addition, for ADSP-2116x DSPs, the `PEy` data registers are all scratch registers. [Table 1-17 on page 1-141](#) lists these registers.

Table 1-17. Additional ADSP-2116x DSP Scratch Registers

|        |      |     |     |     |     |     |        |        |
|--------|------|-----|-----|-----|-----|-----|--------|--------|
| s0     | s1   | s2  | s3  | s4  | s5  | s6  | s7     | s8     |
| s9     | s10  | s11 | s12 | s13 | s14 | s15 | USTAT3 | USTAT4 |
| ASTATy | STKy |     |     |     |     |     |        |        |

## Stack Registers

The cc21k C/C++ run-time environment reserves a set of registers for controlling the run-time stack. These registers may be modified for stack management, but they must be saved and restored. [Table 1-18](#) lists these registers.

Table 1-18. Pointer Registers

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

## Alternate Registers

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

The C/C++ run-time model uses register I6 as the frame pointer and register I7 as the stack pointer. Setting the DAG register that contains I6 and I7 from a background register to an active register directly affects the stack operation. The C/C++ run-time model does not have an understanding of background registers.

If the background I6 and I7 registers are active and an interrupt occurs, the C/C++ run-time model still uses I6 and I7 to update the stack. This results in faulty stack handling.



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

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

### Managing the Stack

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

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

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

[Figure 1-3 on page 1-143](#) shows an example section of a run-time stack. In the figure, the currently executing routine, `Current()`, was called by `Previous()`, and `Current()` in turn calls `Next()`. The state of the stack is as if `Current()` has pushed all the arguments for `Next()` onto the stack and is just about to call `Next()`.



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



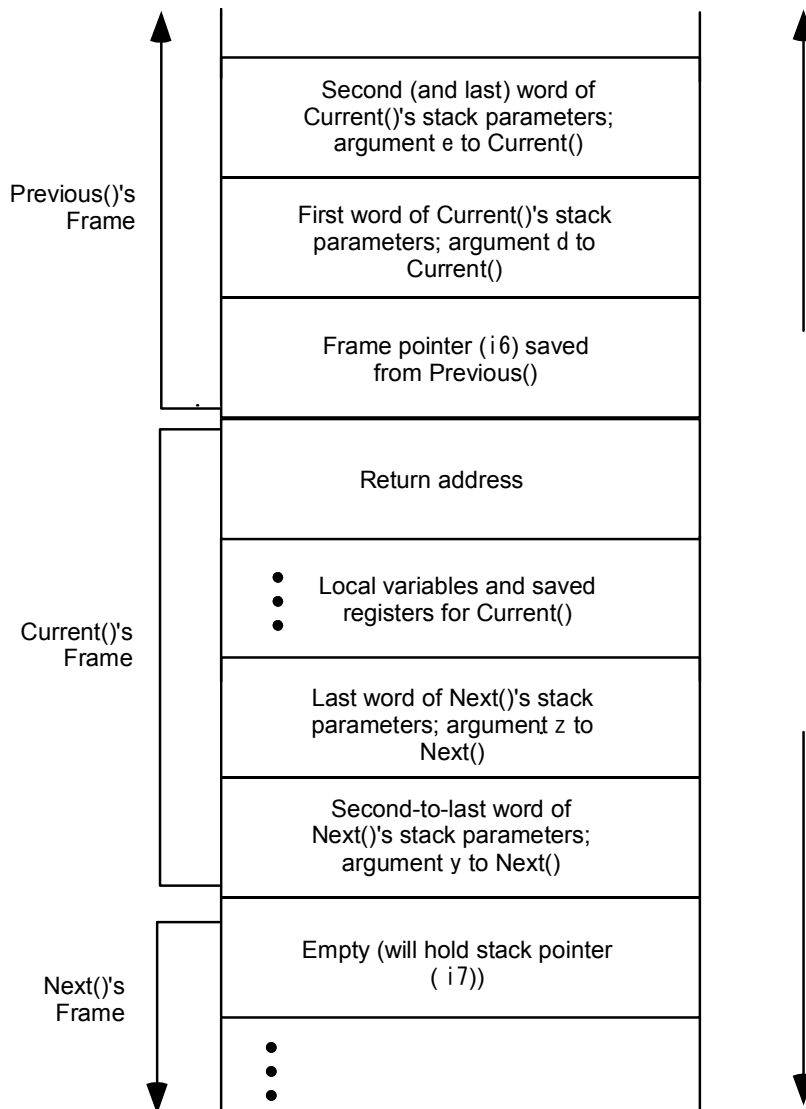


Figure 1-3. Example Run-Time Stack

The prototypes for the functions in [Figure 1-3](#) are as follows:

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

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

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

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

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

As you write assembly routines, note that the operations to create a stack frame are the responsibility of the called function, and you can use the `entry` or `leaf_entry` macros to perform these operations. For more information on these macros, see [“Using Mixed C/C++ and Assembly Support Macros” on page 1-159](#).

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

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

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

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

As you write assembly routines, note that the operations to restore stack and frame pointers are the responsibility of the called function, and you can use the `exit` or `leaf_exit` macros to perform these operations. For more information on these macros, see [“Using Mixed C/C++ and Assembly Support Macros” on page 1-159](#).

In the following code examples ([Listing 1-1](#) and [Listing 1-2](#)), observe how the function calls in the C code translate to stack management tasks in the compiled (assembly) version of the code. The comments have been added to the compiled code to indicate the function prologue and function epilogue.

### Listing 1-1. Stack Management, Example C Code

```
/* Stack management - C code */

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

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

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

### Listing 1-2. Stack Management, Example ADSP-2106x Assembly Code

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

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

    dm(_return_c)=r0;
```

```

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

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

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

```

The next two sections, [“Transferring Function Arguments and Return Value” on page 1-147](#) and [“Using Macros to Manage the Stack” on page 1-173](#), provide additional detail on function call requirements.

## Transferring Function Arguments and Return Value

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

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

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

## C/C++ Run-Time Model and Environment

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

Table 1-19 lists the rules that cc21k uses for passing parameters in registers to functions and the rules that your assembly code must use for returns.

Table 1-19. Parameter and Return Value Transfer Registers

| Register | Parameter Type Passed Or Returned                                                                                                          |
|----------|--------------------------------------------------------------------------------------------------------------------------------------------|
| r4       | Pass first 32-bit data type parameter                                                                                                      |
| r8       | Pass second 32-bit data type parameter                                                                                                     |
| r12      | Pass third 32-bit data type parameter                                                                                                      |
| stack    | Pass fourth and remaining parameters; see exceptions to this rule on page 1-147                                                            |
| r0       | Return int, long, char, float, short, pointer, and one-word structure parameters                                                           |
| r0, r1   | Return long double and two-word structure parameters. Place MSW in r0 and LSW in r1                                                        |
| r1       | Return the address of results that are longer than two words; r1 contains the first location in the block of memory containing the results |

Consider the following function prototype example.

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

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

This next example illustrates the effects of passing doubles.

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

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

The following example illustrates the effects of variable arguments on parameter passing.

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

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

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

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

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

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

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

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

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

## C/C++ Run-Time Model and Environment

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

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

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

### Using Data Storage Formats

The C/C++ run-time environment uses the data formats that appear in the [Table 1-20](#), [Table 1-21 on page 1-151](#), [Figure 1-4 on page 1-151](#), and [Figure 1-5 on page 1-152](#).

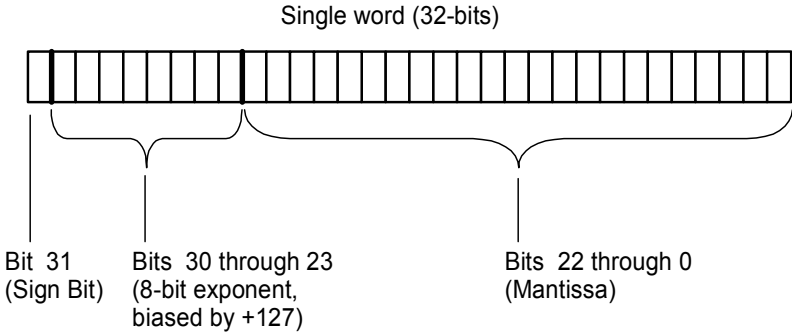
Table 1-20. Data Storage Formats and Data Type Sizes

| Applied Type      | Number Representation                                                                                             |
|-------------------|-------------------------------------------------------------------------------------------------------------------|
| int               | 32-bit two's complement                                                                                           |
| long int          | 32-bit two's complement                                                                                           |
| short int         | 32-bit two's complement                                                                                           |
| unsigned int      | 32-bit unsigned magnitude                                                                                         |
| unsigned long int | 32-bit unsigned magnitude                                                                                         |
| char              | 32-bit two's complement                                                                                           |
| unsigned char     | 32-bit unsigned magnitude                                                                                         |
| float             | 32-bit IEEE single-precision                                                                                      |
| double            | 32-bit IEEE single-precision<br>or 64-bit IEEE double-precision if you compile with the<br>-double-size-64 switch |
| long double       | 64-bit IEEE double-precision                                                                                      |



Table 1-21. Data Storage Formats and Data Storage

| Data        | Big Endian Storage Format                                                                                                                                                                                     |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| long double | Writes 64-bit IEEE double-precision data with the most significant word closer to address 0x0000, proceeds toward the top of memory with the rest (see <a href="#">Figure 1-5 on page 1-152</a> for details). |



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

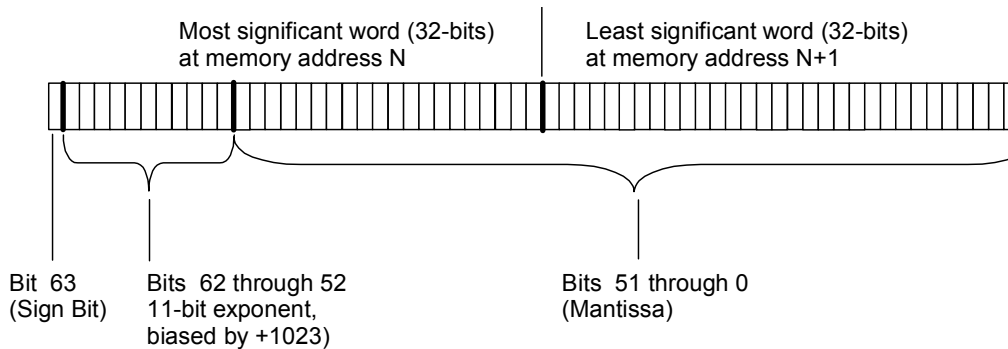
$$-1^{\text{Sign}} \times 1.\text{Mantissa} \times 2^{(\text{Exponent} - 127)}$$

Where:

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

Figure 1-4. Floating-Point (32-Bit IEEE Single-Precision) Storage

## C/C++ Run-Time Model and Environment



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

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

Where:

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

Figure 1-5. Floating-Point (64-Bit IEEE Double-Precision) Storage

### Using the Run-Time Header

The run-time header is an assembly language procedure that initializes the processor and sets up processor features to support the C/C++ run-time environment. The source code for the default run-time header is in the `020_hdr.asm` for ADSP-21020 DSPs, `060_hdr.asm` for ADSP-21060 and

ADSP-21062 DSPs, `065L_hdr.asm` for ADSP-21061 and ADSP-21065L DSPs, `160_hdr.asm` for ADSP-21160 DSPs, and `161_hdr.asm` for ADSP-21161 DSPs.

This run-time header performs the following operations:

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

# C/C++ and Assembly Interface

This section describes how to call assembly language subroutines from within C or C++ programs and how to call C or C++ functions from within assembly language programs.

 Before attempting to do either of these calls, be sure to familiarize yourself with the information about the C/C++ run-time model (including details about the stack, data types, and how arguments are handled) in [“C/C++ Run-Time Environment” on page 1-123](#). At the end of this reference, a series of examples demonstrate how to mix C/C++ and assembly code.

## Calling Assembly Subroutines from C/C++ Programs

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

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

The run-time model defines some registers as *scratch* registers and others as *preserved* or *dedicated* registers. The scratch registers can be used within the assembly language program without worrying about their previous contents. If you need more room (or are working with existing code) and

wish to use the preserved registers, you must first save their contents and then restore those contents before returning. Do not use the dedicated registers for other than their intended purpose; the compiler, libraries, debugger, and interrupt routines all depend on having a stack available as defined by those registers.

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

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

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

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

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

int global_a;
float global_b;
int * global_p;

// the function itself:

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

## C/C++ and Assembly Interface

```
global_b = b;
global_p = p;

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

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

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

## Calling C/C++ Functions from Assembly Programs

You may want to call C or C++-callable library and other functions from within an assembly language program. As discussed in [“Calling Assembly Subroutines from C/C++ Programs” on page 1-154](#), you may wish to create a test function to do this in C or C++, and then use the code generated by the compiler as a reference when creating your assembly language program and the argument setup. Using volatile global variables may help clarify the essential code in your test function.

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

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

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

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

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

## C/C++ and Assembly Interface

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

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

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

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

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

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

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

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

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

    rts;
```



## Using Mixed C/C++ and Assembly Support Macros

This section lists C/C++ and assembly interface support macros in the `asm_sprt.h` system header file. Use these macros for interfacing assembly language modules with C or C++ functions. [Table 1-22](#) lists the macros.



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

Table 1-22. Interface Support Macro Summary

|                       |                       |                          |                        |
|-----------------------|-----------------------|--------------------------|------------------------|
| <code>entry</code>    | <code>exit</code>     | <code>leaf_entry</code>  | <code>leaf_exit</code> |
| <code>ccall(x)</code> | <code>reads(x)</code> | <code>puts</code>        | <code>gets(x)</code>   |
| <code>alter(x)</code> | <code>save_reg</code> | <code>restore_reg</code> |                        |

## Interface Support Macros, Defined

This section describes and shows syntax for the C/C++ and assembly interface support macros.

### **entry**

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

### **exit**

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

### **leaf\_entry**

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

### **leaf\_exit**

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

### **ccall(x)**

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

### **reads(x)**

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

### **puts=x**

The `puts` macro expands into an instruction that pushes the value in register `x` onto the stack.

### **gets(x)**

The `gets` macro expands into an instruction that pops a value off of the stack and puts the value in the indicated register:

```
register = gets(x);
```

The value is located at an offset `x` from the stack pointer.

**alter(x)**

The `alter` macro expands into an instruction that adjusts the stack pointer by adding the immediate value `x`. With a positive value for `x`, `alter` pops `x` words from the top of the stack. You could use `alter` to clear `x` number of parameters off the stack after a call.

**save\_reg**

The `save_reg` macro expands into a series of instructions that push the register file registers (`r0-r15`) onto the run-time stack.

**restore\_reg**

The `restore_reg` macro expands into a series of instructions that pop the register file registers (`r0-r15`) off the run-time stack.

The following code example shows the `asm_sprt.h` used for defining C/C++/assembly interface support macros.

```
/* asm_sprt.h - C/C++/Assembly Interface Support Macros */
/* asm_sprt.h - $Date: 10/09/97 6:28p $ */

#ifndef __ASM_SPRT_DEFINED
#define __ASM_SPRT_DEFINED

#define entry                /* nothing */
#define leaf_entry          /* nothing */

#ifdef __ADSP21020__
#define ccall(x) \
    r2=i6; i6=i7; \
    jump (pc, x) (db); \
    dm(i7,m7)=r2; \
    dm(i7,m7)=PC;
#define leaf_exit \
    i12=dm(m7,i6);\
    jump (m14,i12) (db); \
    i7=i6; i6=dm(0,I6);
```

## C/C++ and Assembly Interface

```
#define exit leaf_exit
#else
#define ccall(x) \
    cjump (x) (DB); \
    dm(i7,m7)=r2; \
    dm(i7,m7)=PC;

#define leaf_exit \
    i12=dm(m7,i6); \
    jump (m14,i12) (db); \
    nop; \
    RFRAME
#define exit leaf_exit
#endif

#define reads(x)dm(x, i6)
#define putsdm(i7, m7)
#define gets(x)dm(x, i7)
#define alter(x)modify(i7, x)

#define save_reg \
    puts=r0;\
    puts=r1;\
    puts=r2;\
    puts=r3;\
    puts=r4;\
    puts=r5;\
    puts=r6;\
    puts=r7;\
    puts=r8;\
    puts=r9;\
    puts=r10;\
    puts=r11;\
    puts=r12;\
    puts=r13;\
    puts=r14;\
    puts=r15;

#define restore_reg \
    r15=gets(1);\
```

```

    r14=gets(2);\
    r13=gets(3);\
    r12=gets(4);\
    r11=gets(5);\
    r10=gets(6);\
    r9 =gets(7);\
    r8 =gets(8);\
    r7 =gets(9);\
    r6 =gets(10);\
    r5 =gets(11);\
    r4 =gets(12);\
    r3 =gets(13);\
    r2 =gets(14);\
    r1 =gets(15);\
    r0 =gets(16);\
    alter(16);

#endif

```

## Using Mixed C/C++ and Assembly Naming Conventions

It is necessary to be able to use C or C++ symbols (function or variable names) in assembly routines and use assembly symbols in C or C++ code. This section describes how to name and use C/C++ and assembly symbols.

To name an assembly symbol that corresponds to a C or C++ symbol, add an underscore prefix to the C/C++ symbol name when declaring the symbol in assembly. For example, the C/C++ symbol `main` becomes the assembly symbol `_main`.

To use a C function or variable in an assembly routine, declare it as global in the C program. Import the symbol into the assembly routine by declaring the symbol with the `.EXTERN` assembler directive.

The external name of a C++ function encodes information about its type and parameters. Function “signature” enables the overloading of functions and operators that the C++ language supports. To reference a function in

## C/C++ and Assembly Interface

a C++ module, declare it with the `extern "C"` specifier in order to use the naming convention of C. Note that C++ data symbols use the same convention as C.

To use an assembly function or variable in your C program, declare the symbol with the `.GLOBAL` assembler directive in the assembly routine and import the symbol by declaring the symbol as `extern` in the C program.

To use an assembly function in your C++ module, declare the symbol with the `.GLOBAL` assembler directive in the assembly routine and import the symbol by declaring the symbol as `extern "C"` in the C++ program. For example, to reference the `_funcmult` assembly routine from a C++ program, you declare it as `extern "C" int funcmult(int a, int b)` in the C++ program.

Table 1-23 shows several examples of the C/Assembly interface naming conventions. Each row shows how assembler code can reference the corresponding C item.

Table 1-23. C Naming Conventions For Symbols

| In the C Program                            | In the Assembly Subroutine                                 |
|---------------------------------------------|------------------------------------------------------------|
| <code>int c_var; /*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><br><code>_asm_func:</code> |

Table 1-24 on page 1-165 shows several examples of the C++/Assembly interface naming conventions. Each row shows how assembler code can reference the corresponding C++ item.

Table 1-24. C++ Naming Conventions for Symbols

| In the C++ Program                                         | In the Assembly Subroutine                     |
|------------------------------------------------------------|------------------------------------------------|
| <code>extern "C" { int c_var; } /*declared global*/</code> | <code>.extern _c_var;</code>                   |
| <code>extern "C" void c_func(void){...}</code>             | <code>.extern _c_func;</code>                  |
| <code>extern "C" int asm_var;</code>                       | <code>.global _asm_var;</code>                 |
| <code>extern "C" void asm_func(void);</code>               | <code>.global _asm_func;<br/>_asm_func:</code> |

## Implementing C++ Member Functions in Assembly

If an assembly language implementation is desired for a C++ member function, the simplest way is to use C++ to provide the proper interface between C++ and assembly.

In the class definition, write a simple member function to call the assembly-implemented function (subroutine). This call can establish any interface between C++ and assembly, including passing a pointer to the class instance. Since the call to the assembly subroutine resides in the class definition, the compiler inlines the call (inlining adds no overhead to compiler performance). From an efficiency point of view, the assembly language function is called directly from the user code.

As for any C++ function, ensure that a prototype for the assembly-implemented function is included in your program. As discussed in [“Using Mixed C/C++ and Assembly Naming Conventions” on page 1-163](#), you declare your assembly language subroutine’s name with the `.GLOBAL` directive in the assembly portion and import the symbol by declaring it as `extern "C"` in the C++ portion of the code.

Note that using this method you avoid name mangling—you choose your own identifier for the external function. Access to internal class information can be done either in the C++ portion or in the assembly portion. If the assembly subroutine needs only limited access to the class members, it

## C/C++ and Assembly Interface

is easier to select those in the C++ code and pass them as explicit arguments. This way the assembly code does not need to know how data is allocated within a class instance.

```
#include <stdio.h>
/* Prototype for external assembly routine: */
/* C linkage does not have name mangling */
extern "C" int cc_array(int);

class CC {
private:
    int av;
public:
    CC({});
    CC(int v) : av(v){};
    int a() {return av;};
                                /* Assembly routine call: */
    int array() {return cc_array(av);};
};

int main()
{
    CC samples(11);
    CC points;
    points = CC(22);
    int j, k;
    j = samples.a();
    k = points.array();    // Test asm call

    printf ( "Val is %d\n", j );
    printf ( "Array is %d\n", k );

    return 1;
}
/* In a separate assembly file: */
.section /pm seg_pmco;
.global _cc_array;
_cc_array:
modify(i7,-3);
dm(-4,i6)=r3;
dm(-2,i6)=r4;
r3=r4;
r0=r3+r3;
```



```

r3=dm(-4,i6);
i12=dm(m7,i6);
jump(m14,i12)(DB);
rframe;
nop;

```

## Writing C/C++ Callable SIMD Subroutines

You can write assembly subroutines that use the ADSP-2116x DSP's SIMD mode and call them from your C programs. The routine may use SIMD mode (PEYEN bit=1) for all code between the function prologue and epilogue, placing the chip in SISD mode (PEYEN bit =0) before the function epilogue or returning from the function.



While it is possible to write subroutines that can be called in SIMD mode (the chip is in SIMD mode before the call and after the return), the compiler does not support a SIMD call interface at this time. For example, trying to call a subroutine from a `#pragma SIMD_for` loop prevents the compiler from executing the loop in SIMD mode because the compiler does not support SIMD mode calls.

Because transfers between memory and data registers are doubled in SIMD mode (each explicit transfer has a matching implicit transfer), it is recommended that you access the stack in SISD mode to prevent corrupting the stack. For more information on SIMD mode memory accesses, see the *Memory* chapter in the *SHARC DSP Hardware Reference* for an appropriate ADSP-2116x DSP.

If you are using SIMD subroutines, your interrupt handler must provide additional support. This support in the interrupt service routine entails saving-restoring the PEYEN bit and placing the DSP in the mode (SISD or SIMD) that the interrupt service routine needs. Interrupt handlers often use the MMASK register to expedite these mode changes.

### C++ Programming Examples

This section provides the following examples for C++-specific features:

- [“Using Fract Support” on page 1-168](#)
- [“Using Complex Support” on page 1-169](#)

Note that the `cc21k` compiler runs in C mode by default. To run the compiler in C++ mode, you select the corresponding option on the command line, or check it in the `Project Options` dialog box of the VisualDSP++ environment.

For example, the following command line

```
cc21k -c++ fdot.c -T062.ldf
```

runs `cc21k` with:

|                         |                                                                                                                         |
|-------------------------|-------------------------------------------------------------------------------------------------------------------------|
| <code>-c++</code>       | Specifies that the following source file is written in ANSI/ISO standard C++ extended with the Analog Devices keywords. |
| <code>fdot.c</code>     | Specifies the source file for your program.                                                                             |
| <code>-T 062.ldf</code> | Specifies the Linker Description File for the ADSP-21062 DSP system.                                                    |

### Using Fract Support

[Listing 1-3 on page 1-169](#) demonstrates the compiler support for the `fract` type and associated arithmetic operators, such as `+` and `*`. The dot product algorithm is expressed using the standard arithmetic operators. The code demonstrates how two variable-length arrays are initialized with fractional literals.

For more information about the fractional data type and arithmetic, see [“C++ Fractional Type Support” on page 1-96](#).

## Listing 1-3. Dot Product Using Fract Arithmetic Example — C++ Code

```

fract fdot (int array_size, fract *x, fract *y)
{
    int j;
    fract s;
    s = 0;
    for (j=0; j < array_size; j++)
    {
        s += x[j] * y[j];
    }
    return s;
}

int main(void)
{
    set_saturate_mode();
    fdot (N,x,y);
}

```

## Using Complex Support

The Mandelbrot fractal set is defined by the following iteration on complex numbers:

$$z := z * z + c$$

The  $c$  values belong to the set for which the above iteration does not diverge to infinity. The canonical set is defined when  $z$  starts from zero.

[Listing 1-4](#) demonstrates the Mandelbrot generator expressed in a simple algorithm using the C++ library `complex` class.

## Listing 1-4. Mandelbrot Generator Example — C++ code

```

#include <complex>
int iterate (complex<double> c, complex<double> z, int max)
{
    int n;

```

```
for (n = 0; n<max && abs(z)<2.0; n++)
{
    z = z * z + c;
}
return (n == max ? 0 : n);
}
```

[Listing 1-5](#) shows a C version of the inner computational function of the Mandelbrot generator, which extracts performance and programming penalties (compared with the C++ version).

### Listing 1-5. Mandelbrot Generator Example — C code

```
int    iterate (double creal, double cimag,
double zreal, double zimag, int max)
{
    double real, imag;
    int n;
    real = zreal * zreal;
    imag = zimag * zimag;
    for (n = 0; n<max && (real+imag)<4.0; n++)
    {
        zimag = 2.0 * zreal * zimag + cimag;
        zreal = real - imag + creal;
        real = zreal * zreal;
        imag = zimag * zimag;
    }
    return (n == max ? 0 : n);
}
```

## Mixed C/C++/Assembly Programming Examples

This section shows examples of types of mixed C/C++/assembly programming in order of increasing complexity. The examples in this section are as follows:

- [“Using Inline Assembly \(Add\)” on page 1-172](#)
- [“Using Macros to Manage the Stack” on page 1-173](#)
- [“Using Scratch Registers \(Dot Product\)” on page 1-174](#)
- [“Using Void Functions \(Delay\)” on page 1-176](#)
- [“Using the Stack for Arguments and Return \(Add 5\)” on page 1-177](#)
- [“Using Registers for Arguments and Return \(Add 2\)” on page 1-178](#)
- [“Using Non-Leaf Routines That Make Calls \(RMS\)” on page 1-179](#)
- [“Using Call Preserved Registers \(Pass Array\)” on page 1-181](#)

Note that leaf assembly routines are routines that return without making any calls. Non-leaf assembly routines call other routines before returning to the caller.

Note that you can use `cc21k` to compile your C or C++ program and assemble your assembly language modules. This ensures that the assembly of your modules complies with the C/C++ run-time environment.

For example, the following `cc21k` command line

```
cc21k my_prog.c my_sub1.asm -T 062.ldf -Wremarks
```

runs `cc21k` with:

## C/C++ and Assembly Interface

|                          |                                                                                  |
|--------------------------|----------------------------------------------------------------------------------|
| <code>my_prog.c</code>   | Selects a C language source file for your program                                |
| <code>my_sub1.asm</code> | Selects an assembly language module to be assembled and linked with your program |
| <code>-T 062.ldf</code>  | Selects a linker description file describing your DSP system                     |
| <code>-Wremarks</code>   | Selects diagnostic compiler warnings                                             |

### Using Inline Assembly (Add)

The following example shows how to write a simple routine in ADSP-21xxx DSP assembly code that properly interfaces to the C/C++ environment. It uses the `asm()` construct to pass inline assembly code to the compiler.

```
int i,j,k,l;
main()
{
    l = add(i,j,k);
}

/* Per the run-time environment, function add() passes arg x in
   r4, arg y in r8, and arg z in r12. Then, func adds args and
   puts return in r0. */

add(int x, int y, int z)
{
    asm("r0=%0+%1; r0=r0+%2"::"d"(x),"d"(y),"d"(z));
}
```

## Using Macros to Manage the Stack

[Listing 1-6](#) and [Listing 1-7 on page 1-174](#) show how macros can simplify function calls between C, C++, and assembly functions. The assembly function uses the `entry`, `exit`, and `ccall` macros to keep track of return addresses and manage the run-time stack. [For more information, see “Managing the Stack” on page 1-142.](#)

### Listing 1-6. Subroutine Return Address Example — C Code

```
/* Subroutine Return Address Example—C code: */
/* assembly and c functions prototyped here */
void asm_func(void);
void c_func(void);

                                /* c_var defined here as a global */
                                /* used in .asm file as _c_var      */
int c_var=10;

                                /* asm_var defined in .asm file as _asm_var */
extern int asm_var;

main ()
{
    asm_func();                /* call to assembly function      */
}

                                /* this function gets called from asm file */
void c_func(void)
{
    if (c_var != asm_var)
        exit(1);
    else
        exit(0);
}
```

Listing 1-7. Subroutine Return Address Example—Assembly Code

```
/* Subroutine Return Address Example—Assembly code: */
#include <asm_sprt.h>
.section/dm seg_dmda;
.var _asm_var=0;          /* asm_var is defined here */
.global _asm_var;         /* global for the C function */

.section/pm seg_pmco;
.global _asm_func;        /* _asm_func is defined here */
.extern _c_func;          /* c_func from the C file */
.extern _c_var;           /* c_var from the C file */

_asm_func:
entry;                   /* entry macro from asm_sprt */

    r8=dm(_c_var);        /* access the global C var */
    dm(_asm_var)=r8;       /* set _asm_var to _c_var */

    ccall(_c_func);        /* call the C function */

    exit;                 /* exit macro from asm_sprt */
```

### Using Scratch Registers (Dot Product)

To write assembly functions that can be called from a C or C++ program, your assembly code must follow the conventions of the C/C++ run-time environment and use the conventions for naming functions. The following assembly function demonstrates how to comply with these specifications.

This function computes the dot product of two vectors. The two vectors and their lengths are passed as arguments. Because the function uses only scratch registers (as defined by the run-time environment) for intermediate values and takes advantage of indirect addressing, the function does not need to save or restore any registers.



```

/* dot(int n, dm float *x, pm float *y);
Computes the dot product of two floating-point vectors of length
n. One is stored in dm and the other in pm. Length n must be
greater than 2.*/

#include <asm_sprt.h>

.section/pm seg_pmco;
/* By convention, the assembly function name is the C function
name with a leading underscore; "dot()" in C becomes "_dot" in
assembly */

.global _dot;
_dot:

leaf_ entry;

r0=r4-1,i4=r8;
/* Load first vector address into I register, and load r0 with
length -1 */

r0=r0-1,i12=r12;
/* Load second vector address into I register and load r0 with
length-2 (because the 2 iterations outside feed and drain the
pipe */

f12=f12-f12,f2=dm(i4,m6),f4=pm(i12,m14);
/* Zero the register that will hold the result and start feeding
pipe */

f8=f2*f4, f2=dm(i4,m6),f4=pm(i12,m14);
/* Second data set into pipeline, also do first multiply */

lcntr=r0, do dot_loop until lce;
/* Loop length-2 times, three-stage pipeline: read, mult, add */

dot_loop:
    f8=f2*f4, f12=f8+f12,f2=dm(i4,m6),f4=pm(i12,m14);
    f8=f2*f4, f12=f8+f12;
    f0=f8+f12;

```

## C/C++ and Assembly Interface

```
/* drain the pipe and end with the result in r0, where it'll be
returned */
```

```
leaf_exit;
/* restore the old frame pointer and return */
```

### Using Void Functions (Delay)

The simplest kind of assembly routine is one with no arguments and no return value, which corresponds to C/C++ functions prototyped as `void my_function(void)`. Such routines could be used to monitor an external event or used to perform an operation on a global variable.

It is important when writing such assembly routines to pay close attention to register usage. If the routine uses any call-preserved or compiler reserved registers (as defined in the run-time environment), the routine must save the register and restore it before returning. Because the following example does not need many registers, this routine uses only scratch registers (also defined in the run-time environment) that do not need to be saved.

Note that in the example all symbols that need to be accessed from C/C++ contain a leading underscore. Because the assembly routine name `_delay` and the global variable `_del_cycle` must both be available to C and C++ programs, they contain a leading underscore in the assembly code.

```
/* Simple Assembly Routines Example - _delay */
/* void delay (void);
An assembly language subroutine to delay N cycles, where N is the
value of the global variable del_cycle */

#include <asm_sprt.h>;

.section/pm seg_pmco;
.extern _del_cycle;
.global _delay;
_delay:
    leaf_entry;                /* first line of any leaf func */
```

```

    R4 = DM (_del_cycle);
/* Here, register r4 is used because it is a scratch register and
doesn't need to be preserved */
    LCNTR = R4, D0 d_loop UNTIL LCE;
    d_loop:
    nop;
    leaf_exit;                /* last line of any leaf func */

```

## Using the Stack for Arguments and Return (Add 5)

A more complicated kind of routine is one that has parameters but no return values. The following example adds together the five integers passed as parameters to the function.

```

/* Assembly Routines With Parameters Example — _add5 */
/* void add5 (int a, int b, int c, int d, int e);
An assembly language subroutine that adds 5 numbers */

#include <asm_sprt.h>
.section/pm seg_pmco;
.extern _sum_of_5; /* variable where sum will be stored */
.global _add5;

_add5:
leaf_entry;
/* the calling routine passes the first three parameters in
registers r4, r8, r12 */

r4 = r4 + r8;      /* add the first and second parameter */
r4 = r4 + r12;     /* adds the third parameter */

/* the calling routine places the remaining parameters
(fourth/fifth) on the run-time stack; these parameters can be
accessed using the reads() macro */

r8 = reads(1);     /* put the fourth parameter in r8 */
r4 = r4 + r8;      /* adds the fourth parameter */
r8 = reads(2);     /* put the fifth parameter in r8 */
r4 = r4 + r8;      /* adds the fifth parameter */

```

```
dm(_sum_of_5) = r4;
/* place the answer in the global variable */

leaf_exit;
```

### Using Registers for Arguments and Return (Add 2)

There is another class of assembly routines in which the routines have both parameters and return values. The following example of such a routine adds two numbers and returns the sum. Note that this routine follows the run-time environment specification for passing function parameters (in registers r4 and r8) and passing the return value (in register r0).

```
/* Routine With Parameters & Return Value -add2_ */
/* int add2 (int a, int b);
An assembly language subroutine that adds two numbers and returns
the sum */

#include <asm_sprt.h>

.section/pm seg_pmco;
.global _add2;
_add2:
leaf_entry;

/* per the run-time environment, the calling routine passes the
first two parameters passed in registers r4 and r8; the return
value goes in register r0 */

r0 = r4 + r8;
/* add the first and second parameter, store in r0*/

leaf_exit;
```

## Using Non-Leaf Routines That Make Calls (RMS)

A more complicated example, which calls another routine, computes the root mean square of two floating-point numbers, such as

$$z = \sqrt{x^2 + y^2}$$

Although it is straight forward to develop your own function that calculates a square-root in ADSP-21xxx assembly language, the following example demonstrates how to call the square root function from the C/C++ run-time library, `sqrtf`. In addition to demonstrating a C run-time library call, this example shows some useful calling macros.

```
/* Non-Leaf Assembly Routines Example - _rms */
/* float rms(float x, float y); An assembly language subroutine
to return the rms z = (x^2 + y^2)^(1/2) */

#include <asm_sprt.h>

.section/pm seg_pmco;
.extern _sqrtf;
.global _rms;
_rms:
    entry;                /* first line of non-leaf routine */

    f4 = f4 * f4;
    f8 = f8 * f8;
    f4 = f4 + f8;
/* f4 contains argument to be passed to sqrtf function */

    ccall (_sqrtf);
/* use the ccall() macro to make a function call in a C environ-
ment; f0 contains the result returned by the _sqrtf function. In
turn, _rms returns the result to its caller in f0 (and it is
already there) */
    exit;                /* last line of non-leaf routine */
```

If a called function takes more than three single word parameters, the remaining parameters must be pushed on to the stack and popped off the stack after the function call. The following function could call the `_add5` routine shown in [“Using the Stack for Arguments and Return \(Add 5\)” on page 1-177](#). Note that the last parameter must be pushed on the stack first.

```
/* Non-Leaf Assembly Routines Example - _calladd5 */
/* int calladd5 (void); An assembly language subroutine that
calls another routine with more than 3 parameters.
This example adds the numbers 1, 2, 3, 4, and 5. */

#include <asm_sprt.h>
.section/pm_seg_pmco;
.extern _add5;
.extern _sum_of_5;
.global _calladd5;
_calladd5:

entry;
    r4 = 5;
/* the fifth parameter is stored in r4 for pushing onto stack */
    puts=r4;          /* put fifth parameter in stack */
    r4 = 4;
/* the fourth parameter is stored in r4 for pushing onto stack */
    puts=r4;          /* put fourth parameter in stack */
    r4 = 1;           /* the first parameter is sent in r4 */
    r8 = 2;           /* the second parameter is sent in r8 */
    r12 = 3;          /* the third parameter is sent in r12 */

    ccall (_add5);
/* use the ccall macro to make a function call in a C environment */
    alter(2);
/* call the alter() macro to remove the two arguments from the
stack */
    r0 = dm(_sum_of_5);
/* _sum_of_5 is where add5 stored its result */
    exit;
```

## Using Call Preserved Registers (Pass Array)

Some functions need to make use of registers that the run-time environment defines as *call preserved registers*. These registers, whose contents are preserved across function calls, are useful for variables whose lifetime spans a function call. The following example performs an operation on the elements of a C array using call preserved registers.

```
/* Non-Leaf Assembly Routines Example - _pass_array */
/* void pass_array(
float function(float),
float *array,
int length);
An assembly language routine that operates on a C array */

#include <asm_sprt.h>
.section/pm seg_pmco;
.global _pass_array;
_pass_array:
    entry;
    puts = i8;
/* This function uses a call preserved register, i8, because it
could be used by multiple functions, and this way it does not
have to be stored for every function call */

    r0 = i1;
    puts = r0;      /* i1 is also call preserved */

    i8 = r4;
/* read the first argument, the address of the function to call
*/

    i1 = r8;
/* read the second argument, the C array containing the data to
be processed */

    r0 = r12;
/* read third argument, the number of data points in the array */

    lcntr=r0, do pass_array_loop until lce;
```

## C/C++ and Assembly Interface

```
/* loop through data points */

f4=dm(i1,m5);
/* get data point from array, store it in f4 as a parameter for
the function call */

ccall(m13,i8);/* call the function */
pass_array_loop:
    dm(i1,m6)=f0;
/* store the return value back in the array */
    i1 = gets(1);    /* restore the value of i1 */
    i8 = gets(2);    /* restore the value of i8 */

exit;
```



## 2 C/C++ RUN-TIME LIBRARY

The C and C++ run-time libraries are collections of functions, macros, and class templates that you can call from your source programs. Many functions are implemented in the ADSP-21xxx assembly language. C and C++ programs depend on library functions to perform operations that are basic to the C and C++ programming environments. These operations include memory allocations, character and string conversions, and math calculations. Using the library simplifies your software development by providing code for a variety of common needs.

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

- “C and C++ Run-Time Libraries Guide” (starting on [page 2-3](#)) contains introductory information about the ANSI/ISO Standard C and C++ libraries. It also provides information about the ANSI-standard header files and built-in functions that are included with this release of the `cc21k` compiler.
- “C Run-Time Library Reference” (starting on [page 2-29](#)) contains reference information about the C run-time library functions included with this release of the `cc21k` compiler.

The cc21k compiler provides a broad collection of library functions, including those required by the ANSI standard and additional functions of value for DSP programming supplied by Analog Devices. In addition to the standard C library, this release of the compiler software includes the abridged C++ library, a conforming subset of the standard C++ library. The abridged C++ library includes the embedded C++ and embedded standard template libraries.

This chapter describes the standard C/C++ library functions in the current release of the run-time libraries. Chapter 3, “[DSP Library for ADSP-2106x and ADSP-21020 Processors](#)”, and Chapter 4, “[DSP Library for ADSP-2116x Processors](#)”, describe a number of signal processing, matrix, and statistical functions that assist DSP code development.

 For more information on the algorithms on which many of the C library’s math functions are based, see Cody, W. J. and W. Waite, *Software Manual for the Elementary Functions*, Englewood Cliffs, New Jersey: Prentice Hall, 1980. For more information on the C++ library portion of the ANSI/ISO Standard for C++, see Plauger, P. J. (Preface), *The Draft Standard C++ Library*, Englewood Cliffs, New Jersey: Prentice Hall, 1994, (ISBN: 0131170031).

The C++ library reference information in HTML format is included on the software distribution CD-ROM. To access the reference files from VisualDSP++, use the **Help Topics** command (**Help** menu) and select the **Reference** book icon. From the **Online Manuals** topic, you can open any of the library files. You can also manually access the HTML files using a web browser.

## C and C++ Run-Time Libraries Guide

The C and C++ run-time libraries contain routines that you can call from your source program. This section describes how to use the libraries and provides information on the following topics:

- [“Calling Library Functions” on page 2-3](#)
- [“Linking Library Functions” on page 2-4](#)
- [“Working with Library Header Files” on page 2-6](#)
- [“Using Compiler’s Built-In C Library Functions” on page 2-19](#)
- [“Abridged C++ Library Support” on page 2-21](#)

For information on the C library’s contents, see [“C Run-Time Library Reference”](#) (starting on [on page 2-29](#)). For information on the Abridged C++ library’s contents, see [“Abridged C++ Library Support”](#) (starting on [on page 2-21](#)) and on-line Help.

### Calling Library Functions

To use a C/C++ library function, call the function by name and give the appropriate arguments. The name and arguments for each function appear on the function’s reference page. The reference pages appear in the [“C Run-Time Library Reference”](#) (starting on [on page 2-29](#)) section and in the [“C++ Run-Time Library”](#) topic of the on-line Help.

Like other functions you use, library functions should be declared. Declarations are supplied in header files. For more information about the header files, see [“Working with Library Header Files” on page 2-6](#).

-  Function names are C/C++ function names. If you call a C or C++ run-time library function from an assembly program, you must use the assembly version of the function name (prefix an underscore on the name). For more information on the naming conventions, see [“C/C++ and Assembly Interface” on page 1-154](#).
-  You can use the archiver, `elfar`, described in the *VisualDSP++ 3.0 Linker and Utilities Manual for SHARC DSPs*, to build library archive files of your own functions.

### Linking Library Functions

The C/C++ run-time library is organized as four libraries:

- C run-time library — Comprises all the functions that are defined by the ANSI standard
- C++ run-time library
- DSP run-time library — Contains additional library functions supplied by Analog Devices that provide services commonly required by DSP applications
- I/O library — Supports a subset of the C standard's I/O functionality

Each library has two versions: one set is suitable for running on the ADSP-2106x and ADSP-21020 DSPs and the other is suitable for running on the ADSP-2116x DSPs. [Table 2-1 on page 2-5](#) catalogues the names of each of the libraries and also lists other files that are used while linking a program.

Table 2-1. C and C++ Files and Libraries

| Description                                                                        | ADSP-2106x DSPs                                                                                                | ADSP-2116x DSPs                                                    |
|------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------|
| C run-time library functions                                                       | libc.dlb<br>libc020.dlb (21020 only)                                                                           | libc160.dlb (21160 only)<br>libc161.dlb (21161 only)               |
| Threadsafe C run-time library functions                                            | libcmtdlb                                                                                                      | libc160mt.dlb (21160 only)<br>libc161mt.dlb (21161 only)           |
| C++ run-time library functions                                                     | libcpp.dlb                                                                                                     | libcpp.dlb                                                         |
| Threadsafe C++ run-time library functions                                          | libcppmt.dlb                                                                                                   | libcppmt.dlb                                                       |
| C++ run-time library support functions                                             | libcppprt.dlb                                                                                                  | libcppprt.dlb                                                      |
| DSP run-time library functions                                                     | libdsp.dlb<br>libdsp020.dlb (21020 only)                                                                       | libdsp160.dlb (21160 only)<br>libdsp161.dlb (21161 only)           |
| I/O library functions                                                              | libio.dlb,<br>libio020.dlb (21020 only)                                                                        | libio.dlb                                                          |
| Threadsafe I/O library functions                                                   | libiomtdlb                                                                                                     | libiomtdlb                                                         |
| Start-up file for C programs — calls set-up routines and main                      | 020_hdr.doj (21020 only)<br>060_hdr.doj (21060 only)<br>061_hdr.doj (21061 only)<br>065L_hdr.doj (21065L only) | 160_hdr.doj (21160 only)<br>161_hdr.doj (21161 only)               |
| Start-up file for C++ programs — calls set-up routines and main                    | 060_cpp_hdr.doj (21060 only)<br>061_cpp_hdr.doj (21061 only)<br>065L_cpp_hdr.doj (21065L only)                 | 160_cpp_hdr.doj (21160 only)<br>161_cpp_hdr.doj (21161 only)       |
| Start-up file for multi-threaded C++ applications — calls set-up routines and main | 060_cpp_hdr_mt.doj (21060 only)<br>061_cpp_hdr_mt.doj (21061 only)<br>065L_cpp_hdr_mt.doj (21065L only)        | 160_cpp_hdr_mt.doj (21160 only)<br>161_cpp_hdr_mt.doj (21161 only) |

The libraries and start-up files are installed within subdirectories of your VisualDSP++ installation. The files that are used to build applications for the ADSP-2106x DSP architecture are installed within the directory `21k\lib` and those used for the ADSP-2116x DSP architecture are installed within the directory `211xx\lib`.

 An alternative set of libraries and start-up files are available to circumvent the ADSP-2116x DSP's Shadow Write Anomaly. The libraries and binary files are installed within the subdirectory `211xx\lib\swfa` and will be used by the linker in place of the default libraries if the “`-swf_screening`” switch (see [on page 1-44](#)) is specified.

When you call a run-time library function, the call creates a reference that the linker resolves when linking your program. One way to direct the linker to the library's location is to use the default Linker Description File (ADSP-`<your_target>.ldf`).

If you are not using the default LDF, 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 -ldsp` will add `libc.dlb` and `libdsp.dlb` to the list of libraries to be searched by the linker. For more information on the `.LDF` file, see the *VisualDSP++ 3.0 Linker and Utilities Manual for SHARC DSPs*.

## Working with Library Header Files

When you use a library object (function, macro, or class template) in your program, you include the corresponding header file with the `#include` preprocessor command. The header file for each library function is identified in the **Synopsis** section of the reference page. Header files contain prototypes. The compiler uses these prototypes to check that each library object is called with the correct arguments.

## Standard C Library Header Files

The following C standard header files are supplied with this release of the ADSP-21xxx DSP compiler. You should use a C standard text to augment the information supplied in this chapter.

[Table 2-2](#) lists the header files that contain ANSI standard run-time environment macros for error handling, standard definitions, limits, and floating-point variables. These do not contain individual functions; they consist of macros and type definitions. See the [“Standard C Library Header File Descriptions” on page 2-8](#) for more detailed descriptions.

Table 2-2. ANSI Standard Run-Time Environment Macros

| Header File           | Description                               |
|-----------------------|-------------------------------------------|
| <code>errno.h</code>  | Error Handling.                           |
| <code>float.h</code>  | Floating-point implementation parameters. |
| <code>limits.h</code> | Implementation limits.                    |
| <code>stddef.h</code> | Standard definitions.                     |

Ten C standard header files that contain ANSI standard functions are supplied with the present release of the SHARC compiler. [Table 2-3](#) provides a list of the Standard C Library function header files. See the [“Standard C Library Header File Descriptions” on page 2-8](#) for more detailed descriptions.

