

# 1 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” on page 1-2 provides 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” on page 1-77 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 supplied by Analog Devices that are of value in signal processing applications. In addition to the standard C library, this release of the compiler

## C and C++ Run-Time Libraries Guide

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 that are supported in the current release of the run-time libraries. Chapter 2, “DSP Run-Time Library” describes a number of signal processing, matrix, and statistical functions that assist code development.

 For more information on the algorithms on which many of the C library’s math functions are based, see W. J. Cody 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 Abridged C++ library software documentation is located on the VisualDSP++ installation CD in the <install\_path>\Docs\Reference folder. Viewing or printing these files requires a browser, such as Internet Explorer 6.0 (or higher). You can copy these files from the installation CD onto another disk.

## 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 1-3
- “Linking Library Functions” on page 1-4
- “Library Attributes” on page 1-14
- “Working With Library Header Files” on page 1-19

- “Calling Library Functions from an ISR” on page 1-36
- “Using the Libraries in a Multi-Threaded Environment” on page 1-37
- “Using Compiler Built-In C Library Functions” on page 1-38
- “Abridged C++ Library Support” on page 1-40
- “Measuring Cycle Counts” on page 1-47
- “File I/O Support” on page 1-57

For information on the C library’s contents, see “C Run-Time Library Reference” on page 1-77. For information on the Abridged C++ library’s contents, see “Abridged C++ Library Support” on page 1-40.

### 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” on page 1-77.

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 1-19.

Function names are C/C++ function names. If you call a C/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

## C and C++ Run-Time Libraries Guide

more information on the naming conventions, see Chapter 1 of the *VisualDSP++ 5.0 Compiler Manual*, in the section “C/C++ and Assembly Interface.”

 You can use the archiver, `elfar`, described in the *VisualDSP++ 5.0 Linker and Utilities Manual*, 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

In general, several versions of the C/C++ run-time library are supplied in binary form; for example, variants are available for different SHARC architectures and are listed in Table 1-1, Table 1-2, Table 1-3, and Table 1-4. Some versions of these binary files are also built for running in a multi-threaded environment; these binaries have `mt` in their filename.

In addition to regular run-time libraries, VisualDSP++ 5.0 also has `libio*_lite.dlb` libraries which provide smaller versions of `LibIO` (the I/O run-time support library) with more limited functionality. These smaller `LibIO` libraries can be used by specifying the switch `-flags-link -MD__LIBIO_LITE` on the build command line.

## C/C++ Run-Time Library

Table 1-1 contains a list of the run-time libraries and start-up files that have been built for the ADSP-21020 and ADSP-2106x processors, and are installed in the subdirectory `21k\lib`.

Table 1-1. C/C++ Files and Libraries for ADSP-210xx Processors

Description	Library Name	Comments
C run-time library	libc.dlb libc020.dlb libcmt.dlb	ADSP-21020 processor only
C++ run-time library	libc++.dlb libc++mt.dlb	
C++ run-time library with exception handling support	libc++eh.dlb libc++ehmt.dlb	
Legacy library	libc++prt.dlb libc++prmt.dlb libc++prteh.dlb libc++prtehmt.dlb libeh.dlb libehmt.dlb	These libraries contain no functions and are only provided for the purpose of linking against a legacy .ldf file
DSP run-time library	libdsp.dlb libdsp020.dlb	ADSP-21020 processor only
I/O run-time library	libio.dlb libio020.dlb libiomt.dlb	ADSP-21020 processor only
I/O run-time library with no support for alternative device drivers or <code>printf("%a")</code>	libio_lite.dlb libio020_lite.dlb libio_litemt.dlb  libio32.dlb libio64.dlb	ADSP-21020 processor only  Legacy library Legacy library
C start-up file — calls set-up routines and <code>main()</code>	020_hdr.doj 060_hdr.doj  061_hdr.doj 065L_hdr.doj	ADSP-21020 processor only ADSP-21060/062 processors only ADSP-21061 processor only ADSP-21065L processor only

## C and C++ Run-Time Libraries Guide

Table 1-1. C/C++ Files and Libraries for ADSP-210xx Processors (Cont'd)

Description	Library Name	Comments
C start-up file with EZ-kit — calls set-up routines and main()	061_hdr_ezkit.doj 065L_hdr_ezkit.doj	ADSP-21061 processor only ADSP-21065L processor only
C++ start-up file — calls set-up routines and main()	060_cpp_hdr.doj 061_cpp_hdr.doj 065L_cpp_hdr.doj  060_cpp_hdr_mt.doj 061_cpp_hdr_mt.doj 065L_cpp_hdr_mt.doj	ADSP-21060/062 processors only ADSP-21061 processor only ADSP-21065L processor only  ADSP-21060/062 processors only ADSP-21061 processor only ADSP-21065L processor only
C++ start-up file with EZ-kit — calls set-up routines and main()	061_cpp_hdr_ezkit.doj 065L_cpp_hdr_ezkit.doj  061_cpp_hdr_ezkit_mt.doj 065L_cpp_hdr_ezkit_mt.doj	ADSP-21061 processor only ADSP-21065L processor only  ADSP-21061 processor only ADSP-21065L processor only

## C/C++ Run-Time Library

The binary files that have been built for ADSP-2116x processors are catalogued in Table 1-2.