Table 2-3. ANSI Standard Run-Time Functions

| Header File           | Description           |
|-----------------------|-----------------------|
| <code>assert.h</code> | Diagnostics.          |
| <code>ctype.h</code>  | Character handling.   |
| <code>locale.h</code> | Localization.         |
| <code>math.h</code>   | Basic math functions. |

## C and C++ Run-Time Libraries Guide

Table 2-3. ANSI Standard Run-Time Functions (Cont'd)

| Header File           | Description         |
|-----------------------|---------------------|
| <code>setjmp.h</code> | Non-local jumps.    |
| <code>signal.h</code> | Signal handling.    |
| <code>stdarg.h</code> | Variable arguments. |
| <code>stdio.h</code>  | Input/Output.       |
| <code>stdlib.h</code> | Standard library.   |
| <code>string.h</code> | String handling.    |

### Standard C Library Header File Descriptions

This section provides descriptions of the header files contained in the C library. The header files are listed in alphabetical order.

#### **`assert.h`**

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

#### **`ctype.h`**

The `ctype.h` header file contains functions for character handling, such as `isalpha`, `tolower`, etc.

#### **`errno.h`**

The `errno.h` header file provides access to `errno` and also defines macros for associated error codes.



## **float.h**

The `float.h` header file contains definitions of size and precision values for each C floating point data type. The `FLT_ROUNDS` macro defined in the header file is set to the C run-time environment definition of the rounding mode for `float` variables which is round-toward-nearest. The rounding mode for `long double` variables is truncation.

## **limits.h**

The `limits.h` header file contains definitions of maximum and minimum values for each C data type other than floating-point.

## **locale.h**

The `locale.h` header file contains definitions for expressing numeric, monetary, time, and other data.

## **math.h**

The `math.h` header file includes trigonometric, power, logarithmic, exponential, and other miscellaneous functions. The library contains the functions specified by the C standard along with implementations for `float`.

This header file also provides prototypes for a number of additional math functions provided by Analog Devices, such as `favg`, `fmax`, `fclip`, and `copysign`. Refer to Chapter 3, [“DSP Library for ADSP-2106x and ADSP-21020 Processors”](#), and Chapter 4, [“DSP Library for ADSP-2116x Processors”](#), for more information about these additional functions.

The `math.h` header contains prototypes for both single and double precision functions. For every `double` mathematical function, there is a corresponding `float` function that has the same name as the `double` function but with a suffix of `'f'`. For example, the name of the single-precision sine function is `sinf`. When the compiler is treating `double` as 32 bits, the header file will arrange that all references to the `double` functions (with

## C and C++ Run-Time Libraries Guide

un-suffixed names) are directed to the equivalent `float` function (with the suffix `f`). This lets you use the un-suffixed names with arguments of type `double`, regardless of whether doubles are 32 or 64 bits long.

The `math.h` header file also defines the macro `HUGE_VAL`. This macro 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.

A domain error occurs when an input argument is outside the domain of the function. “[C Run-Time Library Reference](#)” (starting [on page 2-29](#)) lists the specific cases that cause `errno` to be set to `EDOM`, and the associated return values.

A range error occurs when the result of a function cannot be represented in the return type. If the result overflows, the function returns the value `HUGE_VAL` with the appropriate sign. If the result underflows, the function returns a zero without indicating a range error.

### **setjmp.h**

The `setjmp.h` header file contains `setjmp` and `longjmp` for non-local jumps.

### **signal.h**

The `signal.h` header file provides function prototypes for the standard ANSI `signal.h` routines and also for several ADSP-21xxx DSP extensions, such as `interrupt()` and `clear_interrupt()`. The signal handling functions process conditions (hardware signals) that can occur during program execution. They determine the way that your C program responds to these signals. The functions are designed to process such signals as external interrupts and timer interrupts.

### **Interrupt Dispatchers**

There are currently four types of the interrupt dispatcher, each providing different levels of functionality and performance. There are also variants of these to avoid the use of self-modifying code and to disable circular buffering when the interrupt handler is called. These variants are discussed at the end of this section.

For the normal interrupt dispatcher, use the `interrupt()` or `signal()` functions to set up the interrupt. This dispatcher provides the following services:

- Saves all scratch registers and the loop stack. DO loop and interrupt nesting is allowed because data is pushed onto the stack. Requires approximately 125 cycles for interrupt overhead.
- Sends the interrupt number type to the ISR as a parameter

For the fast interrupt dispatcher, use the `interruptf()` or `signalf()` functions. This dispatcher provides the following services:

- Does not save the loop stack; DO loop handling is restricted to six levels (specified in hardware). If the interrupt service routine (ISR) uses one level of nesting, your code cannot exceed five levels.
- Interrupt nesting is not restricted (20 levels available). Does not send the interrupt number type to the ISR as a parameter. Requires approximately 60 cycles for interrupt overhead.

For the super-fast interrupt dispatcher, use the `interrupts()` or `signals()` functions. This dispatcher provides the following services:

- Does not save the loop stack, therefore do loop handling is restricted to six levels (specified in hardware). Interrupt nesting is disabled. This dispatcher does not send the interrupt number type to the ISR as a parameter.
- Uses the secondary register set. This dispatcher requires approximately 30 cycles for interrupt overhead.

The final interrupt dispatcher is intended for use with user-written assembly functions or C functions that have been compiled using “`#pragma interrupt`”. Use the `interruptss()` or `signalss()` function to utilize this dispatcher. This dispatcher provides the following services:

- Relies on the compiler (or assembly routine) to save and restore all appropriate registers.
- Does not save the loop stack, therefore DO loop handling is restricted to six levels (specified in hardware).
- Interrupt nesting is not restricted (20 levels available). This dispatcher does not send the interrupt number type to the ISR as a parameter.

For more details on the use of this interrupt dispatcher and “`#pragma interrupt`”, see [“Interrupt Handler Pragma” on page 1-94](#).

### Avoiding Self-Modifying Code

The interrupt set-up functions use self-modifying code to set up the interrupts as this offers savings in execution time and code size.

Non-self-modifying variants of all these functions are supplied and are suffixed with “`nsm`”. For example, to utilize the fast interrupt dispatcher, use the `interruptfnsm()` or `signalfnsm()` functions. The choice of a non self-modifying function has no effect on the dispatcher used and no effect on the overall interrupt handling performance.

### Disabling Circular Buffers

A variant of the normal interrupt dispatcher that disables circular buffering (by zeroing L0 - L5 and L8 - L15) is supplied. This variant is used by calling `interruptcb()` or `signalcb()`. Additional 54 cycles are required to save, zero and restore the L registers in the dispatcher. Note that “`#pragma interrupt`” dispatcher automatically handles any appropriate L registers.

A version of the normal interrupt dispatcher that avoids self-modifying code and disables circular buffering is supplied and has the same name as the standard version, but is suffixed with “cbnsm”. It is used by calling `interruptcbnsm()` or `signalcbnsm()` function.

### **stdarg.h**

The `stdarg.h` header file contains definitions needed for functions that accept a variable number of arguments. Callers of such functions must include a prototype.

### **stddef.h**

The `stddef.h` header file contains a few common definitions useful for portable programs, such as `size_t`.

### **stdio.h**

The `stdio.h` header file contains a subset of the C standard's I/O functionality. Always include the header file in your source if you use any of its facilities because the header file contains dual support for type `double` — support for when it is 32 bits and support for when it is 64 bits. Failure to include the header file results in a linker failure as the compiler must see a correct function prototype in order to generate the correct calling sequence.

The following facilities are not available in this software release:

- Stream positioning functions `fgetpos`, `fseek`, `fsetpos`, `ftell`, and `rewind`
- File handling functions `remove`, `rename`, `tmpfile`, and `tmpnam`
- `printf` and `scanf` functions do not support values of type `long double` when type `double` is the same size as type `float`



Each opened stream is allocated a buffer which either contains data from an input file or output from a program. For text streams, this data is held in the form of 8-bit characters that are packed into 32-bit memory locations. Due to internal mechanisms used to unpack and pack this data, the buffer must not reside at a memory location that is greater than the address 0x3fffffff. Since the `stdio` library allocates buffers from the heap, this restriction implies that the heap should not be placed at address 0x40000000 or above. The restriction may be avoided by using the `setvbuf` function to allocate the buffer from alternative memory, as in the following example.

```
#include <stdio.h>

char buffer[BUFSIZ];
setvbuf(stdout,buffer,_IOLBF,BUFSIZ);
printf("Hello World\n");
```

This example assumes that the buffer will reside at a memory location that is less than 0x40000000.

### Implementation of the `stdio` Library

The default I/O library (`libio.dlb`) uses a simple interface to communicate with a host environment, which may be a simulator, debugger, or emulator. The intent is to place minimal demands on the host environment while still providing as much functionality to the DSP as possible.

All I/O operations are channeled through the C function `_primIO`. The assembly label has two underscores, `__primIO`. The source for this function, and all the other library routines, can be found under the base installation for VisualDSP++ in the subdirectory `...\lib\src\libio_src`.

The `__primIO` function accepts no arguments. Instead, it examines the I/O control block at the label `_PrimIOCB`. Without external intervention by a host environment, the `__primIO` routine simply returns, which indicates failure of the request. Two schemes for host interception of I/O requests are provided.

The first scheme is to modify control flow into and out of the `__primIO` routine. Typically, this would be achieved by a break point mechanism available to a debugger/simulator. Upon entry to `__primIO`, the data for the request will reside in a control block at the label `_PrimIOCB`. If this scheme is used, the host should arrange to intercept control when it enters the `__primIO` routine, and, after servicing the request, return control to the calling routine.

The second scheme involves communicating with the DSP process through a pair of simple semaphores. This scheme is most suitable for an externally-hosted development board. Under this scheme, the host system should clear the data word whose label is `__lone_SHARC`; this will cause `__primIO` to assume that a host environment is present and able to communicate with the DSP process. If `__primIO` sees that `__lone_SHARC` is cleared, then upon entry (for example, when an I/O request is made) it will set a non-zero value into the word labeled `__Godot`. The `__primIO` routine will then busy-wait until this word is reset to zero by the host. The non-zero value of `__Godot` raised by `__primIO` is the address of the I/O control block.

### **Data Packing For Primitive I/O**

The DSP implementation is based on a word-addressable machine, with each word comprising a fixed number of 8-bit bytes. All `READ` and `WRITE` requests specify a move of some number of 8-bit bytes; that is, the relevant fields count 8-bit bytes, not words. Packing is always little endian, the first byte of a file read or written is the low-order byte of the first word transferred.

## C and C++ Run-Time Libraries Guide

Data packing is set to four bytes per word for the SHARC DSP architecture. Data packing can be changed on the DSP side to accommodate other DSP architectures by modifying the constant `BITS_PER_WORD`, defined in `_wordsize.h`.

Note that the file name provided in an `OPEN` request uses the DSP's "native" string format, normally one byte per word. Data packing applies only to `READ` and `WRITE` requests.

### Data Structure for Primitive I/O

The I/O control block is declared in `_primio.h`.

```
typedef struct
{
    enum
    {
        PRIM_OPEN = 100,
        PRIM_READ,
        PRIM_WRITE,
        PRIM_CLOSE
    } op;

    int    fileID;
    int    flags;
    unsigned char *buf;    /* data buffer, or file name */
    int    nDesired;       /* number of characters to read */
                                /* or write */
    int    nCompleted;     /* number of characters actually */
                                /* read or written */
    void    *more;         /* for future use */
}
PrimIOCB_T;
```

The first field, `op`, identifies which of the four currently supported operations is being requested.

The file ID for an open file is a non-negative integer assigned by the debugger or other "host" mechanism. The file ID values 0, 1, and 2 are pre-assigned to `stdin`, `stdout`, and `stderr`, respectively. No open request is required for these file IDs.



The `flags` field is a bit field containing other information for special requests. Meaningful bit values for an `OPEN` operation are:

```
M_OPENR = 0x0001    /* open for reading */
M_OPENW = 0x0002    /* open for writing */
M_OPENA = 0x0004    /* open for append */
M_TRUNCATE = 0x0008 /* truncate to zero length if file exists */
M_CREATE = 0x0010   /* create the file if necessary */
M_BINARY = 0x0020   /* binary file (vs. text file) */
```

For a `READ` operation, the low-order four bits of the flag value contain the number of bytes packed into each word of the read buffer, and the rest of the value are reserved for future use.

For a `WRITE` operation, the low-order four bits of the flag value contain the number of bytes packed into each word of the write buffer, and the rest of the value form a bit field, for which only the following bit is currently defined:

```
M_ALIGN_BUFFER = 0x10
```

If this bit is set for a `WRITE` request, the `WRITE` operation is expected to be aligned on a DSP word boundary by writing padding `NULLs` to the file before the buffer contents are transferred.

The `flags` field is currently unused for a `CLOSE` request.

The `buf` field contains a pointer to the file name for an open request, or a pointer to the data buffer for a read or write request.

For a read or write request, `nDesired` is the number of bytes the program is requesting to transfer, `__primIO` is expected to set `nCompleted` to the number of bytes actually transferred.

The `more` field is reserved for future use, and currently is always set to `NULL` before calling `__primIO`.

## C and C++ Run-Time Libraries Guide

### stdlib.h

The `stdlib.h` header file offers general utilities specified by the C standard. These include some integer math functions, such as `abs`, `div`, and `rand`; general string-to-numeric conversions; memory allocation functions, such as `malloc` and `free`; and termination functions, such as `exit`. This library also contains miscellaneous functions such as `bsearch` and `qsort`.

This header file also provides prototypes for a number of additional integer math functions provided by Analog, such as `avg`, `max`, and `clip`.

Table 2-4 is a summary of the additional library functions defined by the header file.

Table 2-4. Standard Library - Additional Functions

| Description                                  | Prototype                                                                                                                                                                                                                                                                                                                               |
|----------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Average                                      | <code>int avg (int a, int b);</code><br><code>long lavg (long a, long b);</code>                                                                                                                                                                                                                                                        |
| Clip                                         | <code>int clip (int a, int b);</code><br><code>long lclip (long a, long b);</code>                                                                                                                                                                                                                                                      |
| Maximum                                      | <code>int max (int a, int b);</code><br><code>long lmax (long a, long b);</code>                                                                                                                                                                                                                                                        |
| Minimum                                      | <code>int min (int a, int b);</code><br><code>long lmin (long a, long b);</code>                                                                                                                                                                                                                                                        |
| Multiple heaps for dynamic memory allocation | <code>void *heap_calloc(int heap_index, size_t nelem, size_t size);</code><br><code>void heap_free(int heap_index, void *ptr);</code><br><code>void *heap_malloc(int heap_index, size_t size);</code><br><code>void *heap_realloc(int heap_index, void *ptr, size_t size);</code><br><code>int set_alloc_type(char * heap_name);</code> |



Some functions exist as both integer and floating point. The floating-point functions typically have an `f` prefix. Make sure you use the correct type.

A number of functions, including `abs`, `avg`, `max`, `min`, and `clip`, are implemented via intrinsics (provided the header file has been `#include'd`) that map to single-cycle machine instructions.

 If the header file is not included, the library implementation is used instead—at a considerable loss in efficiency.

## **string.h**

The `string.h` header file contains string handling functions, including `strcpy` and `memcpy`.

## **Using Compiler's Built-In C Library Functions**

The C compiler's intrinsic (built-in) functions are functions that the compiler immediately recognizes and replaces with in-line assembly code instead of a function call. 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. The `cc21k` compiler contains a number of intrinsic built-in functions for efficient access to various features of the hardware.

Built-in functions are recognized for cases where the name begins with the string `__builtin`, and the declared prototype of the function matches the prototype that the compiler expects. Built-in functions are declared in system header files. Include the appropriate header file in your program to use these functions. The normal action of the appropriate include file is to `#define` the normal name as mapping to the built-in form.

Typically, in-line assembly code is faster than an average library routine, and it does not incur the calling overhead. The routines in [Table 2-5](#) are built-in C library functions for the `cc21k` compiler:

If you want to use the C run-time library functions of the same name, compile with the `-no-builtin` compiler switch (see [on page 1-35](#)).

Table 2-5. Compiler Built-in Functions

|                       |                   |                   |
|-----------------------|-------------------|-------------------|
| abs                   | avg               | clip              |
| copysign <sup>1</sup> | copysignf         | fabs <sup>1</sup> |
| fabsf                 | favg <sup>1</sup> | favgf             |
| fclip <sup>1</sup>    | fclipf            | fmax <sup>1</sup> |
| fmaxf                 | fmin <sup>1</sup> | fminf             |
| labs                  | lavg              | lclip             |
| lmax                  | lmin              | max               |
| min                   |                   |                   |

<sup>1</sup> These functions will only be compiled as a built-in function if `double` is the same size as `float`.

For a certain category of library function, the compiler relaxes the normal rule whereby pointers that are passed as arguments must address Data Memory (DM). For functions in this category, any argument that is a pointer may also address Program Memory (PM). When the compiler recognizes that certain arguments reference PM, it generates a call to an appropriate version of the function in the run-time library.

Table 2-6 contains a list of library functions that may be called with pointers to Program Memory. Note that this facility is only available provided that the `-no-builtin` compiler switch has not been specified.

Table 2-6. Dual Memory Capable Functions

|         |           |         |
|---------|-----------|---------|
| atof    | atoi      | atol    |
| memchr  | memcmp    | memcpy  |
| memmove | memset    | modf    |
| modff   | setlocale | strcat  |
| strchr  | strcmp    | strcoll |

Table 2-6. Dual Memory Capable Functions (Cont'd)

|         |         |         |
|---------|---------|---------|
| strcpy  | strcspn | strlen  |
| strncat | strncmp | strncpy |
| strpbrk | strrchr | strspn  |
| strstr  | strtod  | strtok  |
| strtol  | strtoul | strxfrm |

## Abridged C++ Library Support

When in C++ mode, the cc21k compiler can call a large number of functions from the Abridged Library, a conforming subset of C++ library.



C++ is not supported for ADSP-21020 DSPs.

The Abridged Library has two major components: embedded C++ library (EC++) and embedded standard template library (ESTL). The embedded C++ library is a conforming implementation of the embedded C++ library as specified by the Embedded C++ Technical Committee.

This section lists and briefly describes the following components of the Abridged Library:

- [“Embedded C++ Library Header Files” on page 2-22](#)
- [“C++ Header Files for C Library Facilities” on page 2-24](#)
- [“Embedded Standard Template Library Header Files” on page 2-26](#)

For more information on the Abridged Library, see online Help.

### Embedded C++ Library Header Files

The following section provides a brief description of the header files in the embedded C++ library.

#### **complex**

The `complex` header file defines a template class `complex` and a set of associated arithmetic operators. Predefined types include `complex_float` and `complex_long_double`.

This implementation does not support the full set of complex operations as specified by the C++ standard. In particular, it does not support either the transcendental functions or the I/O operators `<<` and `>>`.

The `complex` header and the C library header file `complex.h` refer to two different and incompatible implementations of the complex data type.

#### **exception**

The `exception` header file defines the `exception` and `bad_exception` classes and several functions for exception handling.

#### **fract**

The `fract` header file defines the `fract` data type, which supports fractional arithmetic, assignment, and type-conversion operations. The header file is fully described under [“C++ Fractional Type Support” on page 1-96](#). An example that demonstrates its use appears under [“C++ Programming Examples” on page 1-168](#).

#### **fstream**

The `fstream` header file defines the `filebuf`, `ifstream`, and `ofstream` classes for external file manipulations.

### **omanip**

The `omanip` header file declares several `ostream` manipulators. Each manipulator accepts a single argument.

### **ios**

The `ios` header file defines several classes and functions for basic `ostream` manipulations. Note that most of the `ostream` header files include `ios.h`.

### **iosfwd**

The `iosfwd` header file declares forward references to various `ostream` template classes defined in other standard header files.

### **ostream**

The `ostream` header file declares most of the `ostream` objects used for the standard stream manipulations.

### **istream**

The `istream` header file defines the `istream` class for `ostream` extractions. Note that most of the `ostream` header files include `istream.h`.

### **new**

The `new` header file declares several classes and functions for memory allocations and deallocations.

### **ostream**

The `ostream` header file defines the `ostream` class for `ostream` insertions.

## C and C++ Run-Time Libraries Guide

### **sstream**

The `sstream` header file defines the `stringbuf`, `istringstream`, and `ostringstream` classes for various string object manipulations.

### **stdexcept**

The `stdexcept` header file defines a variety of classes for exception reporting.

### **streambuf**

The `streambuf` header file defines the `streambuf` classes for basic operations of the `iostream` classes. Note that most of the `iostream` header files include `streambuf.h`.

### **string**

The `string` header file defines the `string` template and various supporting classes and functions for string manipulations.



Objects of the `string` type should not be confused with the NULL-terminated C strings.

### **strstream**

The `strstream` header file defines the `strstreambuf`, `istrstream`, and `ostrstream` classes for `iostream` manipulations on allocated, extended, and freed character sequences.

## **C++ Header Files for C Library Facilities**

For each C standard library header there is a corresponding standard C++ header. If the name of a C standard library header file is `foo.h`, then the name of the equivalent C++ header file will be `cfoo`. For example, the C++ header file `<cstdio>` provides the same facilities as the C header file `<stdio.h>`.



[Table 2-7](#) lists the C++ header files that provide access to the C library facilities.

Normally, the C standard headers files may be used to define names in the C++ global namespace while the equivalent C++ header files define names in the standard namespace. However, the standard namespace is not supported in this release of the compiler, and the effect of including one of the C++ header files listed in [Table 2-7](#) is the same as including the equivalent C standard library header file.

Table 2-7. C++ Header Files for C Library Facilities

| Header               | Description                                     |
|----------------------|-------------------------------------------------|
| <code>cassert</code> | Enforces assertions during function executions  |
| <code>cctype</code>  | Classifies characters                           |
| <code>cerrno</code>  | Tests error codes reported by library functions |
| <code>cfloat</code>  | Tests floating-point type properties            |
| <code>climits</code> | Tests integer type properties                   |
| <code>locale</code>  | Adapts to different cultural conventions        |
| <code>cmath</code>   | Provides common mathematical operations         |
| <code>csetjmp</code> | Executes non-local goto statements              |
| <code>csignal</code> | Controls various exceptional conditions         |
| <code>cstdarg</code> | Accesses a variable number of arguments         |
| <code>cstddef</code> | Defines several useful data types and macros    |
| <code>cstdio</code>  | Performs input and output                       |
| <code>cstdlib</code> | Performs a variety of operations                |
| <code>cstring</code> | Manipulates several kinds of strings            |

### Embedded Standard Template Library Header Files

Templates and the associated header files are not part of the embedded C++ standard, but they are supported by the `cc21k` compiler in C++ mode. The embedded standard template library header files are:

#### **algorithm**

The `algorithm` header file defines numerous common operations on sequences.

#### **deque**

The `deque` header file defines a deque template container.

#### **functional**

The `functional` header file defines numerous function objects.

#### **hash\_map**

The `hash_map` header file defines two hashed map template containers.

#### **hash\_set**

The `hash_set` header file defines two hashed set template containers.

#### **iterator**

The `iterator` header file defines common iterators and operations on iterators.

#### **list**

The `list` header file defines a list template container.

### **map**

The `map` header file defines two map template containers.

### **memory**

The `memory` header file defines facilities for managing memory.

### **numeric**

The `numeric` header file defines several numeric operations on sequences.

### **queue**

The `queue` header file defines two queue template container adapters.

### **set**

The `set` header file defines two set template containers.

### **stack**

The `stack` header file defines a stack template container adapter.

### **utility**

The `utility` header file defines an assortment of utility templates.

### **vector**

The `vector` header file defines a vector template container.

The Embedded C++ library also includes several header files for compatibility with traditional C++ libraries:

## C and C++ Run-Time Libraries Guide

### **fstream.h**

The `fstream.h` header file defines several `iostream` template classes that manipulate external files.

### **iomanip.h**

The `iomanip.h` header file declares several `istream`s manipulators that take a single argument.

### **iostream.h**

The `iostream.h` header file declares the `iostream` objects that manipulate the standard streams.

### **new.h**

The `new.h` header file declares several functions that allocate and free storage.

## C Run-Time Library Reference

The C run-time library is a collection of functions that you can call from your C programs. This section lists the functions in alphabetical order.



The information that follows applies to all of the functions in the library.

### Notation Conventions

An interval of numbers is indicated by the minimum and maximum, separated by a comma, and enclosed in two square brackets, two parentheses, or one of each. A square bracket indicates that the endpoint is included in the set of numbers; a parenthesis indicates that the endpoint is not included.

### Reference Format

Each function in the library has a reference page. These pages have the following format:

**Name** and Purpose of the function

**Synopsis**—Required header file and functional prototype

**Description**—Function specification

**Error Conditions**—How the function indicates an error

**Example**—Typical function usage

**See Also**—Related functions

### **abort**

abnormal program end

#### **Synopsis**

```
#include <stdlib.h>
void abort (void);
```

#### **Description**

The `abort` function causes an abnormal program termination by raising the SIGABRT exception. 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 integer argument.

**Note:** `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](#), [fabsf](#), [labs](#)

### **acos, acosf**

arc cosine

#### **Synopsis**

```
#include <math.h>
double acos (double x);
float acosf (float x);
```

#### **Description**

The `acos` and `acosf` functions return the arc cosine of  $x$ . The input must be in the range  $[-1, 1]$ . The output, in radians, is in the range  $[0, \pi]$ .

The `acos` and `acosf` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of  $2^{-20}$  over its input range.

#### **Error Conditions**

The `acos` and `acosf` functions indicate 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 x;
float y;

y = acos (0.0);      /* y =  $\pi/2$  */
x = acosf (0.0);     /* x =  $\pi/2$  */
```

#### **See Also**

[cos, cosf](#)



**asin, asinf**

arc sine

**Synopsis**

```
#include <math.h>
double asin (double x);
float asinf (float x);
```

**Description**

The `asin` and `asinf` functions return the arc sine of the first 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` and `asinf` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of  $2^{-20}$  over its input range.

**Error Conditions**

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

**Example**

```
#include <math.h>
double y;
float x;

y = asin (1.0);           /* y =  $\pi/2$  */
x = asinf (1.0);          /* x =  $\pi/2$  */
```

**See Also**

[sin, sinf](#)

### atan, atanf

arc tangent

#### Synopsis

```
#include <math.h>
double atan (double x);
float atanf (float x);
```

#### Description

The `atan` and `atanf` functions return the arc tangent of the first argument. The output, in radians, is in the range  $-\pi/2$  to  $\pi/2$ .

The `atan` and `atanf` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of  $2^{-20}$  over its input range.

#### Error Conditions

The `atan` and `atanf` functions do not return error conditions.

#### Example

```
#include <math.h>
double y;
float x;

y = atan (0.0);      /* y = 0.0 */
x = atanf (0.0);     /* x = 0.0 */
```

#### See Also

[atan2](#), [atan2f](#), [tan](#), [tanf](#)

**atan2, atan2f**

arc tangent of quotient

**Synopsis**

```
#include <math.h>
double atan2 (double x, double y);
float atan2f (float x, float y);
```

**Description**

The `atan2` and `atan2f` functions compute the arc tangent of the input value `x` divided by input value `y`. The output, in radians, is in the range  $-\pi$  to  $\pi$ .

The `atan2` and `atan2f` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of  $2^{-20}$  over its input range.

**Error Conditions**

The `atan2` and `atan2f` functions return a zero and set `errno` to `EDOM` if `x=0` and `y <> 0`.

**Example**

```
#include <math.h>
double a;
float b;

a = atan2 (0.0, 0.5);      /* the error condition: a = 0.0 */
b = atan2f (1.0, 0.0);    /* b =  $\pi/2$  */
```

**See Also**

[atan](#), [atanf](#), [tan](#), [tanf](#)

### 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 nonzero value if the function cannot be registered.

#### Example

```
#include <stdlib.h>
extern void goodbye(void);

if (atexit(goodbye))
    exit(1);
```

#### See Also

[abort](#), [exit](#)

## atof

convert string to a double

### Synopsis

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

### Description

The `atof` function converts a character string 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. This function requires the use of the compiler's `-double-size-64` switch.

### 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](#)

### atoi

convert string to integer

#### Synopsis

```
#include <stdlib.h>
int atoi (const char *nptr);
```

#### Description

The `atoi` function converts a character string to an integer value. The character string to be converted is pointed to by the input pointer, `nptr`. The function clears any leading characters for which `isspace` would return true. Conversion begins at the first digit (with an optional preceding sign) and terminates at the first non-digit.

#### Error Conditions

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

#### Example

```
#include <stdlib.h>
int i;

i = atoi ("5");    /* i == 5 */
```

#### See Also

[atof](#), [atol](#), [strtod](#), [strtoul](#), [strtol](#)

## 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

[atoi](#), [atof](#), [strtod](#), [strtol](#), [strtoul](#)

## C Run-Time Library Reference

### avg

returns mean of two values

#### Synopsis

```
#include <stdlib.h>
int avg (int x, int y);
```

#### Description

This function is an Analog Devices extension to the ANSI standard.

The avg function adds two arguments and divides the result by two. The avg function is a built-in function which is implemented with an  $R_n = (R_x + R_y) / 2$  instruction.

#### Error Conditions

The avg function does not return an error code.

#### Example

```
#include <stdlib.h>
int i;

i = avg (10, 8);    /* returns 9 */
```

#### See Also

[lavg](#)



## **bsearch**

perform binary search in a sorted array

### **Synopsis**

```
#include <stdlib.h>
void *bsearch (const void *key, const void *base,
               size_t nelem, size_t size,
               int (*compare)(const void *, const void *));
```

### **Description**

The `bsearch` function executes a binary search operation on a pre-sorted array, where:

- `key` is a pointer to the element to search for.
- `base` points to the start of the array.
- `nelem` is the number of elements in the array.
- `size` is the size of each element of the array.
- `*compare` points to the function used to compare two elements. It takes as parameters a pointer to the key and a pointer to an array element. The function should return a value less than, equal to, or greater than zero according to whether the first parameter is less than, equal to, or greater than the second.

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

### **Error Conditions**

The `bsearch` function returns a null pointer if the key is not found in the array.

## 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 dynamically allocates a range of memory and initializes all locations 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.

The object is allocated from the current heap, which is the default heap unless `set_alloc_type` has been called to change the current heap to an alternate heap.

### **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](#), [heap\\_calloc](#), [heap\\_free](#), [heap\\_lookup\\_name](#), [heap\\_malloc](#), [heap\\_realloc](#), [malloc](#), [realloc](#), [set\\_alloc\\_type](#)

### ceil, ceilf

ceiling

#### Synopsis

```
#include <math.h>
double ceil (double x);
float ceilf (float x);
```

#### Description

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

#### Error Conditions

The `ceil` and `ceilf` functions do not return an error condition.

#### Example

```
#include <math.h>
double y;
float x;

y = ceil (1.05);      /* y = 2.0 */
x = ceilf (-1.05);    /* y = -1.0 */
```

#### See Also

[floor, floorf](#)

## **circindex**

perform circular buffer operation on loop index

### **Synopsis**

```
#include <21020.h>
#include <21060.h>
#include <210651.h>
#include <21160.h>
#include <21161.h>
```

```
int circindex(int index, int incr, int num_items);
```

### **Description**

The `circindex` function is used within a loop in order to implement a circular buffer operation in C/C++. When optimization is enabled, the operation will be implemented using the appropriate hardware features (B registers and L registers) of the SHARC DSP architecture. The `circindex` function is used to increment or decrement an index in a loop and this index should be used to access memory locations.

The argument `index` represents the index variable, `incr` represents the value by which the index should be incremented on each iteration, and `num_items` represents the size of the circular buffer.



It is not currently possible to perform circular buffer operations in SIMD mode.

### **Error Conditions**

The `circindex` function does not return an error code.

### **Example**

```
#include <21060.h>
#include <stdio.h>
```

## C Run-Time Library Reference

```
int x[10] = {1,2,3,4,5,6,7,8,9,10};
int y[10] = {2,3,4,5,6,7,8,9,10,11};

int dot (const int *a, const int *b)
{
    int i, ci = 0;
    long s = 0;

    /* This will calculate the product for the first 5 elements
     * in each array only. As the loop count is 10, each sum will
     * be calculated twice.
     * Note that each array is indexed using 'ci'. */

    for (i = 0; i < 10; i++) {
        s += a[ci] * b[ci];
        ci = circindex(ci, 1, 5);    // Increment the index
    }

    return s;
}

void
main()
{
    int result;
    result = dot(x,y);
    printf("Result is %d\n", result);    // Result is 140
}
```

### See Also

[circptr](#)

## circptr

perform circular buffer operation on a pointer

### Synopsis

```
#include <21020.h>
#include <21060.h>
#include <210651.h>
#include <21160.h>
#include <21161.h>
```

```
void* circptr(void * ptr, size_t incr, void * base, size_t buflen);
```

### Description

The `circptr` function is used within a loop in order to implement a circular buffer operation in C/C++. When optimization is enabled, the operation will be implemented using the appropriate hardware features (B registers and L registers) of the SHARC DSP architecture. The `circptr` function is used to increment or decrement a pointer variable in a loop.

The argument `ptr` represents the pointer that is being used for the circular buffer, `incr` represents the value by which the circular buffer should be incremented, `base` represents the array on which the circular buffer will operate, and `buflen` represents the size of the circular buffer.



It is not currently possible to perform circular buffer operations in SIMD mode.

### Error Conditions

The `circptr` function does not return an error code.

### Example

```
#include <21060.h>
#include <stdio.h>
```

## C Run-Time Library Reference

```
int x[10] = {1,2,3,4,5,6,7,8,9,10};
int y[10] = {2,3,4,5,6,7,8,9,10,11};

int dot (const int *a, const int *b)
{
    int i, ci = 0;
    long s = 0;
    const int *cba, *cbb;

    /* This will calculate the product for the first 5 elements
     * in each array only. As the loop count is 10,each sum will be
     * calculated twice.
     * Note that each array is indexed using 'ci'.*/

    cba = a;
    cbb = b;
    for (i = 0; i < 10; i++) {
        s += *cba * *cbb;
        cba = circptr(cba, 1, a, 5);    // Increment cba
        cbb = builtin_circptr(cbb, 1, b, 5);    // Increment cbb
    }

    return s;
}

void
main()
{
    int result;
    result = dot(x,y);
    printf("Result is %d\n", result);    // Result is 140
}
```

### See Also

[circindex](#)



## clear\_interrupt

clear a pending signal

### Synopsis

```
#include <signal.h>
int clear_interrupt (int sig);
```

### Description

This function is an Analog Devices extension to the ANSI standard.

The `clear_interrupt` function clears the signal `sig` in the `IRPTL` register. [Table 2-8](#), [Table 2-9 on page 2-50](#), and [Table 2-10 on page 2-51](#) show `sig` arguments of the processor signals for appropriate ADSP-21xxx DSPs.

The `clear_interrupt` function does not work for interrupts that set any status bits in the `STKY` register, such as floating-point overflow.

Table 2-8. ADSP-21020 DSP Signals

| SIG Value | Description                                          |
|-----------|------------------------------------------------------|
| SIG_SOVF  | Status stack or Loop stack overflow or PC stack full |
| SIG_TMZ0  | Timer = 0 (high priority option)                     |
| SIG_IRQ3  | Interrupt 3                                          |
| SIG_IRQ2  | Interrupt 2                                          |
| SIG_IRQ1  | Interrupt 1                                          |
| SIG_IRQ0  | Interrupt 0                                          |
| SIG_CB7   | Circular buffer 7 overflow                           |
| SIG_CB15  | Circular buffer 15 overflow                          |
| SIG_TMZ   | Timer = 0 (low priority option)                      |
| SIG_FIX   | Fixed-point overflow                                 |

## C Run-Time Library Reference

Table 2-8. ADSP-21020 DSP Signals (Cont'd)

|          |                                    |
|----------|------------------------------------|
| SIG_FLTO | Floating point overflow exception  |
| SIG_FLTU | Floating point underflow exception |
| SIG_FLTI | Floating point invalid exception   |
| SIG_USR0 | User software interrupt 0          |
| SIG_USR1 | User software interrupt 1          |
| SIG_USR2 | User software interrupt 2          |
| SIG_USR3 | User software interrupt 3          |
| SIG_USR4 | User software interrupt 4          |
| SIG_USR5 | User software interrupt 5          |

Table 2-9. ADSP-2106x DSP Signals

| SIG Value             | Definition                                            |
|-----------------------|-------------------------------------------------------|
| SIG_SOVF              | Status stack or Loop stack overflow or PC stack full  |
| SIG_TMZ0              | Timer = 0 (high priority option)                      |
| SIG_VIRPTI            | Vector Interrupt                                      |
| SIG_IRQ2              | Interrupt 2                                           |
| SIG_IRQ1              | Interrupt 1                                           |
| SIG_IRQ0              | Interrupt 0                                           |
| SIG_SPR0I             | DMA Channel 0 - SPORT0 Receive                        |
| SIG_SPR1I             | DMA Channel 1 - SPORT1 Receive (or Link Buffer 0)     |
| SIG_SPT0I             | DMA Channel 2 - SPORT0 Transmit                       |
| SIG_SPT1I             | DMA Channel 3 - SPORT1 Transmit (or Link Buffer 1)    |
| <sup>1</sup> SIG_LP2I | DMA Channel 4 - Link Buffer 2                         |
| <sup>1</sup> SIG_LP3I | DMA Channel 5 - Link Buffer 3                         |
| SIG_EP0I              | DMA Channel 6 - Ext. Port Buffer 0 (or Link Buffer 4) |

Table 2-9. ADSP-2106x DSP Signals (Cont'd)

| SIG Value             | Definition                                            |
|-----------------------|-------------------------------------------------------|
| SIG_EP1I              | DMA Channel 7 - Ext. Port Buffer 1 (or Link Buffer 5) |
| <sup>1</sup> SIG_EP2I | DMA Channel 8 - Ext. Port Buffer 2                    |
| <sup>1</sup> SIG_EP3I | DMA Channel 9 - Ext. Port Buffer 3                    |
| <sup>1</sup> SIG_LSRQ | Link port service request                             |
| SIG_CB7               | Circular buffer 7 overflow                            |
| SIG_CB15              | Circular buffer 15 overflow                           |
| SIG_TMZ               | Timer = 0 (low priority option)                       |
| SIG_FIX               | Fixed point overflow                                  |
| SIG_FLTO              | Floating point overflow exception                     |
| SIG_FLTU              | Floating point underflow exception                    |
| SIG_FLTI              | Floating point invalid exception                      |
| SIG_USR0              | User software interrupt 0                             |
| SIG_USR1              | User software interrupt 1                             |
| SIG_USR2              | User software interrupt 2                             |
| SIG_USR3              | User software interrupt 3                             |

<sup>1</sup> Signal is not present on the ADSP-21061 and ADSP-21065L processors.

Table 2-10. ADSP-2116x DSP Signals

| SIG Value  | Definition                                           | DSP Restrictions |
|------------|------------------------------------------------------|------------------|
| SIG_IICDI  | Illegal input condition detected                     |                  |
| SIG_SOVF   | Status stack or Loop stack overflow or PC stack full |                  |
| SIG_TMZ0   | Timer = 0 (high priority option)                     |                  |
| SIG_VIRPTI | Vector interrupt                                     |                  |
| SIG_IRQ2   | Interrupt 2                                          |                  |

## C Run-Time Library Reference

Table 2-10. ADSP-2116x DSP Signals (Cont'd)

| SIG Value | Definition       | DSP Restrictions |
|-----------|------------------|------------------|
| SIG_IRQ1  | Interrupt 1      |                  |
| SIG_IRQ0  | Interrupt 0      |                  |
|           |                  |                  |
| SIG_SPR0I | SPORT0 Receive   | 21160 only       |
| SIG_SPR1I | SPORT1 Receive   | 21160 only       |
| SIG_SPT0I | SPORT0 Transmit  | 21160 only       |
| SIG_SPT1I | SPORT0 Transmit  | 21160 only       |
|           |                  |                  |
| SIG_SP0I  | SPORT0 DMA       | 21161 only       |
| SIG_SP1I  | SPORT1 DMA       | 21161 only       |
| SIG_SP2I  | SPORT2 DMA       | 21161 only       |
| SIG_SP3I  | SPORT3 DMA       | 21161 only       |
|           |                  |                  |
| SIG_LP0I  | Link Buffer 0    |                  |
| SIG_LP1I  | Link Buffer 1    |                  |
|           |                  |                  |
| SIG_LP2I  | Link Buffer 2    | 21160 only       |
| SIG_LP3I  | Link Buffer 3    | 21160 only       |
| SIG_LP4I  | Link Buffer 4    | 21160 only       |
| SIG_LP5I  | Link Buffer 5    | 21160 only       |
|           |                  |                  |
| SIG_SPIRI | SPI Receive DMA  | 21161 only       |
| SIG_SPITI | SPI Transmit DMA | 21161 only       |
|           |                  |                  |

Table 2-10. ADSP-2116x DSP Signals (Cont'd)

| SIG Value | Definition                         | DSP Restrictions |
|-----------|------------------------------------|------------------|
| SIG_EP0I  | Ext. Port Buffer 0                 |                  |
| SIG_EP1I  | Ext. Port Buffer 1                 |                  |
| SIG_EP2I  | Ext. Port Buffer 2                 |                  |
| SIG_EP3I  | Ext. Port Buffer 3                 |                  |
| SIG_LSRQ  | Link port service request          |                  |
| SIG_CB7   | Circular buffer 7 overflow         |                  |
| SIG_CB15  | Circular buffer 15 overflow        |                  |
| SIG_TMZ   | Timer = 0 (low priority option)    |                  |
| SIG_FIX   | Fixed-point overflow               |                  |
| SIG_FLTO  | Floating-point overflow exception  |                  |
| SIG_FLTU  | Floating-point underflow exception |                  |
| SIG_FLTI  | Floating-point invalid exception   |                  |
| SIG_USR0  | User software interrupt 0          |                  |
| SIG_USR1  | User software interrupt 1          |                  |

## Error Conditions

The `clear_interrupt` function returns a 1 if the interrupt was pending; otherwise 0 is returned.

## Example

```
#include <signal.h>
clear_interrupt (SIG_IRQ2);
/* clear the interrupt 2 latch */
```

## See Also

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

### clip

clip  $\times$  by  $y$

#### Synopsis

```
#include <stdlib.h>
int clip (int value1, int value2);
```

#### Description

This function is an Analog Devices extension to the ANSI standard.

The `clip` function returns its first argument if it is less than the absolute value of its second argument, otherwise it returns the absolute value of its second argument if the first is positive, or minus the absolute value if the first argument is negative. The `clip` function is a built-in function which is implemented with an `Rn = CLIP Rx BY Ry` instruction.

#### Error Conditions

The `clip` function does not return an error code.

#### Example

```
#include <stdlib.h>
int i;

i = clip (10, 8);    /* returns 8 */
i = clip (8, 10);    /* returns 8 */
i = clip (-10, 8);   /* returns -8 */
```

#### See Also

[fclip](#), [fclipf](#)

**cos, cosf**

cosine

**Synopsis**

```
#include <math.h>
double cos (double x);
float cosf (float x);
```

**Description**

The `cos` and `cosf` functions return the cosine of the first argument. The input is interpreted as radians; the output is in the range  $[-1, 1]$ .

The `cos` and `cosf` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of  $2^{-20}$  over its input range. Although the `cos` and `cosf` functions accept input over the entire floating-point range, the accuracy of the result decreases significantly for an input greater than  $\pi^{12}/2$ .

**Error Conditions**

The `cos` and `cosf` functions do not return an error condition.

**Example**

```
#include <math.h>
double y;
float x;

y = cos (3.14159);    /* y = -1.0 */
x = cosf (3.14159);  /* x = -1.0 */
```

**See Also**

[acos](#), [acosf](#), [sin](#), [sinf](#)

### cosh, coshf

hyperbolic cosine

#### Synopsis

```
#include <math.h>
double cosh (double x);
float coshf (float x);
```

#### Description

The `cosh` and `coshf` functions return the hyperbolic cosine of their argument.

The `cosh` and `coshf` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of  $2^{-20}$  over its input range.

#### Error Conditions

The `cosh` and `coshf` functions return `HUGE_VAL` and set `errno` to `ERANGE` if the input exceeds  $2^{12}$ .

#### Example

```
#include <math.h>
double x, y;
float v, w;

y = cosh (x);
v = coshf (w);
```

#### See Also

[sinh, sinh](#)



**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`, then

```
result.quot * denom + result.rem == numer
```

**Error Conditions**

If `denom` is zero, the behavior of the `div` function is undefined.

**Example**

```
#include <stdlib.h>
div_t result;

result = div (5, 2);    /* result.quot = 2, result.rem = 1 */
```

**See Also**

[fmod](#), [fmodf](#), [ldiv](#), [modf](#), [modff](#)

### 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 `R0`, and control is passed to the label `__lib_prog_term`, which is defined in the run-time startup file.

#### Error Conditions

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

#### Example

```
#include <stdlib.h>

exit (EXIT_SUCCESS);
```

#### See Also

[abort](#), [atexit](#)

## **exp, expf**

exponential

### **Synopsis**

```
#include <math.h>
double exp (double x);
float expf (float x);
```

### **Description**

The `exp` and `expf` functions compute the exponential value  $e$  to the power of its argument.

### **Error Conditions**

For underflow errors the `exp` and `expf` functions return zero.

### **Example**

```
#include <math.h>
double y;
float x;

y = exp (1.0);    /* y = 2.71828 */
x = expf (1.0);   /* x = 2.71828 */
```

### **See Also**

[log](#), [logf](#), [pow](#), [powf](#)

### **fabs, fabsf**

absolute value

#### **Synopsis**

```
#include <math.h>
double fabs (double x);
float fabsf (float x);
```

#### **Description**

The `fabs` and `fabsf` functions return the absolute value of the argument.

#### **Error Conditions**

The `fabs` and `fabsf` functions do not return error conditions.

#### **Example**

```
#include <math.h>
double y;
float x;

y = fabs (-2.3);    /* y = 2.3 */
y = fabs (2.3);    /* y = 2.3 */
x = fabsf (-5.1);  /* x = 5.1 */
```

#### **See Also**

[abs](#), [labs](#)

## floor, floorf

floor

### Synopsis

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

### Description

The `floor` and `floorf` functions return the largest integral value that is not greater than their argument.

### Error Conditions

The `floor` and `floorf` functions do not return error conditions.

### Example

```
#include <math.h>
double y;
float z;

y = floor (1.25);    /* y = 1.0 */
y = floor (-1.25);   /* y = -2.0 */
z = floorf (10.1);   /* z = 10.0 */
```

### See Also

[ceil, ceilf](#)

### **fmod, fmodf**

floating-point modulus

#### **Synopsis**

```
#include <math.h>
double fmod (double x, double y);
float fmodf (float x, float y);
```

#### **Description**

The `fmod` and `fmodf` functions compute the floating-point remainder that results from dividing the first argument by the second argument.

The result is less than the second argument and has the same sign as the first argument. If the second argument is equal to zero, `fmod` and `fmodf` return zero.

#### **Error Conditions**

The `fmod` and `fmodf` functions do not return an error condition.

#### **Example**

```
#include <math.h>
double y;
float x;

y = fmod (5.0, 2.0);    /* y = 1.0 */
x = fmodf (4.0, 2.0);  /* x = 0.0 */
```

#### **See Also**

[div](#), [ldiv](#), [modf](#), [modff](#)

## 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`, `realloc`, `heap_calloc`, `heap_malloc`, or `heap_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](#), [heap\\_calloc](#), [heap\\_free](#), [heap\\_lookup\\_name](#), [heap\\_malloc](#),  
[heap\\_realloc](#), [malloc](#), [realloc](#), [set\\_alloc\\_type](#)

### **frexp, frexpf**

separate fraction and exponent

#### **Synopsis**

```
#include <math.h>
double frexp (double x, int *exp_ptr);
float frexpf (float x, int *exp_ptr);
```

#### **Description**

The `frexp` and `frexpf` functions separate a floating-point input into a normalized fraction and a (base 2) exponent. The functions return a fraction in the interval  $[\frac{1}{2}, 1)$ , and store a power of 2 in the integer pointed to by the second argument. If the input is zero, then both the value stored and the value returned are zero.

#### **Error Conditions**

The `frexp` and `frexpf` functions do not return an error condition.

#### **Example**

```
#include <math.h>
double y;
float x;
int exponent;

y = frexp (2.0, &exponent);    /* y = 0.5, exponent = 2 */
x = frexpf (4.0, &exponent);   /* x = 0.5, exponent = 3 */
```

#### **See Also**

[modf, modff](#)



## getenv

get string definition from operating system

### Synopsis

```
#include <stdlib.h>
char *getenv (const char *name);
```

### Description

The `getenv` function polls the operating system to see if a string is defined. There is no default operating system for the SHARC DSPs, so `getenv` always returns NULL.

### Error Conditions

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

### Example

```
#include <stdlib.h>
char *ptr;

ptr = getenv ("ADI_DSP");    /* ptr = NULL */
```

### See Also

[system](#)

### heap\_calloc

allocate and initialize memory in a heap

#### Synopsis

```
#include <stdlib.h>
void *heap_calloc(int heap_index, size_t nelem, size_t size);
```

#### Description

This function is an Analog Devices extension to the ANSI standard.

The `heap_calloc` function allocates from the heap identified by `heap_index`, an array containing `nelem` elements of `size`, and stores zeros in all bytes of the array. If successful, it returns a pointer to this array; otherwise, it returns a null pointer. You can safely convert the return value to an object pointer of any type whose size in bytes is not greater than `size`. The memory may be deallocated with the `free` or `heap_free` function.

For more information on creating multiple run-time heaps, see [“Using Multiple Heaps” on page 1-130](#).

#### Error Conditions

The `heap_calloc` function returns the null pointer if unable to allocate the requested memory.

#### Example

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
    char *buf;
    int index;
    /* Obtain the heap index for “seg_hp2” */
    index = heap_lookup_name(“seg_hp2”);
```

```
if (index < 0) {
    printf("Heap with name seg_hp2 not found\n");
    return 1;
}

/* Allocate memory for 128 characters from seg_hp2 */
buf = (char *)heap_calloc(index,128,sizeof(char));
if (buf != 0) {
    printf("Allocated space from %p\n", buf);
    free(buf);    /* free can be used to release the memory */
} else {
    printf("Unable to allocate from seg_hp2\n");
}
return 0;
}
```

### See Also

[calloc](#), [free](#), [heap\\_free](#), [heap\\_lookup\\_name](#), [heap\\_malloc](#), [heap\\_realloc](#),  
[malloc](#), [realloc](#), [set\\_alloc\\_type](#)

### heap\_free

return memory to a heap

#### Synopsis

```
#include <stdlib.h>
void heap_free(int heap_index, void *ptr);
```

#### Description

This function is an Analog Devices extension to the ANSI standard.

If `ptr` is not a null pointer, the `heap_free` function deallocates the object whose address is `ptr`; otherwise, it does nothing. The argument `heap_index` must be the index of the heap from which the object pointed to by `ptr` was originally allocated. If the object was not allocated from the specified heap, then the behavior is undefined.

The `heap_free` function is somewhat faster than `free`, but `free` must be used if the heap from which the object was allocated is not known with certainty.

For more information on creating multiple run-time heaps, see [“Using Multiple Heaps” on page 1-130](#).

#### Error Conditions

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

#### Example

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
    char *buf;
    int index;
```

```
/* Obtain the heap index for "seg_hp2" */
index = heap_lookup_name("seg_hp2");
if (index < 0) {
    printf("Heap with name seg_hp2 not found\n");
    return 1;
}

/* Allocate memory for 128 characters from seg_hp2 */
buf = (char *)heap_calloc(index,128,sizeof(char));
if (buf != 0) {
    printf("Allocated space from %p\n", buf);
    heap_free(index, buf);    /* heap_free can be used */
                             /* to release the memory */
} else {
    printf("Unable to allocate from seg_hp2\n");
}
return 0;
}
```

**See Also**

[calloc](#), [free](#), [heap\\_calloc](#), [heap\\_lookup\\_name](#), [heap\\_malloc](#), [heap\\_realloc](#), [malloc](#), [realloc](#), [set\\_alloc\\_type](#)

### heap\_lookup\_name

obtain primary heap identifier

#### Synopsis

```
#include <stdlib.h>
int heap_lookup_name(char * user_id);
```

#### Description

This function is an Analog Devices extension to the ANSI standard.

The `heap_lookup_name` function returns the primary heap identifier of the heap with user identifier `user_id`, if there is such a heap; otherwise, -1 is returned. The primary heap identifier is the index of the heap descriptor record in the heap descriptor table. The user identifier for a heap is determined by a field in the heap descriptor record. The default heap always has user identifier 0.

For more information on multiple run-time heaps, see [“Using Multiple Heaps” on page 1-130](#).

#### Error Conditions

The function returns -1 if the specified user identifier was not found, otherwise it returns the primary heap identifier of the specified heap.

#### Example

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

void func2(int pm * b);

func()
{
    int pm * x;
```

```
int loop, pm_heapID;

pm_heapID = heap_lookup_name("seg_heap");

if (pm_heapID < 0) {
    printf("Lookup failed\n");
    return 1;
}

x = (int pm *)heap_malloc(pm_heapID, 1000);
                        // Get 1K words of PM heap space
if (x == NULL) {
    printf("heap_malloc failed\n");
    return 1;
}

for (loop = 0; loop < 1000; loop++)
    x[loop] = loop;

func2(x);              // Do something with x
}
```

**See Also**

[calloc](#), [free](#), [heap\\_calloc](#), [heap\\_free](#), [heap\\_malloc](#), [heap\\_realloc](#), [malloc](#), [realloc](#), [set\\_alloc\\_type](#)

### heap\_malloc

allocate memory from a heap

#### Synopsis

```
#include <stdlib.h>
void *heap_malloc(int heap_index, size_t size);
```

#### Description

This function is an Analog Devices extension to the ANSI standard.

The `heap_malloc` function allocates an object of `size` from the heap identified by `heap_index`. It returns the address of the object if successful; otherwise, it returns a null pointer. You can safely convert the return value to an object pointer of any type whose size in bytes is not greater than `size`.

The block of memory is uninitialized. The memory may be deallocated with the `free` or `heap_free` function.

For more information on creating multiple run-time heaps, see [“Using Multiple Heaps” on page 1-130](#).

#### Error Conditions

The `heap_malloc` function returns the null pointer if unable to allocate the requested memory.

#### Example

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
    char *buf;
    int index;
```



```
/* Obtain the heap index for "seg_hp2" */
index = heap_lookup_name("seg_hp2");
if (index < 0) {
    printf("Heap with name seg_hp2 not found\n");
    return 1;
}

/* Allocate memory for 128 characters from seg_hp2 */
buf = (char *)heap_malloc(index,128);
if (buf != 0) {
    printf("Allocated space from %p\n", buf);
    free(buf); /* free can be used to release the memory */
} else {
    printf("Unable to allocate from seg_hp2\n");
}
return 0;
}
```

**See Also**

[calloc](#), [free](#), [heap\\_calloc](#), [heap\\_free](#), [heap\\_lookup\\_name](#), [heap\\_realloc](#),  
[malloc](#), [realloc](#), [set\\_alloc\\_type](#)

### heap\_realloc

change memory allocation from a heap

#### Synopsis

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

#### Description

This function is an Analog Devices extension to the ANSI standard.

The `heap_realloc` function allocates from the heap, identified by `heap_index`, an object of `size`, obtaining initial stored values from the object whose address is `ptr`. It returns the address of the object if successful; otherwise, it returns a null pointer. You can safely convert the return value to an object pointer of any type whose size in bytes is not greater than `size`.

If `ptr` is not a null pointer, it must be the address of an existing object that you first allocate by calling `calloc`, `malloc`, `realloc`, `heap_calloc`, `heap_malloc`, or `heap_realloc`. The heap identified by `heap_index` must be the same as the heap from which the object was originally allocated; if it is not the same, the behavior is undefined. If the existing object is not larger than the newly allocated object, `heap_realloc` copies the entire existing object to the initial part of the allocated object. (The values stored in the remainder of the object are indeterminate.)

Otherwise, the function copies only the initial part of the existing object that fits in the allocated object. If `heap_realloc` succeeds in allocating a new object, it deallocates the existing object. Otherwise, the existing object is left unchanged.

If `ptr` is a null pointer, the values stored in the newly created object are indeterminate.

If `size` is 0 and `ptr` is not a null pointer, the object pointed to by `ptr` is deallocated and a null pointer is returned. However, if the object was originally allocated from a different heap from the heap identified by `heap_index`, the behavior is undefined.

The `heap_realloc` function is somewhat faster than `realloc` for resizing or deallocating an existing object, but `realloc` must be used if the heap from which the object was originally allocated is not known with certainty.

The allocated memory may be deallocated with the `free` or `heap_free` function.

For more information on creating multiple run-time heaps, see [“Using Multiple Heaps” on page 1-130](#).

## Error Conditions

The `heap_realloc` function returns the null pointer if unable to allocate the requested memory.

## Example

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

int main()
{
    int index,ok,prev;
    char *buf,*upd;

    /* Obtain the heap index for the user identifier 2 */
    index = heap_lookup_name("seg_hp2");
    if (index < 0) {
        printf("Heap with name seg_hp2 not found\n");
        return 1;
    }

    /* Allocate memory for 128 characters from seg_hp2 */
```

## C Run-Time Library Reference

```
buf = (char *)heap_malloc(index,128);
if (buf != 0) {
    strcpy(buf,"hello");
    /* Change allocated size to 256 */
    upd = (char *)heap_realloc(index,buf,256);
    if (upd != 0) {
        printf("reallocated string for %s\n",upd);
        heap_free(index,upd);      /* Return to seg_hp2 */
    } else {
        free(buf);      /* free can be used to release buf */
    }
} else {
    printf("Unable to allocate from seg_hp2\n");
}
return 0;
}
```

### See Also

[calloc](#), [free](#), [heap\\_calloc](#), [heap\\_free](#), [heap\\_lookup\\_name](#), [heap\\_malloc](#),  
[malloc](#), [realloc](#), [set\\_alloc\\_type](#)

## interrupt

define interrupt handling

### Synopsis

```
#include <signal.h>
void (*interrupt (int sig, void(*func)(int))) (int);
void (*interruptnsm (int sig, void(*func)(int))) (int);
void (*interruptf (int sig, void(*func)(int))) (int);
void (*interruptfnsnsm (int sig, void(*func)(int))) (int);
void (*interrupts (int sig, void(*func)(int))) (int);
void (*interruptsnsm (int sig, void(*func)(int))) (int);
void (*interruptcb (int sig, void(*func)(int))) (int);
void (*interruptcbnsnsm (int sig, void(*func)(int))) (int);
void (*ininterruptss int sig, void(*func)(int))) (int);
void (*interruptssnsm int sig, void(*func)(int))) (int);
```

### Description

The `interrupt` function determines how a signal received during program execution is handled. The `interrupt` function executes the function pointed to by `func` at every `interrupt sig`; the `signal` function executes the function only once. The `func` argument must be one of the following that are listed in [Table 2-11](#). The `interrupt` function causes the receipt of the signal number `sig` to be handled in one of the following ways:

Table 2-11. Interrupt Handling

| Func Value       | Action                                                    |
|------------------|-----------------------------------------------------------|
| SIG_DFL          | The default action is taken.                              |
| SIG_IGN          | The signal is ignored.                                    |
| Function address | The function pointed to by <code>func</code> is executed. |

## C Run-Time Library Reference

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.

The differences between the functions `interrupt`, `interruptf`, `interrupts`, `interruptcb`, `interruptnsm`, `interruptfnsm`, `interruptsnsm`, `interruptcbnsm`, `interruptss`, and `interruptssnsm` are discussed under [“signal.h” on page 2-10](#).

### 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>

interrupt (SIG_IRQ2, irq2_handler);
/* enable interrupt 2 whose handling routine is pointed to by
irq2_handler */

interrupt (SIG_IRQ2, SIG_IGN);
/* disable interrupt 2 */
```

### See Also

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

## 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, the `isalnum` function returns a zero. If the argument is alphanumeric, `isalnum` returns a nonzero 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 argument is an alphabetic character (A-Z or a-z). If the argument is not alphabetic, `isalpha` returns a zero. If the argument is alphabetic, `isalpha` returns a nonzero 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](#), [islower](#), [isupper](#)



## 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 argument character is a decimal digit (0-9). If the argument is not a digit, `isdigit` returns a zero. If the argument is a digit, `isdigit` returns a non-zero value.

#### Error Conditions

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

#### 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 a 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

The `isinf` function does not return or set any error conditions.

#### Example

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

static int fail=0;

main(){
    /* test int isinf(double) */
    union {
        double d; float f; unsigned long l;
    } u;

    #ifdef __DOUBLES_ARE_FLOATS__
        u.l=0xFF800000L; if ( isinf(u.d)==0 ) fail++;
        u.l=0xFF800001L; if ( isinf(u.d)!=0 ) fail++;
        u.l=0x7F800000L; if ( isinf(u.d)==0 ) fail++;
        u.l=0x7F800001L; if ( isinf(u.d)!=0 ) fail++;
    #endif
```

```
/* test int isinff(float) */
    u.l=0xFF800000L; if ( isinff(u.f)==0 ) fail++;
    u.l=0xFF800001L; if ( isinff(u.f)!=0 ) fail++;
    u.l=0x7F800000L; if ( isinff(u.f)==0 ) fail++;
    u.l=0x7F800001L; if ( isinff(u.f)!=0 ) fail++;

/* print pass/fail message */
if ( fail==0 )
    printf("Test passed\n");
else
    printf("Test failed: %d\n", fail);
}
```

## See Also

[isnan](#)

### islower

detect lowercase character

### Synopsis

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

### Description

The `islower` function determines if the argument is a lowercase character (a-z). If the argument is not lowercase, `islower` returns a zero. If the argument is lowercase, `islower` returns a non-zero value.

### Error Conditions

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

### Example

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

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    printf ("%2s", islower (ch) ? "lowercase" : "");
    putchar ('\n');
}
```

### See Also

[isalpha](#), [isupper](#)

**isnan**

test for not-a-number (NaN)

**Synopsis**

```
#include <math.h>
int isnanf(float x);
int isnan(double x);
```

**Description**

The `isnan` function returns a zero if the argument is not set to an IEEE NaN (Not a Number); otherwise, the function will return a non-zero value.

**Error Conditions**

The `isnan` function does not return or set any error conditions.

**Example**

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

static int fail=0;

main(){
    /* test int isnan(double) */
    union {
        double d; float f; unsigned long l;
    } u;

    #ifdef __DOUBLES_ARE_FLOATS__
        u.l=0xFF800000L; if ( isnan(u.d)!=0 ) fail++;
        u.l=0xFF800001L; if ( isnan(u.d)==0 ) fail++;
        u.l=0x7F800000L; if ( isnan(u.d)!=0 ) fail++;
        u.l=0x7F800001L; if ( isnan(u.d)==0 ) fail++;
    #endif
```

## C Run-Time Library Reference

```
/* 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](#), [isgraph](#)

### isupper

detect uppercase character

#### Synopsis

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

#### Description

The `isupper` function determines if the argument is an uppercase character (A-Z). If the argument is not an uppercase character, `isupper` returns a zero. If the argument is an uppercase character, `isupper` returns a non-zero value.

#### Error Conditions

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

#### Example

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

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    printf ("%2s", isupper (ch) ? "uppercase" : "");
    putchar ('\n');
}
```

#### See Also

[isalpha](#), [islower](#)

## isxdigit

detect hexadecimal digit

### Synopsis

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

### Description

The `isxdigit` function determines if the argument character is a hexadecimal digit character (A-F, a-f, or 0-9).

If the argument is not a hexadecimal digit, `isxdigit` returns a zero. If the argument is a hexadecimal digit, `isxdigit` returns a non-zero value.

### Error Conditions

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

### Example

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

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    printf ("%2s", isxdigit (ch) ? "hexadecimal" : "");
    putchar ('\n');
}
```

### See Also

[isalnum](#), [isdigit](#)

### labs

absolute value

#### Synopsis

```
#include <stdlib.h>
long int labs (long int j);
```

#### Description

The `labs` function returns the absolute value of its integer argument.

**Note:** `labs (LONG_MIN) == LONG_MIN`.

#### 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](#), [fabsf](#)

## lavg

return mean of two values

### Synopsis

```
#include <stdlib.h>
long int lavg (long int value1, long int value2);
```

### Description

This function is an Analog Devices extension to the ANSI standard.

The `lavg` function adds two arguments and divides the result by two. The `lavg` function is a built-in function which is implemented with an `Rn=(Rx+Ry)/2` instruction.

### Error Conditions

The `lavg` function does not return an error code.

### Example

```
#include <stdlib.h>
long int i;
i = lavg (10, 8);    /* returns 9 */
```

### See Also

[avg](#), [favg](#), [favgf](#)

### lclip

clip  $\times$  by  $y$

#### Synopsis

```
#include <stdlib.h>
long int lclip (long int value1, long int value2);
```

#### Description

This function is an Analog Devices extension to the ANSI standard.

The `lclip` function returns its first argument if it is less than the absolute value of its second argument, otherwise it returns the absolute value of its second argument if the first is positive, or minus the absolute value if the first argument is negative. The `lclip` function is a built-in function which is implemented with an `Rn = CLIP Rx BY Ry` instruction.

#### Error Conditions

The `lclip` function does not return an error code.

#### Example

```
#include <stdlib.h>
long int i;

i = lclip (10, 8);    /* returns 8 */
i = lclip (8, 10);   /* returns 8 */
i = lclip (-10, 8);  /* returns -8 */
```

#### See Also

[clip](#), [fclip](#), [fclipf](#)



## ldexp, ldexpf

multiply by power of 2

### Synopsis

```
#include <math.h>
double ldexp (double x, int n);
float ldexpf (float x, int n);
```

### Description

The `ldexp` and `ldexpf` functions return the value of the floating-point argument multiplied by  $2^n$ . The `ldexp` and `ldexpf` functions add the value of `n` to the exponent of `x`.

### Error Conditions

If the result overflows, `ldexp` and `ldexpf` return `HUGE_VAL` with the proper sign and set `errno` to `ERANGE`. If the result underflows, a zero is returned.

### Example

```
#include <math.h>
double y;
float x;

y = ldexp (0.5, 2);    /* y = 2.0 */
x = ldexpf (1.0, 2);   /* x = 4.0 */
```

### See Also

[exp](#), [expf](#), [pow](#), [powf](#)

### **ldiv**

long division

#### **Synopsis**

```
#include <stdlib.h>
ldiv_t ldiv (long int numer, long int denom);
```

#### **Description**

The `ldiv` function divides `numer` by `denom`, and returns a structure of type `ldiv_t`. The type `ldiv_t` is defined as:

```
typedef struct {
    long int quot;
    long int rem;
} ldiv_t;
```

where `quot` is the quotient of the division and `rem` is the remainder, such that if `result` is of type `ldiv_t`, then

```
result.quot * denom + result.rem == numer
```

#### **Error Conditions**

If `denom` is zero, the behavior of the `ldiv` function is undefined.

#### **Example**

```
#include <stdlib.h>
ldiv_t result;

result = ldiv (7L, 2L);    /* result.quot = 3, result.rem = 1 */
```

#### **See Also**

[div](#), [fmod](#), [fmodf](#)

## lmax

return larger of two values

### Synopsis

```
#include <stdlib.h>
long int lmax (long int value1, long int value2);
```

### Description

This function is an Analog Devices extension to the ANSI standard.

The `lmax` function returns the larger of its two arguments. The `lmax` function is a built-in function which is implemented with an `Rn = MAX(Rx, Ry)` instruction.

### Error Conditions

The `lmax` function does not return an error code.

### Example

```
#include <stdlib.h>
long int i;

i = lmax (10, 8);    /* returns 10 */
```

### See Also

[fmax](#), [fmaxf](#), [fmin](#), [fminf](#), [lmax](#), [lmin](#), [max](#), [min](#)

### lmin

return the smaller of two values

### Synopsis

```
#include <stdlib.h>
long int lmin (long int value1, long int value2);
```

### Description

This function is an Analog Devices extension to the ANSI standard.

The `lmin` function returns the smaller of its two arguments. The `lmin` function is a built-in function which is implemented with an `Rn = MIN(Rx,Ry)` instruction.

### Error Conditions

The `lmin` function does not return an error code.

### Example

```
#include <stdlib.h>
long int i;

i = lmin (10, 8);    /* returns 8 */
```

### See Also

[fmax](#), [fmaxf](#), [fmin](#), [fminf](#), [lmax](#), [max](#), [min](#)

## localeconv

get pointer for formatting to current locale

### Synopsis

```
#include <locale.h>
struct lconv *localeconv (void);
```

### Description

The `localeconv` function returns a pointer to an object of type `struct lconv`. This pointer is used to set the components of the object with values used in formatting numeric quantities in the current locale.

With the exception of `decimal_point`, those members of the structure with type `char*` may use “ ” to indicate that a value is not available. Expected values are strings. Those members with type `char` may use `CHAR_MAX` to indicate that a value is not available. Expected values are non-negative numbers.

The program may not alter the structure pointed to by the return value but subsequent calls to `localeconv` may do so. Also, calls to `setlocale` with the category arguments of `LC_ALL`, `LC_MONETARY` and `LC_NUMERIC` may overwrite the structure.

Table 2-12. Members of the `lconv` Struct

| Member                             | Description                                                                                                                                                      |
|------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>char *currency_symbol</code> | Currency symbol applicable to the locale                                                                                                                         |
| <code>char *decimal_point</code>   | Used to format nonmonetary quantities                                                                                                                            |
| <code>char *grouping</code>        | Used to indicate the number of digits in each nonmonetary grouping                                                                                               |
| <code>char *int_curr_symbol</code> | Used as international currency symbol (ISO 4217:1987) for that particular locale plus the symbol used to separate the currency symbol from the monetary quantity |

Table 2-12. Members of the lconv Struct (Cont'd)

| Member                  | Description                                                                                                                                                               |
|-------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| char *mon_decimal_point | Used for decimal point format monetary quantities                                                                                                                         |
| char *mon_grouping      | Used to indicate the number of digits in each monetary grouping                                                                                                           |
| char *mon_thousands_sep | Used to group monetary quantities prior to the decimal point                                                                                                              |
| char *negative_sign     | Used to indicate a negative monetary quantity                                                                                                                             |
| char *positive_sign     | Used to indicate a positive monetary quantity                                                                                                                             |
| char *thousands_sep     | Used to group nonmonetary quantities prior to the decimal point                                                                                                           |
| char frac_digits        | Number of digits displayed after the decimal point in monetary quantities in other than international format                                                              |
| char int_frac_digits    | Number of digits displayed after the decimal point in international monetary quantities                                                                                   |
| char p_cs_precedes      | If set to 1, the currency_symbol precedes the positive monetary quantity. If set to 0, the currency_symbol succeeds the positive monetary quantity.                       |
| char n_cs_precedes      | If set to 1, the currency_symbol precedes the negative monetary quantity. If set to 0, the currency_symbol succeeds the negative monetary quantity.                       |
| char n_sign_posn        | Indicates the positioning of negative_sign for monetary quantities.                                                                                                       |
| char n_sep_by_space     | If set to 1, the currency_symbol is separated from the negative monetary quantity. If set to 0, the currency_symbol is not separated from the negative monetary quantity. |
| char p_sep_by_space     | If set to 1, the currency_symbol is separated from the positive monetary quantity. If set to 0, the currency_symbol is not separated from the positive monetary quantity. |

For `grouping` and `non_grouping`, an element of `CHAR_MAX` indicates that no further grouping will be performed, a 0 indicates that the previous element should be used to group the remaining digits, and any other integer value is used as the number of digits in the current grouping.

The definitions of the values for `p_sign_posn` and `n_sign_posn` are:

- parentheses surround `currency_symbol` and `quantity`
- sign string precedes `currency_symbol` and `quantity`
- sign string succeeds `currency_symbol` and `quantity`
- sign string immediately precedes `currency_symbol`
- sign string immediately succeeds `currency_symbol`

### Error Conditions

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

### Example

```
#include <locale.h>
struct lconv *c_locale;

c_locale = localeconv ();    /* Only the C locale is */
                             /* currently supported */
```

### See Also

[setlocale](#)

### log, logf

natural logarithm

#### Synopsis

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

#### Description

The `log` and `logf` functions compute the natural (base e) logarithm of their argument.

The `log` and `logf` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of  $2^{-20}$  over its input range.

#### Error Conditions

The `log` and `logf` functions return zero and set `errno` to `EDOM` if the input value is zero or negative.

#### Example

```
#include <math.h>
double y;
float x;

y = log (1.0);           /* y = 0.0 */
x = logf (2.71828);      /* x = 1.0 */
```

#### See Also

[exp](#), [expf](#), [log10](#), [log10f](#)



## log10, log10f

base 10 logarithm

### Synopsis

```
#include <math.h>
double log10 (double x);
float log10f (float x);
```

### Description

The `log10` and `log10f` functions produce the base 10 logarithm of their argument.

The `log10` and `log10f` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of  $2^{-20}$  over its input range.

### Error Conditions

The `log10` and `log10f` functions indicate a domain error (set `errno` to `EDOM`) and return zero if the input is zero or negative.

### Example

```
#include <math.h>
double y;
float x;

y = log10 (100.0);    /* y = 2.0 */
x = log10f (10.0);    /* x = 1.0 */
```

### See Also

[log](#), [logf](#), [pow](#), [powf](#)

### 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>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
jmp_buf env;
int res;
```

```
if ((res == setjmp(env)) != 0) {
    printf ("Problem %d reported by func ()", res);
    exit (EXIT_FAILURE);
}
func ();

void func (void)
{
    if (errno != 0) {
        longjmp (env, errno);
    }
}
```

### See Also

[setjmp](#)

## C Run-Time Library Reference

### 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.

The object is allocated from the current heap, which will be the default heap unless `set_alloc_type` has been called to change the current heap to an alternate heap.

#### 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](#), [heap\\_calloc](#), [heap\\_free](#), [heap\\_lookup\\_name](#), [heap\\_malloc](#),  
[heap\\_realloc](#), [realloc](#), [set\\_alloc\\_type](#)

## max

return larger of two values

### Synopsis

```
#include <stdlib.h>
int max (int value1, int value2);
```

### Description

This function is an Analog Devices extension to the ANSI standard.

The `max` function returns the larger of its two arguments. The `max` function is a built-in function which is implemented with an  $R_n = \text{MAX}(R_x, R_y)$  instruction.

### Error Conditions

The `max` function does not return an error code.

### Example

```
#include <stdlib.h>
int i;

i = max (10, 8);    /* returns 10 */
```

### See Also

[fmax](#), [fmaxf](#), [fmin](#), [fminf](#), [lmax](#), [lmin](#), [min](#)

### memchr

find first occurrence of character

#### Synopsis

```
#include <string.h>
void *memchr (const void *s1, int c, size_t n);
```

#### Description

The `memchr` function compares the range of memory pointed to by `s1` with the input character `c` and returns a pointer to the first occurrence of `c`. A null pointer is returned if `c` does not occur in the first `n` characters.

#### Error Conditions

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

#### Example

```
#include <string.h>
char *ptr;

ptr = memchr ("TESTING", 'E', 7);
/* ptr points to the E in TESTING */
```

#### See Also

[strchr](#), [strrchr](#)

## memcmp

compare objects

### Synopsis

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

### Description

The `memcmp` function compares the first `n` characters of the objects pointed to by `s1` and `s2`. It returns a positive value if the `s1` object is lexicically greater than the `s2` object, a negative value if the `s2` object is lexicically greater than the `s1` object, and a zero if the objects are the same.

### Error Conditions

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

### Example

```
#include <string.h>
char string1 = "ABC";
char string2 = "BCD";
int result;

result = memcmp (string1, string2, 3);    /* result < 0 */
```

### See Also

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

### memcpy

copy characters from one object to another

#### Synopsis

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

#### Description

The `memcpy` function copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. The behavior of `memcpy` is undefined if the two objects overlap. For more information, see [“memmove” on page 2-113](#).

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";

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

#### See Also

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



## memmove

copy characters from one object to another

### Synopsis

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

### Description

The `memmove` function copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. The entire object is copied correctly even if the objects overlap.

The `memmove` function returns a pointer to `s1`.

### Error Conditions

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

### Example

```
#include <string.h>
char *ptr, *str = "ABCDE";

ptr = str + 2;
memmove (ptr, str, 5);    /* *ptr = "ABCDE" */
```

### See Also

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

### memset

set range of memory to a character

#### Synopsis

```
#include <string.h>
void *memset (void *s1, int c, size_t n);
```

#### Description

The `memset` function sets a range of memory to the input character `c`. The first `n` characters of `s1` are set to `c`.

The `memset` function returns a pointer to `s1`.

#### Error Conditions

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

#### Example

```
#include <string.h>
char string1[50];

memset (string1, '\0', 50);  /* set string1 to 0 */
```

#### See Also

[memcpy](#), [memmove](#)

## min

return smaller of two values

### Synopsis

```
#include <stdlib.h>
int min (int value1, int value2);
```

### Description

This function is an Analog Devices extension to the ANSI standard.

The `min` function returns the smaller of its two arguments. The `min` function is a built-in function which is implemented with an `Rn=MIN(Rx,Ry)` instruction.

### Error Conditions

The `min` function does not return an error code.

### Example

```
#include <stdlib.h>
int i;

i = min (10, 8);    /* returns 8 */
```

### See Also

[fmax](#), [fmaxf](#), [fmin](#), [fminf](#), [lmax](#), [lmin](#), [max](#)

### **modf, modff**

separate integral and fractional parts

#### **Synopsis**

```
#include <math.h>
double modf (double x, double *intptr);
float modff (float x, float *intptr);
```

#### **Description**

The `modf` and `modff` functions separate 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 `intptr`. The integral and fractional portions have the same sign as the input.

#### **Error Conditions**

The `modf` and `modff` functions do not return error conditions.

#### **Example**

```
#include <math.h>
double y, n;
float m, p;

y = modf (-12.345, &n);    /* y = -0.345, n = -13.0 */
m = modff (11.75, &p);    /* m = 0.75, p = 11.0    */
```

#### **See Also**

[frexp, frexpf](#)

**pow, powf**

raise to a power

**Synopsis**

```
#include <math.h>
double pow (double x, double y);
float powf (float x, float y);
```

**Description**

The `pow` and `powf` functions compute the value of the first argument raised to the power of the second argument.

The `pow` and `powf` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of  $2^{-20}$  over its input range.

**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` and zero is returned.

**Example**

```
#include <math.h>
double z;
float x;

z = pow (4.0, 2.0);      /* z = 16.0 */
x = powf (4.0, 2.0);    /* x = 16.0 */
```

**See Also**

[exp](#), [expf](#), [ldexp](#), [ldexpf](#)

### qsort

quicksort

#### Synopsis

```
#include <stdlib.h>
void qsort (void *base, size_t nelem, size_t size,
            int (*compar) (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 `compar`, 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, where

- `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.
- `compar` is a pointer to a function that is called by `qsort` to compare two elements of the array. The function should return a value less than, equal to, or greater than zero, according to whether the first argument is less than, equal to, or greater than the second.

## Return Value

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);
int raisensm(int sig);
```

#### Description

This function is an Analog Devices extension to the ANSI standard.

The `raise` function sends the signal `sig` to the executing program. The `raise` function forces interrupts wherever possible and simulates an interrupt otherwise. The `sig` argument must be one of the signals listed in [Table 2-8 on page 2-49](#), [Table 2-9 on page 2-50](#) or [Table 2-10 on page 2-51](#).



The `raise` function uses self-modifying code. If this is not suitable for your application, then use the `raisensm` function instead. The choice of function has no effect on the dispatcher used and no effect on the overall interrupt handling performance.

#### Error Conditions

The `raise` function returns a zero if successful or a nonzero value if it fails.

#### Example

```
#include <signal.h>
raise (SIG_IRQ2);    /* invoke the interrupt 2 handler */
```

#### 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  $[0, 2^{32} - 1]$ .

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  $2^{32} - 1$ .

### Error Conditions

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

### Example

```
#include <stdlib.h>
int i;
i = rand ();
```

### See Also

[srand](#)

### realloc

change memory allocation

#### Synopsis

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

#### Description

The `realloc` function changes the memory allocation of the object pointed to by `ptr` to `size`. Initial values for the new object are taken from those in the object pointed to by `ptr`. If the size of the new object is greater than the size of the object pointed to by `ptr`, then the values in the newly allocated section are undefined. If `ptr` is a non-NULL pointer that was not allocated with `malloc` or `calloc`, the behavior is undefined. If `ptr` is a null pointer, `realloc` imitates `malloc`. If `size` is zero and `ptr` is not a null pointer, `realloc` imitates `free`.

If `ptr` is non-NULL, then the object is re-allocated from the heap that the object was originally allocated from. If `ptr` is NULL, then the object is allocated from the current heap, which will be the default heap unless `set_alloc_type` has been called to change the current heap to an alternate heap.

#### Error Conditions

If memory cannot be allocated, `ptr` remains unchanged and `realloc` returns a null pointer.

#### Example

```
#include <stdlib.h>
int *ptr;
```

```
ptr = (int *)malloc (10);           /* intervening code */
ptr = (int *)realloc (ptr, 20);     /* the size is now 20 */
```

### See Also

[calloc](#), [free](#), [heap\\_calloc](#), [heap\\_free](#), [heap\\_lookup\\_name](#), [heap\\_malloc](#),  
[heap\\_realloc](#), [malloc](#), [set\\_alloc\\_type](#)

### setjmp

define a run-time label

#### Synopsis

```
#include <setjmp.h>
int setjmp (jmp_buf env);
```

#### Description

The `setjmp` function saves the calling environment in the `jmp_buf` argument. The effect of the call is to declare a run-time label that can be jumped to via a subsequent call to `longjmp`.

When `setjmp` is called, it immediately returns with a result of zero to indicate that the environment has been saved in the `jmp_buf` argument. If, at some later point, `longjmp` is called with the same `jmp_buf` argument, `longjmp` will restore the environment from the argument. The execution will then resume at the statement immediately following the corresponding call to `setjmp`. The effect is as if the call to `setjmp` has returned for a second time but this time the function will return 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 label `setjmp` does not return an error condition.

#### Example

See [“longjmp” on page 2-106](#)

#### See Also

[longjmp](#)

## setlocale

set the current locale

### Synopsis

```
#include <locale.h>
char *setlocale (int category, const char *locale);
```

### Description

The `setlocale` function uses the parameters `category` and `locale` to select a current locale. The possible values for the `category` argument are those macros defined in `locale.h` beginning with “LC\_”. The only `locale` argument supported at this time is the “C” locale. If a null pointer is used for the `locale` argument, `setlocale` returns a pointer to a string which is the current locale for the given category argument. A subsequent call to `setlocale` with the same `category` argument and the string supplied by the previous `setlocale` call returns the locale to its original status. The string pointed to may not be altered by the program but may be overwritten by subsequent `setlocale` calls.

### Error Conditions

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

### Example

```
#include <locale.h>

setlocale (LC_ALL, "C");
/* sets the locale to the "C" locale */
```

### See Also

[localeconv](#)

### set\_alloc\_type

set heap for dynamic memory allocation

#### Synopsis

```
#include <stdlib.h>
int set_alloc_type(char * heap_name);
```

#### Description

The `set_alloc_type` function is an Analog Devices extension to the ANSI standard. The `set_alloc_type` function specifies a heap from which `malloc` and `calloc` should subsequently allocate memory. The `heap_name` argument should be the name of the segment containing the heap as a string. For more information on creating multiple heaps, see [“Using Multiple Heaps” on page 1-130](#).



The `set_alloc_type` function is not available in multithreaded environments.

#### Error Conditions

The `set_alloc_type` function returns a non-zero value if the heap specified cannot be found.

#### Example

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

char *mymem, *stdmem;

int allocate()
{
    int res;
    res = set_alloc_type("seg_heap");
    if (res != 0) {
```

```
        printf("Failed to switch heaps\n");
        return 1;
    }

    mymem = malloc(10);          /* mymem is allocated on "seg_heap" */
    if (mymem == NULL) {
        printf("Failed to allocate memory from seg_heap\n");
        return 1;
    }

    set_alloc_type("seg_heap");
    if (res != 0) {
        printf("Failed to switch heaps\n");
        return 1;
    }

    stdmem = malloc(10); /* stdmem is allocated on the default heap */
    if (stdmem == NULL) {
        printf("Failed to allocate memory from the default heap\n");
        return 1;
    }

    printf("Memory was allocated at %p %p\n", mymem, stdmem);
    return 0;
}
```

### See Also

[calloc](#), [free](#), [heap\\_calloc](#), [heap\\_free](#), [heap\\_lookup\\_name](#), [heap\\_malloc](#),  
[heap\\_realloc](#), [malloc](#), [realloc](#)

### signal

define signal handling

#### Synopsis

```
#include <signal.h>
void (*signal (int sig, void (*func)(int))) (int);
void (*signalnsm (int sig, void (*func)(int))) (int);
void (*signalnf (int sig, void (*func)(int))) (int);
void (*signalfnsm (int sig, void (*func)(int))) (int);
void (*signals (int sig, void (*func)(int))) (int);
void (*signalsnsm (int sig, void (*func)(int))) (int);
void (*signalcb (int sig, void (*func)(int))) (int);
void (*signalcbnsm (int sig, void (*func)(int))) (int);
void (*signalss (int sig, void (*func)(int))) (int);
void (*signalssnsm (int sig, void (*func)(int))) (int);
```

#### Description

The `signal`, `signalnsm`, `signalnf`, `signalfnsm`, `signals`, `signalsnsm`, `signalcb`, `signalcbnsm`, `signalss` or `signalssnsm` functions determine how a signal that is received during program execution is handled. The specified signal function causes the corresponding interrupt dispatcher to be used when handling the interrupt (refer to “[signal.h](#)” on page 2-10 for more information).

The `signal` function returns the value of the previously installed interrupt or signal handler action. The `sig` argument must be one of the values that are listed in either [Table 2-8 on page 2-49](#), [Table 2-9 on page 2-50](#) or [Table 2-10 on page 2-51](#). The `signal` function causes the receipt of the signal number `sig` to be handled in one of the ways listed in [Table 2-11 on page 2-77](#). The function pointed to by `func` is executed once when the signal is received. Handling is then returned to the default state.



The differences between the actions taken by the supplied standard interrupt dispatchers, `interrupt`, `interruptnsm`, `interruptf`, `interruptfnsm`, `interrupts`, `interruptsnsm`, `interruptcb`, and `interruptcbnsm`, are discussed under [“signal.h” on page 2-10](#).

## **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>

signal (SIG_IRQ2, irq2_handler);    /* enable interrupt 2 */
signal (SIG_IRQ2, SIG_IGN);         /* disable interrupt 2 */
```

## **See Also**

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

### sin, sinf

sine

#### Synopsis

```
#include <math.h>
double sin (double x);
float sinf (float x);
```

#### Description

The `sin` and `sinf` functions return the sine of  $x$ . The input is interpreted as radians; the output is in the range  $[-1, 1]$ .

The `sin` and `sinf` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of  $2^{-20}$  over its input range. Although the `sin` and `sinf` functions accept input over the entire floating-point range, the accuracy of the result decreases significantly for inputs greater than  $\pi^{12}/2$ .

#### Error Conditions

The `sin` and `sinf` functions do not return an error condition.

#### Example

```
#include <math.h>
double y;
float x;

y = sin (3.14159);    /* y = 0.0 */
x = sinf (3.14159);  /* x = 0.0 */
```

#### See Also

[asin, asinf](#)

## **sinh, sinh**

hyperbolic sine

### **Synopsis**

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

### **Description**

The `sinh` and `sinh` functions return the hyperbolic sine of  $x$ .

The `sinh` and `sinh` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of  $2^{-20}$  over its input range.

### **Error Conditions**

For input values greater than  $2^{12}$ , the `sinh` and `sinh` functions return `HUGE_VAL` and set `errno` to `ERANGE` to indicate overflow.

### **Example**

```
#include <math.h>
double x, y;
float z, w;

y = sinh (x);
z = sinh (w);
```

### **See Also**

[cosh](#), [cosh](#)

### sqrt, sqrtf

square root

#### Synopsis

```
#include <math.h>
double sqrt (double x);
float sqrtf (float x);
```

#### Description

The `sqrt` and `sqrtf` functions return the positive square root of  $x$ .

The `sqrt` and `sqrtf` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of  $2^{-20}$  over its input range.

#### Error Conditions

The `sqrt` and `sqrtf` functions return zero for negative input values and set `errno` to `EDOM` to indicate a domain error.

#### Example

```
#include <math.h>
double y;
float x;

y = sqrt (2.0);      /* y = 1.414..... */
x = sqrtf (2.0);     /* x = 1.414..... */
```

#### See Also

[rsqrt, rsqrtf](#)

## **srand**

random number seed

### **Synopsis**

```
#include <stdlib.h>
void srand (unsigned int seed);
```

### **Description**

The `srand` function is used to set the seed value for the `rand` function. A particular seed value always produces the same sequence of pseudo-random numbers.

### **Error Conditions**

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

### **Example**

```
#include <stdlib.h>

srand (22);
```

### **See Also**

[rand](#)

### strcat

concatenate strings

#### Synopsis

```
#include <string.h>
char *strcat (char *s1, const char *s2);
```

#### Description

The `strcat` function appends a copy of the null-terminated string pointed to by `s2` to the end of the null-terminated string pointed to by `s1`. It returns a pointer to the new `s1` string, which is null-terminated. The behavior of `strcat` is undefined if the two strings overlap.

#### Error Conditions

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

#### Example

```
#include <string.h>
char string1[50];

string1[0] = 'A';
string1[1] = 'B';
string1[2] = '\0';
strcat (string1, "CD");    /* new string is "ABCD" */
```

#### See Also

[strncat](#)

## strchr

find first occurrence of character in string

### Synopsis

```
#include <string.h>
char *strchr (const char *s1, int c);
```

### Description

The `strchr` function returns a pointer to the first location in `s1`, a null-terminated string, that contains the character `c`.

### Error Conditions

The `strchr` function returns a null pointer if `c` is not part of the string.

### Example

```
#include <string.h>
char *ptr1, *ptr2;

ptr1 = "TESTING";
ptr2 = strchr (ptr1, 'E');
/* ptr2 points to the E in TESTING */
```

### See Also

[memchr](#), [strrchr](#)

### **strcmp**

compare strings

#### **Synopsis**

```
#include <string.h>
int strcmp (const char *s1, const char *s2);
```

#### **Description**

The `strcmp` function lexicographically compares the null-terminated strings pointed to by `s1` and `s2`. It returns a positive value if the `s1` string is greater than the `s2` string, a negative value if the `s2` string is greater than the `s1` string, and a zero if the strings are the same.

#### **Error Conditions**

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

#### **Example**

```
#include <string.h>
char string1[50], string2[50];

if (strcmp (string1, string2))
    printf ("%s is different than %s \n", string1, string2);
```

#### **See Also**

[memchr](#), [strncmp](#)



## strcoll

compare strings

### Synopsis

```
#include <string.h>
int strcoll (const char *s1, const char *s2);
```

### Description

The `strcoll` function compares the string pointed to by `s1` with the string pointed to by `s2`. The comparison is based on the locale macro, `LC_COLLATE`. Because only the C locale is defined in the ADSP-21xxx 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>
char *strcpy (char *s1, const char *s2);
```

#### 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 function `strcspn` returns the length of the initial segment of `s1` which consists entirely of characters not in the string pointed to by `s2`. The string pointed to by `s2` is treated as a set of characters. The order of the characters in the string is not significant.

### Error Conditions

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

### Example

```
#include <string.h>
char *ptr1, *ptr2;
size_t len;

ptr1 = "Tried and Tested";
ptr2 = "aeiou";
len = strcspn (ptr1, ptr2);  /* len = 2 */
```

### See Also

[strlen](#), [strspn](#)

### strerror

get string containing error message

#### Synopsis

```
#include <string.h>
char *strerror (int errnum);
```

#### Description

The function `strerror` returns a pointer to a string containing an error message by mapping the number in `errnum` to that string. Only one error is currently defined in the C environment for ADSP-21xxx DSPs.

#### 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 `s1` (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

[strcspn](#), [strspn](#)

### 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 character (`'\0'`).

#### Error Conditions

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

#### Example

```
#include <string.h>
char string1[50], *ptr;

string1[0] = '\0';
strncat (string1, "MOREFUN", 4);
/* string1 equals "MORE" */
```

#### See Also

[strcat](#)

## strncmp

compare characters in strings

### Synopsis

```
#include <string.h>
int strncmp (const char *s1, const char *s2, size_t n);
```

### Description

The `strncmp` function lexicographically compares up to `n` characters of the null-terminated strings pointed to by `s1` and `s2`. It returns a positive value if the `s1` string is greater than the `s2` string, a negative value if the `s2` string is greater than the `s1` string, and a zero if the strings are the same.

### Error Conditions

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

### Example

```
#include <string.h>
char *ptr1;

ptr1 = "TEST1";
if (strncmp (ptr1, "TEST", 4) == 0)
    printf ("%s starts with TEST\n", ptr1);
```

### See Also

[memcmp](#), [strcmp](#)

### strncpy

copy characters from one string to another

#### Synopsis

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

#### Description

The `strncpy` function copies up to `n` characters of the null-terminated string pointed to by `s2` into the space pointed to by `s1`. If the last character copied from `s2` is not a null, the result 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 the null character until all `n` characters have been written.

#### Error Conditions

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

#### Example

```
#include <string.h>
char string1[50];

strncpy (string1, "MOREFUN", 4);
/* MORE is copied into string1 */
string1[4] = '\0'; /* must null-terminate string1 */
```

#### See Also

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



## strpbrk

find character match in two strings

### Synopsis

```
#include <string.h>
char *strpbrk (const char *s1, const char *s2);
```

### Description

The `strpbrk` function returns a pointer to the first character in `s1` that is also found in `s2`. The string pointed to by `s2` is treated as a set of characters. The order of the characters in the string is not significant.

### Error Conditions

In the event that no character in `s1` matches any in `s2`, a null pointer is returned.

### Example

```
#include <string.h>
char *ptr1, *ptr2, *ptr3;

ptr1 = "TESTING";
ptr2 = "SHOP"
ptr3 = strpbrk (ptr1, ptr2);
/* ptr3 points to the S in TESTING */
```

### See Also

[strcmp](#), [strncmp](#)

### strrchr

find last occurrence of character in string

#### Synopsis

```
#include <string.h>
char *strrchr (const char *s1, int c);
```

#### Description

The `strrchr` function returns a pointer to the last occurrence of character `c` in the null-terminated input string `s1`.

#### Error Conditions

The `strrchr` function returns a null pointer if `c` is not found.

#### Example

```
#include <string.h>
char *ptr1, *ptr2;

ptr1 = "TESTING";
ptr2 = strrchr (ptr1, 'T');
/* ptr2 points to the second T of TESTING */
```

#### See Also

[memchr](#), [strchr](#)

## strspn

length of segment of characters in both strings

### Synopsis

```
#include <string.h>
size_t strspn (const char *s1, const char *s2);
```

### Description

The `strspn` function returns the length of the initial segment of `s1` which consists entirely of characters in the string pointed to by `s2`. The string pointed to by `s2` is treated as a set of characters. The order of the characters in the string is not significant.

### Error Conditions

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

### Example

```
#include <string.h>
size_t len;
char *ptr1, *ptr2;

ptr1 = "TESTING";
ptr2 = "ERST";
len = strspn (ptr1, ptr2);    /* len = 4 */
```

### See Also

[strcspn](#), [strlen](#)

### **strstr**

find string within string

#### **Synopsis**

```
#include <string.h>
char *strstr (const char *s1, const char *s2);
```

#### **Description**

The `strstr` function returns a pointer to the first occurrence in the string pointed to by `s1` of the characters in the string pointed to by `s2`. This excludes the terminating null character in `s1`.

#### **Error Conditions**

If the string is not found, `strstr` returns a null pointer. If `s2` points to a string of zero length, `s1` is returned.

#### **Example**

```
#include <string.h>
char *ptr1, *ptr2;

ptr1 = "TESTING";
ptr2 = strstr (ptr1, "E");
/* ptr2 points to the E in TESTING */
```

#### **See Also**

[strchr](#)

## strtod

convert string to a double

### Synopsis

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

### Description

The `strtod` function extracts a value from the string pointed to by `nptr` as a double. The `strtod` function expects `nptr` to point to a string, such as:

*[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 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, 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. ERANGE is stored in `errno` in the case of either 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

[atoi](#), [atol](#), [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"          */
```

## C Run-Time Library Reference

```
t = strtok (NULL, " ");    /* t points to "phrase"          */
t = strtok (NULL, ",");    /* t points to "to be tested" */
t = strtok (NULL, ".");    /* t points to " today"      */
t = strtok (NULL, ".");    /* t = NULL                  */
```

### See Also

No references to this function.