Table 1-2. C/C++ Files and Libraries for ADSP-2116x Processors

Description	Library Name	Comments
C run-time library	libc160.dlb	ADSP-21160 processor only
	libc161.dlb	ADSP-21161 processor only
	libc160mt.dlb	ADSP-21160 processor only
	libc161mt.dlb	ADSP-21161 processor only
C++ run-time library	libcpp.dlb	
	libcppmt.dlb	
C++ run-time library with exception handling support	libcppenh.dlb	
	libcppenhmt.dlb	
Legacy library	libcppprt.dlb	These libraries contain no functions and are only provided for the purpose of linking against a legacy .ldf file
	libcppprmt.dlb	
	libcppprteh.dlb	
	libcppprtehmt.dlb	
	libeh.dlb	
	libehmt.dlb	
DSP run-time library	libdsp160.dlb	
I/O run-time library	libio.dlb	
	libiomt.dlb	
I/O run-time library with no support for alternative device drivers or printf(“%a”)	libio_lite.dlb	
	libio_litemt.dlb	
	libio32.dlb	Legacy library
	libio64.dlb	Legacy library
C start-up file — calls set-up routines and main()	160_hdr.doj	ADSP-21160 processor only
	161_hdr.doj	ADSP-21161 processor only
C start-up file with EZ-kit — calls set-up routines and main()	160_hdr_ezkit.doj	ADSP-21160 processor only

## C and C++ Run-Time Libraries Guide

Table 1-2. C/C++ Files and Libraries for ADSP-2116x Processors (Cont'd)

Description	Library Name	Comments
C++ start-up file — calls set-up routines and <code>main()</code>	160_cpp_hdr.doj	ADSP-21160 processor only
	161_cpp_hdr.doj	ADSP-21161 processor only
	160_cpp_hdr_mt.doj	ADSP-21160 processor only
	161_cpp_hdr_mt.doj	ADSP-21161 processor only
C++ start-up file with EZ-kit — calls set-up routines and <code>main()</code>	160_cpp_hdr_ezkit.doj	ADSP-21160 processor only
	160_cpp_hdr_ezkit_mt.doj	ADSP-21160 processor only



The run-time libraries and binary files for the ADSP-21160 processors in this table have been compiled with the `-workaround rframe` compiler switch, while those for the ADSP-21161 processors have been compiled with the `-workaround 21161-anomaly-45` switch. An additional set of libraries and binary files that also work around the shadow write FIFO anomaly that affect ADSP-2116x chips is installed in the subdirectory `211xx\lib\swfa`.

Table 1-3 contains a list and a brief description of the library files that have been built for the ADSP-212xx processors. These files are installed in the subdirectory `212xx\lib`.

Table 1-3. C/C++ Files and Libraries for ADSP-212xx Processors

Description	Library Name	Comments
C run-time library	libc26x.dlb	
	libc26xmt.dlb	
C++ run-time library	libcpp.dlb	
	libcppmt.dlb	
C++ run-time library with exception handling support	libcppenh.dlb	
	libcppenhmt.dlb	

## C/C++ Run-Time Library

Table 1-3. C/C++ Files and Libraries for ADSP-212xx Processors (Cont'd)

Description	Library Name	Comments
Legacy library	libcpprt.dlb libcpprtmt.dlb libcpprteh.dlb libcpprtehmt.dlb libeh.dlb libehmt.dlb	These libraries contain no functions and are only provided for the purpose of linking against a legacy .ldf file
DSP run-time library	libdsp26x.dlb	
I/O run-time library	libio.dlb libiomt.dlb	
I/O run-time library with no support for alternative device drivers or printf(“%a”)	libio_lite.dlb libio_litemt.dlb	
C start-up file — calls set-up routines and main()	261_hdr.doj 262_hdr.doj 266_hdr.doj 267_hdr.doj	ADSP-21261 processor only ADSP-21262 processor only ADSP-21266 processor only ADSP-21267 processor only
C++ start-up file — calls set-up routines and main()	261_cpp_hdr.doj 262_cpp_hdr.doj 266_cpp_hdr.doj 267_cpp_hdr.doj  261_cpp_hdr_mt.doj 262_cpp_hdr_mt.doj 266_cpp_hdr_mt.doj 267_cpp_hdr_mt.doj	ADSP-21261 processor only ADSP-21262 processor only ADSP-21266 processor only ADSP-21267 processor only  ADSP-21261 processor only ADSP-21262 processor only ADSP-21266 processor only ADSP-21267 processor only

The libraries located in `212xx\lib` are built without any workarounds enabled. There are directories within the `212xx\lib` directory named `2126x_rev_<revision>` that contain libraries built for that specific revision, for example, `2126x_rev_0.0`. A single revision library directory may support more than one specific silicon revision; as an example, `2126x_rev_0.0` supports revisions 0.0, 0.1 and 0.2 of ADSP-2126x processors.

## C and C++ Run-Time Libraries Guide

In addition, a library directory called `2126x_any` is supplied. Libraries in this directory will contain workarounds for all relevant anomalies on all revisions of ADSP-2126x processors.

The `-si-revision` switch can be used to specify a silicon revision—VisualDSP++ will use the appropriate libraries to build the application.

Table 1-4 describes the library files that have been built for the ADSP-213xx processors, and which are installed in the subdirectory `213xx\lib`.

Table 1-4. C/C++ Files and Libraries for ADSP-213xx Processors

Description	Library Name	Comments
C run-time library	libc36x.dlb libc36xmt.dlb libc37x.dlb libc37xmt.dlb	
C++ run-time library	libcpp.dlb libcppmt.dlb	
C++ run-time library with exception handling support	libcppenh.dlb libcppenhmt.dlb	
Legacy library	libcppprt.dlb libcppprmt.dlb libcppprteh.dlb libcppprtehmt.dlb libeh.dlb libehmt.dlb	These libraries contain no functions and are only provided for the purpose of linking against a legacy <code>.ldf</code> file.
DSP run-time library	libdsp36x.dlb libdsp37x.dlb	
I/O run-time library	libio.dlb libiomt.dlb	
I/O run-time library with no support for alternative device drivers or <code>printf("%a")</code>	libio_lite.dlb libio_litemt.dlb	

## C/C++ Run-Time Library

Table 1-4. C/C++ Files and Libraries for ADSP-213xx Processors (Cont'd)