## strtol

convert string to long integer

### Synopsis

```
#include <stdlib.h>
long int strtol (const char *nptr, char **endptr, int base);
```

### Description

The `strtol` function returns as a `long int` the value that was represented by the string `nptr`. If `endptr` is not a null pointer, `strtol` stores a pointer to the unconverted remainder in `*endptr`.

The `strtol` function breaks down the input into three sections:

- White space (as determined by `isspace`)
- 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 `strtol` 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`, provided that `endptr` is not a null pointer. If the correct value results in an overflow, positive or negative (as appropriate) `LONG_MAX` is returned. If the correct value results in an underflow, `LONG_MIN` is returned. `ERANGE` is stored in `errno` in the case of either overflow or underflow.

### Example

```
#include <stdlib.h>
#define base 10
char *rem;
long int i;
i = strtol ("2345.5", &rem, base);
/* i=2345, rem=".5" */
```

### See Also

[atoi](#), [atol](#), [strtod](#), [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 a pointer to the invalid string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer. 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](#), [strtol](#)

## 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`. (See “[setlocale](#)” on page 2-125). It places the result in the array pointed to by `s1`.



The transformation is such that if `s1` and `s2` were transformed and used as arguments to `strcmp`, the result would be identical to the result derived from `strcoll` using `s1` and `s2` as arguments. However, since only C locale is implemented, this function does not perform any transformations other than the number of characters.

The string stored in the array pointed to by `s1` is never more than `n` characters including the terminating NULL character. `strxfrm` returns 1. If this returned value is `n` or greater, the result stored in the array pointed to by `s1` is indeterminate. `s1` can be a NULL pointer if `n` is zero.

### 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

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

## system

send string to operating system

### Synopsis

```
#include <stdlib.h>
int system (const char *string);
```

### Description

The `system` function normally sends a string to the operating system. In the context of the ADSP-21xxx DSP run-time environment, `system` always returns zero.

### Error Conditions

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

### Example

```
#include <stdlib.h>
system ("string");    /* always returns zero */
```

### See Also

[getenv](#)

### **tan, tanf**

tangent

#### **Synopsis**

```
#include <math.h>
double tan (double x);
float tanf (float x);
```

#### **Description**

The `tan` and `tanf` functions return the hyperbolic tangent of the argument that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of  $2^{-20}$  over its input range.

#### **Error Conditions**

The `tan` and `tanf` functions do not return an error condition.

#### **Example**

```
#include <math.h>
double y;
float x;

y = tan (3.14159/4.0);    /* y = 1.0 */
x = tanf (3.14159/4.0);  /* x = 1.0 */
```

#### **See Also**

[atan, atanf](#)



## **tanh, tanhf**

hyperbolic tangent

### **Synopsis**

```
#include <math.h>
double tanh (double x);
float tanhf (float x);
```

### **Description**

The `tanh` and `tanhf` functions return the hyperbolic tangent of the argument that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of  $2^{-20}$  over its input range.

### **Error Conditions**

The `tanh` and `tanhf` functions do not return an error condition.

### **Example**

```
#include <math.h>
double x, y;
float z, w;

y = tanh (x);
z = tanhf (w);
```

### **See Also**

[cosh](#), [coshf](#), [sinh](#), [sinhf](#)

### tolower

convert from uppercase to lowercase

#### Synopsis

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

#### Description

The `tolower` function converts the input character to lowercase if it is uppercase; otherwise, it returns the character.

#### Error Conditions

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

#### Example

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

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    if (isupper (ch))
        printf ("tolower=%#04x", tolower (ch));
    putchar ('\n');
}
```

#### See Also

[islower](#), [isupper](#), [toupper](#)

## toupper

convert from lowercase to uppercase

### Synopsis

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

### Description

The `toupper` function converts the input character to uppercase if it is in lowercase; otherwise, it returns the character.

### Error Conditions

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

### Example

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

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    if (islower (ch))
        printf ("toupper=%#04x", toupper (ch));
    putchar ('\n');
}
```

### See Also

[islower](#), [isupper](#), [tolower](#)

### **va\_arg**

get next argument in variable-length list of arguments

#### **Synopsis**

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

#### **Description**

The `va_arg` macro is used to walk through the variable length list of arguments to a function.

After starting to process a variable-length list of arguments with `va_start`, call `va_arg` with the same `va_list` variable to extract arguments from the list. Each call to `va_arg` returns a new argument from the list.

Substitute a type name corresponding to the type of the next argument for the type parameter in each call to `va_arg`. After processing the list, call `va_end`.

The header file `stdarg.h` defines a pointer type called `va_list` that is used to access the list of variable arguments.

The function calling `va_arg` is responsible for determining the number and types of arguments in the list. It needs this information to determine how many times to call `va_arg` and what to pass for the type parameter each time. There are several common ways for a function to determine this type of information. The standard C `printf` function reads its first argument looking for %-sequences to determine the number and types of its extra arguments. In the example below, all of the arguments are of the same type (`char*`), and a termination value (NULL) is used to indicate the end of the argument list. Other methods are also possible.

If a call to `va_arg` is made after all arguments have been processed, or if `va_arg` is called with a type parameter that is different from the type of the next argument in the list, the behavior of `va_arg` is undefined.

## Error Conditions

The `va_arg` macro does not return an error condition.

## Example

```
#include <stdarg.h>
#include <string.h>
#include <stdlib.h>

char *concat(char *s1, ...)
{
    int len = 0;
    char *result;
    char *s;
    va_list ap;

    va_start (ap, s1);
    s = s1;
    while (s) {
        len += strlen (s);
        s = va_arg (ap, char *);
    }
    va_end (ap);

    result = malloc (len + 7);
    if (!result)
        return result;
    *result = '\0';
    va_start (ap, s1);
    s = s1;
    while (s) {
        strcat (result, s);
        s = va_arg (ap, char *);
    }
    va_end (ap);
    return result;
}
```

## See Also

[va\\_end](#), [va\\_start](#)

### **va\_end**

finish variable-length argument list processing

#### **Synopsis**

```
#include <stdarg.h>
void va_end (va_list ap);
```

#### **Description**

The `va_end` macro can only be invoked after the `va_start` macro. 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 2-164](#)

#### **See Also**

[va\\_arg](#), [va\\_start](#)

## **va\_start**

initialize the variable-length argument list processing

### **Synopsis**

```
#include <stdarg.h>
void va_start (va_list ap, parmN);
```

### **Description**

The `va_start` macro is used in a function declared to take a variable number of arguments to start processing those variable arguments. The first argument to `va_start` should be a variable of type `va_list`, which is used by `va_arg` to walk through the arguments. The second argument is the name of the last *named* parameter in the function's parameter list; the list of variable arguments immediately follows this parameter. The `va_start` macro must be invoked before either the `va_arg` or `va_end` macro can be invoked.

### **Error Conditions**

The `va_start` macro does not return an error condition.

### **Example**

See “[va\\_arg](#)” on page 2-164

### **See Also**

[va\\_arg](#), [va\\_end](#)





# 3 DSP LIBRARY FOR ADSP-2106X AND ADSP-21020 PROCESSORS

This chapter describes the DSP run-time library for ADSP-2106x and ADSP-21020 processors which contains a collection of functions that provide services commonly required by DSP applications; these functions are in addition to the C/C++ run-time library functions that are described in Chapter 2, “[C Run-Time Library Reference](#)”. The services provided by the DSP library functions include support for interrupt handling, signal processing, and access to hardware registers. All these services are Analog Devices extensions to ANSI standard C.

The chapter contains:

- “[DSP Run-Time Library Guide](#)” (starting on [page 3-2](#)) contains introductory information about the ADI special header files and built-in functions that are included with this release of the cc21k compiler.
- “[DSP Run-Time Library Reference](#)” (starting on [page 3-13](#)) contains the complete reference information for each DSP run-time library function included with this release of the cc21k compiler.

For more information on the algorithms on which many of the C library's math functions are based, see Cody, W. J. and W. Waite, *Software Manual for the Elementary Functions*, Englewood Cliffs, New Jersey: Prentice Hall, 1980.

# DSP Run-Time Library Guide

The DSP run-time library contains routines that you can call from your source program. This section describes how to use the library and provides information on the following topics:

- [“Calling DSP Library Functions” on page 3-2](#)
- [“Linking DSP Library Functions” on page 3-3](#)
- [“Working With Library Source Code” on page 3-3](#)
- [“DSP Header Files” on page 3-4](#)
- [“Built-In DSP Functions” on page 3-11](#)

For information on the contents of the DSP library, see [“DSP Run-Time Library Reference” on page 3-13](#) and on-line Help.

## Calling DSP Library Functions

To use a DSP library function, call the function by name and give the appropriate arguments. The names and arguments for each function are described in the function's reference page in the section [“DSP Run-Time Library Reference” on page 3-13](#).

Like other functions you use, library functions should be declared. Declarations are supplied in header files, as described in the section, [“Working With Library Source Code” on page 3-3](#).



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, which is the function name prefixed with an underscore. For more information on naming conventions, see [“C/C++ and Assembly Interface” on page 1-154](#).



You can use the archiver, described in the *VisualDSP++ 3.0 Linker and Utilities Manual for SHARC DSPs*, to build library archive files of your own functions.

### Linking DSP Library Functions

When your C code calls a DSP run-time library function, the call creates a reference that the linker resolves when linking your program. One way to direct the linker to the location of the DSP library is to use the default Linker Description File (ADSP-21<your\_target>.ldf). The default Linker Description File will automatically direct the linker to the library `libdsp.dlb` in the `21k\lib` subdirectory of your VisualDSP++ installation. If not using the default .LDF file, then either add `libdsp.dlb` to the .LDF file used for your project, or alternatively use the compiler's `-ldsp` switch to specify that `libdsp.dlb` is to be added to the link line.



When compiling for the ADSP-21020 DSP, the name of the DSP run-time library is `libdsp020.dlb`.

### Working With Library Source Code

The source code for the functions and macros in the DSP run-time 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 `...\21k\lib\src`. The directory contains the source for the C run-time library, for the DSP run-time library, and for the I/O run-time library, as well as the source for the main program start-up functions. If you do not intend to modify any of the run-time library functions, you can delete this directory and its contents to conserve disk space.

The source code is provided so you can customize any particular function for your own needs. To modify these files, you need proficiency in ADSP-21xxx assembly language and an understanding of the run-time environment, as explained in [“C/C++ Run-Time Model and Environ-](#)

ment” on page 1-123). 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. Note that Analog Devices supports the run-time library functions only as provided.

## DSP Header Files

This section briefly describes DSP header files supplied with this release of the cc21k compiler.

### 21020.h — ADSP-21020 DSP Functions

The `21020.h` header file includes the ADSP-21020 processor-specific functions of the DSP library, such as `poll_flag_in()`, `timer_set()`, and `idle()`. The `timer_set()`, `timer_on()`, and `timer_off()` functions are also available as in-line functions.

### 21060.h — ADSP-2106x DSP Functions

The `21060.h` header file includes the ADSP-2106x processor-specific functions of the DSP library, such as `poll_flag_in()`, `timer_set()`, and `idle()`. The `timer_set()`, `timer_on()`, and `timer_off()` functions are also available as in-line functions.

### 21065L.h — ADSP-21065L DSP Functions

The `21065L.h` header file includes the ADSP-21065L processor-specific functions of the DSP library, such as `poll_flag_in()` and `idle()`. The header file also includes support for the two programmable timers in the form of in-line functions.

## **asm\_sprt.h — Mixed C/Assembly Support**

The `asm_sprt.h` header file consists of the ADSP-21xxx assembly language macros, not C functions. They are used in your assembly routines that interface with C functions. For more information, see [“Using Mixed C/C++ and Assembly Support Macros”](#) on page 1-159.

## **comm.h — A-law and $\mu$ -law Companders**

The `comm.h` header file includes the voice-band compression and expansion communication functions of the DSP library.

## **complex.h — Basic Complex Arithmetic Functions**

The `complex.h` header file contains the type definition and some basic functions for `complex_float` variables.

The following structure is used to represent complex numbers in rectangular coordinates:

```
typedef struct {  
    float re;  
    float im;  
} complex_float;
```

Additional support for `complex_float` is available via the `cvector.h` header file.

## **cvector.h — Complex Vector Functions**

The `cvector.h` header file contains functions for basic arithmetical operations on vectors of type `complex_float`. Support is provided for the dot product operation, as well as for adding, subtracting, and multiplying a vector by either a scalar or vector.

### **def21020.h — ADSP-21020 DSP Bit Definitions**

The `def21020.h` header file includes macro definitions to enable usage of symbolic names for the system register bits for the ADSP-21020 processors. It also contains macro definitions for the IOP register addresses and bit fields.

### **def21060.h — ADSP-21060 DSP Bit Definitions**

The `def21060.h` header file includes macro definitions to enable usage of symbolic names for the system register bits for the ADSP-21060 processors. It also contains macro definitions for the IOP register addresses and bit fields.

### **def21061.h — ADSP-21061 DSP Bit Definitions**

The `def21061.h` header file includes macro definitions to enable usage of symbolic names for the system register bits for the ADSP-21061 processors. It also contains macro definitions for the IOP register addresses and bit fields.

### **def21062.h — ADSP-21062 DSP Bit Definitions**

The `def21062.h` header file includes macro definitions to enable usage of symbolic names for the system register bits for the ADSP-21062 processors. It also contains macro definitions for the IOP register addresses and bit fields.

### **def21065L.h — ADSP-21065L DSP Bit Definitions**

The `def21065L.h` header file includes macro definitions to enable usage of symbolic names for the system register bits for the ADSP-21065L processors. It also contains macro definitions for the IOP register addresses and bit fields.

## **dma.h — DMA Support Functions**

The `dma.h` header file provides definitions and setup, status, enable and disable functions for DMA operations.

## **filters.h — DSP Filters**

The `filters.h` header file includes the digital signal processing filter functions of the DSP library.

## **macros.h — Circular Buffers**

The `macro.h` header file consists of ADSP-21xxx assembly language macros, not C functions. Some are used to manipulate the circular buffer features of the ADSP-21xxx processors.

## **math.h — Math Functions**

The standard math functions defined in the `math.h` header file have been augmented by implementations for the `float` data type and some additional functions that are Analog Devices extensions to the ANSI standard.

[Table 3-1](#) provides a summary of the additional library functions defined by the `math.h` header file.

Table 3-1. Math Library - Additional Functions

| Description | Prototype                                                                                              |
|-------------|--------------------------------------------------------------------------------------------------------|
| sign copy   | <code>double copysign (double x, double y);</code><br><code>float copysignf (float x, float y);</code> |
| cotangent   | <code>double cot (double x);</code><br><code>float cotf (float x);</code>                              |
| average     | <code>double favg (double x, double y);</code><br><code>float favgf (float x, float y);</code>         |
| clip        | <code>double fclip (double x, double y);</code><br><code>float fclipf (float x, float y);</code>       |

Table 3-1. Math Library - Additional Functions (Cont'd) (Cont'd)

| Description               | Prototype                                                                                        |
|---------------------------|--------------------------------------------------------------------------------------------------|
| detect Infinity           | <code>int isinf (double x);</code><br><code>int isinff (float x);</code>                         |
| detect NaN                | <code>int isnan (double x);</code><br><code>int isnanf (float x);</code>                         |
| maximum                   | <code>double fmax (double x, double y);</code><br><code>float fmaxf (float x, float y);</code>   |
| minimum                   | <code>double fmin (double x, double y);</code><br><code>float fminf (float x, float y);</code>   |
| reciprocal of square root | <code>double rsqrt (double x, double y);</code><br><code>float rsqrtf (float x, float y);</code> |



The following functions are only available if `double` is the same size as `float`: `copysign`, `favg`, `fclip`, `fmax`, `fmin`.

### **matrix.h — Matrix Functions**

The `matrix.h` header file contains functions for operating on real matrices; specifically it defines functions for adding and subtracting two matrices and for multiplying a matrix by either a scalar or a matrix. Functions are also available to compute the inverse and transpose of a matrix.

### **saturate.h — Saturation Mode Arithmetic**

The `saturate.h` header file defines the interface for the saturated arithmetic operations. See [“Saturated Arithmetic” on page 1-99](#) for further information.

### **sport.h — Serial Port Support Functions**

The `sport.h` header file provides definitions and setup, enable, and disable functions for the ADSP-2106x DSP serial ports.



## stats.h — Statistical Functions

The `stats.h` header file includes various statistics functions of the DSP library, such as `mean()` and `autocorr()`.

## sysreg.h — Register Access

The `sysreg.h` header file defines a set of built-in functions that provide efficient access to the SHARC system registers from C. The supported functions are fully described in the section [“Access to System Registers” on page 1-87](#).

## trans.h — Fast Fourier Transforms

The `trans.h` header file includes the Fast Fourier Transform (FFT) functions of the DSP library. Note that `cfftN` stands for the entire family of Fast Fourier Transform functions `cfft65536`, `cfft32768`, 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.



If you only wish to input the real part of a signal, ensure that the imaginary input array is filled with zeros before calling the function.

The functions first bit-reverse the input arrays and then process them with an optimized block floating-point FFT (or IFFT) routine.

The `rfftN` functions work like `cfftN` functions, except they operate on input arrays of real data only. This is equivalent to a `cfftN` whose imaginary input component is set to zero.

## vector.h — Vector Functions

The `vector.h` header file contains functions for operating on vectors of type `float`. Support is provided for the dot product operation and well as for adding, subtracting, and multiplying a vector by either a scalar or vector.

## window.h — Window Generators

The `window.h` header file contains various functions to generate windows based on various methodologies. The functions, defined in the `window.h` header file, are listed in [Table 3-2 on page 3-10](#).

For all window functions, a stride parameter `a` can be used to space the window values. The window length parameter `n` equates to the number of elements in the window. Therefore, for a stride `a` of 2 and a length `n` of 10, an array of length 20 is required, where every second entry is untouched.

Table 3-2. Window Generator Functions

| Description              | Prototype                                                                 |
|--------------------------|---------------------------------------------------------------------------|
| generate bartlett window | <code>void gen_bartlett<br/>(float w[], int a, int n)</code>              |
| generate blackman window | <code>void gen_blackman<br/>(float w[], int a, int n)</code>              |
| generate gaussian window | <code>void gen_gaussian<br/>(float w[], float alpha, int a, int n)</code> |
| generate hamming window  | <code>void gen_hamming<br/>(float w[], int a, int n)</code>               |
| generate hanning window  | <code>void gen_hanning<br/>(float w[], int a, int n)</code>               |
| generate harris window   | <code>void gen_harris<br/>(float w[], int a, int n)</code>                |
| generate kaiser window   | <code>void gen_kaiser<br/>(float w[], float beta, int a, int n)</code>    |

Table 3-2. Window Generator Functions

| Description                 | Prototype                                         |
|-----------------------------|---------------------------------------------------|
| generate rectangular window | void gen_rectangular<br>(float w[], int a, int n) |
| generate triangle window    | void gen_triangle<br>(float w[], int a, int n)    |

## Built-In DSP Functions

The C/C++ compiler supports built-in functions (also known as intrinsic functions) that enable efficient use of hardware resources. Knowledge of these functions is built into the compiler. Your program uses them via normal function call syntax. The compiler notices the invocation and replaces a call to a DSP library function with one or more machine instructions, just as it does for normal operators like “+” and “\*”.

Built-in functions are declared in system header files and have names which begin with double underscores, `__builtin`.



Identifiers beginning with “\_\_” are reserved by the C standard, so these names do not conflict with user defined identifiers.

These functions are specific to individual architectures. The built-in DSP library functions supported at this time on the ADSP-2106x and ADSP-21020 architectures are listed in [Table 3-3](#). Refer to [“Using Compiler’s Built-In C Library Functions” on page 2-19](#) for more information on this topic.

Table 3-3. Built-in DSP Functions

|                       |                        |
|-----------------------|------------------------|
| <code>copysign</code> | <code>copysignf</code> |
| <code>favg</code>     | <code>favgf</code>     |
| <code>fmax</code>     | <code>fmaxf</code>     |
| <code>fmin</code>     | <code>fminf</code>     |



Functions `copysign`, `favg`, `fmax`, and `fmin` will only be compiled as a built-in function if `double` is the same size as `float`.

The compiler also supports a set of built-in functions for which no line machine instructions are substituted. This set of built-in functions is characterized by defining one or more pointers in their argument list.

For this set of built-in functions, the compiler relaxes the normal rule whereby any pointer that is passed to a library function must address Data Memory (DM). The compiler recognizes when certain pointers address Program Memory (PM) and will generate a call to an appropriate version of the run-time library function. [Table 3-4](#) lists library functions that may be called with pointers that address Program Memory.

Table 3-4. Library Functions Called with Pointers

|                        |                          |
|------------------------|--------------------------|
| <code>histogram</code> | <code>matadd</code>      |
| <code>matsub</code>    | <code>mean</code>        |
| <code>matmul</code>    | <code>matscalmult</code> |
| <code>rms</code>       | <code>transpm</code>     |



Use the `-no-builtin` compiler switch (see [on page 1-35](#)) to disable this feature.

## DSP Run-Time Library Reference

The DSP run-time library is a collection of functions that you can call from your C programs. This section lists the functions in alphabetical order. Note the following items that apply to all 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.

**Function Benchmarks and Specifications.** All functions have been timed from setup, to invocation, to results storage of returned value. This includes all register storing, parameter passing, etc. Most functions execute slightly faster if you pass constants as arguments instead of variables.

**Restrictions.** When polymorphic functions are used and the function returns a pointer to program memory, cast the output of the function to pm. For example,

```
(char pm *)
```

**Reference Format.** Each function in the library has a reference page. These pages follow the following format:

**Name** and Purpose of the function

**Synopsis**—Required header file and functional prototype

**Description**—Function specification

**Error Conditions**—Method function uses to indicate an error

**Example**—Typical function usage

**See Also**—Related functions

### **a\_compress**

A-law compression

#### **Synopsis**

```
#include <comm.h>
int a_compress (int x);
```

#### **Description**

The `a_compress` function takes a linear 13-bit signed speech sample and compresses it according to ITU recommendation G.711. The value returned is an 8-bit sample that can be sent directly to an A-law codec.

#### **Error Conditions**

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

#### **Example**

```
#include <comm.h>
int sample, compress;

compress = a_compress (sample);
```

#### **See Also**

[a\\_expand](#), [mu\\_compress](#)

## **a\_expand**

A-law expansion

### **Synopsis**

```
#include <comm.h>
int a_expand (int compress_x);
```

### **Description**

The `a_expand` function takes an 8-bit compressed speech sample and expands it according to ITU recommendation G.711 (A-law definition). The value returned is a linear 13-bit signed sample.

### **Error Conditions**

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

### **Example**

```
#include <comm.h>
int compressed_sample, expanded;

expanded = a_expand (compressed_sample);
```

### **See Also**

[a\\_compress](#), [mu\\_expand](#)

## autocoh

autocoherence

### Synopsis

```
#include <stats.h>
float *autocoh (float dm out[],
               const float dm in[], /* Input vector a */
               int samples,         /* Input samples */
               int lags);           /* Lag count */
```

### Description

The `autocoh` function computes the autocohereence of the floating-point input, `in[]`. The autocohereence of an input signal is its autocorrelation minus its mean. The function returns a pointer to the output array, `out[]` of length `lags`.

### Error Conditions

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

### Example

```
#include <stats.h>
#define SAMPLES 1024

float excitation[SAMPLES], response[16];
int lags = 16;

autocoh (response, excitation, SAMPLES, lags);
```

### See Also

[autocorr](#), [crosscoh](#), [crosscorr](#)



## autocorr

autocorrelation

### Synopsis

```
#include <stats.h>
float *autocorr (float dm out[],
                 const float dm in[],    /* Input vector */
                 int samples,           /* input samples */
                 int lags);              /* Lag count */
```

### Description

The `autocorr` function performs an autocorrelation of a signal. Autocorrelation is the cross-correlation of a signal with a copy of itself. It provides information about the time variation of the signal. The signal to be autocorrelated is given by the `in[]` input array. The number of samples of the autocorrelation sequence to be produced is given by `lags`. The length of the input sequence is given by `samples`. This function returns a pointer to the `out[]` output data array of length `lags`.

The `autocorr` function is used in digital signal processing applications such as speech analysis.

### Error Conditions

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

### Example

```
#include <stats.h>
float r[10], s[160];

autocorr (r, s, 160, 10);
/* compute first 10 autocorr coefficients of array s */
```

### See Also

[autocoh](#), [crosscoh](#), [crosscorr](#)

## biquad

biquad filter section

### Synopsis

```
#include <filters.h>
float biquad (float sample,
             const float pm coeffs[],
             float dm state[],
             int sections);
```

### Description

The `biquad` function implements a cascaded biquad filter. The function produces the filtered response of its input data. The parameter `sections` specifies the number of `biquad` sections.

Each biquad section is represented by five coefficients that must be stored in the order B2, B1, B0, A2, A1. The value of A0 is assumed to be 1.0, and A1 and A2 should be scaled accordingly. The coefficients should be stored in the array `coeffs` which must be located in program memory (PM). The definition of the `coeffs` array is:

```
float pm coeffs[5*sections];
```



When importing coefficients from a filter design tool that employs a transposed direct form II, the A coefficients have to be negated. For example, if a filter design tool returns  $A = [1.0, 0.2, -0.9]$ , then the A coefficients have to be modified to  $A = [1.0, -0.2, 0.9]$ .

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 equal to  $2*sections + 1$ . The definition is:

```
float dm state[2*sections + 1];
```

Each filter has its own state array which should be initialized to zero before calling the `biquad` function for the first time, and should not otherwise be modified by the calling program.

### Error Conditions

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

### Example

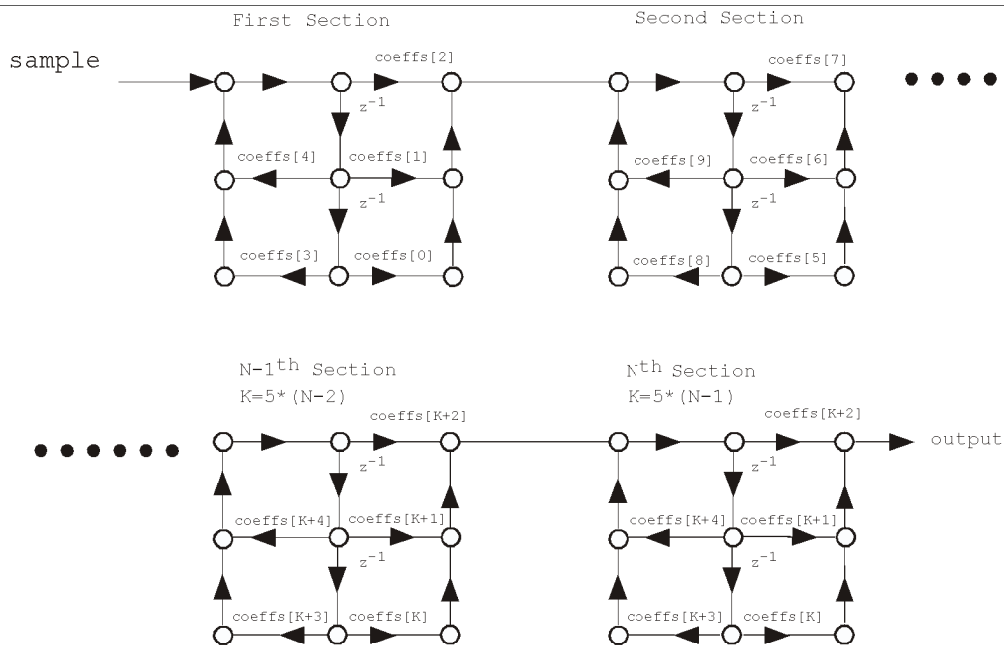
```
#include <filters.h>
#define TAPS 9

float sample, output, state[2*TAPS+1];
float pm coeffs[5*TAPS];
int i;

for (i = 0; i < 2*TAPS+1; i++)
    state[i] = 0; /* initialize state array */

output = biquad (sample, coeffs, state, TAPS);
```

## DSP Run-Time Library Reference



$N$  = the number of biquad sections.

The algorithm shown here is adapted from Oppenheim, Alan V. and Ronald Schaffer, *Digital Signal Processing*, Englewood Cliffs, New Jersey: Prentice Hall, 1975.

See Also

[fir](#), [iir](#)

## **cabsf**

complex absolute value

### **Synopsis**

```
#include <complex.h>
float cabsf (complex_float z);
```

### **Description**

The `cabsf` function returns the floating-point absolute value of its complex input.

The absolute value of a complex number is evaluated with the following formula.

$$y = \sqrt{(\text{Re}(x))^2 + (\text{Im}(x))^2}$$

where `x` is the `complex_float` input and `y` is the `float` output.

### **Error Conditions**

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

### **Example**

```
#include <complex.h>
complex_float cnum;
float answer;

cnum.re = 12.0;
cnum.im = 5.0;

answer = cabsf (cnum);    /* answer = 13.0 */
```

### **See Also**

[fabs](#), [fabsf](#), [labs](#)

## cexpf

complex exponential

### Synopsis

```
#include <complex.h>
complex_float cexpf (complex_float z);
```

### Description

The `cexpf` function computes the complex exponential value  $e$  to the power of the first argument. The exponential of a complex value is evaluated with the following formula.

$$\begin{aligned}\text{Re}(y) &= \text{expf}(\text{Re}(x)) * \text{cosf}(\text{Im}(x)); \\ \text{Im}(y) &= \text{expf}(\text{Re}(x)) * \text{sinf}(\text{Im}(x));\end{aligned}$$

where  $x$  is the `complex_float` input and  $y$  is the `complex_float` output.

### Error Conditions

For underflow errors the `cexpf` function returns zero.

### Example

```
#include <complex.h>
complex_float cnum;
complex_float answer;

cnum.re = 1.0;
cnum.im = 0.0;

answer = cexpf (cnum);      /* answer = (2.7182 + 0i) */
```

### See Also

[log](#), [logf](#), [pow](#), [powf](#)

## cfftN

N-point complex input fast Fourier transform

### Synopsis

```
#include <trans.h>
float *cfft65536 (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *cfft32768 (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *cfft16384 (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *cfft8192 (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *cfft4096 (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *cfft2048 (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *cfft1024 (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *cfft512 (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *cfft256 (const float dm real_input[],
```

## DSP Run-Time Library Reference

```
const float dm imag_input[],
float dm real_output[], float dm imag_output[]];

float *cfft128 (const float dm real_input[],
const float dm imag_input[],
float dm real_output[], float dm imag_output[]];

float *cfft64 (const float dm real_input[],
const float dm imag_input[],
float dm real_output[], float dm imag_output[]];

float *cfft32 (const float dm real_input[],
const float dm imag_input[],
float dm real_output[], float dm imag_output[]];

float *cfft16 (const float dm real_input[],
const float dm imag_input[],
float dm real_output[], float dm imag_output[]];

float *cfft8 (const float dm real_input[],
const float dm imag_input[],
float dm real_output[], float dm imag_output[]];
```

### Description

Each of these 14 `cfftN` functions computes the N-point radix-2 fast Fourier transform (CFFT) of its floating-point input (where N is 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768 or 65536).

There are fourteen 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 they operate on. Call a particular function by substituting the number of points for N, as in

```
cfft8 (r_inp, i_inp, r_outp, i_outp);
```

The input to `cfftN` are two floating-point arrays of N points. The array `real_input` contains the real components of the complex signal, and the array `imag_input` contains the imaginary components.



If there are fewer than N actual data points, you must pad the arrays with zeros to make N samples. However, better results occur with less zero padding.

The input data should be windowed (if necessary) before calling the function because no preprocessing is performed on the data.

The `cfftN()` functions return a pointer to the `real_output` array.

### Error Conditions

The `cfftN` functions do not return any error conditions.

### Example

```
#include <trans.h>
#define N 2048

float real_input[N], imag_input[N];
float real_output[N], imag_output[N];

/* Real input array is filled from a converter or other source */

cfft2048 (real_input, imag_input, real_output, imag_output);
/* Arrays are filled with FFT data */
```

### See Also

[ifftN](#), [rfftN](#)

## copysign, copysignf

copy the sign of the floating-point operand (IEEE arithmetic function)

### Synopsis

```
#include <math.h>
double copysign (double x, double y);
float copysignf (float x, float y);
```

### Description

The `copysign` and `copysignf` functions copy the sign of the second argument `y` to the first argument `x` without changing either its exponent or mantissa. The `copysignf` function is a built-in function which is implemented with an `Fn=Fx COPYSIGN Fy` instruction.

### Error Conditions

This function does not return an error code.

### Example

```
#include <math.h>
double x;
float y;

x = copysign (0.5, -10.0);      /* x = -0.5 */
y = copysignf (-10.0, 0.5f);    /* y = 10.0 */
```

### See Also

No references to this function.



The double precision function `copysign` is only available when using the `-double-size-32` switch (see [on page 1-27](#)) and actually calls the single precision function `copysignf`.

## **cot, cotf**

cotangent

### **Synopsis**

```
#include <math.h>
double cot (double x);
float cotf (float x);
```

### **Description**

The `cot` and `cotf` functions return the cotangent of their argument. The input is interpreted as radians.

The `cot` and `cotf` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of  $2^{-20}$  over its input range.

### **Error Conditions**

The `cot` and `cotf` functions do not return an error condition.

### **Example**

```
#include <math.h>
double x, y;
float v, w;

y = cot (x);
v = cotf (w);
```

### **See Also**

[tan, tanf](#)

## crosscoh

cross-coherence

### Synopsis

```
#include <stats.h>
float *crosscoh (float dm out[],
                const float dm x[],
                const float dm y[],
                int samples,
                int lags);
```

### Description

The `crosscoh` function computes the cross-coherence of two floating point inputs, `x[]` and `y[]`. The cross-coherence is the cross-correlation minus the product of the mean of `x` and the mean of `y`. The length of the input arrays is given by `samples`. This function returns a pointer to the output data array, `out[]`, of length `lags`.

### Error Conditions

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

### Example

```
#include <stats.h>
#define SAMPLES 1024
float excitation[SAMPLES], response[16], y[SAMPLES];
int lags = 16;

crosscoh (response, excitation, y, SAMPLES, lags);
```

### See Also

[autocoh](#), [autocorr](#), [crosscorr](#)

## crosscorr

cross-correlation

### Synopsis

```
#include <stats.h>
float *crosscorr (float dm out[],
                  const float dm x[],
                  const float dm y[],
                  int samples,
                  int lags);
```

### Description

The `crosscorr` function performs a cross-correlation between two signals. The cross-correlation is the sum of the scalar products of the signals in which the signals are displaced in time with respect to one another. The signals to be correlated are given by input `x[]` and `y[]` arrays. The length of the input arrays is given by `samples`. This function returns a pointer to the output data array, `out[]`, of length `lags`.

The `crosscorr` function is used in digital signal processing applications such as speech analysis.

### Error Conditions

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

### Example

```
#include <stats.h>
float r[10], s[160];
float p[160];

crosscorr (r, s, p, 160, 10);
```

### See Also

[autocoh](#), [autocorr](#), [crosscoh](#)

## cvecdot

complex vector dot product

### Synopsis

```
#include <cvector.h>
complex_float cvecdot (const complex_float dm x_input[],
                      const complex_float dm y_input[],
                      int samples);
```

### Description

The `cvecdot` function computes the complex dot product of the complex vectors, `x_input` and `y_input`, which are `samples` in size. The scalar result is returned by the function.

### Error Conditions

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

### Example

```
#include <cvector.h>
complex_float answer, x[100], y[100];

answer = cvecdot (x, y, 100);
```

### See Also

[vecdot](#)

## cvecsadd

complex vector scalar addition

### Synopsis

```
#include <cvector.h>
complex_float *cvecsadd (const complex_float dm input[],
                        complex_float scalar,
                        complex_float dm output[],
                        int samples);
```

### Description

The `cvecsadd` function computes the sum of each element of the complex vector, `input`, added to the complex scalar. Both the input and output vectors are `samples` in length. The function returns a pointer to the output vector.

### Error Conditions

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

### Example

```
#include <cvector.h>
complex_float answer[50], x[50], y;

cvecsadd (x, y, answer, 50);
```

### See Also

[vecvadd](#)

### **cvecsmlt**

complex vector scalar multiply

#### **Synopsis**

```
#include <cvector.h>
complex_float *cvecsmlt (const complex_float dm input[],
                        complex_float scalar,
                        complex_float dm output[],
                        int samples);
```

#### **Description**

The `cvecsmlt` function computes the product of each element of the complex vector, `input`, multiplied by the complex scalar. Both the input and output vectors are `samples` in length. The function returns a pointer to the output vector.

#### **Error Conditions**

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

#### **Example**

```
#include <cvector.h>
complex_float answer[50], x[50], y;

cvecsmlt (x, y, answer, 50);
```

#### **See Also**

[vecvmlt](#)



## cvecssub

complex vector scalar subtraction

### Synopsis

```
#include <cvector.h>
complex_float *cvecssub (const complex_float dm input[],
                        complex_float scalar,
                        complex_float dm output[],
                        int samples);
```

### Description

The `cvecssub` function computes the difference of each element of the complex vector, `input`, minus the complex scalar. Both the input and output vector are `samples` in length. The function returns a pointer to the output vector.

### Error Conditions

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

### Example

```
#include <cvector.h>
complex_float answer[50], x[50], y

cvecssub (x, y, answer, 50);
```

### See Also

[vecvsub](#)

## cvecvadd

complex vector addition

### Synopsis

```
#include <cvector.h>
complex_float *cvecvadd (const complex_float dm x_input[],
                        const complex_float dm y_input[],
                        complex_float dm output[],
                        int samples);
```

### Description

The `cvecvadd` function computes the sum of each of the elements of the complex vectors, `x_input` and `y_input`, and places the results in `output`. All three vectors are `samples` in length. The function returns a pointer to the output vector.

### Error Conditions

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

### Example

```
#include <cvector.h>
complex_float answer[50], x[50], y[50];

cvecvadd (x, y, answer, 50);
```

### See Also

[vecvadd](#)

## cvecvmlt

complex vector multiply

### Synopsis

```
#include <cvector.h>
complex_float *cvecvmlt (const complex_float dm x_input[],
                        const complex_float dm y_input[],
                        complex_float dm output[],
                        int samples);
```

### Description

The `cvecvmlt` function computes the product of each of the elements of the complex vectors, `x_input` and `y_input`, and places the results in `output`. All three vectors are `samples` in length. The function returns a pointer to the output vector.

### Error Conditions

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

### Example

```
#include <cvector.h>
complex_float answer[50], x[50], y[50];

cvecvmlt (x, y, answer, 50);
```

### See Also

[vecvmlt](#)

## **cvecvsub**

complex vector subtraction

### **Synopsis**

```
#include <cvector.h>
complex_float *cvecvsub (const complex_float dm x_input[],
                        const complex_float dm y_input[],
                        complex_float dm output[],
                        int samples);
```

### **Description**

The `cvecvsub` function computes the difference of each of the elements of the complex vectors, `x_input` and `y_input`, and places the results in `output`. All three vectors are `samples` in length. The function returns a pointer to the output vector.

### **Error Conditions**

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

### **Example**

```
#include <cvector.h>
complex_float answer[50], x[50], y[50];

cvecvsub (x, y, answer, 50);
```

### **See Also**

[vecvsub](#)

## favg, favgf

return mean of two values

### Synopsis

```
#include <math.h>
double favg (double x, double y);
float favgf (float x, float y);
```

### Description

The `favg` and `favgf` functions return the mean of its two arguments. The `favgf` function is a built-in function which is implemented with an `Fn=(Fx+Fy)/2` instruction.

### Error Conditions

The `favg` and `favgf` functions do not return an error code.

### Example

```
#include <math.h>
float x;
x = favgf (10.0f, 8.0f);    /* returns 9.0f */
```

### See Also

[avg](#), [lavg](#)



The double-precision function `favg` is only available when using the `-double-size-32` switch (see [on page 1-27](#)) and actually calls the single-precision function `favgf`.

### **fclip, fclipf**

clip  $x$  by  $y$

#### **Synopsis**

```
#include <math.h>
double fclip (double x, double y);
float fclipf (float x, float y);
```

#### **Description**

The `fclip` and `fclipf` functions return the first argument if it is less than the absolute value of the second argument, otherwise it returns the absolute value of the second argument if the first is positive, or minus the absolute value if the first argument is negative. The `fclipf` function is a built-in function which is implemented with an `Fn=CLIP Fx BY Fy` instruction.

#### **Error Conditions**

The `fclip` and `fclipf` functions do not return an error code.

#### **Example**

```
#include <math.h>
float y;

y = fclipf (5.1f, 8.0f);    /* returns 5.1f */
```

#### **See Also**

[clip](#), [lclip](#)



The double-precision function `fclip` is only available when using the `-double-size-32` switch (see [on page 1-27](#)) and actually calls the single-precision function `fclipf`.

## **fir**

finite impulse response (FIR) filter

### **Synopsis**

```
#include <filters.h>
float fir (float sample,
          const float pm coeffs[],
          float dm state[],
          int taps);
```

### **Description**

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 floating-point 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 should be stored in reverse order with `coeffs[0]` containing the last (`taps-1`) coefficient and `coeffs[taps-1]` containing the first coefficient. The `coeffs` array must be located in program memory data space so that the single-cycle dual-memory fetch of the processor can be used.

The `state` array contains a pointer to the delay line as its first element, followed by the delay line values. The length of the `state` array is therefore one 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.

## Error Conditions

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

## Example

```
#include <filters.h>
#define TAPS 10
float y;
float pm coeffs[TAPS];    /* coeffs array must be initialized */
                          /* and in PM memory */
float state[TAPS+1];
int i;

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

y = fir (0.775, coeffs, state, TAPS);
                          /* y holds the filtered output */
```

## See Also

[biquad](#), [iir](#)



## fmax, fmaxf

return larger of two values

### Synopsis

```
#include <math.h>
double fmax (double x, double y);
float fmaxf (float x, float y);
```

### Description

The `fmax` and `fmaxf` functions return the larger of its two arguments. The `fmaxf` function is a built-in function which is implemented with an `Fn=MAX(Fx,Fy)` instruction.

### Error Conditions

The `fmax` and `fmaxf` functions do not return an error code.

### Example

```
#include <math.h>
float y;

y = fmaxf (5.1f, 8.0f);           /* returns 8.0f */
```

### See Also

[fmin](#), [fminf](#), [lmax](#), [lmin](#), [max](#), [min](#)



The double-precision function `fmax` is only available when using the `-double-size-32` switch (see [on page 1-27](#)) and actually calls the single-precision function `fmaxf`.

### fmin, fminf

return smaller of two values

#### Synopsis

```
#include <math.h>
double fmin (double x, double y);
float fminf (float x, float y);
```

#### Description

The `fmin` and `fminf` functions return the smaller of their two arguments. The `fminf` function is a built-in function which is implemented with an `Fn=MIN(Fx,Fy)` instruction.

#### Error Conditions

The `fmin` and `fminf` functions do not return an error code.

#### Example

```
#include <math.h>
float y;

y = fminf (5.1f, 8.0f);      /* returns 5.1f */
```

#### See Also

[fmax](#), [fmaxf](#), [lmax](#), [lmin](#), [max](#), [min](#)



The double-precision function `fmin` is only available when using the `-double-size-32` switch (see [on page 1-27](#)) and actually calls the single-precision function `fminf`.

## gen\_bartlett

generate bartlett window

### Synopsis

```
#include <window.h>
void gen_bartlett(
    float dm w[], /* Window vector */
    int a,         /* Address stride in samples for window vector */
    int N          /* Length of window vector */ );
```

### Description

The `gen_bartlett` function generates a vector containing the Bartlett window. The length is specified by parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`.

The Bartlett window is similar to the Triangle window (see [“gen\\_triangle” on page 3-53](#)) but has the following different properties

- The Bartlett window always returns a window with two zeros on either end of the sequence. Therefore, for odd `n`, the center section of a `N+2` Bartlett window equals a `N` Triangle window.
- For even `n`, the Bartlett window is the convolution of two rectangular sequences. There is no standard definition for the Triangle window for even `n`; the slopes of the Triangle window are slightly steeper than those of the Bartlett window.

### Algorithm

$$w[n] = 1 - \left| \frac{n - \frac{N-1}{2}}{\frac{N-1}{2}} \right|$$

where `n = {0, 1, 2, ..., N-1}`

## DSP Run-Time Library Reference

### Domain

$a > 0$ ;  $N > 0$

### Error Conditions

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

### See Also

[gen\\_blackman](#), [gen\\_gaussian](#), [gen\\_hamming](#), [gen\\_hanning](#), [gen\\_harris](#),  
[gen\\_kaiser](#), [gen\\_rectangular](#), [gen\\_triangle](#)

## gen\_blackman

generate blackman window

### Synopsis

```
#include <window.h>
void gen_blackman(
    float dm w[], /* Window vector */
    int a, /* Address stride in samples for window vector */
    int N /* Length of window vector */ );
```

### Description

The `gen_blackman` function generates a vector containing the Blackman window. The length of the window required is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`.

### Algorithm

$$w[n] = 0.42 - 0.5 \cos\left(\frac{2\pi n}{N-1}\right) + 0.08 \cos\left(\frac{4\pi n}{N-1}\right)$$

where  $n = \{0, 1, 2, \dots, N-1\}$

### Domain

$a > 0$ ;  $N > 0$

### Error Conditions

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

### See Also

[gen\\_bartlett](#), [gen\\_gaussian](#), [gen\\_hamming](#), [gen\\_hanning](#), [gen\\_harris](#),  
[gen\\_kaiser](#), [gen\\_rectangular](#), [gen\\_triangle](#)

## gen\_gaussian

generate gaussian window

### Synopsis

```
#include <window.h>
void gen_gaussian(
float dm w[], /* Window vector */
float alpha, /* Gaussian alpha parameter */
int a, /* Address stride in samples for window vector */
int N /* Length of window vector */);
```

### Description

The `gen_gaussian` function generates a vector containing the Gaussian window. The length of the window required is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`.

### Algorithm

$$w(n) = \exp \left[ -\frac{1}{2} \left( \alpha \frac{n - N/2 - 1/2}{N/2} \right)^2 \right]$$

where  $n = \{0, 1, 2, \dots, N-1\}$  and  $\alpha$  is an input parameter

### Domain

$a > 0$ ;  $N > 0$ ;  $\alpha > 0.0$

### Error Conditions

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

### See Also

[gen\\_bartlett](#), [gen\\_blackman](#), [gen\\_hamming](#), [gen\\_hanning](#), [gen\\_harris](#),  
[gen\\_kaiser](#), [gen\\_rectangular](#), [gen\\_triangle](#)

## gen\_hamming

generate hamming window

### Synopsis

```
#include <window.h>
void gen_hamming(
    float dm w[], /* Window vector */
    int a, /* Address stride in samples for window vector */
    int N /* Length of window vector */ );
```

### Description

The `gen_hamming` function generates a vector containing the Hamming window. The length of the window required is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`.