Description	Library Name	Comments
C start-up file — calls set-up routines and <code>main()</code>	363_hdr.doj	ADSP-21363 processor only
	364_hdr.doj	ADSP-21364 processor only
	365_hdr.doj	ADSP-21365 processor only
	366_hdr.doj	ADSP-21366 processor only
	367_hdr.doj	ADSP-21367 processor only
	368_hdr.doj	ADSP-21368 processor only
	369_hdr.doj	ADSP-21369 processor only
	371_hdr.doj	ADSP-21371 processor only
C++ start-up file — calls set-up routines and <code>main()</code>	363_cpp_hdr.doj	ADSP-21363 processor only
	364_cpp_hdr.doj	ADSP-21364 processor only
	365_cpp_hdr.doj	ADSP-21365 processor only
	366_cpp_hdr.doj	ADSP-21366 processor only
	367_cpp_hdr.doj	ADSP-21367 processor only
	368_cpp_hdr.doj	ADSP-21368 processor only
	369_cpp_hdr.doj	ADSP-21369 processor only
	371_cpp_hdr.doj	ADSP-21371 processor only
	375_cpp_hdr.doj	ADSP-21375 processor only
	363_cpp_hdr_mt.doj	ADSP-21363 processor only
	364_cpp_hdr_mt.doj	ADSP-21364 processor only
	365_cpp_hdr_mt.doj	ADSP-21365 processor only
	366_cpp_hdr_mt.doj	ADSP-21366 processor only
	367_cpp_hdr_mt.doj	ADSP-21367 processor only
	368_cpp_hdr_mt.doj	ADSP-21368 processor only
	369_cpp_hdr_mt.doj	ADSP-21369 processor only
	371_cpp_hdr_mt.doj	ADSP-21371 processor only
	375_cpp_hdr_mt.doj	ADSP-21375 processor only

Table 1-5 contains a list and a brief description of the library files that have been built for the ADSP-214xx processors. These files are installed in the subdirectory `214xx\lib`. The libraries are built in short-word mode by default, though there are versions which have been built in normal-word mode; these binaries have `nwc` in their filename.

## C and C++ Run-Time Libraries Guide

Table 1-5. C/C++ Files and Libraries for ADSP-214xx Processors

Description	Library Name	Comments
C run-time library	libc.dlb libcmt.dlb libc_nwc.dlb libcmt_nwc.dlb	
C++ run-time library	libcpp.dlb libcppmt.dlb libcpp_nwc.dlb libcppmt_nwc.dlb	
C++ run-time library with exception handling support	libcppeh.dlb libcppehmt.dlb libcppeh_nwc.dlb libcppehmt_nwc.dlb	
DSP run-time library	libdsp.dlb libdsp_nwc.dlb	
I/O run-time library	libio.dlb libiomt.dlb libio_nwc.dlb libiomt_nwc.dlb	
I/O run-time library with no support for alternative device drivers or printf(“%a”)	libio_lite.dlb libio_lite.dlb libio_lite_nwc.dlb libio_lite_nwc.dlb	

## C/C++ Run-Time Library

Table 1-5. C/C++ Files and Libraries for ADSP-214xx Processors (Cont'd)

Description	Library Name	Comments
C start-up file — calls set-up routines and <code>main()</code>	21462_hdr.doj	ADSP-21462 processor only
	21465_hdr.doj	ADSP-21465 processor only
	21467_hdr.doj	ADSP-21467 processor only
	21469_hdr.doj	ADSP-21469 processor only
	21479_hdr.doj	ADSP-21479 processor only
	21489_hdr.doj	ADSP-21489 processor only
C++ start-up file — calls set-up routines and <code>main()</code>	21462_cpp_hdr.doj	ADSP-21462 processor only
	21462_cpp_hdr_mt.doj	ADSP-21462 processor only
	21465_cpp_hdr.doj	ADSP-21465 processor only
	21465_cpp_hdr_mt.doj	ADSP-21465 processor only
	21467_cpp_hdr.doj	ADSP-21467 processor only
	21467_cpp_hdr_mt.doj	ADSP-21467 processor only
	21469_cpp_hdr.doj	ADSP-21469 processor only
	21469_cpp_hdr_mt.doj	ADSP-21469 processor only
	21479_cpp_hdr.doj	ADSP-21479 processor only
	21479_cpp_hdr_mt.doj	ADSP-21479 processor only
	21489_cpp_hdr.doj	ADSP-21489 processor only
	21489_cpp_hdr_mt.doj	ADSP-21489 processor only

The libraries located in `214xx\lib` are built without any workarounds enabled. In addition, a library directory called `21469_rev_any` is supplied. Libraries in this directory contain workarounds for all relevant anomalies on all revisions of ADSP-214xx processors.

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` file, then either add the appropriate library/libraries to the `.ldf` file used for your project, or use the compiler's `-l` switch to specify the library to be added to the link line. For example, the switches `-lc -ldsp 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++ 5.0 Linker and Utilities Manual*.

## C and C++ Run-Time Libraries Guide

### Library Attributes

The run-time libraries make use of file attributes. (See Chapter 1 of the VisualDSP++ 5.0 Compiler Manual for more details on how to use file attributes.) Each library function has a defined set of file attributes that are listed in Table 1-6. For each object `obj` in the run-time libraries the following is true:

Table 1-6. Run-time Library Object Attributes

Attribute name	Meaning of attribute and value
<code>libGroup</code>	A potentially multi-valued attribute. Each value is the name of a header file that either defines <code>obj</code> , or that defines a function that calls <code>obj</code> .
<code>libName</code>	The name of the library that contains <code>obj</code> , without the processor identifier. For example, suppose that <code>obj</code> were part of <code>libdsp160.dlb</code> , then the value of the attribute would be <code>libdsp</code> .
<code>libFunc</code>	The name of all the functions in <code>obj</code> . <code>libFunc</code> will have multiple values -both the C, and assembly linkage names will be listed. <code>libFunc</code> will also contain all the published C and assembly linkage names of objects in <code>obj</code> 's library that call into <code>obj</code> .
<code>prefersMem</code>	One of three values: <code>internal</code> , <code>external</code> or <code>any</code> . If <code>obj</code> contains a function that is likely to be application performance critical, it will be marked as <code>internal</code> . Most DSP run-time library functions fit into the <code>internal</code> category. If a function is deemed unlikely to be essential for achieving the necessary performance it will be marked as <code>external</code> (all the I/O library functions fall into this category). The default <code>.ldf</code> files use this attribute to place code and data optimally.

Table 1-6. Run-time Library Object Attributes (Cont'd)

Attribute name	Meaning of attribute and value
prefersMemNum	Analogous to prefersMem but takes a numeric string value. The attribute can be used in .ldf files to provide a greater measure of control over the placement of binary object files than is available using the prefersMem attribute. The values "30", "50", and "70" correspond to the prefersMem values external, any, and internal respectively. The default .ldf files use the prefersMem attribute in preference to the prefersMemNum attribute to specify the optimum placement of files.
FuncName	Multi-valued attribute whose values are all the assembler linkage names of the defined names in obj.