### Algorithm

$$w[n] = 0.54 - 0.46 \cos\left(\frac{2\pi n}{N}\right)$$

where  $n = \{0, 1, 2, \dots, N-1\}$

### Domain

$a > 0$ ;  $N > 0$

### Error Conditions

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

### See Also

[gen\\_bartlett](#), [gen\\_blackman](#), [gen\\_gaussian](#), [gen\\_hanning](#), [gen\\_harris](#),  
[gen\\_kaiser](#), [gen\\_rectangular](#), [gen\\_triangle](#)

## gen\_hanning

generate hanning window

### Synopsis

```
#include <window.h>
void gen_hanning(
    float dm w[], /* Window vector */
    int a, /* Address stride in samples for window vector */
    int N /* Length of window vector */ );
```

### Description

The `gen_hanning` function generates a vector containing the Hanning window. The length of the window required is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`. This window is also known as the Cosine window.

### Algorithm

$$w[n] = 0.5 - 0.5 \cos\left(\frac{2\pi n}{N-1}\right)$$

where  $n = \{0, 1, 2, \dots, N-1\}$

### Domain

$a > 0$ ;  $N > 0$

### Error Conditions

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

### See Also

[gen\\_bartlett](#), [gen\\_blackman](#), [gen\\_gaussian](#), [gen\\_hamming](#), [gen\\_harris](#),  
[gen\\_kaiser](#), [gen\\_rectangular](#), [gen\\_triangle](#)



## gen\_harris

generate harris window

### Synopsis

```
#include <window.h>
void gen_harris(
    float dm w[], /* Window vector */
    int a,         /* Address stride in samples for window vector */
    int N          /* Length of window vector */
    */ );
```

### Description

The `gen_harris` function generates a vector containing the Harris window. The length of the window required is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`. This window is also known as the Blackman-Harris window.

### Algorithm

$$w[n] = 0.35875 - 0.48829 * \cos\left(\frac{2\pi n}{N-1}\right) + 0.14128 * \cos\left(\frac{4\pi n}{N-1}\right) - 0.01168 * \cos\left(\frac{6\pi n}{N-1}\right)$$

where  $n = \{0, 1, 2, \dots, N-1\}$

### Domain

$a > 0$ ;  $N > 0$

### Error Conditions

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

### See Also

[gen\\_bartlett](#), [gen\\_blackman](#), [gen\\_gaussian](#), [gen\\_hamming](#), [gen\\_hanning](#),  
[gen\\_kaiser](#), [gen\\_rectangular](#), [gen\\_triangle](#)

## gen\_kaiser

generate kaiser window

### Synopsis

```
#include <window.h>
void gen_kaiser(
    float dm w[], /* Window vector */
    float beta, /* Kaiser beta parameter */
    int a, /* Address stride in samples for window vector */
    int N /* Length of window vector */);
```

### Description

The `gen_kaiser` function generates a vector containing the Kaiser window. The length of the window required is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`. The  $\beta$  value is specified by parameter `beta`.

### Algorithm

$$w[n] = \frac{I_0 \left[ \beta \left( 1 - \left[ \frac{n - \alpha}{\alpha} \right]^2 \right)^{1/2} \right]}{I_0(\beta)}$$

where  $n = \{0, 1, 2, \dots, N-1\}$ ,  $\alpha = (N - 1) / 2$ , and  $I_0(\beta)$  represents the zeroth-order modified Bessel function of the first kind.

### Domain

$a > 0$ ;  $N > 0$ ;  $\beta > 0.0$

### Error Conditions

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

### See Also

[gen\\_bartlett](#), [gen\\_blackman](#), [gen\\_gaussian](#), [gen\\_hamming](#), [gen\\_hanning](#),  
[gen\\_harris](#), [gen\\_rectangular](#), [gen\\_triangle](#)

## gen\_rectangular

generate rectangular window

### Synopsis

```
#include <window.h>
void gen_rectangular(
    float dm w[], /* Window vector */
    int a, /* Address stride in samples for window vector */
    int N /* Length of window vector */ );
```

### Description

The `gen_rectangular` function generates a vector containing the Rectangular window. The length of the window required is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be  $N \cdot a$ .

### Algorithm

$$w[n] = 1$$

where  $n = \{0, 1, 2, \dots, N-1\}$

### Domain

$a > 0$ ;  $N > 0$

### Error Conditions

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

### See Also

[gen\\_bartlett](#), [gen\\_blackman](#), [gen\\_gaussian](#), [gen\\_hamming](#), [gen\\_hanning](#),  
[gen\\_harris](#), [gen\\_kaiser](#), [gen\\_triangle](#)

## gen\_triangle

generate triangle window

### Synopsis

```
#include <window.h>
void gen_triangle(
float dm w[], /* Window vector */
int a, /* Address stride in samples for window vector */
int N /* Length of window vector */ );
```

### Description

The `gen_triangle` function generates a vector containing the Triangle window. The length of the window required is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`.

Refer to the Bartlett window (described [on page 3-43](#)) regarding the relationship between it and the Triangle window.

### Algorithm

For even `n`, the following equation applies:

$$w[n] = \begin{cases} \frac{2n+1}{N} & n < N/2 \\ \frac{2N-2n-1}{N} & n > N/2 \end{cases}$$

where `n = {0, 1, 2, ..., N-1}`

For odd `n`, the following equation applies:

$$w[n] = \begin{cases} \frac{2n+2}{N+1} & n < N/2 \\ \frac{2N-2n}{N+1} & n > N/2 \end{cases}$$

## DSP Run-Time Library Reference

where  $n = \{0, 1, 2, \dots, N-1\}$

### Domain

$a > 0; N > 0$

### Error Conditions

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

### See Also

[gen\\_bartlett](#), [gen\\_blackman](#), [gen\\_gaussian](#), [gen\\_hamming](#), [gen\\_hanning](#),  
[gen\\_harris](#), [gen\\_kaiser](#), [gen\\_rectangular](#)

## histogram

histogram

### Synopsis

```
#include <stats.h>
int *histogram (int out[],
                const int in[],
                int out_len,
                int samples,
                int bin_size);
```

### Description

The `histogram` function computes a scaled-integer histogram of its input array. The `bin_size` parameter is used to adjust the width of each individual bin in the output array. For example, a `bin_size` of 5 indicates that the first location of the output array holds the number of occurrences of a 0, 1, 2, 3 or 4.

The output array is first zeroed by the function, and then each sample in the input array is multiplied by  $1/\text{bin\_size}$  and truncated. The appropriate bin in the output array is incremented. This function returns a pointer to the output array.

All values within the input array must be within range. In order to achieve maximum performance, out of bounds checking is not performed by this function.

### Error Conditions

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

### Example

```
#include <stats.h>
#define SAMPLES 1024
```

## DSP Run-Time Library Reference

```
int length = 2048;  
int excitation[SAMPLES], response[2048];  
histogram (response, excitation, length, SAMPLES, 5);
```

### See Also

[mean](#), [var](#)



## idle

execute ADSP-21xxx processor's IDLE instruction

### Synopsis

```
#include <21060.h>
void idle (void);
```

### Description

The `idle` function invokes the processor's `idle` instruction once and returns. The `idle` instruction causes the processor to stop and respond only to interrupts. For a complete description of the `idle` instruction, please refer to the *ADSP-2106x SHARC User's Manual*.



In previous releases of the VisualDSP++ software (prior to release 2.1), the `idle` function repeatedly executed the `idle` instruction. This function has been changed to give you more control over the amount of time spent in the `idle` state.

### Error Conditions

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

### Example

```
#include <21060.h>
idle ();
```

### See Also

[interrupt](#), [signal](#)

## ifftN

N-point inverse complex input fast Fourier transform (IFFT)

### Synopsis

```
#include <trans.h>
float *ifft65536 (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *ifft32768 (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *ifft16384 (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *ifft8192  (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *ifft4096  (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *ifft2048  (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *ifft1024  (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *ifft512   (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);
```

```
float *ifft256 (const float dm real_input[],
               const float dm imag_input[],
               float dm real_output[], float dm imag_output[]);

float *ifft128 (const float dm real_input[],
               const float dm imag_input[],
               float dm real_output[], float dm imag_output[]);

float *ifft64 (const float dm real_input[],
               const float dm imag_input[],
               float dm real_output[], float dm imag_output[]);

float *ifft32 (const float dm real_input[],
               const float dm imag_input[],
               float dm real_output[], float dm imag_output[]);

float *ifft16 (const float dm real_input[],
               const float dm imag_input[],
               float dm real_output[], float dm imag_output[]);

float *ifft8 (const float dm real_input[],
               const float dm imag_input[],
               float dm real_output[], float dm imag_output[]);
```

### Description

Each of these 14 `ifftN` functions computes the N-point radix-2 inverse fast Fourier transform (IFFT) of its floating-point input (where N is 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768 or 65536).

There are 14 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);
```

The input to `ifftN` are two floating-point arrays of N points. The array `real_input` contains the real components of the inverse FFT input and the array `imag_input` contains the imaginary components.

## DSP Run-Time Library Reference

If there are fewer than  $N$  actual data points, you must pad the arrays with zeros to make  $N$  samples. However, better results occur with less zero padding. The input data should be windowed (if necessary) before calling the function because no preprocessing is performed on the data.

The time-domain signal generated by the `ifftN` functions is stored in the arrays `real_output` and `imag_output`. The array `real_output` will contain the real component of the complex output signal while the array `imag_output` will contain the imaginary component. The output will be scaled by  $N$ , the number of points in the inverse FFT. The functions return a pointer to the `real_output` array.

### Error Conditions

The `ifftN` functions do not return error conditions.

### Example

```
#include <trans.h>
#define N 2048

float real_input[N], imag_input[N];
float real_output[N], imag_output[N];

/* Real input arrays filled from a previous xfft2048() or
   other source */
ifft2048 (real_input, imag_input, real_output, imag_output);
/* Arrays are filled with FFT data */
```

### See Also

[cfftN](#), [rfftN](#)

## iir

infinite impulse response (IIR) filter

### Synopsis

```
#include <filters.h>
float iir (float sample,
          const float pm a_coeffs[],
          const float pm b_coeffs[],
          float dm state[],
          int taps);
```

### Description

The `iir` function implements an infinite impulse response (IIR) filter based on the Oppenheim and Schaffer direct form II. The function returns the filtered response of the input data `sample`. The characteristics of the filter are dependent upon a set of coefficients, a delay line, and the length of the filter. The length of filter is specified by the argument `taps`.

The set of IIR filter coefficients is composed of a-coefficients and b-coefficients. The  $a_0$  coefficient is assumed to be 1.0, and the remaining a-coefficients should be scaled accordingly and stored in the array `a_coeffs` in reverse order. The length of the `a_coeffs` array is `taps` and therefore `a_coeffs[0]` should contain  $a_{taps}$ , and `a_coeffs[taps-1]` should contain  $a_1$ .

The b-coefficients are stored in the array `b_coeffs`, also in reverse order. The length of the `b_coeffs` is `taps+1`, and so `b_coeffs[0]` will contain  $b_{taps}$  and `b_coeffs[taps]` will contain  $b_0$ .

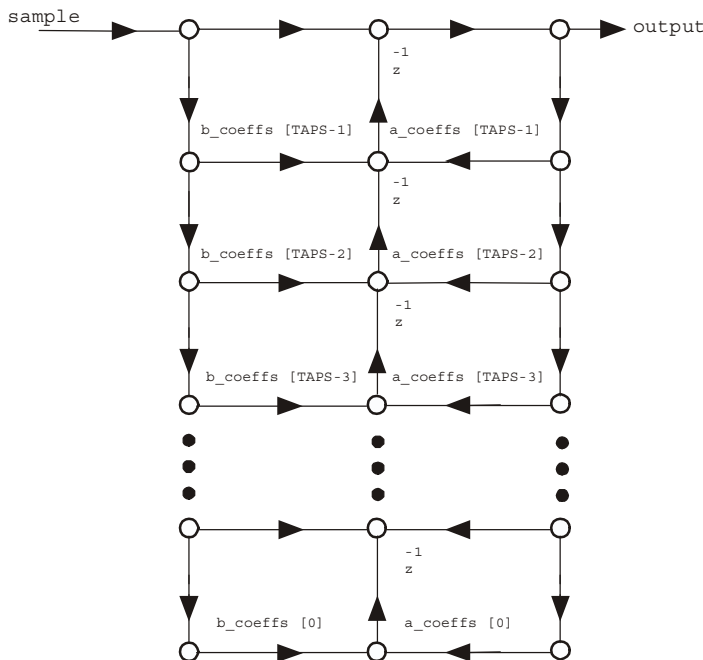
Both the `a_coeffs` and `b_coeffs` arrays must be located in program memory (PM) so that the single-cycle dual-memory fetch of the processor can be used.



When importing coefficients from a filter design tool that employs a transposed direct form II, the a-coefficients have to be negated. For example, if a filter design tool returns  $A = [1.0, 0.2, -0.9]$ , then the a-coefficients will first have to be inverted to  $A = [1.0, -0.2, 0.9]$ .

Each filter should have its own delay line which the function maintains in the `state` array. The array should be initialized to zero before calling the function for the first time and should not otherwise be modified by the calling program. The length of the `state` array should be `taps+1` as the function uses the array to store a pointer to the current delay line.

The flow graph (below) corresponds to the `irr()` routine as part of the DSP Run-Time Library.



The `b_coeffs` array should equal `TAPS+1`  
 The `a_coeffs` array should equal `TAPS`

The `biquad` function should be used instead of the `iir` function if a multi-stage filter is required

### Error Conditions

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

### Example

```
#include <filters.h>
#define TAPS 10
float pm a_coeffs[TAPS], b_coeffs[TAPS+1];
float in_sample, output, state[TAPS+1];
int i;
for (i = 0; i < TAPS+1; i++)
    state[i] = 0;                /* initialize state array */
output = iir (in_sample, a_coeffs, b_coeffs, state, TAPS);
```

### See Also

[biquad](#), [fir](#)

## matadd

matrix addition

### Synopsis

```
#include <matrix.h>
float *matadd (float output[][],
               const float x_input[][],
               const float y_input[][],
               int r,
               int s);
```

### Description

The **matadd** function performs a matrix addition of the input matrices **x\_input[][]** and **y\_input[][]**, returning the result in **output[][]**. The **matadd** function returns a pointer to the output matrix. The dimensions of these matrices are **x\_input[r][s]**, **y\_input[r][s]**, and **output[r][s]**.

### Error Conditions

The **matadd** function does not return an error condition.

### Example

```
#include <matrix.h>
float x[10][20], y[10][20];
float z[10][20];

matadd (z, x, y, 10, 20);
```

### See Also

[matsub](#)



## matinv

real matrix inversion

### Synopsis

```
#include <matrix.h>
float *matinv (float dm *output,
               const float dm *input,
               int n);
```

### Description

The `matinv` function employs Gauss-Jordan elimination with full pivoting to compute the inverse of the input matrix `input` and store the result in the matrix `output`. The dimensions of the matrices `input` and `output` are `[n][n]`.

The function returns a pointer to the output matrix.

### Error Conditions

If no inverse exists for the input matrix, the function will return a null pointer.

### Example

```
#include <matrix.h>

#define N 8
float a[N][N];
float a_inv[N][N];

matinv((float *) (a_inv), (float *) (a), N);
```

### See Also

No references available.

## matmul

matrix multiplication

### Synopsis

```
#include <matrix.h>
float *matmul (float z[],[],
               const float x[],[],
               const float y[],[],
               int r,
               int s,
               int t);
```

### Description

The `matmul` function performs a matrix multiplication of the input matrices `x[][]` and `y[][]`, returning a pointer to the `z[][]` output matrix. The dimensions of these matrices are `x[r][s]`, `y[s][t]`, and `z[r][t]`. The matrix multiplication is defined by the following equation (for `i=0` to `r-1`, for `j=0` to `t-1`):

$$z[i][j] = \sum_{k=0}^{s-1} x[i][k] * y[k][j]$$

### Error Conditions

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

### Example

```
#include <matrix.h>
float x[10][5], y[5][10];
float z[10][10];

matmul (z, x, y, 10, 5, 10);
```

### See Also

[matscalmult](#)

## matscalmult

multiply matrix by scalar

### Synopsis

```
#include <matrix.h>
float *matscalmult (float output[][],
                   const float input[][],
                   float scalar,
                   int r,
                   int s);
```

### Description

The `matscalmult` function performs a scaled multiplication of the `input[][]` matrix, returning the result in `output[][]`. The `input[][]` matrix is multiplied by the input value `scalar`. The `matscalmult` function returns a pointer to the output matrix. The dimensions of these matrices are `input[r][s]`, and `output[r][s]`.

### Error Conditions

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

### Example

```
#include <matrix.h>
float x[10][5], z[10][5];

matscalmult (z, x, 0.5, 10, 5);
/* multiplies the matrix x by 0.5 */
```

### See Also

[matmul](#)

## matsub

matrix subtraction

### Synopsis

```
#include <matrix.h>
float *matsub (float output[],[],
               const float x_input[],[],
               const float y_input[],[],
               int r,
               int s);
```

### Description

The `matsub` function subtracts the elements of the input matrix `y_input[][]` from the input matrix `x_input[][]`, returning the result in `output[][]`. The `matsub` function returns a pointer to the output matrix. The dimensions of these matrices are `x_input[r][s]`, `y_input[r][s]`, and `output[r][s]`.

### Error Conditions

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

### Example

```
#include <matrix.h>

float x[10][5], y[10][5];
float z[10][5];

matsub (z, x, y, 10, 5);
```

### See Also

[matadd](#)

### mean

mean of an array of floating-point numbers

#### Synopsis

```
#include <stats.h>
float mean (const float in[], int length);
```

#### Description

The `mean` function returns the mean of its floating-point input array.

#### Error Conditions

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

#### Example

```
#include <stats.h>
float result, input[256];

result = mean (input, 256);
```

#### See Also

[var](#)

### **mu\_compress**

μ-law compression

#### **Synopsis**

```
#include <comm.h>
int mu_compress (int x);
```

#### **Description**

The `mu_compress` function takes a linear 14-bit signed speech sample and compresses it according to ITU recommendation G.711. The value returned is an 8-bit sample that can be sent directly to a μ-law codec.

#### **Error Conditions**

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

#### **Example**

```
#include <comm.h>
int linear, compressed;

compressed = mu_compress (linear);
```

#### **See Also**

[a\\_compress](#), [mu\\_expand](#)

## mu\_expand

$\mu$ -law expansion

### Synopsis

```
#include <comm.h>
int mu_expand (int x);
```

### Description

The `mu_expand` function takes an 8-bit compressed speech sample and expands it according to ITU recommendation G.711 ( $\mu$ -law definition). The value returned is a linear 14-bit signed sample.

### Error Conditions

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

### Example

```
#include <comm.h>
int compressed_sample, expanded;

expanded = mu_expand (compressed_sample);
```

### See Also

[a\\_expand](#), [mu\\_compress](#)

### poll\_flag\_in

test input flag

#### Synopsis

```
#include <21060.h>
int poll_flag_in (int flag, int mode);
```

#### Description

The `poll_flag_in` function tests the specified flag (0, 1, 2, 3) for the specified transition (0=low to high, 1=high to low, 2=flag high, 3=flag low, 4=any transition, 5=read flag). The function returns a zero *after* the specified transition has occurred in modes 0-3. In Mode 4, it returns the state of the flag after the transition. In Mode 5, it returns the value of the flag without waiting.

Table 3-5. `poll_flag_in` Macros and Values

| Flag Macro | Value | Mode Macro         | Value |
|------------|-------|--------------------|-------|
| READ_FLAG0 | 0     | FLAG_IN_LO_TO_HI   | 0     |
| READ_FLAG1 | 1     | FLAG_IN_HI_TO_LOW  | 1     |
| READ_FLAG2 | 2     | FLAG_IN_HI         | 2     |
| READ_FLAG3 | 3     | FLAG_IN_LOW        | 3     |
| READ_FLAG3 | 3     | FLAG_IN_TRANSITION | 4     |
| READ_FLAG3 | 3     | RETURN_FLAG_STATE  | 5     |

This function assumes that the flag direction in the `MODE2` register is already set as an input (the default state at reset).



### Error Conditions

The `poll_flag_in` function returns a negative value for an invalid flag or transition mode.

### Example

```
#include <21060.h>
poll_flag_in (0, 3);
    /* return zero after transition has occurred */
```

### See Also

[interrupt](#), [set\\_flag](#)

## rfftN

N-point real input fast Fourier transform

### Synopsis

```
#include <trans.h>
float *rfft65536 (const float dm real_input[],
                  float dm real_output[], float dm imag_output[]);

float *rfft32768 (const float dm real_input[],
                  float dm real_output[], float dm imag_output[]);

float *rfft16384 (const float dm real_input[],
                  float dm real_output[], float dm imag_output[]);

float *rfft8192  (const float dm real_input[],
                  float dm real_output[], float dm imag_output[]);

float *rfft4096  (const float dm real_input[],
                  float dm real_output[], float dm imag_output[]);

float *rfft2048  (const float dm real_input[],
                  float dm real_output[], float dm imag_output[]);

float *rfft1024  (const float dm real_input[],
                  float dm real_output[], float dm imag_output[]);

float *rfft256   (const float dm real_input[],
                  float dm real_output[], float dm imag_output[]);

float *rfft128   (const float dm real_input[],
                  float dm real_output[], float dm imag_output[]);

float *rfft64    (const float dm real_input[],
                  float dm real_output[], float dm imag_output[]);

float *rfft32    (const float dm real_input[],
                  float dm real_output[], float dm imag_output[]);
```

```
float *rfft16    (const float dm real_input[],  
                  float dm real_output[], float dm imag_output[]);  
  
float *rfft8     (const float dm real_input[],  
                  float dm real_output[], float dm imag_output[]);
```

### Description

Each of these fourteen `rfftN` functions are similar to the `cfftN` functions, except that they only take real inputs. They compute the N-point radix-2 fast Fourier transform (RFFT) of their floating-point input (where N is 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768 or 65536).

There are fourteen distinct functions in this set. They all perform the same function with same type and number of arguments. Their only difference is the size of the arrays on which they operate.

Call a particular function by substituting the number of points for N; for example,

```
rfft8 (r_inp, r_outp, i_outp);
```

The input to `rfftN` is a floating-point array of N points. If there are fewer than N actual data points, you must pad the array with zeros to make N samples. However, better results occur with less zero padding. The input data should be windowed (if necessary) before calling the function because no preprocessing is performed on the data.

The `rfftN` functions return a pointer to the `real_output` array.

### Error Conditions

The `rfftN` functions do not return any error conditions.

# DSP Run-Time Library Reference

## Example

```
#include <trans.h>
#define N 2048
float real_input[N];
float real_output[N], imag_output[N];

/* Real input array fills from a converter or other source */

rfft2048 (real_input, real_output, imag_output);
/* Arrays are filled with FFT data */
```

## See Also

[cfftN](#), [ifftN](#)

## **rms**

root mean square

### **Synopsis**

```
#include <stats.h>
float rms (const float in[], int length);
```

### **Description**

The `rms` function returns the square root of the mean of the square of its floating-point input array.

### **Error Conditions**

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

### **Example**

```
#include <stats.h>
float input[256], results;

results = rms (input, 256);
```

### **See Also**

[mean](#), [var](#)

## rsqrt, rsqrtf

reciprocal square root

### Synopsis

```
#include <math.h>
double rsqrt (double x);
float rsqrtf (float x);
```

### Description

The `rsqrt` and `rsqrtf` functions return the reciprocal positive square root of their argument.

The `rsqrt` and `rsqrtf` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of  $2^{-20}$  over its input range.

### Error Conditions

The `rsqrt` and `rsqrtf` functions return zero for a negative input.

### Example

```
#include <math.h>
double y;

y = rsqrt (2.0);    /* y = 0.707 */
```

### See Also

[sqrt, sqrtf](#)

## set\_flag

set ADSP-21xxx DSP flags

### Synopsis

```
#include <21060.h>
int set_flag (int flag, int mode);
```

### Description

The `set_flag` function is used to set the ADSP-21xxx DSP flags to the desired output value.

The function accepts as input a flag number [0-3] and a mode. The mode can be specified as a macro (defined in `21060.h`) or a value [0-3].

Table 3-6. Flag Function Macros and Values

| Flag Macro | Value | Mode Macro | Value |
|------------|-------|------------|-------|
| SET_FLAG0  | 0     | SET_FLAG   | 0     |
| SET_FLAG1  | 1     | CLR_FLAG   | 1     |
| SET_FLAG2  | 2     | TGL_FLAG   | 2     |
| SET_FLAG3  | 3     | TST_FLAG   | 3     |

In addition to setting the flag to the specified value, the function also sets the `MODE2` register to specify that the flag is used for output, not input.

If the `TST_FLAG` macro (or a 3) is specified as the mode, the current value (0 or 1) of the flag is returned as the result of the function.

The `set_flag` function returns a zero upon success (except as noted in the previous paragraph).

# DSP Run-Time Library Reference

## Error Conditions

The `set_flag` function returns a non-zero for an error.

## Example

```
#include <21060.h>
set_flag (SET_FLAG0, CLR_FLAG);
set_flag (SET_FLAG0, SET_FLAG);
```

## See Also

[poll\\_flag\\_in](#)



### **set\_semaphore**

set bus lock semaphore

#### **Synopsis**

```
#include <21060.h>
int set_semaphore (
    void dm *semaphore, int set_value, int timeout);
```

#### **Description**

The `set_semaphore` function is used to control bus lock in multiprocessor ADSP-21xxx systems.

- -1 is returned if the bus is locked and the bus lock timeout exceeded.
- 0 is returned if the bus is not locked and a semaphore set.

#### **Error Conditions**

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

#### **See Also**

No references to this function.

## timer\_off

disable ADSP-21xxx DSP timer

### Synopsis

```
#include <21060.h>
unsigned int timer_off (void);
```

### Description

The `timer_off` function disables the ADSP-21xxx 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 <21060.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](#)



The `timer_off` function is not available for the ADSP-21065L chip. Refer to [timer0\\_off](#), [timer1\\_off](#) to disable the ADSP-21065L programmable timers.



The function is supplied only as an inlined procedure; that is, the compiler substitutes the appropriate statements for any reference to the procedure. Therefore, any source that references `timer_off` must include the `21060.h` header file.

## timer0\_off, timer1\_off

disable ADSP-21065L DSP timers

### Synopsis

```
#include <21065l.h>
unsigned int timer0_off (void);
unsigned int timer1_off (void);
```

### Description

The `timer0_off` and `timer1_off` functions disable the ADSP-21065L programmable timers and return the current value of the `TCOUNT0` and `TCOUNT1` registers respectively.

### Error Conditions

The `timer0_off` and `timer1_off` functions do not return an error condition.

### Example

```
#include <21065l.h>
unsigned int hold_tcount;
hold_tcount = timer0_off ();
/* hold_tcount contains value of TCOUNT0 */
/* register AFTER timer 0 has stopped */
```

### See Also

[timer0\\_on](#), [timer1\\_on](#), [timer0\\_set](#), [timer1\\_set](#)



The functions are supplied only as inlined procedures; that is, the compiler substitutes the appropriate statements for any reference to the procedures. Therefore, any source that references either `timer0_off` or `timer1_off` must include the `21065l.h` header file.

## timer\_on

enable ADSP-21xxx DSP timer

### Synopsis

```
#include <21060.h>
unsigned int timer_on (void);
```

### Description

The `timer_on` function enables the ADSP-21xxx timer and returns the current value of the `TCOUNT` register.

### Error Conditions

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

### Example

```
#include <21060.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](#)



The `timer_on` function is not available for the ADSP-21065L chip. Refer to [“timer0\\_on, timer1\\_on” on page 3-85](#) to enable the ADSP-21065L programmable timers.



The function is supplied only as an inlined procedure; that is, the compiler substitutes the appropriate statements for any reference to the procedure. Therefore, any source that references `timer_on` must include the `21060.h` header file.

## timer0\_on, timer1\_on

enable ADSP-21065L DSP timers

### Synopsis

```
#include <210651.h>
unsigned int timer0_on (void);
unsigned int timer1_on (void);
```

### Description

The `timer0_on` and `timer1_on` functions enable the ADSP-21065L programmable timers and return the current value of the `TCOUNT0` and `TCOUNT1` registers, respectively.

### Error Conditions

The `timer0_on` and `timer1_on` functions do not return an error condition.

### Example

```
#include <210651.h>
unsigned int hold_tcount;
hold_tcount = timer0_on ();
/* hold_tcount contains value of TCOUNT0 */
/* register when timer 0 starts          */
```

### See Also

[timer0\\_off](#), [timer1\\_off](#), [timer0\\_set](#), [timer1\\_set](#)



The functions are supplied only as inlined procedures; that is, the compiler substitutes the appropriate statements for any reference to the procedures. Therefore, any source that references either `timer0_on` or `timer1_on` must include the `210651.h` header file.

## timer\_set

initialize ADSP-21xxx DSP timer

### Synopsis

```
#include <21060.h>
int timer_set (unsigned int tperiod,
               unsigned int tcount);
```

### Description

The `timer_set` function sets the ADSP-21xxx DSP timer registers `TPERIOD` and `TCOUNT`. The function returns a 1 if the timer is enabled, or a zero if the timer is disabled.

 Each interrupt call takes approximately 50 cycles on entrance and 50 cycles on return. If `TPERIOD` and `TCOUNT` registers are set too low, you may incur an initializing overhead that could create an infinite loop.

### Error Conditions

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

### Example

```
#include <21060.h>
if (timer_set (1000, 1000) != 1)
    timer_on ();    /* enable timer */
```

### See Also

[timer\\_on](#), [timer\\_off](#)

 The `timer_set` function is not available for the ADSP-21065L chip. Refer to [timer0\\_set](#), [timer1\\_set](#) to initialize the ADSP-21065L programmable timers.



The function is supplied only as an inlined procedure; that is, the compiler substitutes the appropriate statements for any reference to the procedure. Therefore, any source that references `timer_set` must include the `21060.h` header file.

## timer0\_set, timer1\_set

initialize ADSP-21065L DSP timers

### Synopsis

```
#include <21065l.h>
int timer0_set (unsigned int tperiod,
               unsigned int tcount,
               unsigned int tscale);
int timer1_set (unsigned int tperiod,
               unsigned int tcount,
               unsigned int tscale);
```

### Description

The `timer0_set` and `timer1_set` functions set the ADSP-21065L timer registers `TPERIOD0`, `TCOUNT0`, `TPWIDTH0` and `TPERIOD1`, `TCOUNT1`, `TPWIDTH1` respectively. The functions return a 1 if the corresponding timer is enabled, or a zero if the timer is disabled.

 Each interrupt call takes approximately 50 cycles on entrance and 50 cycles on return. If `TPERIOD` and `TCOUNT` registers are set too low, you may incur an initializing overhead that could create an infinite loop.

### Error Conditions

The `timer0_set` and `timer1_set` functions do not return an error condition.

### Example

```
#include <21065l.h>
unsigned int hold_tcount;
if (timer0_set (200, 1, 150) != 1)
    timer0_on ();    /* enable timer 0 */
```



### See Also

[timer0\\_off](#), [timer1\\_off](#), [timer0\\_on](#), [timer1\\_on](#)



The functions are supplied only as inlined procedures; that is, the compiler substitutes the appropriate statements for any reference to the procedures. Therefore, any source that references either `timer0_set` or `timer1_set` must include the `210651.h` header file.

## transpm

real matrix transpose

### Synopsis

```
#include <matrix.h>
void transpm (float *output,
              const float *input,
              int rows,
              int columns);
```

### Description

The `transpm` function computes the linear algebraic transpose of the input matrix `input` and stores the result in the matrix `output`. The dimensions of the input matrix are `[rows][columns]`, and the dimensions of the output matrix are `[columns][rows]`.

The function returns a pointer to the output matrix.

### Error Conditions

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

### Example

```
#include <matrix.h>

float a[4][8];
float a_transpose[8][4];

transpm((float *) (a_transpose), (float *) (a), 4, 8);
```

### See Also

No references to this function.

## **var**

variance

### **Synopsis**

```
#include <stats.h>
float var (const float in[], int length);
```

### **Description**

The `var` function returns the variance of its floating-point input array.

### **Error Conditions**

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

### **Example**

```
#include <stats.h>
float input[256], result;

result = var (input, 256);
```

### **See Also**

[mean](#)

## vecdot

vector dot product

### Synopsis

```
#include <vector.h>
float vecdot (const float dm x_input[],
              const float dm y_input[],
              int samples);
```

### Description

The `vecdot` function computes the dot product of the vectors, `x_input` and `y_input`, which are `samples` in size. The scalar result is returned by the function.

### Error Conditions

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

### Example

```
#include <vector.h>
float answer, x[100], y[100];

answer = vecdot (x, y, 100);
```

### See Also

[cvecdot](#)

## vecsadd

vector scalar addition

### Synopsis

```
#include <vector.h>
float *vecsadd (const float dm input[],
               float scalar,
               float dm output[],
               int samples);
```

### Description

The `vecsadd` function computes the sum of each element of the vector, input, added to the scalar. Both the input and output vectors are `samples` in length. The function returns a pointer to the output vector.

### Error Conditions

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

### Example

```
#include <vector.h>
float answer[50], x[50], y;

vecsadd (x, y, answer, 50);
```

### See Also

[cvecsadd](#)

## vecsm1t

vector scalar multiply

### Synopsis

```
#include <vector.h>
float *vecsm1t (const float dm input[],
               float scalar,
               float dm output[],
               int samples);
```

### Description

The `vecsm1t` function computes the product of each element of the vector, `input`, multiplied by the scalar. Both the input and output vectors are `samples` in length. The function returns a pointer to the output vector.

### Error Conditions

The `vecsm1t` function does not return an error condition.

### Example

```
#include <vector.h>
float answer[50], x[50], y;

vecsm1t (x, y, answer, 50);
```

### See Also

[cvecsm1t](#)

## vecssub

vector scalar subtraction

### Synopsis

```
#include <vector.h>
float *vecssub (const float dm input[],
               float scalar,
               float dm output[],
               int samples);
```

### Description

The `vecssub` function computes the difference of each element of the vector, `input`, minus the scalar. Both the input and output vector are `samples` in length. The function returns a pointer to the output vector.

### Error Conditions

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

### Example

```
#include <vector.h>
float answer[50], x[50], y;

vecssub (x, y, answer, 50);
```

### See Also

[cvecssub](#)

## vecvadd

vector addition

### Synopsis

```
#include <vector.h>
float *vecvadd (const float dm x_input[],
               const float dm y_input[],
               float dm output[],
               int samples);
```

### Description

The `vecvadd` function computes the sum of each of the elements of the vectors, `x_input` and `y_input`, and places the results in `output`. All three vectors are `samples` in length. The function returns a pointer to the output vector.

### Error Conditions

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

### Example

```
#include <vector.h>
float answer[50], x[50], y[50];

vecvadd (x, y, answer, 50);
```

### See Also

[cvecvadd](#)



## vecvmlt

vector multiply

### Synopsis

```
#include <vector.h>
float *vecvmlt (const float dm x_input[],
               const float dm y_input[],
               float dm output[],
               int samples);
```

### Description

The `vecvmlt` function computes the product of each of the elements of the vectors, `x_input` and `y_input`, and places the results in `output`. All three vectors are `samples` in length. The function returns a pointer to the output vector.

### Error Conditions

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

### Example

```
#include <vector.h>
float answer[50], x[50], y[50];

vecvmlt (x, y, answer, 50);
```

### See Also

[cvecvmlt](#)

## vecvsub

vector subtraction

### Synopsis

```
#include <vector.h>
float *vecvsub (const float dm x_input[],
               const float dm y_input[],
               float dm output[],
               int samples);
```

### Description

The `vecvsub` function computes the difference of each of the elements of the vectors, `x_input` and `y_input`, and places the results in `output`. All three vectors are `samples` in length. The function returns a pointer to the output vector.

### Error Conditions

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

### Example

```
#include <vector.h>
float answer[50], x[50], y[50];

vecvsub (x, y, answer, 50);
```

### See Also

[cvecvsub](#)

## zero\_cross

count zero crossings

### Synopsis

```
#include <stats.h>
int zero_cross (const float in[], int length);
```

### Description

The `zero_cross` function returns the number of times that a signal represented in the input array crosses over the zero line. If all input values are zero, the function returns a zero.

### Error Conditions

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

### Example

```
#include <stats.h>
float input[256];
int results;

results = zero_cross (input, 256);
```

### See Also

No references to this function.



# 4 DSP LIBRARY FOR ADSP-2116X PROCESSORS

This chapter describes the DSP run-time library for ADSP-2116x processors which contains a collection of functions that provide services commonly required by DSP applications; these functions are in addition to the C/C++ run-time library functions that are described in Chapter 2, “C Run-Time Library Reference”. The services provided by the DSP library functions include support for interrupt handling, signal processing, and access to hardware registers. All these services are Analog Devices extensions to ANSI standard C.

The chapter contains:

- “DSP Run-Time Library Guide” (starting on [page 4-2](#)) contains introductory information about the ADI special header files and built-in functions that are included with this release of the cc21k compiler.
- “DSP Run-Time Library Reference” (starting on [page 4-15](#)) contains the complete reference information for each DSP run-time library function included with this release of the cc21k compiler.

For more information on the algorithms on which many of the C library's math functions are based, see Cody, W. J. and W. Waite, *Software Manual for the Elementary Functions*, Englewood Cliffs, New Jersey: Prentice Hall, 1980.

# DSP Run-Time Library Guide

The DSP run-time library contains routines that you can call from your source program. This section describes how to use the library and provides information on the following topics:

- [“Calling DSP Library Functions” on page 4-2](#)
- [“Linking DSP Library Functions” on page 4-3](#)
- [“Working With Library Source Code” on page 4-3](#)
- [“DSP Header Files” on page 4-4](#)
- [“Built-In DSP Functions” on page 4-12](#)
- [“Implications of Using SIMD Mode” on page 4-13](#)

For information on the contents of the DSP library, see [“DSP Run-Time Library Reference” on page 4-15](#) and on-line Help.

## Calling DSP Library Functions

To use a DSP library function, call the function by name and give the appropriate arguments. The names and arguments for each function are described in the function's reference page in the section [“DSP Run-Time Library Reference” on page 4-15](#).

Like other functions you use, library functions should be declared. Declarations are supplied in header files, as described in the section, [“Working With Library Source Code” on page 4-3](#).



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, which is the func-

tion name prefixed with an underscore. For more information on naming conventions, see “C/C++ and Assembly Interface” on [page 1-154](#).



You can use the archiver, described in the *VisualDSP++ 3.0 Linker and Utilities Manual for SHARC DSPs*, to build library archive files of your own functions.

## Linking DSP Library Functions

When your C code calls a DSP run-time library function, the call creates a reference that the linker resolves when linking your program. One way to direct the linker to the location of the DSP library is to use the default Linker Description File (ADSP-21<your\_target>.ldf). The default Linker Description File will automatically direct the linker to the library `libdsp160.dlb` in the `21xxx\lib` subdirectory of your VisualDSP++ installation. If not using the default .LDF file, then either add `libdsp160.dlb` to the .LDF file used for your project, or alternatively use the compiler's `-ldsp160` switch to specify that `libdsp160.dlb` is to be added to the link line.

## Working With Library Source Code

The source code for the functions and macros in the DSP run-time 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 `...\21xxx\lib\src`. The directory contains the source for the C run-time library, for the DSP run-time library, and for the I/O run-time library, as well as the source for the main program start-up functions. If you do not intend to modify any of the run-time library functions, you can delete this directory and its contents to conserve disk space.

## DSP Run-Time Library Guide

The source code is provided so you can customize any particular function for your own needs. To modify these files, you need proficiency in ADSP-21xxx assembly language and an understanding of the run-time environment, as explained in [“C/C++ Run-Time Model and Environment” on page 1-123](#). 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. Note that Analog Devices supports the run-time library functions only as provided.

## DSP Header Files

The following header files are supplied with this release of the cc21k compiler.

### 21160.h — ADSP-2116x DSP Functions

The `21160.h` header file includes the ADSP-2116x processor-specific functions of the DSP library, such as `poll_flag_in()`, `timer_set()`, and `idle()`. The `timer_set()`, `timer_on()`, and `timer_off()` functions are also available as in-line functions.

### asm\_sprt.h — Mixed C/Assembly Support

The `asm_sprt.h` header file consists of ADSP-21xxx assembly language macros, not C functions. They are used in your assembly routines that interface with C functions. For more information on this header file, see [“Using Mixed C/C++ and Assembly Support Macros” on page 1-159](#).

### comm.h — A-law and $\mu$ -law Companders

The `comm.h` header file includes the voice-band compression and expansion communication functions as supported by the DSP library for ADSP-2106x DSP processors. However, the functions defined by this header file have not been optimized for the ADSP-2116x architecture.



Versions of these functions that have been optimized for the ADSP-2116x architecture are available in the header file `filter.h`. Since these two sets of functions have different arguments, it is important that the user program includes the appropriate header file.

### **complex.h — Basic Complex Arithmetic Functions**

The `complex.h` header file contains the type definition and some basic functions for `complex_float` variables.

The following structure is used to represent complex numbers in rectangular coordinates.

```
typedef struct {  
    float re;  
    float im;  
} complex_float;
```

Additional support for `complex_float` is available via the `cvector.h` header file.

### **cvector.h — Complex Vector Functions**

The `cvector.h` header file contains functions for basic arithmetical operations on vectors of type `complex_float`. Support is provided for the dot product operation, as well as for adding, subtracting, and multiplying a vector by either a scalar or vector.

### **def21160.h — ADSP-21160 DSP Bit Definitions**

The `def21160.h` header file includes macro definitions to enable usage of symbolic names for the system register bits for the ADSP-21160 processors. It also contains macro definitions for the IOP register addresses and bit fields.

### **def21161.h — ADSP-21161 DSP Bit Definitions**

The `def21161.h` header file includes macro definitions to enable usage of symbolic names for the system register bits for the ADSP-21161 processors. It also contains macro definitions for the IOP register addresses and bit field.

### **dma.h — DMA Support Functions**

The `dma.h` header file provides definitions and setup, status, enable, and disable functions for DMA operations.

### **filter.h — DSP Filters and Transformations**

The `filter.h` header file contains filters used in digital signal processing. It also includes the A-law and  $\mu$ -law companders that are used by voice-band compression and expansion applications.

Additionally, the header file contains the Fast Fourier Transform (FFT) functions of the DSP library. The various forms of the FFT functions provided are `cfftN`, `ifftN`, and `rfftN`. Each form is represented by N separate functions, where N represents the number of points supported by the FFT. For example, `cfftN` stands for the functions `cfft65536`, `cfft32768`, and so forth.

The prototypes defined by this header file have been designed to allow the functions to exploit the parallelism within the ADSP-2116x architecture. Equivalent functions that have not been optimized for the ADSP-2116x architecture are supported and are documented in Chapter 3, “[DSP Library for ADSP-2106x and ADSP-21020 Processors](#)”. These functions are defined in separate header files (`comm.h` for the companding functions, `filters.h` for the filters, and `trans.h` for the FFTs). Since these two sets of functions have different arguments, it is important that the user’s program includes the appropriate header file.

The FFT functions are optimized to take advantage of the SIMD (Single Instruction Multiple Data) execution model of the ADSP-2116x SHARC DSP, and the function prototypes have been adjusted accordingly.

Complex arguments and complex results must be defined using the `complex_float` data type (defined in the header file `complex.h`). Using the `complex_float` data type ensures that the real and imaginary parts of a complex value are interleaved in memory and therefore are accessible in a single cycle using the wider data bus of the processor.

The FFT functions have also been optimized with respect to memory usage. In general, Fast Fourier Transform algorithms require access to a temporary working buffer. All the FFT functions defined by the `filter.h` header file assume that the input buffer may be corrupted and therefore may be used as a temporary area. This assumption allows the functions to be more efficient both in terms of memory usage and processor speed.

The `cfftN` functions compute the fast Fourier transform 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 a `complex_float` array of N elements. The routines output an N-element array of type `complex_float`. If you wish only to input the real part of a signal, ensure that the imaginary components of the input array are set to zero before calling the function.

The functions first bit-reverse the input arrays and then process them with an optimized block-floating-point FFT (or IFFT) routine.

The `rffftN` functions work like `cfftN` functions, except they operate only on input arrays of real data. This type of operation is equivalent to a `cfftN` function whose imaginary input component is set to zero.

The header file also defines a complex FFT function that has been implemented using a fast radix-2 algorithm. However, this function, `cfftff`, has certain requirements that may not be appropriate for some applications.

## filters.h — DSP Filters

The `filters.h` header file includes the digital signal processing filter functions as supported by the DSP library for ADSP-2106x DSP processors. However, the functions defined by this header file have not been optimized for the ADSP-2116x architecture. Versions of these functions that have been optimized for the ADSP-2116x architecture are available in the `filter.h` header file. Since these two sets of functions have different arguments, it is important that the user program includes the appropriate header file.

## macro.h – Circular Buffers

The `macro.h` header file consists of ADSP-21xxx assembly language macros, not C functions. Some are used to manipulate the circular buffer features of the ADSP-21xxx processors.

## math.h — Math Functions

The standard math functions defined in `math.h` have been augmented by implementations for the `float` data type and some additional functions that are Analog Devices extensions to the ANSI standard. [Table 4-1](#) provides a summary of the additional library functions defined by the header file.

Table 4-1. Math Library - Additional Functions

| Description | Prototype                                                                                              |
|-------------|--------------------------------------------------------------------------------------------------------|
| sign copy   | <code>double copysign (double x, double y);</code><br><code>float copysignf (float x, float y);</code> |
| cotangent   | <code>double cot (double x);</code><br><code>float cotf (float x);</code>                              |
| average     | <code>double favg (double x, double y);</code><br><code>float favgf (float x, float y);</code>         |

Table 4-1. Math Library - Additional Functions (Cont'd)

| Description               | Prototype                                                              |
|---------------------------|------------------------------------------------------------------------|
| clip                      | double fclip (double x, double y);<br>float fclipf (float x, float y); |
| detect Infinity           | int isinf (double x);<br>int isinff (float x);                         |
| detect NaN                | int isnan (double x);<br>int isnanf (float x);                         |
| maximum                   | double fmax (double x, double y);<br>float fmaxf (float x, float y);   |
| minimum                   | double fmin (double x, double y);<br>float fminf (float x, float y);   |
| reciprocal of square root | double rsqrt (double x, double y);<br>float rsqrtf (float x, float y); |



The following functions are only available if `double` is the same size as `float`: `copysign`, `favg`, `fclip`, `fmax`, and `fmin`.

## matrix.h — Matrix Functions

The `matrix.h` header file contains functions for operating on real matrices; specifically it defines functions for adding and subtracting two matrices and for multiplying a matrix by either a scalar or a matrix. Functions are also available to compute the inverse and transpose of a matrix.

## saturate.h — Saturation Mode Arithmetic

The `saturate.h` header file defines the interface for the saturated arithmetic operations. See [“Saturated Arithmetic” on page 1-99](#) for further information.

### **sport.h — Serial Port Support Functions**

The `sport.h` header file provides definitions and setup, enable, and disable functions for the ADSP-2116x DSP serial ports.

### **stats.h — Statistical Functions**

The `stats.h` header file includes various statistics functions of the DSP library, such as `mean()` and `autocorr()`.

### **sysreg.h — Register Access**

The `sysreg.h` header file defines a set of built-in functions that provide efficient access to the SHARC system registers from C. The supported functions are fully described in [“Access to System Registers” on page 1-87](#).

### **trans.h — Fast Fourier Transforms**

The `trans.h` header file includes Fast Fourier Transform (FFT) functions as supported by the DSP library for ADSP-2106x processors. However, the functions defined by this header file have not been optimized for the ADSP-2116x architecture. Versions of these functions that have been optimized for the ADSP-2116x architecture are available in the header file `filter.h`. Since these two sets of functions have different arguments, it is important that the user program includes the appropriate header file.

### **vector.h — Vector Functions**

The `vector.h` header file contains functions for operating on vectors of type `float`. Support is provided for the dot product operation and well as for adding, subtracting, and multiplying a vector by either a scalar or vector.

## window.h — Window Generators

The `window.h` header file contains various functions to generate windows based on various methodologies. The functions defined in the `window.h` header file are listed in [Table 4-2](#).

For all window functions, a stride parameter `a` can be used to space the window values. The window length parameter `n` equates to the number of elements in the window. Therefore, for a stride `a` of 2 and a length `n` of 10, an array of length 20 is required, where every second entry is untouched.

Table 4-2. Window Generator Functions

| Description                 | Prototype                                                                 |
|-----------------------------|---------------------------------------------------------------------------|
| generate bartlett window    | <code>void gen_bartlett<br/>(float w[], int a, int n)</code>              |
| generate blackman window    | <code>void gen_blackman<br/>(float w[], int a, int n)</code>              |
| generate gaussian window    | <code>void gen_gaussian<br/>(float w[], float alpha, int a, int n)</code> |
| generate hamming window     | <code>void gen_hamming<br/>(float w[], int a, int n)</code>               |
| generate hanning window     | <code>void gen_hanning<br/>(float w[], int a, int n)</code>               |
| generate harris window      | <code>void gen_harris<br/>(float w[], int a, int n)</code>                |
| generate kaiser window      | <code>void gen_kaiser<br/>(float w[], float beta, int a, int n)</code>    |
| generate rectangular window | <code>void gen_rectangular<br/>(float w[], int a, int n)</code>           |
| generate triangle window    | <code>void gen_triangle<br/>(float w[], int a, int n)</code>              |

## Built-In DSP Functions

The C/C++ compiler supports built-in functions (also known as intrinsic functions) that enable efficient use of hardware resources. Knowledge of these functions is built into the compiler. Your program uses them via normal function call syntax. The compiler notices the invocation and replaces a call to a DSP library function with one or more machine instructions—just as it does for normal operators like “+” and “\*”.

Built-in functions are declared in system header files and have names which begin with double underscores, `__builtin`.

 Identifiers beginning with “\_\_” are reserved by the C standard, so these names do not conflict with user defined identifiers.

These functions are specific to individual architectures. The built-in DSP library functions supported at this time on the ADSP-2116x architectures are listed in [Table 4-3](#). Refer to [“Using Compiler’s Built-In C Library Functions” on page 2-19](#) for further information on this topic.

Table 4-3. Built-in DSP Functions

|                       |                        |
|-----------------------|------------------------|
| <code>copysign</code> | <code>copysignf</code> |
| <code>favg</code>     | <code>favgf</code>     |
| <code>fmax</code>     | <code>fmaxf</code>     |
| <code>fmin</code>     | <code>fminf</code>     |

 Functions `copysign`, `favg`, `fmax`, and `fmin` will only be compiled as a built-in function if `double` is the same size as `float`.

 Use the `-no-builtin` compiler switch to disable this feature.

The compiler also supports a set of built-in functions for which no in-line machine instructions are substituted. This set of built-in functions is characterized by defining one or more pointers in their argument list.



For this set of built-in functions, the compiler relaxes the normal rule whereby any pointer that is passed to a library function must address Data Memory (DM). The compiler recognizes when certain pointers address Program Memory (PM) and will generate a call to an appropriate version of the run-time library function. [Table 4-4](#) lists library functions that may be called with pointers that address Program Memory.

Table 4-4. Library Functions Called with Pointers

|           |              |
|-----------|--------------|
| histogram | matadd       |
| matmul    | matscaltmult |
| matsub    | mean         |
| rms       | transpm      |
| var       |              |



Use the `-no-builtin` compiler switch (see [on page 1-35](#)) to disable this feature.

## Implications of Using SIMD Mode

The DSP run-time library for the ADSP-2116x DSP makes extensive use of the DSP's SIMD capabilities. In essence, when running in SIMD mode, data contained in memory is always accessed as two 32-bit words, starting at an even word boundary. Therefore, it is essential that any array that is passed to a DSP library function be allocated on a double-word (even word) boundary.

The `cc21k` compiler normally aligns arrays properly in memory. However, the compiler cannot control the allocation of all arrays that are used as arguments to DSP library functions. For example, the alignment of the array `&a[i]` is controlled by the value of the scalar `i`. If the value of the scalar is odd, then the library function will return incorrect results. A vari-

ant of this example involves the use of pointers to arrays. If the variable `ptr` is initialized using `ptr=&a[i]` and the value of the scalar `i` is odd, then you cannot use `ptr` to pass an array to a DSP library function.

For more information on this topic, refer to [“Restrictions to Using SIMD” on page 1-103](#).

A limited number of DSP library functions, whose arguments involve the use of arrays, do not use the SIMD feature of a ADSP-2116x DSP due to the nature of their algorithm. These library functions are:

```
histogram  
iir  
zero_cross  
matmul
```

Some ADSP-2116x DSP architectures only have a 32-bit external bus and, because of this shorter bus length, the effect of SIMD accesses to external memory on such architectures will not be the same as SIMD accesses to internal memory. For this reason, the DSP library contains a alternative set of functions that do not use the architecture's SIMD capabilities. This alternative set is selected in preference to the standard library functions if the `-no-simd` compiler switch (see [on page 1-37](#)) is specified at compilation time.



The SIMD feature is described in detail in [“SIMD Support” on page 1-99](#).

## DSP Run-Time Library Reference

The DSP run-time library is a collection of functions that you can call from your C programs. This section lists the functions in alphabetical order. Note the following items that apply to all 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.

**Function Benchmarks and Specifications.** All functions have been timed from setup, to invocation, to results storage of returned value. This includes all register storing, parameter passing, etc. Most functions execute slightly faster if you pass constants as arguments instead of variables.

**Restrictions.** When polymorphic functions are used and the function returns a pointer to program memory, cast the output of the function to pm. For example,

```
(char pm *)
```

**Reference Format.** Each function in the library has a reference page. These pages have the following format:

**Name** and Purpose of the function

**Synopsis**—Required header file and functional prototype

**Description**—Function specification

**Error Conditions**—Method function uses to indicate an error

**Example**—Typical function usage

**See Also**—Related functions

## **a\_compress**

A-law compression

### **Synopsis**

```
#include <filter.h>
int *a_compress (const int dm input[],
                 int dm output[],
                 int length);
```

### **Description**

The `a_compress` function takes an array of linear 13-bit signed speech samples and compresses them according to ITU recommendation G.711. The array returned contains 8-bit samples that can be sent directly to an A-law codec. This function returns a pointer to the `output` data array.

The function uses serial port 0 to perform companding. Therefore, serial port 0 must not be in use when this routine is called.

### **Error Conditions**

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

### **Example**

```
#include <filter.h>
int data[50], compressed[50];
a_compress (data, compressed, 50);
```

### **See Also**

[a\\_expand](#), [mu\\_compress](#)

## **a\_expand**

A-law expansion

### **Synopsis**

```
#include <filter.h>
int *a_expand (const int dm input[],
               int dm output[],
               int length);
```

### **Description**

The `a_expand` function takes an array of 8-bit compressed speech samples and expands them according to ITU recommendation G.711 (A-law definition). The array returned contains linear 13-bit signed samples. This function returns a pointer to the `output` data array.

The function uses serial port 0 to perform companding. Therefore, serial port 0 must not be in use when this routine is called.

### **Error Conditions**

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

### **Example**

```
#include <filter.h>
int expanded_data[50], compressed_data[50];

a_expand (compressed_data, expanded_data, 50);
```

### **See Also**

[a\\_compress](#), [mu\\_expand](#)

## autocoh

autocoherence

### Synopsis

```
#include <stats.h>
float *autocoh (float dm out[],
               const float dm in[],
               int samples,
               int lags);
```

### Description

The `autocoh` function computes the autocohereence of the floating-point input, `in[]`. The autocohereence of an input signal is its autocorrelation minus its mean. The function returns a pointer to the output array, `out[]` of length `lags`.

### Error Conditions

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

### Example

```
#include <stats.h>
#define SAMPLES 1024

float excitation[SAMPLES], response[16];
int lags = 16;

autocoh (response, excitation, SAMPLES, lags);
```

### See Also

[autocorr](#), [crosscoh](#), [crosscorr](#)



By default, this function uses SIMD; refer to [“Implications of Using SIMD Mode” on page 4-13](#) for more information.

## autocorr

autocorrelation

### Synopsis

```
#include <stats.h>
float *autocorr (float dm out[],
                const float dm in[],
                int samples,
                int lags);
```

### Description

The `autocorr` function performs an autocorrelation of a signal. Autocorrelation is the cross-correlation of a signal with a copy of itself. It provides information about the time variation of the signal. The signal to be autocorrelated is given by the `in[]` input array. The number of samples of the autocorrelation sequence to be produced is given by `lags`. The length of the input sequence is given by `samples`. This function returns a pointer to the `out[]` output data array of length `lags`.

The `autocorr` function is used in digital signal processing applications such as speech analysis.

### Error Conditions

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

### Example

```
#include <stats.h>
float r[10], s[160];

autocorr (r, s, 160, 10);
/* compute first 10 autocorr coefficients of array s */
```

### See Also

[autocoh](#), [crosscoh](#), [crosscorr](#)



By default, this function uses SIMD; refer to [“Implications of Using SIMD Mode” on page 4-13](#) for more information.



## biquad

biquad filter section

### Synopsis

```
#include <filter.h>
float biquad (float sample,
             const float pm coeffs[],
             float dm state[],
             int sections);
```

### Description

The `biquad` function implements a cascaded biquad filter. The function produces the filtered response of its input data. The parameter `sections` specifies the number of biquad sections.

Each biquad section is represented by five coefficients that must be stored in the order B2, B1, B0, A2, A1. The value of A0 is assumed to be 1.0, and A1 and A2 should be scaled accordingly. The coefficients should be stored in the array `coeffs` which must be located in program memory (PM). The definition of the `coeffs` array is:

```
float pm coeffs[5*sections];
```



When importing coefficients from a filter design tool that employs a transposed direct form II, the A coefficients have to be negated. For example, if a filter design tool returns  $A = [1.0, 0.2, -0.9]$ , then the A coefficients have to be modified to  $A = [1.0, -0.2, 0.9]$ .

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 equal to  $2*sections + 1$ . The definition is:

```
float dm state[2*sections + 1];
```

## DSP Run-Time Library Reference

Each filter has its own state array which should be initialized to zero before calling the `biquad` function for the first time, and should not otherwise be modified by the calling program.

### Error Conditions

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

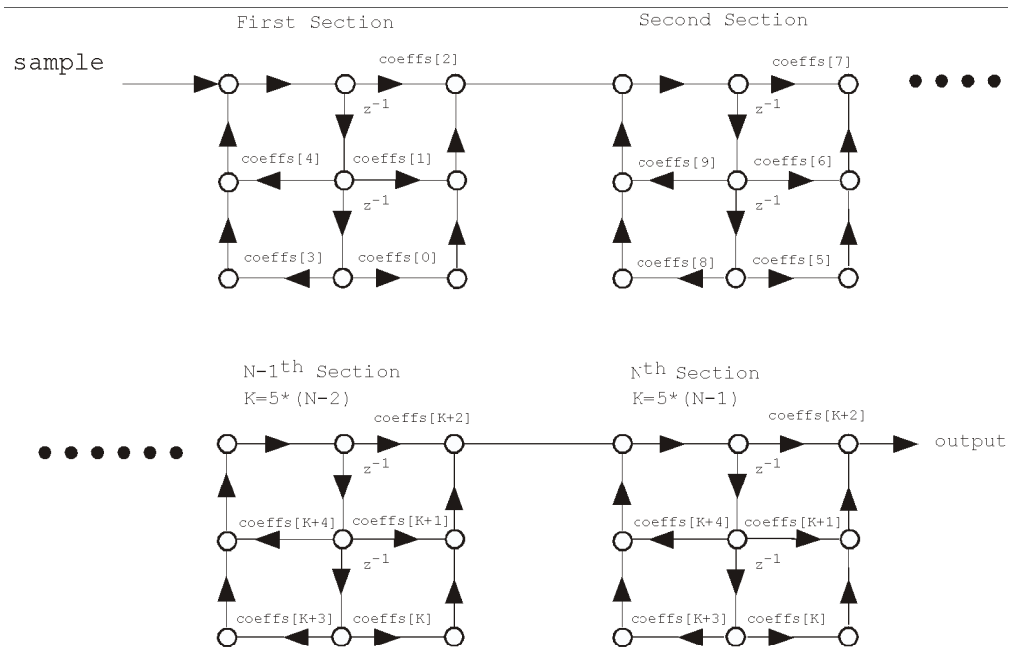
### Example

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

float sample, output, state[2*TAPS+1];
float pm coeffs[5*TAPS];
int i;

for (i = 0; i < 2*TAPS+1; i++)
    state[i] = 0;    /* initialize state array */

output = biquad (sample, coeffs, state, TAPS);
```



$N$  = the number of biquad sections.

The algorithm shown here is adapted from Oppenheim, Alan V. and Ronald Schaffer, *Digital Signal Processing*, Englewood Cliffs, New Jersey: Prentice Hall, 1975.

See Also

[fir](#), [iir](#)

## **cabsf**

complex absolute value

### **Synopsis**

```
#include <complex.h>
float cabsf (complex_float z);
```

### **Description**

The `cabsf` function returns the floating-point absolute value of its complex input. The absolute value of a complex number is evaluated as:

$$y = \sqrt{(\operatorname{Re}(x))^2 + (\operatorname{Im}(x))^2}$$

where `x` is the `complex_float` input and `y` is the `float` output.

### **Error Conditions**

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

### **Example**

```
#include <complex.h>
complex_float cnum;
float answer;

cnum.re = 12.0;
cnum.im = 5.0;

answer = cabsf (cnum);      /* answer = 13.0 */
```

### **See Also**

[fabs](#), [fabsf](#), [labs](#)

## cexpf

complex exponential

### Synopsis

```
#include <complex.h>
complex_float cexpf (complex_float z);
```

### Description

The `cexpf` function computes the complex exponential value  $e$  to the power of the first argument. The exponential of a complex value is evaluated with the following formula:

$$\begin{aligned}\text{Re}(y) &= \expf(\text{Re}(x)) * \cosf(\text{Im}(x)); \\ \text{Im}(y) &= \expf(\text{Re}(x)) * \sinf(\text{Im}(x));\end{aligned}$$

where `x` is the `complex_float` input and `y` is the `complex_float` output.

### Error Conditions

For underflow errors the `cexpf` function returns zero.

### Example

```
#include <complex.h>
complex_float cnum;
complex_float answer;

cnum.re = 1.0;
cnum.im = 0.0;

answer = cexpf (cnum);      /* answer = (2.7182 + 0i) */
```

### See Also

[log](#), [logf](#), [pow](#), [powf](#)

## cfftN

N-point complex input fast Fourier transform

### Synopsis

```
#include <filter.h>
complex_float *cfft65536 (complex_float dm input[],
                           complex_float dm output[]);

complex_float *cfft32768 (complex_float dm input[],
                           complex_float dm output[]);

complex_float *cfft16384 (complex_float dm input[],
                           complex_float dm output[]);

complex_float *cfft8192  (complex_float dm input[],
                           complex_float dm output[]);

complex_float *cfft4096  (complex_float dm input[],
                           complex_float dm output[]);

complex_float *cfft2048  (complex_float dm input[],
                           complex_float dm output[]);

complex_float *cfft1024  (complex_float dm input[],
                           complex_float dm output[]);

complex_float *cfft512   (complex_float dm input[],
                           complex_float dm output[]);

complex_float *cfft256   (complex_float dm input[],
                           complex_float dm output[]);

complex_float *cfft128   (complex_float dm input[],
                           complex_float dm output[]);

complex_float *cfft64    (complex_float dm input[],
                           complex_float dm output[]);
```

```
complex_float *cfft16    (complex_float dm input[],  
                           complex_float dm output[]);  
  
complex_float *cfft8     (complex_float dm input[],  
                           complex_float dm output[]);
```

### Description

The `cfftN` functions are optimized to take advantage of the SIMD execution model of the ADSP-2116x DSPs, and require complex arguments to ensure that the real and imaginary parts are interleaved in memory and are therefore accessible in a single cycle using the wider data bus of the processor.

Each of these 14 `cfftN` functions computes the N-point radix-2 fast Fourier transform (CFFT) of its complex input (where N is 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768 or 65536).

There are fourteen 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 they operate on. Call a particular function by substituting the number of points for N, as in

```
cfft8 (input, output);
```

The input to `cfftN` is a floating-point 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 `cfftN()` function returns a pointer to the `output` array.

### Error Conditions

The `cfftN` functions do not return any error conditions.

### Example

```
#include <filter.h>
#define N 2048
complex_float input[N], output[N];

/* Input array is filled from a converter or other source */

cfft2048 (input, output);
/* Array is filled with FFT data */
```

### See Also

[cfftN](#), [ifftN](#), [rfftN](#)



The `cfftN` functions use the input array as an intermediate workspace. If the input data is to be preserved it must first be copied to a safe location before calling these functions.



By default, this function uses SIMD; refer to [“Implications of Using SIMD Mode” on page 4-13](#) for more information.



## cfft

fast N-point complex input FFT

### Synopsis

```
#include <filter.h>
void cfft (float data_real[], float data_imag[],
          float temp_real[], float temp_imag[],
          const float twid_real[],
          const float twid_imag[],
          int n);
```

### Description

The `cfft` function transforms the time domain complex input signal sequence to the frequency domain by using the accelerated version of the Discrete Fourier Transform known as a Fast Fourier Transform or FFT. It will decimate in frequency using an optimized radix-2 algorithm.

The array `data_real` contains the real part of a complex input signal, and the array `data_imag` contains the imaginary part of the signal. On output, the function will overwrite the data in these arrays and will store the real part of the FFT in `data_real`, and the imaginary part of the FFT in `data_imag`. If the input data is to be preserved, it must first be copied to a safe location before calling this function. The argument `n` represents the number of points in the FFT; it must be a power of 2 and must be at least 64.

The `cfft` function has been designed for optimum performance and requires that the arrays `data_real` and `data_imag` are aligned on an address boundary that is a multiple of the FFT size. For certain applications, this alignment constraint may not be appropriate; in such cases, the application should call the `cfftN` function instead with no loss of facility (apart from performance).

## DSP Run-Time Library Reference

The arrays `temp_real` and `temp_imag` are used as intermediate temporary buffers and should each be of size `n`.

The twiddle table is passed in using the arrays `twid_real` and `twid_imag`. The array `twid_real` contains the +cosine factors, and the array `twid_imag` contains the -sine factors; each array should be of size `n/2`. The `twidfft` function may be used to initialize the twiddle table arrays.

It is recommended that the arrays containing real parts (`data_real`, `temp_real`, and `twid_real`) are allocated in separate memory blocks from the arrays containing imaginary parts (`data_imag`, `temp_imag`, and `twid_imag`); otherwise, the performance of the function will degrade.



The `cfft` function has been highly optimized and should therefore not be used with any application that relies on the `-reserve` switch (see [on page 1-42](#)) for correct operation.

### Error Conditions

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

### Example

```
#include <filter.h>

#define FFT_SIZE 1024

#pragma align 1024
float dm input_r[FFT_SIZE];
#pragma align 1024
float pm input_i[FFT_SIZE];

float dm temp_r[FFT_SIZE];
float pm temp_i[FFT_SIZE];
float dm twid_r[FFT_SIZE/2];
float pm twid_i[FFT_SIZE/2];

twidfft(twid_r, twid_i, FFT_SIZE);
```

```
cfft_f(input_r, input_i,  
        temp_r, temp_i,  
        twid_r, twid_i, FFT_SIZE);
```

### See Also

[cfftN](#), [ifftN](#), [rfftN](#)



The `cfft_f` function has been implemented to make highly efficient use of the ADSP-2116x processor's SIMD capabilities. The DSP library therefore does not contain a version of this function that does not use SIMD. Refer to [“Implications of Using SIMD Mode” on page 4-13](#) for more information.

## copysign, copysignf

copy the sign of the floating-point operand (IEEE arithmetic function)

### Synopsis

```
#include <math.h>
double copysign (double x, double y);
float copysignf (float x, float y);
```

### Description

The `copysign` and `copysignf` functions copy the sign of the second argument `y` to the first argument `x` without changing either its exponent or mantissa. The `copysignf` function is a built-in function which is implemented with an `Fn=Fx COPYSIGN Fy` instruction.

### Error Conditions

This function does not return an error code.

### Example

```
#include <math.h>
double x;
float y;

x = copysign (0.5, -10.0);      /* x = -0.5 */
y = copysignf (-10.0, 0.5f);    /* y = 10.0 */
```

### See Also

No references to this function.



The double precision function `copysign` is only available when using the `-double-size-32` switch (see [on page 1-27](#)) and actually calls the single precision function `copysignf`.

**cot, cotf**

cotangent

**Synopsis**

```
#include <math.h>
double cot (double x);
float cotf (float x);
```

**Description**

The `cot` and `cotf` functions return the cotangent of their argument. The input is interpreted as radians.

The `cot` and `cotf` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of  $2^{-20}$  over its input range.

**Error Conditions**

The `cot` and `cotf` functions do not return an error condition.

**Example**

```
#include <math.h>
double x, y;
float v, w;

y = cot (x);
v = cotf (w);
```

**See Also**

[tan, tanf](#)

## crosscoh

cross-coherence

### Synopsis

```
#include <stats.h>
float *crosscoh (float dm out[],
                const float dm x[],
                const float dm y[],
                int samples,
                int lags);
```

### Description

The `crosscoh` function computes the cross-coherence of two floating point inputs, `x[]` and `y[]`. The cross-coherence is the cross-correlation minus the product of the mean of `x` and the mean of `y`. The length of the input arrays is given by `samples`. This function returns a pointer to the output data array, `out[]`, of length `lags`.

### Error Conditions

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

### Example

```
#include <stats.h>
#define SAMPLES 1024

float excitation[SAMPLES], response[16], y[SAMPLES];
int lags = 16;

crosscoh (response, excitation, y, SAMPLES, lags);
```

### See Also

[autocoh](#), [autocorr](#), [crosscorr](#)



By default, this function uses SIMD; refer to [“Implications of Using SIMD Mode” on page 4-13](#) for more information.

## **crosscorr**

cross-correlation

### **Synopsis**

```
#include <stats.h>
float *crosscorr (float dm out[],
                  const float dm x[],
                  const float dm y[],
                  int samples,
                  int lags);
```

### **Description**

The `crosscorr` function performs a cross-correlation between two signals. The cross-correlation is the sum of the scalar products of the signals in which the signals are displaced in time with respect to one another. The signals to be correlated are given by input `x[]` and `y[]` arrays. The length of the input arrays is given by `samples`. This function returns a pointer to the output data array, `out[]`, of length `lags`.

The `crosscorr` function is used in digital signal processing applications such as speech analysis.

### **Error Conditions**

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

### **Example**

```
#include <stats.h>
float r[10], s[160];
float p[160];

crosscorr (r, s, p, 160, 10);
```



### See Also

[autocoh](#), [autocorr](#), [crosscoh](#)



By default, this function uses SIMD; refer to [“Implications of Using SIMD Mode” on page 4-13](#) for more information.

## cvecdot

complex vector dot product

### Synopsis

```
#include <cvector.h>
complex_float cvecdot (const complex_float dm x_input[],
                      const complex_float dm y_input[],
                      int samples);
```

### Description

The `cvecdot` function computes the complex dot product of the complex vectors, `x_input` and `y_input`, which are `samples` in size. The scalar result is returned by the function.

### Error Conditions

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

### Example

```
#include <cvector.h>
complex_float answer, x[100], y[100];

answer = cvecdot (x, y, 100);
```

### See Also

[vecdot](#)

## cvecsadd

complex vector scalar addition

### Synopsis

```
#include <cvector.h>
complex_float *cvecsadd (const complex_float dm input[],
                        complex_float scalar,
                        complex_float dm output[],
                        int samples);
```

### Description

The `cvecsadd` function computes the sum of each element of the complex vector, `input`, added to the complex scalar. Both the input and output vectors are `samples` in length. The function returns a pointer to the output vector.

### Error Conditions

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

### Example

```
#include <cvector.h>
complex_float answer[50], x[50], y;

cvecsadd (x, y, answer, 50);
```

### See Also

[vecvadd](#)



By default, this function uses SIMD; refer to [“Implications of Using SIMD Mode” on page 4-13](#) for more information.

### **cvecsmlt**

complex vector scalar multiply

#### **Synopsis**

```
#include <cvector.h>
complex_float *cvecsmlt (const complex_float dm input[],
                        complex_float scalar,
                        complex_float dm output[],
                        int samples);
```

#### **Description**

The `cvecsmlt` function computes the product of each element of the complex vector, `input`, multiplied by the complex scalar. Both the input and output vectors are `samples` in length. The function returns a pointer to the output vector.

#### **Error Conditions**

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

#### **Example**

```
#include <cvector.h>
complex_float answer[50], x[50], y;

cvecsmlt (x, y, answer, 50);
```

#### **See Also**

[vecvmlt](#)

## cvecssub

complex vector scalar subtraction

### Synopsis

```
#include <cvector.h>
complex_float *cvecssub (const complex_float dm input[],
                        complex_float scalar,
                        complex_float dm output[],
                        int samples);
```

### Description

The `cvecssub` function computes the difference of each element of the complex vector, `input`, minus the complex scalar. Both the input and output vector are `samples` in length. The function returns a pointer to the output vector.

### Error Conditions

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

### Example

```
#include <cvector.h>
complex_float answer[50], x[50], y

cvecssub (x, y, answer, 50);
```

### See Also

[vecvsub](#)



By default, this function uses SIMD; refer to [“Implications of Using SIMD Mode” on page 4-13](#) for more information.

## cvecvadd

complex vector addition

### Synopsis

```
#include <cvector.h>
complex_float *cvecvadd (const complex_float dm x_input[],
                        const complex_float dm y_input[],
                        complex_float dm output[],
                        int samples);
```

### Description

The `cvecvadd` function computes the sum of each of the elements of the complex vectors, `x_input` and `y_input`, and places the results in `output`. All three vectors are `samples` in length. The function returns a pointer to the output vector.

### Error Conditions

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

### Example

```
#include <cvector.h>
complex_float answer[50], x[50], y[50];

cvecvadd (x, y, answer, 50);
```

### See Also

[vecvadd](#)



By default, this function uses SIMD; refer to [“Implications of Using SIMD Mode” on page 4-13](#) for more information.

## cvecvmlt

complex vector multiply

### Synopsis

```
#include <cvector.h>
complex_float *cvecvmlt (const complex_float dm x_input[],
                          const complex_float dm y_input[],
                          complex_float dm output[],
                          int samples);
```

### Description

The `cvecvmlt` function computes the product of each of the elements of the complex vectors, `x_input` and `y_input`, and places the results in `output`. All three vectors are `samples` in length. The function returns a pointer to the output vector.

### Error Conditions

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

### Example

```
#include <cvector.h>
complex_float answer[50], x[50], y[50];

cvecvmlt (x, y, answer, 50);
```

### See Also

[vecvmlt](#)



By default, this function uses SIMD; refer to [“Implications of Using SIMD Mode” on page 4-13](#) for more information.

## cvecvsub

complex vector subtraction

### Synopsis

```
#include <cvector.h>
complex_float *cvecvsub (const complex_float dm x_input[],
                        const complex_float dm y_input[],
                        complex_float dm output[],
                        int samples);
```

### Description

The `cvecvsub` function computes the difference of each of the elements of the complex vectors, `x_input` and `y_input`, and places the results in `output`. All three vectors are `samples` in length. The function returns a pointer to the output vector.

### Error Conditions

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

### Example

```
#include <cvector.h>
complex_float answer[50], x[50], y[50];

cvecvsub (x, y, answer, 50);
```

### See Also

[vecvsub](#)



By default, this function uses SIMD; refer to [“Implications of Using SIMD Mode” on page 4-13](#) for more information.



**favg, favgf**

return mean of two values

**Synopsis**

```
#include <math.h>
double favg (double x, double y);
float favgf (float x, float y);
```

**Description**

The `favg` and `favgf` functions return the mean of its two arguments. The `favgf` function is a built-in function which is implemented with an `Fn=(Fx+Fy)/2` instruction.

**Error Conditions**

The `favg` and `favgf` functions do not return an error code.

**Example**

```
#include <math.h>
float x;
x = favgf (10.0f, 8.0f);    /* returns 9.0f */
```

**See Also**

[avg](#), [lavg](#)



The double-precision function `favg` is only available when using the `-double-size-32` switch (see [on page 1-27](#)) and actually calls the single-precision function `favgf`.

### **fclip, fclipf**

clip  $x$  by  $y$

#### **Synopsis**

```
#include <math.h>
double fclip (double x, double y);
float fclipf (float x, float y);
```

#### **Description**

The `fclip` and `fclipf` functions return the first argument if it is less than the absolute value of the second argument, otherwise it returns the absolute value of the second argument if the first is positive, or minus the absolute value if the first argument is negative. The `fclipf` function is a built-in function which is implemented with an `Fn=CLIP Fx BY Fy` instruction.

#### **Error Conditions**

The `fclip` and `fclipf` functions do not return an error code.

#### **Example**

```
#include <math.h>
float y;
y = fclipf (5.1f, 8.0f); /* returns 5.1f */
```

#### **See Also**

[clip](#), [lclip](#)



The double precision function `fclip` is only available when using the `-double-size-32` switch (see [on page 1-27](#)) and actually calls the single precision function `fclipf`.

## fft\_mag

compute FFT magnitude

### Synopsis

```
#include <filter.h>
float *fft_mag (const complex_float dm input[],
               float dm output[],
               int samples);
```

### Description

The `fft_mag` function computes the normalized power spectrum from the complex results of an FFT operation. The complex input array is `samples` in size and the result is `samples/2` in size. The result evaluates as

$$\text{magnitude}(z) = \frac{((\text{Re}(z))^2 + (\text{Im}(z))^2)}{\left(\frac{(\text{samples})}{2}\right)^2}$$

The `fft_mag` function returns a pointer to the output array.

### Error Conditions

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

### Example

```
#include <filter.h>
complex_float fft_data[1024];
float spectra[512];

fft_mag (fft_data, spectra, 1024);
```

### See Also

[cfftN](#), [rfftN](#)

### **fir**

finite impulse response (FIR) filter

### **Synopsis**

```
#include <filter.h>
float *fir (const float dm input[],
           float dm output[],
           const float pm coeffs[],
           float dm state[],
           int samples,
           int taps);
```

### **Description**

The `fir` function is optimized to take advantage of the SIMD execution model of the ADSP-2116x DSPs.

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 and stores the result in the vector `output`. 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 floating-point input array to the filter is `samples` in length. 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, and so `coeffs[0]` contains the last coefficient and `coeffs[taps-1]` contains the first coefficient. The array must be located in program memory data space so that the single-cycle dual-memory fetch of the processor can be used.

The `state` array contains a pointer to the delay line as its first element, followed by the delay line values. The length of the `state` array is therefore 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 function returns a pointer to the output vector.

## Error Conditions

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

## Example

```
#include <filter.h>

#define TAPS 10

float x[100], y[100];
float pm coeffs[TAPS];      /* coeffs array must be      */
                             /* initialized and in PM memory */

float state[TAPS+1];
int i;

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

fir (x, y, coeffs, state, 100, TAPS);
                             /* y holds the filtered output */
```

## See Also

[biquad](#), [iir](#)



By default, this function uses SIMD; refer to [“Implications of Using SIMD Mode” on page 4-13](#) for more information.

## fmax, fmaxf

return larger of two values

### Synopsis

```
#include <math.h>
double fmax (double x, double y);
float fmaxf (float x, float y);
```

### Description

The `fmax` and `fmaxf` functions return the larger of its two arguments. The `fmaxf` function is a built-in function which is implemented with an `Fn=MAX(Fx,Fy)` instruction.

### Error Conditions

The `fmax` and `fmaxf` functions do not return an error code.

### Example

```
#include <math.h>
float y;

y = fmaxf (5.1f, 8.0f);    /* returns 8.0f */
```

### See Also

[fmin](#), [fminf](#), [lmax](#), [lmin](#), [max](#), [min](#)



The double-precision function `fmax` is only available when using the `-double-size-32` switch (see [on page 1-27](#)) and actually calls the single-precision function `fmaxf`.

## fmin, fminf

return smaller of two values

### Synopsis

```
#include <math.h>
double fmin (double x, double y);
float fminf (float x, float y);
```

### Description

The `fmin` and `fminf` functions return the smaller of their two arguments. The `fminf` function is a built-in function which is implemented with an `Fn=MIN(Fx,Fy)` instruction.

### Error Conditions

The `fmin` and `fminf` functions do not return an error code.

### Example

```
#include <math.h>
float y;

y = fminf (5.1f, 8.0f);  /* returns 5.1f */
```

### See Also

[fmax](#), [fmaxf](#), [lmax](#), [lmin](#), [max](#), [min](#)



The double-precision function `fmin` is only available when using the `-double-size-32` switch (see [on page 1-27](#)) and actually calls the single-precision function `fminf`.

### gen\_bartlett

generate bartlett window

#### Synopsis

```
#include <window.h>
void gen_bartlett(
    float dm w[], /* Window vector */
    int a, /* Address stride in samples for window vector */
    int N /* Length of window vector */ );
```

#### Description

The `gen_bartlett` function generates a vector containing the Bartlett window. The length is specified by parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be  $N \cdot a$ .

The Bartlett window is similar to the Triangle window (see [“gen\\_triangle” on page 4-62](#)) but has the following different properties

- The Bartlett window always returns a window with two zeros on either end of the sequence. Therefore, for odd  $n$ , the center section of a  $N+2$  Bartlett window equals a  $N$  Triangle window.
- For even  $n$ , the Bartlett window is the convolution of two rectangular sequences. There is no standard definition for the Triangle window for even  $n$ ; the slopes of the Triangle window are slightly steeper than those of the Bartlett window.

#### Algorithm

$$w[n] = 1 - \left| \frac{n - \frac{N-1}{2}}{\frac{N-1}{2}} \right|$$

where  $n = \{0, 1, 2, \dots, N-1\}$



### Domain

$a > 0$ ;  $N > 0$

### Error Conditions

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

### See Also

[gen\\_blackman](#), [gen\\_gaussian](#), [gen\\_hamming](#), [gen\\_hanning](#), [gen\\_harris](#),  
[gen\\_kaiser](#), [gen\\_rectangular](#), [gen\\_triangle](#)

## gen\_blackman

generate blackman window

### Synopsis

```
#include <window.h>
void gen_blackman(
    float dm w[], /* Window vector */
    int a, /* Address stride in samples for window vector */
    int N /* Length of window vector */ );
```

### Description

The `gen_blackman` function generates a vector containing the Blackman window. The length of the window required is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`.

### Algorithm

$$w[n] = 0.42 - 0.5 \cos\left(\frac{2\pi n}{N-1}\right) + 0.08 \cos\left(\frac{4\pi n}{N-1}\right)$$

where  $n = \{0, 1, 2, \dots, N-1\}$

### Domain

$a > 0$ ;  $N > 0$

### Error Conditions

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

### See Also

[gen\\_bartlett](#), [gen\\_gaussian](#), [gen\\_hamming](#), [gen\\_hanning](#), [gen\\_harris](#),  
[gen\\_kaiser](#), [gen\\_rectangular](#), [gen\\_triangle](#)

## gen\_gaussian

generate gaussian window

### Synopsis

```
#include <window.h>
void gen_gaussian(
    float dm w[], /* Window vector */
    float alpha, /* Gaussian alpha parameter */
    int a, /* Address stride in samples for window vector */
    int N /* Length of window vector */ );
```

### Description

The `gen_gaussian` function generates a vector containing the Gaussian window. The length of the window required is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`.

### Algorithm

$$w(n) = \exp \left[ -\frac{1}{2} \left( \alpha \frac{n - N/2 - 1/2}{N/2} \right)^2 \right]$$

where  $n = \{0, 1, 2, \dots, N-1\}$  and  $\alpha$  is an input parameter

### Domain

$a > 0$ ;  $N > 0$ ;  $\alpha > 0.0$

### Error Conditions

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

### See Also

[gen\\_bartlett](#), [gen\\_blackman](#), [gen\\_hamming](#), [gen\\_hanning](#), [gen\\_harris](#),  
[gen\\_kaiser](#), [gen\\_rectangular](#), [gen\\_triangle](#)

## gen\_hamming

generate hamming window

### Synopsis

```
#include <window.h>
void gen_hamming(
    float dm w[], /* Window vector */
    int a,         /* Address stride in samples for window vector */
    int N          /* Length of window vector */);
```

### Description

The `gen_hamming` function generates a vector containing the Hamming window. The length of the window required is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`.

### Algorithm

$$w[n] = 0.54 - 0.46 \cos\left(\frac{2\pi n}{N-1}\right)$$

where  $n = \{0, 1, 2, \dots, N-1\}$

### Domain

$a > 0$ ;  $N > 0$

### Error Conditions

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

### See Also

[gen\\_bartlett](#), [gen\\_blackman](#), [gen\\_gaussian](#), [gen\\_hanning](#), [gen\\_harris](#),  
[gen\\_kaiser](#), [gen\\_rectangular](#), [gen\\_triangle](#)

## gen\_hanning

generate hanning window

### Synopsis

```

#include <window.h>
void gen_hanning(
    float dm w[], /* Window vector */
    int a,        /* Address stride in samples for window vector */
    int N         /* Length of window vector */ );

```

### Description

The `gen_hanning` function generates a vector containing the Hanning window. The length of the window required is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`. This window is also known as the Cosine window.

### Algorithm

$$w[n] = 0.5 - 0.5 \cos\left(\frac{2\pi n}{N-1}\right)$$

where  $n = \{0, 1, 2, \dots, N-1\}$

### Domain

$a > 0$ ;  $N > 0$

### Error Conditions

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

### See Also

[gen\\_bartlett](#), [gen\\_blackman](#), [gen\\_gaussian](#), [gen\\_hamming](#), [gen\\_harris](#),  
[gen\\_kaiser](#), [gen\\_rectangular](#), [gen\\_triangle](#)

## gen\_harris

generate harris window

### Synopsis

```
#include <window.h>
void gen_harris(
    float dm w[], /* Window vector */
    int a, /* Address stride in samples for window vector */
    int N /* Length of window vector */ );
```

### Description

The `gen_harris` function generates a vector containing the Harris window. The length of the window required is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`. This window is also known as the Blackman-Harris window.

### Algorithm

$$w[n] = 0.35875 - 0.48829 * \cos\left(\frac{2\pi n}{N-1}\right) + 0.14128 * \cos\left(\frac{4\pi n}{N-1}\right) - 0.01168 * \cos\left(\frac{6\pi n}{N-1}\right)$$

where  $n = \{0, 1, 2, \dots, N-1\}$

### Domain

$a > 0$ ;  $N > 0$

### Error Conditions

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

### See Also

[gen\\_bartlett](#), [gen\\_blackman](#), [gen\\_gaussian](#), [gen\\_hamming](#), [gen\\_hanning](#),  
[gen\\_kaiser](#), [gen\\_rectangular](#), [gen\\_triangle](#)

## gen\_kaiser

generate kaiser window

### Synopsis

```
#include <window.h>
void gen_kaiser(
    float dm w[], /* Window vector */
    float beta, /* Kaiser beta parameter */
    int a, /* Address stride in samples for window vector */
    int N /* Length of window vector */ );
```

### Description

The `gen_kaiser` function generates a vector containing the Kaiser window. The length of the window required is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`. The  $\beta$  value is specified by parameter `beta`.

### Algorithm

$$w[n] = \frac{I_0 \left[ \beta \left( 1 - \left[ \frac{n - \alpha}{\alpha} \right]^2 \right)^{1/2} \right]}{I_0(\beta)}$$

where  $n = \{0, 1, 2, \dots, N-1\}$ ,  $\alpha = (N - 1) / 2$ , and  $I_0(\beta)$  represents the zeroth-order modified Bessel function of the first kind.

### Domain

$a > 0$ ;  $N > 0$ ;  $\beta > 0.0$

### Error Conditions

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

## DSP Run-Time Library Reference

### See Also

[gen\\_bartlett](#), [gen\\_blackman](#), [gen\\_gaussian](#), [gen\\_hamming](#), [gen\\_hanning](#),  
[gen\\_harris](#), [gen\\_rectangular](#), [gen\\_triangle](#)



## gen\_rectangular

generate rectangular window

### Synopsis

```

#include <window.h>
void gen_rectangular(
    float dm w[], /* Window vector */
    int a,         /* Address stride in samples for window vector */
    int N          /* Length of window vector */ );

```

### Description

The `gen_rectangular` function generates a vector containing the Rectangular window. The length of the window required is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`.

### Algorithm

$$w[n] = 1$$

where  $n = \{0, 1, 2, \dots, N-1\}$

### Domain

$a > 0$ ;  $N > 0$

### Error Conditions

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

### See Also

[gen\\_bartlett](#), [gen\\_blackman](#), [gen\\_gaussian](#), [gen\\_hamming](#), [gen\\_hanning](#),  
[gen\\_harris](#), [gen\\_kaiser](#), [gen\\_triangle](#)

### gen\_triangle

generate triangle window

#### Synopsis

```
#include <window.h>
void gen_triangle(
float dm w[], /* Window vector */
int a, /* Address stride in samples for window vector */
int N /* Length of window vector */ );
```

#### Description

The `gen_triangle` function generates a vector containing the Triangle window. The length of the window required is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`.

Refer to the Bartlett window (described [on page 4-52](#)) regarding the relationship between it and the Triangle window.

#### Algorithm

For even  $n$ , the following equation applies:

$$w[n] = \begin{cases} \frac{2n+1}{N} & n < N/2 \\ \frac{2N-2n-1}{N} & n > N/2 \end{cases}$$

where  $n = \{0, 1, 2, \dots, N-1\}$

For odd  $n$ , the following equation applies:

$$w[n] = \begin{cases} \frac{2n+2}{N+1} & n < N/2 \\ \frac{2N-2n}{N+1} & n > N/2 \end{cases}$$

where  $n = \{0, 1, 2, \dots, N-1\}$

### Domain

$a > 0; N > 0$

### Error Conditions

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

### See Also

[gen\\_bartlett](#), [gen\\_blackman](#), [gen\\_gaussian](#), [gen\\_hamming](#), [gen\\_hanning](#),  
[gen\\_harris](#), [gen\\_kaiser](#), [gen\\_rectangular](#)

## histogram

histogram

### Synopsis

```
#include <stats.h>
int *histogram (int out[],
                const int in[],
                int out_len,
                int samples,
                int bin_size);
```

### Description

The `histogram` function computes a scaled-integer histogram of its input array. The `bin_size` parameter is used to adjust the width of each individual bin in the output array. For example, a `bin_size` of 5 indicates that the first location of the output array holds the number of occurrences of a 0, 1, 2, 3 or 4.

The output array is first zeroed by the function, then each sample in the input array is multiplied by  $1/\text{bin\_size}$  and truncated. The appropriate bin in the output array is incremented. This function returns a pointer to the output array.

All values within the input array must be within range. In order to achieve maximum performance, out of bounds checking is not performed by this function.

### Error Conditions

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

### Example

```
#include <stats.h>
#define SAMPLES 1024
```

```
int length = 2048;  
int excitation[SAMPLES], response[2048];  
histogram (response, excitation, length, SAMPLES, 5);
```

### See Also

[mean](#), [var](#)

## idle

execute ADSP-21xxx DSP `idle` instruction

### Synopsis

```
#include <21160.h>
void idle (void);
```

### Description

The `idle` function invokes the processor's `idle` instruction once and returns. The `idle` instruction causes the processor to stop and respond only to interrupts. For a complete description of the `idle` instruction, please refer to the *ADSP-21160 SHARC DSP Hardware Reference*.



In previous releases of the VisualDSP++ software (prior to release 2.1), the `idle` function repeatedly executed the `idle` instruction. This function has been changed to give you more control over the amount of time spent in the `idle` state.

### Error Conditions

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

### Example

```
#include <21160.h>
idle ();
```

### See Also

[interrupt](#), [signal](#)

## ifftN

N-point inverse complex input fast Fourier transform (IFFT)

### Synopsis

```
#include <filter.h>
complex_float *ifft65536 (complex_float dm input[],
                           complex_float dm output[]);

complex_float *ifft32768 (complex_float dm input[],
                           complex_float dm output[]);

complex_float *ifft16384 (complex_float dm input[],
                           complex_float dm output[]);

complex_float *ifft8192  (complex_float dm input[],
                           complex_float dm output[]);

complex_float *ifft4096  (complex_float dm input[],
                           complex_float dm output[]);

complex_float *ifft2048  (complex_float dm input[],
                           complex_float dm output[]);

complex_float *ifft1024  (complex_float dm input[],
                           complex_float dm output[]);

complex_float *ifft512   (complex_float dm input[],
                           complex_float dm output[]);

complex_float *ifft256   (complex_float dm input[],
                           complex_float dm output[]);

complex_float *ifft128   (complex_float input[],
                           complex_float dm output[]);

complex_float *ifft64    (complex_float dm input[],
                           complex_float dm output[]);

complex_float *ifft32    (complex_float dm input[],
                           complex_float dm output[]);
```

## DSP Run-Time Library Reference

```
complex_float *ifft16    (complex_float dm input[],  
                           complex_float dm output[]);  
  
complex_float *ifft8     (complex_float dm input[],  
                           complex_float dm output[]);
```

### Description

The `ifftN` functions are optimized to take advantage of the SIMD execution model of the ADSP-2116x DSPs. They require complex arguments to ensure that the real and imaginary parts are interleaved in memory and are therefore accessible in a single cycle using the wider data bus of the processor.

Each of these fourteen `ifftN` functions computes the N-point radix-2 inverse fast Fourier transform (IFFT) of its floating-point input (where N is 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768 or 65536).

There are 14 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 (input, output);
```

The input to `ifftN` is a floating-point array of N points. If there are fewer than N actual data points, you must pad the array with zeros to make N samples. However, better results occur with less zero padding. The input data should be windowed (if necessary) before calling the function because no preprocessing is performed on the data. The functions return a pointer to the `output` array.

### Error Conditions

The `ifftN` functions do not return error conditions.



### Example

```
#include <filter.h>
#define N 2048

complex_float input[N], output[N];

/* Input array is filled from a previous xfft2048 () or
   other source */

ifft2048 (input, output);
/* Arrays are filled with FFT data */
```

### See Also

[cfftN](#), [rfftN](#)



The `ifftN` functions use the input array as an intermediate workspace. If the input data is to be preserved it must first be copied to a safe location before calling these functions.



By default, this function uses SIMD; refer to [“Implications of Using SIMD Mode” on page 4-13](#) for more information.

## iir

infinite impulse response (IIR) filter

### Synopsis

```
#include <filter.h>
float *iir (const float dm input[],
           float dm output[],
           const float pm coeffs[],
           float dm state[],
           int samples,
           int sections);
```

### Description

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 and stores the result in the output vector. The IIR filter is implemented as a cascaded biquad. The characteristics of the filter are dependent on the coefficient values supplied by the calling program.

The floating-point inputs to the filter are contained in the array `input`, which has a size of `samples`.

The parameter `sections` specifies the number of biquad sections.

The `coeffs` array must be 4 times the number of sections in length and it also must be located in program memory. The definition is:

```
float pm coeffs[4*sections];
```

The `coeffs` array must be ordered in the following form:

```
coeffs[] =
    [a2 stage 1, a1 stage 1, b2 stage 1, b1 stage 1, a2 stage 2, ... ]
```

The function assumes that the value of `B0` is 1.0 and so the `B1` and `B2` coefficients should be scaled accordingly. As a consequence of this, all the output generated by the `iir` function has to be scaled by  $1/B0$  to obtain the correct signal amplitude. The function also assumes that the value of the `A0` coefficient is 1.0, and the `A1` and `A2` coefficients should be normalized.