If an object in the run-time library calls into another object in the same library, whether it is internal or publicly visible, the called object will inherit extra libGroup and libFunc values from the caller.

The following example demonstrates how attributes would look in a small example library libfunc.dlb that comprises three objects: func1.doj, func2.doj and subfunc.doj. These objects are built from the following source modules:

**File:** func1.h

```
void func1(void);
```

**File:** func2.h

```
void func2(void);func1.c
```

```
#include "func1.h"
void func1(void) {
    /* Compiles to func1.doj */
    subfunc();
}
```

## C and C++ Run-Time Libraries Guide

**File:** func2.c

```
#include "func2.h"
void func2(void) {
    /* Compiles to func2.doj */
    subfunc();
}
```

**File:** subfunc.c

```
void subfunc(void) {
    /* Compiles to subfunc.doj */
}
```

The objects in `libfunc.dlb` have the attributes as defined in Table 1-7:

Table 1-7. Attribute Values in `libfunc.dlb`

Attribute	Value
func1.doj	
libGroup	func1.h
libName	libfunc
libFunc	_func1
libFunc	func1
FuncName	_func1
prefersMem	any <sup>(1)</sup>
prefersMemNum	50
func2.doj	
libGroup	func2.h
libName	libfunc
libFunc	_func2
libFunc	<b>func2</b>
FuncName	_func2
prefersMem	internal <sup>(2)</sup>
prefersMemNum	30

Table 1-7. Attribute Values in libfunc.dlb (Cont'd)

Attribute	Value
subfunc.doj	
libGroup	func1.h
libGroup	func2.h <sup>(3)</sup>
libName	libfunc
libFunc	_func1
libFunc	func1
libFunc	_func2
libFunc	func2
libFunc	_subfunc
libFunc	subfunc
FuncName	_subfunc
prefersMem	internal <sup>(4)</sup>
prefersMemNum	30

- 1 func1.doj will not be performance critical, based on its normal usage.
- 2 func2.doj will be performance critical in many applications, based on its normal usage.
- 3 libGroup contains the union of the libGroup attributes of the two calling objects.
- 4 prefersMem contains the highest priority of all the calling objects.

## Exceptions to the Attribute Conventions

The library attribute convention has the following exceptions: The C++ support libraries (libcpp\*.dlb) all contain functions that have C++ linkage. Functions written in C++ have their functions names encoded (often referred to as name mangling) to allow for the overloading of parameter types. The function name encoding includes all the parameter types, the return type and the namespace within which the function is declared. Whenever a function's name is encoded, the encoded name is used as the value for the libFunc attribute.

## C and C++ Run-Time Libraries Guide

Table 1-8 lists additional `libGroup` attribute values:

Table 1-8. Additional `libGroup` Attribute Values

Value	Meaning
<code>exceptions_support</code>	Compiler support routines for C++ exceptions.
<code>floating_point_support</code>	Compiler support routines for floating point arithmetic.
<code>integer_support</code>	Compiler support routines for integer arithmetic.
<code>runtime_support</code>	Other run-time functions that do not fit into any of the above categories.
<code>startup</code>	One-time initialization functions called prior to the invocation of <code>main</code> .

Objects with any of the `libGroup` attribute values listed in Table 1-8 will not contain any `libGroup` or `libFunc` attributes from any calling objects.

Table 1-9 presents a summary of the default memory placement using `prefersMem`.

Table 1-9. Default Memory Placement Summary

Library	Placement
<code>libcpp*.dlb</code>	<code>any</code>
<code>idle*.doj</code> <code>libio*.dlb</code>	<code>external</code>
<code>libdsp*.dlb</code>	<code>internal</code> except for the windowing functions and functions which generate a twiddle table which are <code>external</code>
<code>libc*.dlb</code>	<code>any</code> except for the <code>stdio.h</code> functions, which are <code>external</code> , and <code>qsort</code> , which is <code>internal</code>

Most of the functions contained within the DSP run-time library (`libdsp*.dlb`) have `prefersMem=internal`, because it is likely that any function called in this run-time library will make up a significant part of an application's cycle count.

## Mapping Objects to FLASH Memory Using Attributes

When using the Memory Initializer to initialize code and data areas from flash memory, code and data used during the process of initialization must be mapped to flash memory to ensure it is available during boot-up. The `requiredForROMBoot` attribute is specified on library objects that contain such code and data and can be used in the `.ldf` file to perform the required mapping. See the *VisualDSP++ 5.0 Linker and Utilities Manual* for further information on memory initialization.

## Working With Library Header Files

When you use a library function in your program, you should also include the function's header file with the `#include` preprocessor command. The header file for each function is identified in the **Synopsis** section of the function's reference page. Header files contain function prototypes. The compiler uses these prototypes to check that each function is called with the correct arguments.

A list of the header files that are supplied with this release of the `cc21k` compiler appears in Table 1-10. You should use a C standard text to augment the information supplied in this chapter.

Table 1-10. Standard C Run-Time Library Header Files

Header	Purpose	Standard
<code>adi_types.h</code>	Type definitions	Analog extension
<code>assert.h</code>	Diagnostics	ANSI
<code>ctype.h</code>	Character Handling	ANSI
<code>cycle_count.h</code>	Basic Cycle Counting	Analog extension
<code>cycles.h</code>	Cycle Counting with Statistics	Analog extension
<code>device.h</code>	Macros and data structures for alternative device drivers	Analog extension
<code>device_int.h</code>	Enumerations and prototypes for alternative device drivers	Analog extension

## C and C++ Run-Time Libraries Guide

Table 1-10. Standard C Run-Time Library Header Files (Cont'd)

Header	Purpose	Standard
<code>errno.h</code>	Error Handling	ANSI
<code>float.h</code>	Floating Point	ANSI
<code>iso646.h</code>	Boolean Operators	ANSI
<code>limits.h</code>	Limits	ANSI
<code>locale.h</code>	Localization	ANSI
<code>math.h</code>	Mathematics	ANSI
<code>setjmp.h</code>	Non-Local Jumps	ANSI
<code>signal.h</code>	Signal Handling	ANSI
<code>stdarg.h</code>	Variable Arguments	ANSI
<code>stdbool.h</code>	Boolean macros	ANSI
<code>stddef.h</code>	Standard Definitions	ANSI
<code>stdint.h</code>	Exact width integer types	ANSI
<code>stdio.h</code>	Input/Output	ANSI
<code>stdlib.h</code>	Standard Library	ANSI
<code>string.h</code>	String Handling	ANSI
<code>time.h</code>	Date and Time	ANSI

The following sections provide descriptions of the header files contained in the C library. The header files are listed in alphabetical order.