 When importing coefficients from a filter design tool that employs a transposed direct form II, the `A` coefficients have to be negated. For example, if a filter design tool returns `A = [1.0, 0.2, -0.9]`, then the `A` coefficients have to be modified to `A = [1.0, -0.2, 0.9]`.

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:

```
float state[2*sections + 1];
```

Each filter should have its own `state` array which should be cleared to zero before calling the `iir` function for the first time; the array should not otherwise be modified by the user program.

The algorithm used is adapted from *Digital Signal Processing*, Oppenheim and Schaffer, New Jersey, Prentice Hall, 1975.

The function returns a pointer to the `output` vector.

### Error Conditions

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

### Example

```
#include <filter.h>

#define SECTIONS 4
```

## DSP Run-Time Library Reference

```
float input[100], output[100], state[2*SECTIONS + 1];
float pm coeffs[SECTIONS * 4];
int    i;

for (i = 0; i < SECTIONS + 1; i++)
    state[i] = 0;    /* initialize state array */
iir (input, output, coeffs, state, 100, SECTIONS);
```

### See Also

[biquad](#), [fir](#)

## matadd

matrix addition

### Synopsis

```
#include <matrix.h>
float *matadd (float output[][[]],
               const float x_input[][[]],
               const float y_input[][[]],
               int r,
               int s);
```

### Description

The `matadd` function performs a matrix addition of the input matrices `x_input[][[]]` and `y_input[][[]]`, returning the result in `output[][[]]`. The `matadd` function returns a pointer to the output matrix. The dimensions of these matrices are `x_input[r][s]`, `y_input[r][s]`, and `output[r][s]`.

### Error Conditions

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

### Example

```
#include <matrix.h>
float x[10][20], y[10][20];
float z[10][20];

matadd (z, x, y, 10, 20);
```

### See Also

[matsub](#)



By default, this function uses SIMD; refer to [“Implications of Using SIMD Mode” on page 4-13](#) for more information.

## matinv

real matrix inversion

### Synopsis

```
#include <matrix.h>
float *matinv (float dm *output,
               const float dm *input,
               int n);
```

### Description

The `matinv` function employs Gauss-Jordan elimination with full pivoting to compute the inverse of the input matrix `input` and store the result in the matrix `output`. The dimensions of the matrices `input` and `output` are `[n][n]`.

The function returns a pointer to the output matrix.

### Error Conditions

If no inverse exists for the input matrix, the function will return a null pointer.

### Example

```
#include <matrix.h>

#define N 8
float a[N][N];
float a_inv[N][N];

matinv((float *) (a_inv), (float *) (a), N);
```

### See Also

No references available.

## matmul

matrix multiplication

### Synopsis

```
#include <matrix.h>
float *matmul (float output[][[]],
               const float x_input[][[]],
               const float y_input[][[]],
               int r,
               int s,
               int t);
```

### Description

The `matmul` function performs a matrix multiplication of the input matrices `x_input[][[]]` and `y_input[][[]]`, returning a pointer to the `output[][[]]` matrix. The dimensions of these matrices are `x_input[r][s]`, `y_input[s][t]` and `output[r][t]`.

### Error Conditions

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

### Example

```
#include <matrix.h>
float x[10][5], y[5][10];
float z[10][10];

matmul (z, x, y, 10, 5, 10);
```

### See Also

[matscalmult](#)

## matscalmult

multiply matrix by scalar

### Synopsis

```
#include <matrix.h>
float *matscalmult (float output[][[]],
                   const float input[][[]],
                   float scalar,
                   int r,
                   int s);
```

### Description

The `matscalmult` function performs a scaled multiplication of the `input[][[]]` matrix, returning the result in `output[][[]]`. The `input[][[]]` matrix is multiplied by the input value `scalar`. The `matscalmult` function returns a pointer to the output matrix. The dimensions of these matrices are `input[r][s]`, and `output[r][s]`.

### Error Conditions

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

### Example

```
#include <matrix.h>
float x[10][5], z[10][5];

matscalmult (z, x, 0.5, 10, 5);
/* multiplies the matrix x by 0.5 */
```

### See Also

[matmul](#)



By default, this function uses SIMD; refer to [“Implications of Using SIMD Mode” on page 4-13](#) for more information.



## matsub

matrix subtraction

### Synopsis

```
#include <matrix.h>
float *matsub (float output[][[]],
               const float x_input[][[]],
               const float y_input[][[]],
               int r,
               int s);
```

### Description

The `matsub` function subtracts the elements of the input matrix `y_input[][[]]` from the input matrix `x_input[][[]]`, returning the result in `output[][[]]`. The `matsub` function returns a pointer to the output matrix. The dimensions of these matrices are `x_input[r][s]`, `y_input[r][s]`, and `output[r][s]`.

### Error Conditions

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

### Example

```
#include <matrix.h>
float x[10][5], y[10][5];
float z[10][5];

matsub (z, x, y, 10, 5);
```

### See Also

[matadd](#)



By default, this function uses SIMD; refer to [“Implications of Using SIMD Mode” on page 4-13](#) for more information.

## mean

mean of an array of floating-point numbers

### Synopsis

```
#include <stats.h>
float mean (const float input[],
            int samples);
```

### Description

The `mean` function returns the mean of its floating-point input array.

### Error Conditions

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

### Example

```
#include <stats.h>
float result, input[256];

result = mean (input, 256);
```

### See Also

[var](#)



By default, this function uses SIMD; refer to [“Implications of Using SIMD Mode” on page 4-13](#) for more information.

## mu\_compress

μ-law compression

### Synopsis

```
#include <filter.h>
int *mu_compress (const int dm input[],
                  int dm output[],
                  int samples);
```

### Description

The `mu_compress` function takes an array of linear 14-bit signed speech samples and compresses them according to ITU recommendation G.711. The output array returned contains 8-bit samples that can be sent directly to a μ-law codec.

This function uses serial port 0 to perform companding. Therefore, serial port 0 must not be in use when this routine is called.

The function returns a pointer to the compressed data.

### Error Conditions

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

### Example

```
#include <filter.h>
int linear[100], compressed[100];

mu_compress (linear, compressed, 100);
```

### See Also

[a\\_compress](#), [mu\\_expand](#)

## mu\_expand

μ-law expansion

### Synopsis

```
#include <filter.h>
int *mu_expand (const int dm input[],
               int dm output[],
               int samples);
```

### Description

The `mu_expand` function takes an array of 8-bit compressed speech samples and expands them according to ITU recommendation G.711 (μ-law definition). The output returned is an array of linear 14-bit signed samples. The function returns a pointer to the output array.

This function uses serial port 0 to perform companding. Therefore, serial port 0 must not be in use when this routine is called.

The function returns a pointer to the expanded data.

### Error Conditions

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

### Example

```
#include <filter.h>
int compressed_data[100], expanded_data[100];

mu_expand (compressed_data, expanded_data, 100);
```

### See Also

[a\\_expand](#), [mu\\_compress](#)

## poll\_flag\_in

test input flag

### Synopsis

```
#include <21160.h>
int poll_flag_in (int flag, int mode);
```

### Description

The `poll_flag_in` function tests the specified flag (0, 1, 2, 3) for the specified transition (0=low to high, 1=high to low, 2=flag high, 3=flag low, 4=any transition, 5=read flag). The function returns a zero *after* the specified transition has occurred in modes 0-3. In Mode 4, it returns the state of the flag after the transition. In Mode 5, it returns the value of the flag without waiting.

Table 4-5. `poll_flag_in` Macros and Values

| Flag Macro | Value | Mode Macro         | Value |
|------------|-------|--------------------|-------|
| READ_FLAG0 | 0     | FLAG_IN_LO_TO_HI   | 0     |
| READ_FLAG1 | 1     | FLAG_IN_HI_TO_LOW  | 1     |
| READ_FLAG2 | 2     | FLAG_IN_HI         | 2     |
| READ_FLAG3 | 3     | FLAG_IN_LOW        | 3     |
| READ_FLAG3 | 3     | FLAG_IN_TRANSITION | 4     |
| READ_FLAG3 | 3     | RETURN_FLAG_STATE  | 5     |

This function assumes that the flag direction in the `MODE2` register is already set as an input (the default state at reset).

# DSP Run-Time Library Reference

## Error Conditions

The `poll_flag_in` function returns a negative value for an invalid flag or transition mode.

## Example

```
#include <21160.h>
poll_flag_in (0, 3);
    /* return zero after transition has occurred */
```

## See Also

[interrupt](#), [set\\_flag](#)

## rfftN

N-point real input fast Fourier transform

### Synopsis

```
#include <filter.h>
complex_float *rfft65536 (float dm input[],
                           complex_float dm output[]);

complex_float *rfft32768 (float dm input[],
                           complex_float dm output[]);

complex_float *rfft16384 (float dm input[],
                           complex_float dm output[]);

complex_float *rfft8192  (float dm input[],
                           complex_float dm output[]);

complex_float *rfft4096  (float dm input[],
                           complex_float dm output[]);

complex_float *rfft2048  (float dm input[],
                           complex_float dm output[]);

complex_float *rfft1024  (float dm input[],
                           complex_float dm output[]);

complex_float *rfft256   (float dm input[],
                           complex_float dm output[]);

complex_float *rfft128   (float dm input[],
                           complex_float dm output[]);

complex_float *rfft64    (float dm input[],
                           complex_float dm output[]);

complex_float *rfft32    (float dm input[],
                           complex_float dm output[]);

complex_float *rfft16    (float dm input[],
                           complex_float dm output[]);
```

### Description

The `rfftN` functions are optimized to take advantage of the SIMD execution model of the ADSP-2116x DSPs, and require complex output results to ensure that the real and imaginary parts are interleaved in memory and are therefore accessible in a single cycle using the wider data bus of the processor.

Each of these `rfftN` functions are similar to the `cfftN` functions except that they only take real inputs. They compute the N-point radix-2 fast Fourier transform (RFFT) of their floating-point input (where N is 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768 or 65536).

There are thirteen 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.

Call a particular function by substituting the number of points for N, as in the following example.

```
rfft16 (input, output);
```

The input to `rfftN` is a floating-point array of N points. If there are fewer than N actual data points, you must pad the array with zeros to make N samples. However, better results occur with less zero padding. The input data should be windowed (if necessary) before calling the function because no preprocessing is performed on the data.

The complex frequency domain signal generated by the `rfftN` functions is stored in the array output. Because the output signal is symmetric around the midpoint of the frequency domain, the functions only generate  $N/2$  output points.

The following example illustrates how the complete frequency domain signal may be generated:

```
for (i = 0; i < N/2; i++)  
    output[(N-1) - i] = output[i];
```



The `rfftN` functions return a pointer to the output array.

## Error Conditions

The `rfftN` functions do not return any error conditions.

## Example

```
#include <filter.h>
#define N 2048

float input[N];
complex_float output[N];

/* Real input array fills from a converter or other source */

rfft2048 (input, output);
/* Arrays are filled with FFT data */
```

## See Also

[cfftN](#), [cfftF](#), [rfftN](#)



The `rfftN` functions use the input array as an intermediate workspace. If the input data is to be preserved it must first be copied to a safe location before calling these functions.



By default, this function uses SIMD; refer to [“Implications of Using SIMD Mode” on page 4-13](#) for more information.

## rms

root mean square

### Synopsis

```
#include <stats.h>
float rms (const float input[], int length);
```

### Description

The `rms` function returns the square root of the mean of the square of its floating-point input array.

### Error Conditions

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

### Example

```
#include <stats.h>
float input[256], results;

results = rms (input, 256);
```

### See Also

[mean](#), [var](#)



By default, this function uses SIMD; refer to [“Implications of Using SIMD Mode” on page 4-13](#) for more information.

## rsqrt, rsqrtf

reciprocal square root

### Synopsis

```
#include <math.h>
double rsqrt (double x);
float rsqrtf (float x);
```

### Description

The `rsqrt` and `rsqrtf` functions return the reciprocal positive square root of their argument.

The `rsqrt` and `rsqrtf` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of  $2^{-20}$  over its input range.

### Error Conditions

The `rsqrt` and `rsqrtf` functions return zero for a negative input.

### Example

```
#include <math.h>
double y;

y = rsqrt (2.0);    /* y = 0.707 */
```

### See Also

[sqrt, sqrtf](#)

## set\_flag

set ADSP-21xxx DSP flags

### Synopsis

```
#include <21160.h>
int set_flag (int flag, int mode);
```

### Description

The `set_flag` function is used to set the ADSP-21xxx DSP flags to the desired output value.

The function accepts as input a flag number [0-3] and a mode. The mode can be specified as a macro (defined in `21160.h`) or a value [0-3].

Table 4-6. Flag Function Macros and Values

| Flag Macro | Value | Mode Macro | Value |
|------------|-------|------------|-------|
| SET_FLAG0  | 0     | SET_FLAG   | 0     |
| SET_FLAG1  | 1     | CLR_FLAG   | 1     |
| SET_FLAG2  | 2     | TGL_FLAG   | 2     |
| SET_FLAG3  | 3     | TST_FLAG   | 3     |

In addition to setting the flag to the specified value, the function also sets the `MODE2` register to specify that the flag is used for output, not input.

If the `TST_FLAG` macro (or a 3) is specified as the mode, the current value (0 or 1) of the flag is returned as the result of the function.

The `set_flag` function returns a zero upon success (except as noted in the previous paragraph).

### Error Conditions

The `set_flag` function returns a non-zero for an error.

### Example

```
#include <21160.h>

set_flag (SET_FLAG0, CLR_FLAG);
set_flag (SET_FLAG0, SET_FLAG);
```

### See Also

[poll\\_flag\\_in](#)

### set\_semaphore

set bus lock semaphore

#### Synopsis

```
#include <21160.h>
int set_semaphore (
    void dm *semaphore, int set_value, int timeout);
```

#### Description

The `set_semaphore` function is used to control bus lock in multiprocessor ADSP-21xxx systems.

- -1 is returned if the bus is locked and the bus lock timeout is exceeded.
- 0 (zero) is returned if the bus is not locked and a semaphore is set.

#### Error Conditions

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

#### See Also

No references to this function.

## timer\_off

disable ADSP-21xxx DSP timer

### Synopsis

```
#include <21160.h>
unsigned int timer_off (void);
```

### Description

The `timer_off` function disables the ADSP-21xxx 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 <21160.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](#)



The function is supplied only as an in-lined procedure; that is, the compiler substitutes the appropriate statements for any reference to the procedure. Therefore, any source that references `timer_off` must include the `21160.h` header file.

## timer\_on

enable ADSP-21xxx DSP timer

### Synopsis

```
#include <21160.h>
unsigned int timer_on (void);
```

### Description

The `timer_on` function enables the ADSP-21xxx 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 <21160.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](#)



The function is supplied only as an in-lined procedure; that is, the compiler substitutes the appropriate statements for any reference to the procedure. Therefore, any source that references `timer_on` must include the `21160.h` header file.



## timer\_set

initialize ADSP-21xxx DSP timer

### Synopsis

```
#include <21160.h>
int timer_set (unsigned int tperiod,
               unsigned int tcount);
```

### Description

The `timer_set` function sets the ADSP-21xxx timer registers `TPERIOD` and `TCOUNT`. The function returns a 1 if the timer is enabled, or a zero if the timer is disabled.



Each interrupt call takes approximately 50 cycles on entrance and 50 cycles on return. If `TPERIOD` and `TCOUNT` registers are set too low, you may incur an initializing overhead that could create an infinite loop.

### Error Conditions

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

### Example

```
#include <21160.h>
if (timer_set (1000, 1000) != 1)
    timer_on ();    /* enable timer */
```

### See Also

[timer\\_on](#), [timer\\_off](#)



The function is supplied only as an in-lined procedure; that is, the compiler substitutes the appropriate statements for any reference to the procedure. Therefore, any source that references `timer_set` must include the `21160.h` header file.

## transpm

real matrix transpose

### Synopsis

```
#include <matrix.h>
void transpm (float *output,
              const float *input,
              int rows,
              int columns);
```

### Description

The `transpm` function computes the linear algebraic transpose of the input matrix `input` and stores the result in the matrix `output`. The dimensions of the input matrix are `[rows][columns]`, and the dimensions of the output matrix are `[columns][rows]`.

The function returns a pointer to the output matrix.

### Error Conditions

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

### Example

```
#include <matrix.h>

float a[4][8];
float a_transpose[8][4];

transpm((float *)(a_transpose), (float *)(a), 4, 8);
```

### See Also

No references to this function.

## twidfftf

generate FFT twiddle factors for a fast FFT

### Synopsis

```
#include <filter.h>
void twidfftf(float twid_real[], float twid_imag[], int n);
```

### Description

The `twidfftf` function generates complex twiddle factors for the fast FFT function `cfft`. The function calculates +cosine twiddle factors from `cos(0)` to `cos(pi)` and stores them in the vector `twid_real`, and calculates -sine factors from `-sin(0)` to `-sin(pi)` and stores them in the vector `twid_imag`.

The size of the vectors `twid_real` and `twid_imag` should be  $n/2$  where  $n$  must be a power of 2 and represents the size of the FFT.



For maximum efficiency, the `cfft` function requires that the vectors `twid_real` and `twid_imag` are allocated in separate memory blocks.

### Error Conditions

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

### Example

```
#include <filter.h>

#define FFT_SIZE 1024

section("seg_dmdata") float twid_r[FFT_SIZE/2];
section("seg_pmdata") float twid_i[FFT_SIZE/2];

twidfftf(twid_r,twid_i,FFT_SIZE);
```

```
cfft_f(input_r, input_i,  
        temp_r, temp_i,  
        twid_r, twid_i, FFT_SIZE);
```

### See Also

[cfft\\_f](#)

# DSP Run-Time Library Reference

## var

variance

### Synopsis

```
#include <stats.h>
float var (const float input[], int length);
```

### Description

The `var` function returns the variance of its floating-point input array.

### Error Conditions

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

### Example

```
#include <stats.h>
float input[256], result;

result = var (input, 256);
```

### See Also

[mean](#)



By default, this function uses SIMD; refer to [“Implications of Using SIMD Mode” on page 4-13](#) for more information.

## vecdot

vector dot product

### Synopsis

```
#include <vector.h>
float vecdot (const float dm x_input[],
              const float dm y_input[],
              int samples);
```

### Description

The `vecdot` function computes the dot product of the vectors, `x_input` and `y_input`, which are `samples` in size. The scalar result is returned by the function.

### Error Conditions

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

### Example

```
#include <vector.h>
float answer, x[100], y[100];

answer = vecdot (x, y, 100);
```

### See Also

[cvecdot](#)



By default, this function uses SIMD; refer to [“Implications of Using SIMD Mode” on page 4-13](#) for more information.

## vecsadd

vector scalar addition

### Synopsis

```
#include <vector.h>
float *vecsadd (const float dm input[],
               float scalar,
               float dm output[],
               int samples);
```

### Description

The `vecsadd` function computes the sum of each element of the vector, input, added to the scalar. Both the input and output vectors are `samples` in length. The function returns a pointer to the output vector.

### Error Conditions

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

### Example

```
#include <vector.h>
float answer[50], x[50], y;

vecsadd (x, y, answer, 50);
```

### See Also

[cvecsadd](#)



By default, this function uses SIMD; refer to [“Implications of Using SIMD Mode” on page 4-13](#) for more information.



## vecsm1t

vector scalar multiply

### Synopsis

```
#include <vector.h>
float *vecsm1t (const float dm input[],
               float scalar,
               float dm output[],
               int samples);
```

### Description

The `vecsm1t` function computes the product of each element of the vector, `input`, multiplied by the scalar. Both the input and output vectors are `samples` in length. The function returns a pointer to the output vector.

### Error Conditions

The `vecsm1t` function does not return an error condition.

### Example

```
#include <vector.h>
float answer[50], x[50], y;

vecsm1t (x, y, answer, 50);
```

### See Also

[cvecsm1t](#)



By default, this function uses SIMD; refer to [“Implications of Using SIMD Mode” on page 4-13](#) for more information.

## vecssub

vector scalar subtraction

### Synopsis

```
#include <vector.h>
float *vecssub (const float dm input[],
               float scalar,
               float dm output[],
               int samples);
```

### Description

The `vecssub` function computes the difference of each element of the vector, `input`, minus the scalar. Both the input and output vector are `samples` in length. The function returns a pointer to the output vector.

### Error Conditions

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

### Example

```
#include <vector.h>
float answer[50], x[50], y;

vecssub (x, y, answer, 50);
```

### See Also

[cvecssub](#)



By default, this function uses SIMD; refer to [“Implications of Using SIMD Mode” on page 4-13](#) for more information.

## vecvadd

vector addition

### Synopsis

```
#include <vector.h>
float *vecvadd (const float dm x_input[],
               const float dm y_input[],
               float dm output[],
               int samples);
```

### Description

The `vecvadd` function computes the sum of each of the elements of the vectors, `x_input` and `y_input`, and places the results in `output`. All three vectors are `samples` in length. The function returns a pointer to the output vector.

### Error Conditions

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

### Example

```
#include <vector.h>
float answer[50], x[50], y[50];

vecvadd (x, y, answer, 50);
```

### See Also

[cvecvadd](#)



By default, this function uses SIMD; refer to [“Implications of Using SIMD Mode” on page 4-13](#) for more information.

## vecvmlt

vector multiply

### Synopsis

```
#include <vector.h>
float *vecvmlt (const float dm x_input[],
               const float dm y_input[],
               float dm output[],
               int samples);
```

### Description

The `vecvmlt` function computes the product of each of the elements of the vectors, `x_input` and `y_input`, and places the results in `output`. All three vectors are `samples` in length. The function returns a pointer to the output vector.

### Error Conditions

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

### Example

```
#include <vector.h>
float answer[50], x[50], y[50];

vecvmlt (x, y, answer, 50);
```

### See Also

[cvecvmlt](#)



By default, this function uses SIMD; refer to [“Implications of Using SIMD Mode” on page 4-13](#) for more information.

## vecvsub

vector subtraction

### Synopsis

```
#include <vector.h>
float *vecvsub (const float dm x_input[],
               const float dm y_input[],
               float dm output[],
               int samples);
```

### Description

The `vecvsub` function computes the difference of each of the elements of the vectors, `x_input` and `y_input`, and places the results in `output`. All three vectors are `samples` in length. The function returns a pointer to the output vector.

### Error Conditions

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

### Example

```
#include <vector.h>
float answer[50], x[50], y[50];

vecvsub (x, y, answer, 50);
```

### See Also

[cvecvsub](#)



By default, this function uses SIMD; refer to [“Implications of Using SIMD Mode” on page 4-13](#) for more information.

## zero\_cross

count zero crossings

### Synopsis

```
#include <stats.h>
int zero_cross (const float in[], int length);
```

### Description

The `zero_cross` function returns the number of times that a signal represented in the input array crosses over the zero line. If all input values are zero, the function returns a zero.

### Error Conditions

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

### Example

```
#include <stats.h>
float input[256];
int results;

results = zero_cross (input, 256);
```

### See Also

No references to this function.

# I INDEX

## Symbols

#define preprocessor command, [1-121](#)

#include preprocessor command,  
[1-120](#)

#pragma align num, [1-92](#)

#pragma compiler support, [1-91](#)

#pragma interrupt, [1-94](#)

#pragma SIMD\_for command, [1-105](#)

.EXTERN assembler directive, [1-163](#)

.SECTION directive, [1-79](#), [1-125](#)

@ filename (command file) compiler  
switch, [1-21](#)

\_\_lib\_prog\_term label, [2-58](#)

\_\_2106x\_\_ macro, [1-115](#)

\_\_2116x\_\_ macro, [1-115](#)

\_\_ADSP21000\_\_ macro, [1-115](#)

\_\_ADSP21020\_\_ macro, [1-115](#)

\_\_ADSP21060\_\_ macro, [1-116](#)

\_\_ADSP21061\_\_ macro, [1-116](#)

\_\_ADSP21062\_\_ macro, [1-116](#)

\_\_ADSP21065L\_\_ macro, [1-117](#)

\_\_ADSP21160\_\_ macro, [1-117](#)

\_\_ADSP21161\_\_ macro, [1-117](#)

\_\_ANALOG\_EXTENSIONS\_\_  
macro, [1-118](#)

\_\_ARRAY\_OPERATORS macro,  
[1-50](#)

\_\_cplusplus macro, [1-118](#)

\_\_DATE\_\_ macro, [1-118](#)

\_\_DOUBLES\_ARE\_FLOATS\_\_  
macro, [1-28](#), [1-118](#)

\_\_ECC\_\_ macro, [1-118](#)

\_\_EDG\_\_ macro, [1-118](#)

\_\_EDG\_VERSION\_\_ macro, [1-119](#)

\_\_EXPLICIT macro, [1-49](#)

\_\_FILE\_\_ macro, [1-119](#)

\_\_GROUPNAME\_\_ macro, [1-44](#)

\_\_HOSTNAME\_\_ macro, [1-44](#)

\_\_lib\_setup\_processor routine, [1-129](#)

\_\_LINE\_\_ macro, [1-119](#)

\_\_MACHINE\_\_ macro, [1-44](#)

\_\_NO\_BUILTIN macro, [1-119](#)

\_\_primIO function, [2-15](#)

\_\_REALNAME\_\_ macro, [1-44](#)

\_\_SIGNED\_CHARS\_\_ macro, [1-119](#)

\_\_STDC\_\_ macro, [1-119](#)

\_\_STDC\_VERSION\_\_ macro, [1-120](#)

\_\_STRICT\_ANSI\_\_ macro, [1-51](#)

\_\_SYSTEM\_\_ macro, [1-44](#)

\_\_TIME\_\_ macro, [1-120](#)

\_\_TYPENAME macro, [1-52](#)

\_\_USERNAME\_\_ macro, [1-44](#)

\_\_VERSION\_\_ macro, [1-120](#)

\_\_ADI\_THREADS macro, [1-45](#)

# INDEX

`_NO_LONGLONG` macro, [1-119](#)

`_wordsize.h` header file, [2-16](#)

$\mu$ -law

companders

ADSP-2106x, [3-5](#)

ADSP-2116x, [4-4](#)

compression function

ADSP-2106x, [3-70](#)

ADSP-2116x, [4-79](#)

expansion function

ADSP-2106x, [3-71](#)

ADSP-2116x, [4-80](#)

‘for’ statement, [1-52](#)

## Numerics

-21020 (ADSP-21020 macro)

compiler switch, [1-21](#)

-21060 (ADSP-21060 macro)

compiler switch, [1-21](#)

21060.h header file, [3-4](#)

-21061 (ADSP-21061 macro)

compiler switch, [1-22](#)

-21062 (ADSP-21062 macro)

compiler switch, [1-22](#)

-21065L (ADSP-21065L macro)

compiler switch, [1-22](#)

21065L.h header file, [3-4](#)

-21160 (ADSP-21160 macro)

compiler switch, [1-22](#)

21160.h header file, [4-4](#)

-21160rev0 compiler switch, [1-23](#)

-21161 (ADSP-21161 macro)

compiler switch, [1-23](#)

32-bit floating-point arithmetic, [1-54](#)

64-bit floating-point arithmetic, [1-54](#)

## A

-A (assert) compiler switch, [1-23](#)

`a_compress` function, [3-14](#), [4-16](#)

`a_expand` function, [3-15](#), [4-17](#)

abend (see abort function)

abort (abnormal program end)

function, [2-30](#)

Abridged C++ library, [2-21](#)

abs (absolute value, int) function, [2-31](#)

absolute value (see abs, fabs, fabsf, labs functions)

accessing

system registers, [1-87](#)

system registers from C, [4-10](#)

acos, acosf (arc cosine) function, [2-32](#)

ADSP21000 macro, [1-115](#)

ADSP21020 macro, [1-116](#)

ADSP21060 macro, [1-116](#)

ADSP21061 macro, [1-116](#)

ADSP21062 macro, [1-117](#)

ADSP21065L macro, [1-117](#)

ADSP-21065L programmable timers

disabling, [3-83](#)

enabling, [3-85](#)

ADSP-2106x

built-in DSP functions, [3-11](#)

DSP run-time library reference, [3-13](#)  
to [3-99](#)

ADSP-2106x functions

`a_compress`, [3-14](#)

`a_expand`, [3-15](#)

`autocoh`, [3-16](#)



- autocorr, [3-17](#)
- biquad, [3-18](#), [4-21](#)
- cabsf, [3-21](#)
- cexpf, [3-22](#)
- cfftN, [3-23](#)
- copysign, [3-26](#)
- copysignf, [3-26](#)
- cot, cotf, [3-27](#)
- crosscoh, [3-28](#)
- crosscorr, [3-29](#)
- cvecdot, [3-30](#)
- cvecsadd, [3-31](#)
- cvecsmult, [3-32](#)
- cvecssub, [3-33](#)
- cvecvadd, [3-34](#)
- cvecvmult, [3-35](#)
- cvecvsub, [3-36](#)
- favg, [3-37](#)
- favgf, [3-37](#)
- fclip, [3-38](#)
- fclipf, [3-38](#)
- fir, [3-39](#)
- fmax, [3-41](#)
- fmaxf, [3-41](#)
- fmin, [3-42](#)
- fminf, [3-42](#)
- histogram, [3-55](#)
- idle, [3-57](#)
- ifftN, [3-58](#)
- iir, [3-61](#)
- matinv, [3-65](#), [4-74](#)
- matmul, [3-66](#)
- matscaltmult, [3-67](#)
- matsub, [3-68](#)
- mean, [3-69](#)
- mu\_compress, [3-70](#)
- mu\_expand, [3-71](#)
- poll\_flag\_in, [3-72](#)
- rfftN, [3-74](#)
- rms, [3-77](#)
- rsqrt, [3-78](#)
- rsqrtf, [3-78](#)
- set\_flag, [3-79](#)
- set\_semaphore, [3-81](#)
- timer\_off, [3-82](#)
- timer\_on, [3-84](#)
- timer\_set, [3-86](#)
- timer0\_off, [3-83](#)
- timer0\_on, [3-85](#)
- timer0\_set, [3-88](#)
- timer1\_off, [3-83](#)
- timer1\_on, [3-85](#)
- timer1\_set, [3-88](#)
- transpm, [3-90](#)
- var (variance), [3-91](#)
- vecdot, [3-92](#)
- vecsadd, [3-93](#)
- vecsmult, [3-94](#)
- vecssub, [3-95](#)
- vecvadd, [3-96](#)
- vecvmult, [3-97](#)
- vecvsub, [3-98](#)
- zero\_cross, [3-99](#)
- ADSP21160 macro, [1-117](#)
- ADSP21161 macro, [1-118](#)
- ADSP-2116x
  - built-in DSP functions, [4-12](#)
  - DSP run-time library reference, [4-15](#)

# INDEX

- to [4-106](#)
- serial ports, [4-10](#)
- ADSP-2116x functions
  - [a\\_compress](#), [4-16](#)
  - [a\\_expand](#), [4-17](#)
  - [autocoh](#), [4-18](#)
  - [autocorr](#), [4-19](#)
  - [cabsf](#), [4-24](#)
  - [cexpf](#), [4-25](#)
  - [cfft](#), [4-29](#)
  - [cfftN](#), [4-26](#)
  - [copysign](#), [4-32](#)
  - [copysignf](#), [4-32](#)
  - [cot](#), [cotf](#), [4-33](#)
  - [crosscoh](#), [4-34](#), [4-35](#)
  - [crosscorr](#), [4-36](#)
  - [cvecdot](#), [4-38](#)
  - [cvecsadd](#), [4-39](#)
  - [cvecsmult](#), [4-40](#)
  - [cvecssub](#), [4-41](#)
  - [cvecvadd](#), [4-42](#)
  - [cvecvmlt](#), [4-43](#)
  - [cvecvsub](#), [4-44](#)
  - [favg](#), [4-45](#)
  - [favgf](#), [4-45](#)
  - [fclip](#), [4-46](#)
  - [fclipf](#), [4-46](#)
  - [fft\\_mag](#), [4-47](#)
  - [fir](#), [4-48](#)
  - [fmax](#), [4-50](#)
  - [fmaxf](#), [4-50](#)
  - [fmin](#), [4-51](#)
  - [fminf](#), [4-51](#)
  - [histogram](#), [4-64](#)
  - [idle](#), [4-66](#)
  - [ifftN](#), [4-67](#)
  - [iir](#), [4-70](#)
  - [matadd](#), [4-73](#)
  - [matmul](#), [4-75](#)
  - [matscaltmult](#), [4-76](#)
  - [matsub](#), [4-77](#)
  - [mean](#), [4-78](#)
  - [mu\\_compress](#), [4-79](#)
  - [mu\\_expand](#), [4-80](#)
  - [poll\\_flag\\_in](#), [4-81](#)
  - [rfftN](#), [4-83](#)
  - [rms](#), [4-86](#)
  - [rsqrt](#), [4-87](#)
  - [rsqrtf](#), [4-87](#)
  - [set\\_flag](#), [4-88](#)
  - [set\\_semaphore](#), [4-90](#)
  - SIMD execution model, [4-27](#), [4-48](#),  
[4-68](#), [4-84](#)
  - [timer\\_off](#), [4-91](#)
  - [timer\\_on](#), [4-92](#)
  - [timer\\_set](#), [4-93](#)
  - [transpm](#), [4-95](#)
  - [twidfft](#), [4-96](#)
  - [var](#), [4-98](#)
  - [vecdot](#), [4-99](#)
  - [vecsadd](#), [4-100](#)
  - [vecsmult](#), [4-101](#)
  - [vecssub](#), [4-102](#)
  - [vecvadd](#), [4-103](#)
  - [vecvmlt](#), [4-104](#)
  - [vecvsub](#), [4-105](#)
  - [zero\\_cross](#), [4-106](#)

- aggregate assignment support  
    (compiler), [1-85](#)
- aggregate constructor expression, [1-85](#)
- A-law
  - companders
    - ADSP-2106x, [3-5](#)
    - ADSP-2116x, [4-4](#)
  - compression function
    - ADSP-2106x, [3-14](#)
    - ADSP-2116x, [4-16](#)
  - expansion function
    - ADSP-2106x, [3-15](#)
    - ADSP-2116x, [4-17](#)
- algebraic functions (see math functions)
- algorithm header file, [2-26](#)
- ALIGN NUM pragma, [1-92](#)
- aligned-stack (align stack) compiler switch, [1-24](#)
- allocate memory (see calloc, free, malloc, realloc functions)
- alphabetic character test (see isalpha function)
- alphanumeric character test (see isalnum function)
- alter macro, [1-161](#)
- alternate heap interface functions, [1-135](#)
  - entry point names, [1-135](#)
- alternate registers, [1-137](#), [1-141](#)
- alttok (alternative tokens) C++ mode compiler switch, [1-24](#)
- analog (Analog C compilation) compiler switch, [1-20](#)
- ANSI standard compiler, [1-20](#), [1-29](#)
- archiver, [1-2](#)
- arguments and return transfer, [1-147](#)
- array search, binary (see bsearch function)
- array storage, [1-150](#)
- ASCII string (see atof, atoi, atol functions)
- asin (arc sine) function, [2-33](#)
- asinf (arc sine) function, [2-33](#)
- asm compiler keyword, [1-62](#), [1-64](#)
  - construct template operands, [1-68](#)
- asm volatile() construct, [1-71](#)
- asm() construct, [1-64](#), [1-71](#)
  - and macros, [1-74](#)
  - input and output operands, [1-72](#)
  - multiple instructions, [1-71](#)
  - operand, [1-68](#)
  - optimizing, [1-71](#)
  - reordering, [1-71](#)
  - syntax, [1-65](#)
  - template, [1-65](#)
- asm\_sprt.h header file, [1-159](#), [3-5](#), [4-4](#)
- assembler
  - for SHARC processors, [1-2](#)
- assembly language support keyword (asm), [1-172](#)
- assert.h header file, [2-8](#)
- atan (arc tangent) function, [2-34](#)
- atan2 (arc tangent division) function, [2-35](#)
- atan2f (arc tangent division) function, [2-35](#)
- atanf (arc tangent) function, [2-34](#)

# INDEX

atexit (select exit function) function,  
2-36  
atoi (string to int) function, 2-38  
atol (string to long) function, 2-39  
autocoh function, 3-16, 4-18  
autocorr function, 3-17, 4-19  
-auto-inline (auto inline) compiler  
switch, 1-24  
automatic variables, 1-75  
average (mean of 2 int) function, 2-40

## B

background registers, 1-137, 1-141  
bin\_size parameter, 3-55, 4-64  
binary array search (see bsearch  
function)  
biquad function, 3-18, 4-21  
BITS\_PER\_WORD constant, 2-16  
bool (see Boolean type support  
keywords (bool, true, false))  
Boolean type support keywords (bool,  
true, false), 1-80  
boot loader, 1-129  
bsearch (array search, binary) function,  
2-41  
build tools, 1-29  
-build-lib (build library) compiler  
switch, 1-25  
built-in DSP functions  
ADSP-2106x, 3-11  
ADSP-2116x, 4-12  
built-in functions, 1-86  
C compiler, 2-19  
C/C++ compiler support, 4-12

circular buffer, 1-90  
bus lock control, 3-81

## C

-C (comments) compiler switch, 1-25  
-c (compile only) compiler switch,  
1-25  
C language extensions  
C++ style comments, 1-63  
preprocessor generated warnings,  
1-63  
C run-time library reference, 2-29 to  
2-167  
C type functions  
islower, 2-84  
-c++ (C++ mode) compiler switch,  
1-20  
C++ fractional arithmetic, 1-96  
C++ language extension  
fract data type, 1-63  
C++ member functions in assembly,  
1-165  
C++ mode compiler, 1-50  
C++ programming examples, 1-168  
complex support, 1-169  
fract support, 1-168  
C++ style comments, 1-86  
C/assembly interface (see mixed  
C/assembly programming)  
C/C++ data types, 1-53  
C/C++ language extensions, 1-61  
aggregate assignments, 1-63  
asm keyword, 1-64  
bool keyword, 1-62

- dm keyword, 1-74
- false keyword, 1-62
- indexed initializers, 1-63
- inline keyword, 1-63
- non-constant initializers, 1-62
- pm keyword, 1-74
- section keyword, 1-62
- true keyword, 1-62
- variable length arrays, 1-62
- C/C++ mode selection, 1-20
- C/C++ run-time environment, 1-123
  - (see mixed C/C++/assembly programming)
- C/C++ run-time library guide, 2-3 to 2-28
- cabsf (complex absolute value)
  - function, 3-21, 4-24
- call preserved registers, 1-139, 1-181
- calling
  - assembly language subroutines from C/C++ programs, 1-154
  - C/C++ functions from assembly language programs, 1-156
  - C/C++ library functions, 2-3
  - DSP library functions, 3-2, 4-2
- calloc (allocate initialized memory)
  - function, 2-43
- calloc heap function, 1-130
- ccall macro, 1-160, 1-173
- ceil (ceiling) function, 2-44
- ceilf (ceiling) function, 2-44
- cexpf (complex exponential) function, 3-22, 4-25
- cfft (fast N point complex input FFT)
  - function, 4-29
- cfftN (N-point complex input fast Fourier transform) functions, 3-23, 4-26
- char storage, 1-150
- character string search (see strchr function)
- character string search, recursive (see strchr function)
- circbuf (circular buffer) compiler switch, 1-25
- circindex (circular buffer operation on loop index) function, 2-45
- circptr (circular buffer operation on pointer) function, 2-47
- circular buffer operation
  - on a pointer, 2-47
  - on loop index, 2-45
- circular buffers
  - built-in functions, 1-90
  - disabling, 2-12
  - increment of array references, 1-91
  - increment of index, 1-90
  - increment of pointer, 1-90
  - setting features of, 3-7, 4-8
- clear\_interrupt (clear pending)
  - function, 2-49
- clip (x by y, int) function, 2-54
- code optimization
  - enabling, 1-38
  - for size, 1-38
- comm.h header file, 3-5, 4-4

# INDEX

- compare memory range (see memcmp function)
- compare, strings (see strcmp, strcoll, strcspn, strpbrk, strncmp, strstr functions)
- compiler
  - C/C++ language extensions, [1-60](#)
  - command-line interface, [1-4](#)
  - prelinker, [1-57](#)
  - registers, [1-137](#)
- complex header file, [2-22](#)
- complex vector functions, [3-5](#), [4-5](#)
- complex.h header file, [3-5](#), [4-5](#)
- complex\_float
  - operator, [2-22](#)
  - variables, [4-5](#)
- complex\_long\_double operator, [2-22](#)
- compound statement., [1-121](#)
- concatenate, string (see strcat, strncat function)
- const keyword, [1-20](#)
- control character test (see iscntrl function)
- convert, characters (see tolower, toupper functions)
- convert, strings to doubles (see atof, atoi, atol, strtod, strtok, strtol, strtoul, functions)
- convert, strings to long integer (see atof, atoi, atol, strtok, strtol, strtoul, functions)
- copy memory range (see memcpy function)
- copy, string (see strcpy, strncpy function)
- copysign function, [3-26](#)
- copysign, copysignf functions, [4-32](#)
- copysignf function, [3-26](#)
- cos (cosine) function, [2-55](#)
- cosf (cosine) function, [2-55](#)
- cosh (hyperbolic cosine) function, [2-56](#)
- coshf (hyperbolic cosine) function, [2-56](#)
- cot (cotangent) function, [3-27](#), [4-33](#)
- cotf (cotangent) function, [3-27](#), [4-33](#)
- crosscoh function, [3-28](#), [4-34](#)
- crosscorr function, [3-29](#), [4-36](#)
- C-type functions
  - isalnum, [2-79](#)
  - isalpha, [2-80](#)
  - iscntrl, [2-81](#)
  - isdigit, [2-82](#)
  - isgraph, [2-83](#)
  - islower, [2-86](#)
  - isprint, [2-89](#)
  - ispunct, [2-90](#)
  - isspace, [2-91](#)
  - isupper, [2-92](#)
  - isxdigit, [2-93](#)
  - tolower, [2-162](#)
  - toupper, [2-163](#)
- ctype.h header file, [2-8](#)
- cvcddot (complex vector dot product) function, [3-30](#), [4-38](#)
- cvecsadd (complex vector scalar addition) function, [3-31](#), [4-39](#)

cvecsmult (complex vector scalar multiply) function, [3-32](#), [4-40](#)  
 cvecssub (complex vector scalar subtraction) function, [3-33](#), [4-41](#)  
 cvector.h header file, [3-5](#), [4-5](#)  
 cvecvadd (complex vector addition) function, [3-34](#), [4-42](#)  
 cvecvmult (complex vector multiply) function, [3-35](#), [4-43](#)  
 cvecvsub (complex vector subtraction) function, [3-36](#), [4-44](#)

## D

-D (define macro) compiler switch, [1-26](#), [1-45](#)

### data

memory storage, [1-126](#)  
 storage formats, [1-150](#)

data alignment pragmas, [1-92](#)

### data packing

for primitive I/O, [2-15](#)

### data structure

for primitive I/O, [2-16](#)

### data types, [1-53](#)

double, [1-54](#)

float, [1-54](#)

fract, [1-53](#), [1-96](#)

int, [1-54](#)

long, [1-54](#)

data\_imag array, [4-29](#)

data\_real array, [4-29](#)

deallocate memory (see free function)

### debugging

source-level, [1-30](#)

debugging information, [1-30](#), [1-55](#)  
 for header file, [1-26](#)

removing, [1-43](#)

-debug-types compiler switch, [1-26](#)

def21020.h header file, [3-6](#)

def21060.h header file, [3-6](#)

def21061.h header file, [3-6](#)

def21062.h header file, [3-6](#)

def21065l.h header file, [3-6](#)

def21160.h header file, [4-5](#)

def21161.h header file, [4-6](#)

-default-linkage (assembler, C, or C++) compiler switch, [1-26](#)

### defining

run-time label, [2-124](#)

deque header file, [2-26](#)

digit character test (see isdigit function)

### disabling

ADSP-21065L programmable timers, [3-83](#)

circular buffering, [2-12](#)

### dispatcher

super fast interrupt, [1-142](#)

div (division, int) function, [2-57](#)

division (see div, ldiv functions)

dm (see dual memory support

keywords (pm dm))

DMA support functions, [4-6](#)

dma.h header file, [3-7](#), [4-6](#)

-dollar (dollar format) compiler switch, [1-27](#)

double storage format, [1-27](#), [1-150](#)

-double-size-32 (single-precision double) compiler switch, [1-27](#)

# INDEX

- double-size-64 (double-precision double) compiler switch, [1-27](#)
- driver I/O redirection, [1-48](#)
- dry (terse -dry-run) compiler switch, [1-28](#)
- dry-run (verbose dry-run) compiler switch, [1-28](#)
- dual-memory support keywords (pm dm), [1-74](#)

## E

- E (stop after preprocessing) compiler switch, [1-28](#)
- easm21k utility, [1-2](#)
- EDOM macro, [2-10](#)
- EE (run after preprocessing) compiler switch, [1-28](#)
- elfar archive library, [1-2](#)
- elfloader utility, [1-128](#)
- Embedded C++ library header files
  - complex, [2-22](#)
  - exception, [2-22](#)
  - fract, [2-22](#)
  - fstream, [2-22](#)
  - fstreams.h, [2-28](#)
  - iomanip, [2-23](#)
  - ios, [2-23](#)
  - iosfwd, [2-23](#)
  - istream, [2-23](#)
  - istream.h, [2-28](#)
  - istream, [2-23](#)
  - new, [2-23](#)
  - new.h, [2-28](#)
  - ostream, [2-23](#)

- sstream, [2-24](#)
- stdexcept, [2-24](#)
- streambuf, [2-24](#)
- string, [2-24](#)
- strstream, [2-24](#)
- Embedded Standard Template Library, [2-26](#)
- enabling
  - ADSP-21065L programmable timers, [3-85](#)
- end (see atexit, exit functions)
- entry macro, [1-159](#)
- ERANGE macro, [2-10](#)
- errno.h header file, [2-8](#)
- examples
  - inline assembly (add), [1-172](#)
  - registers for arguments and return (add 2), [1-178](#)
  - scratch registers (dot oroduct), [1-174](#)
  - stack for arguments and return (add 5), [1-177](#)
  - void functions (delay), [1-176](#)
- exception header file, [2-22](#)
- exit (program termination) function, [2-58](#)
- exit macro, [1-145](#), [1-159](#)
- exp, expf (exponential) function, [2-59](#)
- expert-linker compiler switch, [1-29](#)
- explicit C++ mode compiler switch, [1-49](#)
- exponential (see exp, expf, ldexp, ldexpf functions)
- extra-keywords (not quite -analog) compiler switch, [1-29](#)



## F

fabs (absolute value, float) function,  
2-60

fabsf (absolute value, float) function,  
2-60

false (see Boolean type support  
keywords (bool, true, false))

far jump return (see longjmp, setjmp  
functions)