### **`adi_types.h`**

The `adi_types.h` header file contains the type definitions for `char_t`, `float32_t`, `float64_t`, and also includes both `stdint.h` and `stdbool.h`.

### assert.h

The `assert.h` header file defines the `assert` macro, which can be used to insert run-time diagnostics into a source file. The macro normally tests (asserts) that an expression is true. If the expression is false, then the macro will first print an error message, and will then call the `abort` function to terminate the application. The message displayed by the `assert` macro will be of the form:

```
ASSERT [expression] fails at "filename": linenumber
```

Note that the message includes the following information:

- `filename` - the name of the source file
- `linenumber` - the current line number in the source file
- `expression` - the expression tested

However if the macro `NDEBUG` is defined at the point at which the `assert.h` header file is included in the source file, then the `assert` macro will be defined as a null macro and no run-time diagnostics will be generated.

The strings associated with `assert.h` can be assigned to slower, more plentiful memory (and therefore free up faster memory) by placing a `default_section` pragma above the sections of code containing the asserts. For example:

```
#pragma default_section(STRINGS,"seg_sram")
```

Note that the pragma will affect the placement of all strings, and not just the ones associated with using the `ASSERT` macro. See the section “Linking Control Pragmas” in Chapter 1 of the *VisualDSP++ 5.0 Compiler Manual* for more information about using the pragma.

## C and C++ Run-Time Libraries Guide

An alternative to using the `default_section` pragma is to use the compiler's `-section` switch (for example `-section strings=seg_sram`). You can accomplish this in one of two ways:

- Use the command line.
- Use the **VisualDSP++ Project Options** dialog box. In the **Compile** category, select the **General** tab. Then type the command in the **Additional options:** field.

### **ctype.h**

The `ctype.h` header file contains functions for character handling, such as `isalpha`, `tolower`, etc.

For a list of library functions that use this header, see Table 1-19 on page 1-73.

### **cycle\_count.h**

The `cycle_count.h` header file provides an inexpensive method for benchmarking C-written source by defining basic facilities for measuring cycle counts. The facilities provided are based upon two macros, and a data type which are described in more detail in the section “Measuring Cycle Counts” on page 1-47.

### **cycles.h**

The `cycles.h` header file defines a set of five macros and an associated data type that may be used to measure the cycle counts used by a section of C-written source. The macros can record how many times a particular piece of code has been executed and also the minimum, average, and maximum number of cycles used. The facilities that are available via this header file are described in the section “Measuring Cycle Counts” on page 1-47.

### **device.h**

The `device.h` header file provides macros and defines data structures that an alternative device driver would require to provide file input and output services for `stdio` library functions. Normally, the `stdio` functions use a default driver to access an underlying device, but alternative device drivers may be registered that may then be used transparently by these functions. This mechanism is described in “Extending I/O Support To New Devices” on page 1-58.

### **device\_int.h**

The `device_int.h` header file contains function prototypes and provides enumerations for alternative device drivers. An alternative device driver is normally provided by an application and may be used by the `stdio` library functions to access an underlying device; an alternative device driver may coexist with, or may replace, the default driver that is supported by the VisualDSP++ simulator and EZ-KIT Lite evaluation systems. Refer to “Extending I/O Support To New Devices” on page 1-58.

### **errno.h**

The `errno.h` header file provides access to `errno` and also defines macros for associated error codes. This facility is not, in general, supported by the rest of the library.

## C and C++ Run-Time Libraries Guide

### float.h

The `float.h` header file defines the properties of the floating-point data types that are implemented by the compiler—that is, `float`, `double`, and `long double`. These properties are defined as macros and include the following for each supported data type:

- the maximum and minimum value (for example, `FLT_MAX` and `FLT_MIN`)
- the maximum and minimum power of ten (for example, `FLT_MAX_10_EXP` and `FLT_MIN_10_EXP`)
- the precision available expressed in terms of decimal digits (for example, `FLT_DIG`)
- a constant that represents the smallest value that may be added to 1.0 and still result in a change of value (for example, `FLT_EPSILON`)


Note that the set of macros that define the properties of the `double` data type will have the same values as the corresponding set of macros for the `float` type when `doubles` are defined to be 32 bits wide, and they will have the same value as the macros for the `long double` data type when `doubles` are defined to be 64 bits wide (use the `-double-size[-32|-64]` compiler switch).

## iso646.h

The `iso646.h` header file defines symbolic names for certain C operators; the symbolic names and their associated value are shown in Table 1-11.

Table 1-11. Symbolic Names Defined in `iso646.h`

Symbolic Name	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>not</code>	<code>!</code>
<code>not_eq</code>	<code>!=</code>
<code>or</code>	<code>  </code>
<code>or_eq</code>	<code> =</code>
<code>xor</code>	<code>^</code>
<code>xor_eq</code>	<code>^=</code>

 The symbolic names have the same name as the C++ keywords that are accepted by the compiler when the `-alttok` switch is specified.

## 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.

For a list of library functions that use this header, see Table 1-20 on page 1-73.

## C and C++ Run-Time Libraries Guide

### 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 the data types `float` and `long double`.

For a list of library functions that use this header, see Table 1-21 on page 1-73.

For every function that is defined to return a `double`, the `math.h` header file also defines corresponding functions that return a `float` and a `long double`. The names of the `float` functions are the same as the equivalent `double` function with `f` appended to its name. Similarly, the names of the `long double` functions are the same as the `double` function with `d` appended to its name.

For example, the header file contains the following prototypes for the `sine` function:

```
float sinf (float x);
double sin (double x);
long double sind (long double x);
```

When the compiler is treating `double` as 32 bits, the header file arranges that all references to the `double` functions are directed to the equivalent `float` function (with the suffix `f`). This allows you to use the un-suffixed names with arguments of type `double`, regardless of whether `doubles` are 32 or 64 bits long.

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 2, “DSP Run-Time Library” for more information about these additional functions.

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” on page 1-77 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.

For a list of library functions that use this header, see Table 1-22 on page 1-74.

### **signal.h**

The `signal.h` header file provides function prototypes for the standard ANSI `signal.h` routines and also for several 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.

For a list of library functions that use this header, see Table 1-23 on page 1-74.

## C and C++ Run-Time Libraries Guide

### stdarg.h

The `stdarg.h` header file contains definitions needed for functions that accept a variable number of arguments. Programs that call such functions must include a prototype for the functions referenced.

For a list of library functions that use this header, see Table 1-24 on page 1-74.

### stdbool.h

The `stdbool.h` header file contains three boolean related macros (`true`, `false` and `__bool_true_false_are_defined`) and an associated data type (`bool`). This header file was introduced in the C99 standard library.

### stddef.h

The `stddef.h` header file contains a few common definitions useful for portable programs, such as `size_t`.

### stdint.h

The `stdint.h` header file contains various exact-width integer types along with associated minimum and maximum values. The `stdint.h` header file was introduced in the C99 standard library.

Table 1-12 describes each type with regard to MIN and MAX macros.

Table 1-12. Exact-Width Integer Types

Type	Common Equivalent	MIN	MAX
<code>int32t</code>	<code>int</code>	<code>INT32_MIN</code>	<code>INT32_MAX</code>
<code>uint32t</code>	<code>unsigned int</code>	<code>0</code>	<code>UINT32_MAX</code>
<code>int_least8_t</code>	<code>int</code>	<code>INT_LEAST8_MIN</code>	<code>INT_LEAST8_MAX</code>
<code>int_least16_t</code>	<code>int</code>	<code>INT_LEAST16_MIN</code>	<code>INT_LEAST16_MAX</code>

Table 1-12. Exact-Width Integer Types (Cont'd)

Type	Common Equivalent	MIN	MAX
int_least32_t	int	INT_LEAST32_MIN	INT_LEAST32_MAX
uint_least8_t	unsigned int	0	UINT_LEAST8_MAX
uint_least16_t	unsigned int	0	UINT_LEAST16_MAX
uint_least32_t	unsigned int	0	UINT_LEAST32_MAX
int_fast8_t	int	INT_FAST8_MIN	INT_FAST8_MAX
int_fast16_t	int	INT_FAST16_MIN	INT_FAST16_MAX
int_fast32_t	int	INT_FAST32_MIN	INT_FAST32_MAX
uint_fast8_t	unsigned int	0	UINT_FAST8_MAX
uint_fast16_t	unsigned int	0	UINT_FAST16_MAX
uint_fast32_t	unsigned int	0	UINT_FAST32_MAX
intmax_t	int	INTMAX_MIN	INTMAX_MAX
intptr_t	int	INTPTR_MIN	INTPTR_MAX
uintmax_t	unsigned int	0	UINTMAX_MAX
uintptr_t	unsigned int	0	UINTPTR_MAX

Table 1-13 describes MIN and MAX macros defined for typedefs in other headings.

Table 1-13. MIN and MAX Macros for typedefs in Other Headings

Type	MIN	MAX
ptrdiff_t	PTRDIFF_MIN	PTRDIFF_MAX
sig_atomic_t	SIG_ATOMIC_MIN	SIG_ATOMIC_MAX
size_t	0	SIZE_MAX
wchar_t	WCHAR_MIN	WCHAR_MAX
wint_t	WINT_MIN	WINT_MAX

## C and C++ Run-Time Libraries Guide

Macros for minimum-width integer constants include: `INT8_C(x)`, `INT16_C(x)`, `INT32_C(x)`, `UINT8_C(x)`, `UINT16_C(x)`, and `UINT32_C(x)`.

Macros for greatest-width integer constants include `INTMAX_C(x)` and `UINTMAX_C(x)`.

### **stdio.h**

The `stdio.h` header file defines a set of functions, macros, and data types for performing input and output. Applications that use the facilities of this header file should link with the I/O library `libio.dlb` in the same way as linking with the C run-time library (see “Linking Library Functions” on page 1-4). The library is thread-safe but it is not interrupt-safe and should not therefore be called either directly or indirectly from an interrupt service routine.

The compiler uses definitions within the header file to select an appropriate set of functions that correspond to the currently selected size of type `double` (either 32 bits or 64 bits). Any source file that uses the facilities of `stdio.h` must therefore include the header file. 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 implementation of the `stdio.h` routines is based on a simple interface with a device driver that provides a set of low-level primitives for `open`, `close`, `read`, `write`, and `seek` operations. By default, these operations are provided by the VisualDSP++ simulator and EZ-KIT Lite systems and this mechanism is outlined in the section “Default Device Driver Interface” on page 1-67.

Alternative device drivers may be registered that can then be used transparently through the `stdio.h` functions. See “Extending I/O Support To New Devices” on page 1-58 for a description of the feature. Applications that do not require this functionality may be built with the `-flags-link -MD__LIBIO_LITE` switch. The switch links the application with a version of the I/O library that does not support the ability to register alternative

device drivers, does not support the `%a` conversion specifier in `printf`, and does not support the `hh`, `j`, `t`, or `z` size qualifiers in `scanf`. Linking with this switch results in a smaller executable.



When creating applications, be aware that the default device driver is activated when:

- A file is opened or closed.
- An input buffer becomes empty.
- An output buffer becomes full or is flushed.
- Interrogating or re-positioning a file pointer
- Deleting a file through the remove library function
- Renaming a file through the rename library function

Under the above conditions, the default device driver will disable interrupts and will halt the DSP while it negotiates with the host to perform the required I/O operation. Once the I/O operation has completed, the default device driver will restart the DSP and re-enable interrupts.