Fast Fourier Transform (FFT)  
functions, 3-9, 4-6, 4-10

fast interrupt dispatcher, 2-11

fast N-point complex input FFT (cfft)  
function, 4-29

favg (mean of two values) function,  
3-37, 4-45

favgf (mean of two values) function,  
3-37, 4-45

fclip function, 3-38, 4-46

fclipf function, 3-38, 4-46

FFT twiddle factors for fast FFT, 4-96

fft\_mag (FFT magnitude) function,  
4-47

FIFO anomaly  
avoiding, 1-44

file  
extensions, 1-5, 1-7  
searches, 1-7

file name  
description, 1-21

fill memory range (see memset  
function)

filter.h header file, 4-6

filters

digital signal processing, 4-6

for ADSP-2106x processors, 3-7, 4-8

for ADSP-2116x processors, 4-6, 4-8

filters.h header file, 3-7, 4-8

final interrupt dispatcher, 2-12

finish processing argument list (see  
va\_end function)

finite impulse response (FIR) filter,  
3-39, 4-48

fir function, 3-39, 4-48

flags, 3-79

-flags (command line input) compiler  
switch, 1-29

float storage format, 1-27, 1-150

float.h header file, 2-9

-float\_to\_int compiler switch, 1-29

floating-point data types, 1-54

floor (integral value) function, 2-61

floorf (integral value) function, 2-61

FLT\_ROUNDS macro, 2-9

fmax function, 3-41, 4-50

fmaxf function, 3-41, 4-50

fmin function, 3-42, 4-51

fminf function, 3-42, 4-51

fmod, fmodf (find remainder)  
functions, 2-62

-fp-associative (floating-point  
associative operation) compiler  
switch, 1-30

fract data type, 1-53, 1-96

fract header file, 2-22

fractional

(fixed-point) arithmetic, 1-96

arithmetic operators, 1-97

# INDEX

- literals, [1-97](#)
- saturated arithmetic, [1-99](#)
- frame pointer, [1-141](#), [1-142](#)
- free (deallocate memory) functions, [2-63](#)
- free heap function, [1-130](#)
- frexp (fraction/exponent) function, [2-64](#)
- frexpf (fraction/exponent) function, [2-64](#)
- fstream header file, [2-22](#)
- fstream.h header file, [2-28](#)
- full-version (display versions) compiler switch, [1-30](#)
- function
  - arguments/return value transfer, [1-147](#)
  - call return address, [1-173](#)
  - declarations with pointers, [1-77](#)
  - entry (prologue), [1-142](#), [1-173](#)
  - exit (epilogue), [1-142](#), [1-173](#)
  - primitive I/O, [2-13](#)
- functional header file, [2-26](#)

## G

- g (generate debug information) compiler switch, [1-30](#)
- gen\_bartlett (generate bartlett window) function, [3-43](#), [4-52](#)
- gen\_blackman (generate blackman window) function, [3-45](#), [4-54](#)
- gen\_gaussian (generate gaussian window) function, [3-46](#), [4-55](#)

- gen\_hamming (generate hamming window) function, [3-47](#), [4-56](#)
- gen\_hanning (generate hanning window) function, [3-48](#), [4-57](#)
- gen\_harris (generate harris window) function, [3-49](#), [4-58](#)
- gen\_kaiser (generate kaiser window) function, [3-50](#), [4-59](#)
- gen\_rectangular (generate rectangular window) function, [3-52](#), [4-61](#)
- gen\_triangle (generate triangle window) function, [3-53](#), [4-62](#)
- get locale pointer (see localeconv function)
- get next argument in list (see va\_arg function)
- getenv (get string definition from operating system) function, [2-65](#)
- gets macro, [1-160](#)
- graphical character test (see isgraph function)

## H

- H (list \*.h) compiler switch, [1-30](#)
- hash\_map header file, [2-26](#)
- hash\_set header file, [2-26](#)
- header files, [1-120](#)
  - ANSI C standard functions, [2-7](#)
  - ANSI standard run-time environment macros, [2-7](#)
  - system, [1-120](#), [1-159](#)
  - user, [1-120](#)
  - working with, [2-6](#)
- header files (ADSP-2102x)

- def21020.h, [3-6](#)
- header files (ADSP-2106x), [3-4](#)
  - 21060.h, [3-4](#)
  - 21065l.h, [3-4](#)
  - asm\_sprt.h, [3-5](#)
  - comm.h, [3-5](#)
  - complext.h, [3-5](#)
  - cvector.h, [3-5](#)
  - def21060.h, [3-6](#)
  - def21061.h, [3-6](#)
  - def21062.h, [3-6](#)
  - def21065l.h, [3-6](#)
  - dma.h, [3-7](#)
  - filters.h, [3-7](#)
  - macros.h, [3-7](#)
  - math.h, [3-7](#)
  - matrix.h, [3-8](#)
  - saturate.h, [3-8](#)
  - sprt.h, [3-8](#)
  - stats.h, [3-9](#)
  - sysreg.h, [3-9](#)
  - trans.h, [3-9](#)
  - vector.h, [3-10](#)
  - window.h, [3-10](#)
- header files (ADSP-2116x), [4-4](#)
  - 21160.h, [4-4](#)
  - asm\_sprt.h, [4-4](#)
  - comm.h, [4-4](#)
  - complex.h, [4-5](#)
  - cvector.h, [4-5](#)
  - def21160.h, [4-5](#)
  - def21161.h, [4-6](#)
  - dma.h, [4-6](#)
  - filter.h, [4-6](#)
  - filters.h, [4-8](#)
  - macro.h, [4-8](#)
  - math.h, [4-8](#)
  - matrix.h, [4-9](#)
  - saturate.h, [4-9](#)
  - sprt.h, [4-10](#)
  - stats.h, [4-10](#)
  - sysreg.h, [4-10](#)
  - trans.h, [4-10](#)
  - vector.h, [4-10](#)
  - window.h, [4-11](#)
- header files (C++ for C facilities)
  - cassert, [2-25](#)
  - cctype, [2-25](#)
  - cerrno, [2-25](#)
  - cfloat, [2-25](#)
  - climits, [2-25](#)
  - locale, [2-25](#)
  - cmath, [2-25](#)
  - csetjmp, [2-25](#)
  - csignal, [2-25](#)
  - cstdarg, [2-25](#)
  - cstddef, [2-25](#)
  - cstdio, [2-25](#)
  - cstdlib, [2-25](#)
  - cstring, [2-25](#)
- heap, [2-66](#)
  - allocating and initializing memory
    - in, [2-66](#)
  - allocating memory from, [2-72](#)
  - allocating uninitialized memory, [2-108](#)
  - alternate, [1-134](#), [1-135](#)
  - changing memory allocation from,

# INDEX

- [2-74](#)
- identifiers, [1-134](#)
- obtaining primary heap identifier, [2-70](#)
- return memory to, [2-68](#)
- setting for dynamic memory allocation, [2-126](#)
- standard functions, [1-130](#)
- heap\_calloc function, [1-130](#), [1-135](#), [2-66](#)
- heap\_free function, [1-130](#), [1-135](#), [2-68](#)
- heap\_lookup\_name function, [2-70](#)
- heap\_malloc function, [1-130](#), [1-135](#), [2-72](#)
- heap\_realloc function, [1-130](#), [1-135](#), [2-74](#)
- heap\_switch function, [1-134](#)
- heaps
  - multiple, [1-130](#)
- help (command-line help) compiler switch, [1-31](#)
- hexadecimal digit test (see isxdigit function)
- HH (list \*.h and compile) compiler switch, [1-31](#)
- histogram function, [3-55](#), [4-64](#)
- HUGE\_VAL macro, [2-10](#)
- hyperbolic (see cosh, coshf, sinh, sinh, tanh, tanhf functions)

## I

- I (include search directory) compiler switch, [1-37](#)

- I (start include directory) compiler switch, [1-31](#), [1-32](#)
- I/O control block, [2-16](#)
- I/O library
  - default, [2-14](#)
- I/O operations, [2-14](#)
- idle function, [3-57](#), [4-66](#)
- IEEE single/double precision formats, [1-150](#)
- ifftN (N-point radix-2 inverse Fast Fourier transform) functions, [3-58](#), [4-67](#)
- iir (infinite impulse response) function, [3-61](#), [4-70](#)
- include (include file) compiler switch, [1-32](#)
- include files, [1-31](#)
- index in a loop, [2-45](#)
- indexed initializer support (compiler), [1-83](#)
- infinite impulse response (IIR) filter, [4-70](#)
- initialization
  - data storage, [1-128](#)
  - section processing, [1-128](#)
- initialize argument list (see va\_start function)
- initializer
  - indexed, [1-83](#)
  - non-constant, [1-83](#)
- initializing
  - DSP timer, [4-93](#)
- inline
  - control, [1-56](#)

- keyword, [1-20](#), [1-63](#)
- inline assembly language support
  - keyword (asm), [1-64](#)
  - construct optimization, [1-71](#)
  - construct template, [1-65](#)
  - constructs with multiple instructions, [1-71](#)
  - macros containing asm, [1-74](#)
- inline function support keyword (inline), [1-62](#), [1-63](#)
- input flag
  - testing, [3-72](#), [4-81](#)
- int storage format, [1-150](#)
- integer data types, [1-54](#)
- interface support macros, [1-159](#)
  - alter, [1-161](#)
  - ccall, [1-160](#)
  - entry, [1-159](#)
  - exit, [1-159](#)
  - gets, [1-160](#)
  - leaf\_entry, [1-160](#)
  - leaf\_exit, [1-160](#)
  - puts, [1-160](#)
  - reads, [1-160](#)
  - restore\_reg, [1-161](#)
  - save\_reg, [1-161](#)
- interfacing C/C++ and assembly (see mixed C/C++/assembly programming)
- intermediate files
  - saving, [1-43](#)
- interprocedural analysis, [1-55](#), [1-57](#)
- interrupt
  - pragma, [1-94](#)
  - table, [1-153](#)
  - vector table area, [1-129](#)
- interrupt (interrupt handling)
  - functions, [2-77](#)
- interrupt dispatchers, [2-11](#)
  - fast, [2-11](#)
  - final, [2-12](#)
  - normal, [2-11](#)
  - super-fast, [2-11](#)
- interruptcb() function, [2-12](#)
- interruptcbnsm() function, [2-13](#)
- interruptf() function, [2-11](#)
- interruptfnsm() function, [2-12](#)
- interrupts (see clear\_interrupt, interruptf, interrupts, signal, raise functions)
- interruptss() function, [2-12](#)
- intrinsic (built-in) functions, [1-86](#)
- inverse (see acos, asin, atan, atan2 functions)
- iomanip header file, [2-23](#)
- iomanip.h header file, [2-28](#)
- ios header file, [2-23](#)
- iosfwd header file, [2-23](#)
- iostream header file, [2-23](#)
- iostream.h header file, [2-28](#)
- IPA, [1-57](#)
- ipa (interprocedural analysis) compiler switch, [1-32](#)
- isalnum (alphanumeric character test)
  - function, [2-79](#)
- isalpha (alphabetic character test)
  - function, [2-80](#)

# INDEX

isctrl (control character test) function, [2-81](#)  
isdigit (digit character test) function, [2-82](#)  
isgraph (graphical character test) function, [2-83](#)  
isinf (test for infinity) function, [2-84](#)  
islower (lower case character test) function, [2-86](#)  
isnan (test for NAN) function, [2-87](#)  
isprint (printable character test) function, [2-89](#)  
ispunct (punctuation character test) function, [2-90](#)  
isspace (white space character test) function, [2-91](#)  
istream header file, [2-23](#)  
isupper (uppercase character test) function, [2-92](#)  
isxdigit (hexadecimal digit test) function, [2-93](#)  
iterator header file, [2-26](#)

## K

keywords (compiler) (see compiler C/C++ extensions)

## L

-L (library search directory) compiler switch, [1-33](#)  
-l (link library) compiler switch, [1-33](#)  
-L (search library) compiler switch, [1-38](#)

labs (absolute value, long) function, [2-94](#)  
language extensions (compiler) (see compiler C/C++ extensions)  
lavg (mean of two values) function, [2-95](#)  
LC\_COLLATE macro, [2-137](#)  
lclip function, [2-96](#)  
ldexp, ldexpf (exponential, multiply) function, [2-97](#)  
ldiv (division, long) function, [2-98](#)  
leaf assembly routines, [1-171](#)  
leaf\_entry macro, [1-160](#)  
leaf\_exit macro, [1-145](#), [1-160](#)  
legacy code, [1-80](#)  
libio.dlb library, [2-14](#)  
library source code  
    working with, [3-3](#), [4-3](#)  
limits.h header file, [2-9](#)  
-line-debug (limited debugging information) compiler switch, [1-33](#)  
linkage pragmas, [1-95](#)  
LINKAGE\_NAME pragma, [1-95](#)  
linking  
    DSP library functions (ADSP-2106x DSPs), [3-3](#)  
    DSP library functions (ADSP-2116x DSPs), [4-3](#)  
list header file, [2-26](#)  
lmax (larger of two values) function, [2-99](#)  
lmin (smaller of two values) function, [2-100](#)

- locale.h header file, [2-9](#)
- localeconv (localization pointer)
  - function, [2-101](#)
- localization (see localeconv, setlocale, strxfrm functions)
- log (log base e) function, [2-104](#)
- log10 (log base 10) function, [2-105](#)
- log10f (log base 10) function, [2-105](#)
- logf (log base e) function, [2-104](#)
- long jump (see longjmp, setjmp functions)
- long storage format, [1-150](#)
- longjmp (far jump return) function, [2-106](#)
- lowercase (see islower, tolower functions)
- M**
- M (make only) compiler switch, [1-34](#)
- macro.h header file, [3-7](#), [4-8](#)
- macros
  - \_\_ARRAY\_OPERATORS, [1-50](#)
  - \_\_EXPLICIT, [1-49](#)
  - \_\_GROUPNAME\_\_, [1-44](#)
  - \_\_HOSTNAME\_\_, [1-44](#)
  - \_\_MACHINE\_\_, [1-44](#)
  - \_\_REALNAME\_\_, [1-44](#)
  - \_\_STRICT\_ANSI\_\_, [1-51](#)
  - \_\_SYSTEM\_\_, [1-44](#)
  - \_\_TYPENAME, [1-52](#)
  - \_\_USERNAME\_\_, [1-44](#)
- ANSI standard run-time
  - environment, [2-7](#)
- asm() C program constructs and, [1-74](#)
- ccall, [1-173](#)
- compound statements as, [1-121](#)
- EDOM, [2-10](#)
- ERANGE, [2-10](#)
- FLT\_ROUNDS, [2-9](#)
- HUGE\_VAL, [2-10](#)
- interface support, [1-159](#)
- LC\_COLLATE, [2-137](#)
- mixed C/assembly support, [1-159](#)
- predefined, [1-115](#)
- SKIP\_SPACES, [1-121](#)
- stack management, [1-173](#)
- malloc (allocate uninitialized memory)
  - function, [2-108](#)
- malloc heap function, [1-130](#)
- managing
  - stacks, [1-142](#)
- map (generate a memory map)
  - compiler switch, [1-34](#)
- map header file, [2-27](#)
- matadd (matrix addition) function, [3-64](#), [4-73](#)
- math functions
  - acos, [2-32](#)
  - additional, [3-7](#), [4-8](#)
  - asin, asinf, [2-33](#)
  - atan, atanf, [2-34](#)
  - atan2, atan2f, [2-35](#)
  - ceil, ceilf, [2-44](#), [2-47](#)
  - cos, cosf, [2-55](#)
  - cosh, coshf, [2-56](#)
  - exp, expf, [2-59](#)
  - fabs, fabsf, [2-60](#)

## INDEX

- floor, floorf, [2-61](#)
- fmod, fmodf, [2-62](#)
- frexp, frexpf, [2-64](#)
- ldexp, ldexpf, [2-97](#)
- log, logf, [2-104](#)
- log10, log10f, [2-105](#)
- modf, modff, [2-116](#)
- pow, powf, [2-117](#)
- rsqrt, rsqrtf, [3-78](#), [4-87](#)
- sin, sinf, [2-130](#)
- sinh, sinh, [2-131](#)
- sqrt, sqrtf, [2-132](#)
- standard, [3-7](#), [4-8](#)
- tan, tanf, [2-160](#)
- tanh, tanh, [2-161](#)
- math.h header file, [2-9](#), [3-7](#), [4-8](#)
- matinv (real matrix inversion)
  - function, [3-65](#), [4-74](#)
- matmul (matrix multiplication)
  - function, [3-66](#), [4-75](#)
- matrix.h header file, [3-8](#), [4-9](#)
- matscaltmult (multiply matrix by scalar)
  - function, [4-76](#)
- matscaltmult (scaled multiplication)
  - function, [3-67](#)
- matsub (matrix subtraction) function,
  - [3-68](#), [4-77](#)
- max (find larger, int) function, [2-109](#)
- mean function, [3-69](#), [4-78](#)
- mem (enable memory initialization)
  - compiler switch, [1-35](#)
- mem21k utility, [1-128](#), [1-129](#)
- memchr (find character) function,
  - [2-110](#)
- memcmp (compare memory range)
  - function, [2-111](#)
- memcpy (copy memory range)
  - function, [2-112](#)
- memmove (move memory range)
  - function, [2-113](#)
- memory
  - header file, [2-27](#)
  - initializer (mem21k), [1-129](#)
  - map file, [1-34](#)
  - placing code in, [1-125](#)
  - used for placing code in, [1-125](#)
- memory functions (see calloc, free, malloc, memcmp, memcpy, memset, memmove, memchar, realloc functions)
- memory section names, [1-125](#)
- memset (fill memory range) function,
  - [2-114](#)
- min (find smaller, int) function, [2-115](#)
- mixed C/assembly naming
  - conventions, [1-163](#)
- mixed C/assembly programming
  - arguments and return, [1-147](#)
  - asm() constructs, [1-64](#), [1-65](#), [1-68](#), [1-71](#)
  - call preserved registers, [1-139](#)
  - compiler registers, [1-137](#)
  - data storage and type sizes, [1-150](#)
  - examples, [1-171](#)
  - return address, [1-173](#)
  - scratch registers, [1-140](#)
  - stack registers, [1-141](#)
  - stack usage, [1-142](#)



- user registers, [1-138](#)
- mixed C/assembly support macros, [1-159](#)
- mixed C/C++/assembly programming, [1-154](#)
- MM (make rules and compile)
  - compiler switch, [1-34](#)
- MODE2 register, [3-72](#), [4-81](#), [4-88](#)
  - setting, [3-79](#)
- modf (modulus, float) function, [2-116](#)
- modff (modulus, float) function, [2-116](#)
- modulus array references, [1-91](#)
- move memory range (see memmove function)
- MQ (output without compile)
  - compiler switch, [1-34](#)
- Mt filename (output make rule)
  - compiler switch, [1-34](#)
- mu\_compress function, [3-70](#), [4-79](#)
- mu\_expand function, [3-71](#), [4-80](#)
- multi-line asm() C program constructs, [1-71](#)
- multiple heaps, [1-130](#)
- N**
- namespace (enable namespaces) C++
  - mode compiler switch, [1-49](#)
- naming conventions
  - assembly and C, [1-164](#)
  - assembly and C/C ++, [1-163](#)
- natural logarithm (see log, logf functions)
- new header file, [2-23](#)
- new.h header file, [2-28](#)
- newforinit (new 'for' initialization)
  - C++ mode compiler switch, [1-49](#)
- newvec (new vector) C++ mode
  - compiler switch, [1-50](#)
- no-aligned-stack (do not align stack)
  - compiler switch, [1-35](#)
- no-alttok (disable tokens) C++ mode
  - compiler switch, [1-35](#)
- no-builtin (no built-in functions)
  - compiler switch, [1-35](#)
- no-char (disable wide char type) C++
  - mode compiler switch, [1-51](#)
- no-circbuf (no circular buffer)
  - compiler switch, [1-35](#)
- no-def (disable definitions) compiler
  - switch, [1-35](#)
- no-demangle C++ mode compiler
  - switch, [1-50](#)
- no-dir-warnings compiler switch, [1-36](#)
- no-dollar (disable dollar format)
  - compiler switch, [1-36](#)
- no-explicit C++ mode compiler
  - switch, [1-50](#)
- no-extra-keywords (not quite -ansi)
  - compiler switch, [1-36](#)
- no-fp-associative compiler switch, [1-36](#)
- no-inline (disable inline keyword)
  - compiler switch, [1-36](#)
- no-mem (disable memory
  - initialization) compiler switch, [1-36](#)

# INDEX

- no-namespace C++ mode compiler switch, [1-50](#)
- non-constant initializer support (compiler), [1-83](#)
- no-newvec (disallow a new vector) C++ mode compiler switch, [1-50](#)
- non-leaf
  - assembly routines, [1-171](#)
  - routines to make calls (RMS), [1-179](#)
- no-restrict (no restrict support) C++ mode compiler switch, [1-37](#)
- normal interrupt dispatcher, [2-11](#)
- normalized fraction (see frexp, frexpf function)
- no-simd (disable SIMD mode) compiler switch, [1-37](#)
- no-std (disable std namespace) C++ compiler switch, [1-50](#)
- no-std-def (disable standard definitions) compiler switch, [1-37](#)
- no-std-inc (disable standard include search) compiler switch, [1-37](#)
- no-std-lib (disable standard library search) compiler switch, [1-38](#)
- not-a-number (NaN) test, [2-87](#)
- nothreads (disable thread-safe build) compiler switch, [1-38](#)
- notstrict (non-strict compilation) C++ mode compiler switch, [1-51](#)
- N-point complex input Fast Fourier transform, [4-26](#)
- N-point inverse complex input Fast Fourier transform (IFFT), [4-67](#)

- N-point real input fast Fourier transform, [4-83](#)
- numeric header file, [2-27](#)

## O

- O (enable optimization) compiler switch, [1-38](#)
- o (output) compiler switch, [1-38](#)
- OPEN operation
  - bit values, [2-17](#)
- optimization
  - controlling, [1-55](#)
  - enabling, [1-38](#)
  - turning on, [1-57](#)
- optimization pragmas, [1-94](#)
- optimize\_for\_space pragma, [1-95](#)
- optimize\_for\_speed pragma, [1-95](#)
- optimize\_off pragma, [1-95](#)
- Os (optimize for size) compiler switch, [1-38](#)
- ostream header file, [2-23](#)

## P

- P (omit line numbers and compile) compiler switch, [1-39](#)
- P (omit line numbers) compiler switch, [1-39](#)
- PACK (ALIGNOPT) pragma, [1-93](#)
- PAD (ALIGNOPT) pragma, [1-93](#)
- passing
  - arguments to driver, [1-43](#)
  - function parameters, [1-147](#)
- path- (tool location) compiler switch, [1-39](#)

- path-install (installation location)
  - compiler switch, [1-39](#)
- path-output (non-temporary files location) compiler switch, [1-40](#)
- path-temp (temporary files location)
  - compiler switch, [1-40](#)
- pch (precompiled header) compiler switch, [1-40](#)
- pchdir (locate PCHRepository)
  - compiler switch, [1-40](#)
- pedantic (ANSI standard warnings)
  - compiler switch, [1-40](#)
- pedantic-errors (ANSI standard errors) compiler switch, [1-41](#)
- placement support keyword (section), [1-79](#)
- pm (see dual memory support keywords (pm dm))
- pointer class support keyword (restrict), [1-62](#), [1-80](#)
- poll\_flag\_in (testing input flag)
  - function, [3-72](#), [4-81](#)
- pow (power,  $x^y$ ) function, [2-117](#)
- power (see exp, expf, pow, powf functions)
- powf (power,  $x^y$ ) function, [2-117](#)
- pplist (preprocessor listing) compiler switch, [1-41](#)
- pragmas
  - ALIGN NUM, [1-92](#)
  - data alignment, [1-92](#)
  - external linkage, [1-95](#)
  - interrupt handler, [1-94](#)
  - LINKAGE\_NAME, [1-95](#)
  - linking, [1-95](#)
  - optimization, [1-95](#)
  - PACK (ALIGNOPT), [1-93](#)
  - PAD (ALIGNOPT), [1-93](#)
  - RETAIN\_NAME, [1-96](#)
  - SIMD\_for, [1-94](#), [1-105](#)
- predefined macros, [1-115](#)
- prelinker, [1-57](#)
- preprocessing
  - program, [1-114](#)
- preprocessor
  - commands, [1-114](#)
  - predefined macros, [1-115](#)
  - warnings, [1-85](#)
- primitive I/O
  - data packing, [2-15](#)
  - data structure, [2-16](#)
- primitive I/O functions, [2-13](#)
- printable character test (see isprint function)
- proc (target processor) compiler switch, [1-41](#)
- program
  - assignments, [1-76](#)
  - memory code storage, [1-126](#)
  - memory data storage, [1-127](#)
- program control functions
  - calloc, [2-43](#)
  - free, [2-63](#)
  - malloc, [2-108](#)
  - realloc, [2-122](#)
- punctuation character test (ispunct)
  - function, [2-90](#)
- puts macro, [1-160](#)

# INDEX

## Q

qsort (quicksort) function, [2-118](#)  
queue header file, [2-27](#)

## R

-R- (disable source path) compiler switch, [1-42](#)  
-R (search for source files) compiler switch, [1-42](#)  
raise (force a signal) function, [2-120](#)  
rand (random number generator) function, [2-121](#)  
random number (see rand, srand functions), [2-121](#)  
READ operation,, [2-17](#)  
reads macro, [1-160](#)  
real matrix inversion, [3-65](#), [4-74](#)  
real matrix transpose, [3-90](#), [4-95](#)  
realloc (allocate used memory) function, [2-122](#)  
realloc heap function, [1-130](#)  
real-time signals (see clear\_interrupt, interruptf, interrupts, poll\_flag\_in, raise, signal functions)  
reciprocal square root function (see rsqrt, rsqrtf function)  
register names, [1-88](#)  
register usage (see mixed C/assembly programming)  
registers  
    alternate, [1-141](#)  
    asm() constructs, [1-68](#)  
    assigning to operands, [1-69](#)  
    call preserved, [1-139](#)

    compiler, [1-137](#)  
    performance, [1-138](#)  
    reserved, [1-42](#)  
    scratch, [1-140](#)  
    stack, [1-141](#)  
    user, [1-138](#)  
-reserve (reserve register) compiler switch, [1-42](#)  
RESOLVE directive, [1-103](#)  
restore\_reg macro, [1-161](#)  
restrict (see pointer class support keyword (restrict))  
-restrict (support restrict keyword) C++ mode compiler switch, [1-43](#)  
restrict operator keyword, [1-80](#)  
RETAIN\_NAME pragma, [1-96](#)  
return value transfer, [1-147](#)  
rfftN functions, [3-74](#), [4-83](#)  
RFRAME anomaly., [1-23](#)  
rms (root mean square) function, [3-77](#), [4-86](#)  
rsqrt (reciprocal square root) math function, [3-78](#)  
rsqrt, rsqrtf (reciprocal square root) math functions, [4-87](#)  
rsqrtf (reciprocal square root) math function, [3-78](#)  
run-time  
    C header, [1-129](#)  
    C/C++ environment (see mixed C/assembly programming)  
    header, [1-153](#)  
    header storage, [1-129](#)  
    heap section, [1-128](#)

- heap storage, [1-128](#)
- label, [2-124](#)
- stack, [1-141](#)
- stack storage, [1-127](#)
- run-time header
  - using, [1-153](#)
- S**
  - S (stop after compilation) compiler switch, [1-43](#)
  - s (strip debug information) compiler switch, [1-43](#)
  - saturate.h header file, [3-8](#), [4-9](#)
  - saturated arithmetic, [1-99](#)
  - save\_reg macro, [1-161](#)
  - save-temps (save intermediate files)
    - compiler switch, [1-43](#)
  - scratch registers, [1-140](#)
  - search character string (see strchr, strrchr functions)
  - search memory, character (see memchar function)
  - search path
    - for include files, [1-31](#)
    - for library files, [1-33](#)
  - secondary registers, [1-137](#), [1-141](#)
  - section keyword, [1-62](#), [1-79](#), [1-125](#)
  - section names, [1-125](#)
  - seg\_init initialization section, [1-128](#)
  - seg\_stak stack section, [1-127](#)
  - segment (see Placement Support Keyword (section))
  - segment keyword (see section keyword)
  - segment legacy keyword, [1-80](#)
  - self-modifying code, [2-12](#)
  - send string to operating system (see system function)
  - serial ports
    - ADSP-2116x DSP, [4-10](#)
  - set header file, [2-27](#)
  - set jump (see longjmp, setjmp functions)
  - set\_alloc\_type (set heap for dynamic memory allocation) function, [2-126](#)
  - set\_alloc\_type function, [1-134](#)
  - set\_flag function, [3-79](#), [4-88](#)
  - set\_semaphore function, [3-81](#), [4-90](#)
  - setjmp (define runtime label) function, [2-124](#)
  - setjmp.h header file, [2-10](#)
  - setlocale (set localization) function, [2-125](#)
  - setvbuf function, [2-14](#)
  - shadow register operation, [1-107](#)
  - short storage format, [1-150](#)
  - show (display command line)
    - compiler switch, [1-43](#)
  - SIGABRT handler, [2-30](#)
  - signal (define signal handling) function, [2-128](#)
  - signal functions
    - clear\_interrupt, [2-49](#)
    - interrupt, [2-77](#)
    - raise, [2-120](#)
    - signal, [2-128](#)
  - signal.h header file, [2-10](#)
  - signalcb() function, [2-12](#)

# INDEX

- signalcbnsm() function, [2-13](#)
- signalf() function, [2-11](#)
- signalfnsm() function, [2-12](#)
- signals (see clear\_interrupt, interruptf, interrupts, poll\_flag\_in, raise, signal functions)
- signalss() function, [2-12](#)
- signed keyword, [1-20](#)
- SIMD mode, [1-99](#)
  - anomaly #40, [1-107](#)
  - C/C++ callable subroutines, [1-167](#)
  - C/C++ constraints in, [1-106](#)
  - C/C++ data alignment rules, [1-112](#)
  - data alignment, [1-103](#), [1-112](#)
  - data increments, [1-110](#)
  - disabling, [1-37](#)
  - loop counter rules, [1-111](#)
  - performance in C/C++, [1-108](#)
  - processing multichannel data in, [1-101](#)
  - processing single channel data in, [1-102](#)
  - restrictions, [1-103](#)
  - SIMD\_for pragma, [1-105](#)
  - using with ADSP-2116x DSPs, [4-13](#)
- SIMD\_for loop., [1-107](#)
- SIMD\_for pragma, [1-94](#), [1-101](#), [1-105](#), [1-110](#)
- sin (sine) function, [2-130](#)
- sine (see sin, sinf, sinh, sinhlf functions)
- sinf (sine) function, [2-130](#)
- sinh (sine hyperbolic) function, [2-131](#)
- sinhf (sine hyperbolic) function, [2-131](#)
- SISD mode, [1-100](#)
- sizeof() operator, [1-82](#)
- SKIP\_SPACES macro, [1-121](#)
- sport.h header file, [3-8](#), [4-10](#)
- sqrt (square root) function, [2-132](#)
- sqrtf (square root) function, [2-132](#)
- srand (random number seed) function, [2-133](#)
- sstream header file, [2-24](#)
- stack
  - frame, [1-142](#)
  - header file, [2-27](#)
  - managing, [1-142](#)
  - managing in memory, [1-142](#)
  - managing with macros, [1-173](#)
  - pointer, [1-141](#)
  - registers, [1-141](#)
- standard
  - heap management functions, [1-130](#)
  - math functions, [3-7](#), [4-8](#)
  - optimizations, [1-30](#)
- standard argument functions
  - va\_arg, [2-164](#)
  - va\_end, [2-166](#)
  - va\_start, [2-167](#)
- standard C library
  - functions, [2-7](#) to [2-19](#)
- standard header files
  - assert.h, [2-8](#)
  - ctype.h, [2-8](#)
  - errno.h, [2-8](#)
  - float.h, [2-9](#)
  - limits.h, [2-9](#)
  - locale.h, [2-9](#)
  - math.h, [2-9](#)

- setjmp.h, [2-10](#)
- signal.h, [2-10](#)
- stdarg.h, [2-13](#)
- stddef.h, [2-13](#)
- stdio.h, [2-13](#)
- stdlib.h, [2-18](#)
- string.h, [2-19](#)
- standard library functions
  - abort, [2-30](#)
  - abs, [2-31](#)
  - acos, acosf, [2-32](#)
  - atexit, [2-36](#)
  - atof, [2-37](#)
  - atoi, [2-38](#)
  - atol, [2-39](#)
  - avg, [2-40](#)
  - bsearch, [2-41](#)
  - calloc, [2-43](#)
  - clip, [2-54](#)
  - div, [2-57](#)
  - exit, [2-58](#)
  - free, [2-63](#)
  - getenv, [2-65](#)
  - heap\_calloc, [2-66](#)
  - heap\_free, [2-68](#)
  - heap\_lookup\_name, [2-70](#)
  - heap\_malloc, [2-72](#)
  - heap\_realloc, [2-74](#)
  - labs, [2-94](#)
  - lavg, [2-95](#)
  - lclip, [2-96](#)
  - ldiv, [2-98](#)
  - lmax, [2-99](#)
  - lmin, [2-100](#)
  - malloc, [2-108](#)
  - max, [2-109](#)
  - min, [2-115](#)
  - qsort, [2-118](#)
  - rand, [2-121](#)
  - realloc, [2-122](#)
  - srand, [2-133](#)
  - strtol, [2-149](#), [2-153](#)
  - strtoul, [2-155](#)
  - system, [2-159](#)
- statistical functions, [4-10](#)
- stats.h header file, [3-9](#), [4-10](#)
- std (std namespace) C++ compiler
  - switch, [1-51](#)
- stdarg.h header file, [2-13](#)
- stddef.h header file, [2-13](#)
- stdexcept header file, [2-24](#)
- stdio header file, [2-13](#)
- stdlib.h header file, [2-18](#)
- stdout output stream, [1-28](#)
- stop (see atexit, exit functions)
- storage format
  - double, [1-27](#)
- strcat (concatenate string) function, [2-134](#)
- strchr (search character string) function, [2-135](#)
- strcmp (compare strings) function, [2-136](#)
- strcoll (compare strings, localized) function, [2-137](#)
- strcpy (copy string) function, [2-138](#)
- strcspn (compare string span) function, [2-139](#)

# INDEX

- streambuf header file, [2-24](#)
- strerror (get error message string)
  - function, [2-140](#)
- strict (strict compilation) C++ mode
  - compiler switch, [1-51](#)
- strictwarn (warn if non-strict) C++ mode
  - mode compiler switch, [1-51](#)
- string compare (see strcmp, strcoll, strcspn, strncmp, strpbrk, strstr functions)
- string concatenate (see strncat, strcat functions)
- string conversion (see atof, atoi, atol, strtok, strtol, strxfrm functions)
- string copy (see strcpy, strncpy function)
- string functions
  - memchar, [2-110](#)
  - memcmp, [2-111](#)
  - memcpy, [2-112](#)
  - memmove, [2-113](#)
  - memset, [2-114](#)
  - strcat, [2-134](#)
  - strchr, [2-135](#)
  - strcmp, [2-136](#)
  - strcoll, [2-137](#)
  - strcpy, [2-138](#)
  - strcspn, [2-139](#)
  - strerror, [2-140](#)
  - strlen, [2-141](#)
  - strncat, [2-142](#)
  - strncmp, [2-143](#)
  - strncpy, [2-144](#)
  - strpbrk, [2-145](#)
  - strrchr, [2-146](#)
  - strspn, [2-147](#)
  - strstr, [2-148](#)
  - strtok, [2-151](#)
  - strxfrm, [2-157](#)
- string header file, [2-24](#)
- string length (see strlen function)
- string.h header file, [2-19](#)
- strlen (string length) function, [2-141](#)
- strncat (concatenate characters from string) function, [2-142](#)
- strncmp (compare characters in strings) function, [2-143](#)
- strncpy (copy characters in string) function, [2-144](#)
- strpbrk (compare strings, pointer break) function, [2-145](#)
- strrchr (search character string, recursive) function, [2-146](#)
- strspn (string span) function, [2-147](#)
- strstr (compare string, string) function, [2-148](#)
- strstream header file, [2-24](#)
- strtod (convert string to a double) function, [2-149](#)
- strtok (convert token to string) function, [2-151](#)
- strtol (convert string to long) function, [2-153](#)
- strtoul (convert string to unsigned long) function, [2-155](#)
- struct alignment, [1-92](#)
- strxfrm (localization transform) function, [2-157](#)



- super-fast interrupt dispatcher, [2-11](#)
- swf\_screening (avoid FIFO anomaly)
  - compiler switch, [1-44](#)
- switches
  - C++ mode compiler
    - explicit (explicit specifier), [1-49](#)
    - namespace (enable namespaces), [1-49](#)
    - newforinit (new 'for' initialization), [1-49](#)
    - newvec (new vector), [1-50](#)
    - no-char (disable wide char type), [1-51](#)
    - no-demangle (disable demangler), [1-50](#)
    - no-namespace (disable namespaces), [1-50](#)
    - no-newvec (disallow a new vector), [1-50](#)
    - no-std (disable std namespace), [1-50](#)
    - notstrict (non-strict compilation), [1-51](#)
    - std (std namespace), [1-51](#)
    - strict (strict standard), [1-51](#)
    - strictwarn (warn if non-strict), [1-51](#)
    - tpautooff (disable automatic template instantiation), [1-52](#)
    - trdforinit (traditional initialization), [1-52](#)
    - typename (type name), [1-52](#)
    - wchar (enable wide char type), [1-52](#)
  - C/C++ mode selection
    - analog (Analog C compilation), [1-20](#)
    - c++ (C++ mode), [1-20](#)
    - traditional (traditional compilation), [1-20](#)
  - compiler common
    - @filename (command file), [1-21](#)
    - 21020 (compile for AD-SP-21020), [1-21](#)
    - 21060 (compile for AD-SP-21060), [1-21](#)
    - 21061 (compile for AD-SP-21061), [1-22](#)
    - 21062 (compile for ADSP-21062), [1-22](#)
    - 21065L (compile for AD-SP-21065L), [1-22](#)
    - 21160 (compile for AD-SP-21160), [1-22](#)
    - 21160rev0 (compile for AD-SP-21160), [1-23](#)
    - 21161 (compile for AD-SP-21161), [1-23](#)
    - A (assert) compiler switch, [1-23](#)
    - aligned-stack (align stack), [1-24](#)
    - alttok (alternative tokens), [1-24](#)
    - auto-inline (auto inline), [1-24](#)
    - build-lib (build library), [1-25](#)
    - C (comments), [1-25](#)
    - c (compile only), [1-25](#)
    - cirbuf (circular buffer), [1-25](#)
    - D (define macro), [1-26](#)
    - debug-types, [1-26](#)

- default-linkage-asm|-c|-c+., 1-26
- dollar (dollar format), 1-27
- dollar (dollar format), 1-27
- double-size-32|64, 1-27
- dry (verbose dry-run), 1-28
- dryrun (terse dry-run), 1-28
- E (stop after preprocessing), 1-28
- EE (run after preprocessing), 1-28
- expert-linker, 1-29
- extra-keywords (enable short-form keywords), 1-29
- flags (command-line input), 1-29
- float\_to\_int, 1-29
- fp-associative (floating-point associative operation), 1-30
- full-version (display versions), 1-30
- g (generate debug information), 1-30
- H (list headers), 1-30
- help (command-line help), 1-31
- HH (list headers and compile), 1-31
- I (start include directory), 1-32
- I directory (include search directory), 1-31
- include (include file), 1-32
- ipa (interprocedural analysis), 1-32
- L (library search directory), 1-33
- l (link library), 1-33
- line-debug (limit debugging information), 1-33
- M (generate make rules only), 1-34
- map filename (generate a memory map), 1-34
- mem (enable memory initialization), 1-35
- MM (generate make rules and compile), 1-34
- MQ (output without compilation), 1-34
- Mt filename (output make rule), 1-34
- no-aligned-stack (disable stack alignment), 1-35
- no-alttok (disable alternative tokens), 1-35
- no-builtin (no built-in functions), 1-35
- no-circbuf (no circular buffer), 1-35
- no-defs (disable defaults), 1-35
- no-dir-warnings, 1-36
- no-dollar (disable dollar format), 1-36
- no-extra-keywords (disable short-form keywords), 1-36
- no-fp-associative, 1-36
- no-inline (disable inline keyword), 1-36
- no-mem (disable memory initialization), 1-36
- no-restrict (disable restrict), 1-37
- no-std-def (disable standard macro definitions), 1-37
- no-std-inc (disable standard in-

- clude search), [1-37](#)
- no-std-lib (disable standard library search), [1-38](#)
- nothreads (disable thread-safe build), [1-38](#)
- O (enable optimizations), [1-38](#)
- o (output file), [1-38](#)
- Os (optimize for size), [1-38](#)
- P (omit line numbers), [1-39](#)
- path- (tool location), [1-39](#)
- path-install (installation location), [1-39](#)
- path-output (non-temporary files location), [1-40](#)
- path-temp (temporary files location), [1-40](#)
- pch (precompiled header), [1-40](#)
- pchdir (locate PCHRepository), [1-40](#)
- pedantic (ANSI standard warnings), [1-40](#)
- pedantic-errors (ANSI standard errors), [1-41](#)
- PP (omit line numbers and compile), [1-39](#)
- pplist (preprocessor listing), [1-41](#)
- proc identifier (processor), [1-41](#)
- R (add source directory), [1-42](#)
- R- (disable source path), [1-42](#)
- reserve (reserve register), [1-42](#)
- restrict (support restrict keyword), [1-43](#)
- S (stop after compilation), [1-43](#)
- s (strip debug information), [1-43](#)
- save-temps (save intermediate files), [1-43](#)
- show (display command line), [1-43](#)
- sourcefile (parameter), [1-21](#)
- swf\_screening (check syntax only), [1-44](#)
- switch-pm (switch tables), [1-44](#)
- syntax-only (system definitions), [1-44](#)
- threads (enable thread-safe build), [1-45](#)
- time (tell time), [1-45](#)
- v (version and verbose), [1-45](#)
- val-global (add global names), [1-46](#)
- version (display version), [1-46](#)
- w (disable all warnings), [1-48](#)
- W number (override error message), [1-46](#)
- warn-protos (warn if incomplete prototype), [1-46](#)
- Wdriver-limit (maximum process errors), [1-47](#)
- Werror-limit (maximum compiler errors), [1-47](#)
- Wremarks (enable diagnostic warnings), [1-47](#)
- write-files (enable driver I/O redirection), [1-48](#)
- write-opts (user options), [1-48](#)
- Wterse (enable terse warnings), [1-47](#)
- xref (cross-reference list), [1-48](#)

# INDEX

- no-explicit (disable explicit specifier)
  - compiler switch, [1-50](#)
- switch-pm (switch tables) compiler switch, [1-44](#)
- syntax-only (check syntax only)
  - compiler switch, [1-44](#)
- sysdef (system definitions) compiler switch, [1-44](#)
- sysreg.h header file, [1-87](#), [3-9](#), [4-10](#)
- sysreg\_bit\_\* interrogation and manipulation functions, [1-89](#)
- sysreg\_read function, [1-87](#)
- sysreg\_write function, [1-87](#)
- system (send string to operating system) function, [2-159](#)
- system registers
  - accessing from C, [1-87](#), [3-9](#), [4-10](#)

## T

- T (linker description file) compiler switch, [1-45](#)
- tan (tangent) function, [2-160](#)
- tanf (tangent) function, [2-160](#)
- tangent (see atan, atanf, atan2, atan2f, cot, cotf, tan, tanf, tanh, tanhf functions)
- tanh (hyperbolic tangent) function, [2-161](#)
- tanhf (hyperbolic tangent) function, [2-161](#)
- TCOUNT registers, [3-83](#), [3-86](#), [4-92](#), [4-93](#)
- template library header files
  - algorithm, [2-26](#)

- deque, [2-26](#)
- functional, [2-26](#)
- hash\_map, [2-26](#)
- hash\_set, [2-26](#)
- iterator, [2-26](#)
- list, [2-26](#)
- map, [2-27](#)
- memory, [2-27](#)
- numeric, [2-27](#)
- queue, [2-27](#)
- set, [2-27](#)
- stack, [2-27](#)
- utility, [2-27](#)
- vector, [2-27](#)
- temporary files, [1-43](#)
- terminate (see atexit, exit functions)
- testing
  - specified input flag, [3-72](#), [4-81](#)
- threads (enable thread-safe build)
  - compiler switch, [1-45](#)
- time (tell time) compiler switch, [1-45](#)
- timer\_off (disable DSP timer)
  - function, [3-82](#), [4-91](#)
- timer\_on (enable DSP timer) function, [3-84](#), [4-92](#)
- timer\_set (initialize DSP timer)
  - function, [4-93](#)
- timer\_set function, [3-86](#)
- timer0\_off function, [3-83](#)
- timer0\_on function, [3-85](#)
- timer0\_set function, [3-88](#)
- timer1\_off function, [3-83](#)
- timer1\_on function, [3-85](#)
- timer1\_set function, [3-88](#)

tokens, string convert (see strtok function)

tolower (convert characters to lower case) function, [2-162](#)

toupper (convert characters to upper case) function, [2-163](#)

-tpautooff (no automatic templates) C++ mode compiler switch, [1-52](#)

TPERIOD register, [3-86](#), [4-93](#)

-traditional (traditional compilation) compiler switch, [1-20](#)

trans.h header file, [3-9](#), [4-10](#)

transferring

- function arguments and return value, [1-147](#)
- function parameters to assembly routines, [1-147](#)

transpm (real matrix transpose) function, [3-90](#), [4-95](#)

-trdforinit (traditional initialization) C++ mode compiler switch, [1-52](#)

trigonometric functions (see math functions)

true (see Boolean type support keywords (bool, true, false))

TST\_FLAG macro, [3-79](#), [4-88](#)

twidfft function, [4-96](#)

type sizes, data, [1-150](#)

-typename (support typename) C++ mode compiler switch, [1-52](#)

## U

-U (undefine macro) compiler switch, [1-26](#), [1-45](#)

uppercase (see isupper, toupper functions)

user registers, [1-138](#)

utility functions

getenv, [2-65](#)

system, [2-159](#)

utility header file, [2-27](#)

## V

-v (version and verbose) compiler switch, [1-45](#)

va\_arg (get next argument in list) function, [2-164](#)

va\_end (finish processing argument list) function, [2-166](#)

va\_start (initialize argument list) function, [2-167](#)

-val-global (add global names) compiler switch, [1-46](#)

var (variance) function, [3-91](#), [4-98](#)

variable length arrays, [1-81](#)

VCSE components, [1-46](#)

vecdot (vector dot product) function, [3-92](#), [4-99](#)

vecsadd (vector scalar addition) function, [3-93](#), [4-100](#)

vecsmult (vector scalar multiply) function, [3-94](#), [4-101](#)

vecssub (vector scalar subtraction) function, [3-95](#), [4-102](#)

vector functions, [3-10](#), [4-10](#)

vector header file, [2-27](#)

vector.h header file, [3-10](#), [4-10](#)

# INDEX

vecvadd (vector addition) function,  
3-96, 4-103  
vecvmlt (vector multiply) function,  
3-97, 4-104  
vecvsub (vector subtraction) function,  
3-98, 4-105  
-verbose (display command line)  
compiler switch, 1-46  
-version (display version) compiler  
switch, 1-46  
VisualDSP++ Kernel (VDK), 1-45  
voice-band compression and  
expansion, 4-4  
volatile C program constructs, 1-71  
volatile keyword, 1-20

## W

-w (disable all warnings) compiler  
switch, 1-48  
-W {...} number (override error  
message) compiler switch, 1-46  
warning messages, 1-85  
-Warn-protos (warn if incomplete  
prototype) compiler switch, 1-46  
-wchar (support wide char type) C++  
mode compiler switch, 1-52

-Wdriver-limit (maximum process  
errors) compiler switch, 1-47  
-Werror-limit (maximum compiler  
errors) compiler switch, 1-47  
white space character test (see isspace  
function)  
window generators, 3-10, 4-11  
window.h header file, 3-10  
-Wremarks (enable diagnostic  
warnings) compiler switch, 1-47  
WRITE operation, 2-17  
-write-files (enable driver I/O pipe)  
compiler switch, 1-48  
-write-opts (enable driver I/O pipe)  
compiler switch, 1-48  
writing macros, 1-121  
-Wterse (enable terse warnings)  
compiler switch, 1-47

## X

-xref (cross-reference list) compiler  
switch, 1-48

## Z

zero padding, 3-60, 4-84  
zero\_cross (count zero crossings)  
function, 3-99, 4-106