While the DSP is stopped, the cycle count registers are not updated and the DSP itself cannot initiate any interrupts; however, signals that correspond to external events can still occur, and these may be activated once the default device driver re-enables interrupts.

The following restrictions apply to this software release:


- The functions `tmpfile` and `tmpnam` are not available
- The functions `rename` and `remove` are only supported under the default device driver supplied by the VisualDSP++ simulator and EZ-KIT Lite systems, and they only operate on the host file system

## C and C++ Run-Time Libraries Guide

- Positioning within a file that has been opened as a text stream is only supported if the lines within the file are terminated by the character sequence `\r\n`
- Support for formatted reading and writing of data of `long double` type is only supported if an application is built with the `-double-size-64` switch

At program termination, the host environment closes down any physical connection between the application and an opened file. However, the I/O library does not implicitly close any opened streams to avoid unnecessary overheads (particularly with respect to memory occupancy). Thus, unless explicit action is taken by an application, any unflushed output may be lost.

Any output generated by `printf` is always flushed but output generated by other library functions, such as `putchar`, `fwrite`, and `fprintf`, is not automatically flushed. Applications should therefore arrange to close down any streams that they open. Note that the function reference `fflush(NULL)`; flushes the buffers of all opened streams.

-  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 resides at a memory location that is less than `0x40000000`.

For a list of library functions that use this header, see Table 1-25 on page 1-74.

### **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 Devices, such as `avg`, `max`, and `clip`. Table 1-14 is a summary of the additional library functions defined by the `stdlib.h` header file.


 Some functions exist as both integer and floating point functions. The floating point functions typically have an `f` prefix. Make sure you use the correct type.

## C and C++ Run-Time Libraries Guide

Table 1-14. Standard Library - Additional Functions

Description	Prototype
Average	<code>int avg (int a, int b);</code> <code>long lavg (long a, long b);</code>
Clip	<code>int clip (int a, int b);</code> <code>long lclip (long a, long b);</code>
Count bits set	<code>int count_ones (int a);</code> <code>int lcount_ones (long a);</code>
Maximum	<code>int max (int a, int b);</code> <code>long lmax (long a, long b);</code>
Minimum	<code>int min (int a, int b);</code> <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> <code>void heap_free(int heap_index, void *ptr);</code> <code>void *heap_malloc(int heap_index, size_t size);</code> <code>void *heap_realloc(int heap_index, void *ptr, size_t size);</code> <code>int set_alloc_type(char * heap_name);</code> <code>int heap_install(void *base, size_t size, int userid, int pmdm);</code> <code>int heap_lookup_name(char *userid);</code> <code>int heap_switch(int heapid);</code>

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.

For a list of library functions that use this header, see Table 1-26 on page 1-75.

### **string.h**

The `string.h` header file contains string handling functions, including `strcpy` and `memcpy`.

For a list of library functions that use this header, see Table 1-27 on page 1-76.

## time.h

The `time.h` header file provides functions, data types, and a macro for expressing and manipulating date and time information. The header file defines two fundamental data types, one of which is `clock_t` and is associated with the number of implementation-dependent processor “ticks” used since an arbitrary starting point; and the other which is `time_t`.

The `time_t` data type is used for values that represent the number of seconds that have elapsed since a known epoch; values of this form are known as a *calendar time*. In this implementation, the epoch starts on 1st January, 1970, and calendar times before this date are represented as negative values.

A calendar time may also be represented in a more versatile way as a broken-down time which is a structured variable of the following form:


```
struct tm { int tm_sec; /* seconds after the minute [0,61] */
            int tm_min; /* minutes after the hour [0,59] */
            int tm_hour; /* hours after midnight [0,23] */
            int tm_mday; /* day of the month [1,31] */
            int tm_mon; /* months since January [0,11] */
            int tm_year; /* years since 1900 */
            int tm_wday; /* days since Sunday [0, 6] */
            int tm_yday; /* days since January 1st [0,365] */
            int tm_isdst; /* Daylight Saving flag */
};
```



This implementation does not support either the Daylight Saving flag in the structure `struct tm`; nor does it support the concept of time zones. All calendar times are therefore assumed to relate to Greenwich Mean Time (Coordinated Universal Time or UTC).

## C and C++ Run-Time Libraries Guide

The header file sets the `CLOCKS_PER_SEC` macro to the number of processor cycles per second and this macro can therefore be used to convert data of type `clock_t` into seconds, normally by using floating-point arithmetic to divide it into the result returned by the `clock` function.

 In general, the processor speed is a property of a particular chip and it is therefore recommended that the value to which this macro is set is verified independently before it is used by an application.

In this version of the C/C++ compiler, the `CLOCKS_PER_SEC` macro is set by one of the following (in descending order of precedence):

- Via the `-DCLOCKS_PER_SEC=<definition>` compile-time switch
- Via the **Processor speed** box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Processor** category
- From the header file `cycles.h`

For a list of library functions that use this header, see Table 1-28 on page 1-76.

## Calling Library Functions from an ISR

Not all C run-time library functions are interrupt-safe (and can therefore be called from an interrupt service routine). For a run-time function to be classified as *interrupt-safe*, it must:

- Not update any global data, such as `errno`, and
- Not write to (or maintain) any private static data

It is recommended therefore that none of the functions defined in the header file `math.h`, nor the string conversion functions defined in `stdlib.h`, be called from an ISR as these functions are commonly defined to update the global variable `errno`. Similarly, the functions defined in the `stdio.h` header file maintain static tables for currently opened streams and

should not be called from an ISR. Additionally, the memory allocation routines `malloc`, `calloc`, `realloc`, `free`, and the C++ operators `new` and `delete` read and update global tables and are not interrupt-safe.

Several other library functions are not interrupt-safe because they make use of private static data. These functions are:

```
asctime  
gmtime  
localtime  
rand  
srand  
strtok
```

While not all C run-time library functions are interrupt-safe, versions of the functions are available that are *thread-safe* and may be used in a VDK multi-threaded environment. These library functions can be found in the run-time libraries that have the suffix `_mt` in their filename.

### Using the Libraries in a Multi-Threaded Environment

It is sometimes desirable for there to be several instances of a given library function to be active at any one time. Two examples of such a requirement are:

- An interrupt or other external event invokes a function, while the application is also executing that function,
- An application that runs in a multi-threaded environment, such as VDK, and more than one thread executes the function concurrently.

## C and C++ Run-Time Libraries Guide

The majority of the functions in the C and C++ run-time libraries are safe in this regard and may be called in either of the above schemes; this is because the functions operate on parameters passed in by the caller and they do not maintain private static storage, and they do not access non-constant global data.

A subset of the library functions however either make use of private storage or they operate on shared resources (such as `FILE` pointers). This can lead to undefined behavior if two instances of a function simultaneously access the same data. The issues associated with calling such library functions via an interrupt or other external event is discussed in the section “Calling Library Functions from an ISR” on page 1-36.

A VisualDSP++ installation contains versions of the C and C++ libraries that may be used in a multi-threaded environment. These libraries have recursive locking mechanisms so that shared resources, such as `stdio` `FILE` tables and buffers, are only updated by a single function instance at any given time. The libraries also make use of local-storage routines for thread-local private copies of data, and for the variable `errno` (each thread therefore has its own copy of `errno`).

The multi-threaded libraries have “`mt`” in their filename and will be used automatically by the default VDK `.ldf` file to build a multi-threaded application.

Note that the DSP run-time library (which is described in Chapter 2, “DSP Run-Time Library”, is thread-safe and may be used in any multi-threaded environment.

## Using Compiler Built-In C Library Functions

The C compiler intrinsic (built-in) functions are functions that the compiler immediately recognizes and replaces with inline 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 inline 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, inline built-in functions are faster than an average library routine, and it does not incur the calling overhead. The routines in Table 1-15 are built-in C library functions for the `cc21k` compiler:

Table 1-15. Compiler Built-in Functions

<code>abs</code>	<code>avg</code>	<code>clip</code>
<code>copysign<sup>1</sup></code>	<code>copysignf</code>	<code>fabs<sup>1</sup></code>
<code>fabsf</code>	<code>favg<sup>1</sup></code>	<code>favgf</code>
<code>fclip<sup>1</sup></code>	<code>fclipf</code>	<code>fmax<sup>1</sup></code>
<code>fmaxf</code>	<code>fmin<sup>1</sup></code>	<code>fminf</code>
<code>labs</code>	<code>lavg</code>	<code>lclip</code>
<code>lmax</code>	<code>lmin</code>	<code>max</code>
<code>min</code>		

<sup>1</sup> These functions are only compiled as a built-in function if `double` is the same size as `float`.

If you want to use the C run-time library functions of the same name, compile with the `-no-builtin` compiler switch.

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

## C and C++ Run-Time Libraries Guide

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 1-16 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 1-16. Dual Memory Capable Functions

<code>atof</code>	<code>atoi</code>	<code>atol</code>	<code>frexp</code>
<code>frexpf</code>	<code>memchr</code>	<code>memcmp</code>	<code>memcpy</code>
<code>memmove</code>	<code>memset</code>	<code>modf</code>	<code>modff</code>
<code>setlocale</code>	<code>strcat</code>	<code>strchr</code>	<code>strcmp</code>
<code>strcoll</code>	<code>strcpy</code>	<code>strcspn</code>	<code>strlen</code>
<code>strncat</code>	<code>strncmp</code>	<code>strncpy</code>	<code>strpbrk</code>
<code>strrchr</code>	<code>strspn</code>	<code>strstr</code>	<code>strtod</code>
<code>strtok</code>	<code>strtol</code>	<code>strtoul</code>	<code>strxfrm</code>

## 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 processors.

The Abridged C++ 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. You can view the Abridged Library Reference by locating the file `docs\cpl_lib\index.html` underneath your VisualDSP++ installation and opening it in a web browser.

This section lists and briefly describes the following components of the Abridged C++ library:

- “Embedded C++ Library Header Files” on page 1-41
- “C++ Header Files for C Library Facilities” on page 1-44
- “Embedded Standard Template Library Header Files” on page 1-45
- “Using Thread-Safe C/C++ Run-Time Libraries with VDK” on page 1-47

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.

## C and C++ Run-Time Libraries Guide

### **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 in Chapter 1 of the *VisualDSP++ 5.0 Compiler Manual*, section “C++ Fractional Type Support”.

### **fstream**

The `fstream` header file defines the `filebuf`, `ifstream`, and `ofstream` classes for external file manipulations.

### **iomanip**

The `iomanip` header file declares several `iostream` manipulators. Each manipulator accepts a single argument.

### **ios**

The `ios` header file defines several classes and functions for basic `iostream` manipulations. Note that most of the `iostream` header files include `ios`.

### **iosfwd**

The `iosfwd` header file declares forward references to various `iostream` template classes defined in other standard header files.

### **iostream**

The `iostream` header file declares most of the `iostream` objects used for the standard stream manipulations.

### **istream**

The `istream` header file defines the `istream` class for `iostream` extractions. Note that most of the `iostream` header files include `istream`.

### **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 `iostream` insertions.

### **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`.

### **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 `ostream` classes for `iostream` manipulations on allocated, extended, and freed character sequences.

## C and C++ Run-Time Libraries Guide

### 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 were `foo.h`, then the name of the equivalent C++ header file would be `cf00`. For example, the C++ header file `<cstdint>` provides the same facilities as the C header file `<stdint.h>`.

Table 1-17 lists the C++ header files that provide access to the C library facilities.

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.

Table 1-17. 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>setjmp</code>	Executes non-local goto statements
<code>signal</code>	Controls various exceptional conditions
<code>stdarg</code>	Accesses a variable number of arguments
<code>stddef</code>	Defines several useful data types and macros
<code>stdio</code>	Performs input and output
<code>stdlib</code>	Performs a variety of operations
<code>string</code>	Manipulates several kinds of strings

 Chapter 2 describes the functions in the DSP run-time libraries. Referencing these functions with a namespace prefix is not supported. All DSP library functions are in the global namespace.

### 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 templates that can be used to create callable types.

#### **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.

## C and C++ Run-Time Libraries Guide

### **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.

## **Header Files for C++ Library Compatibility**

The Embedded C++ library also includes several header files for compatibility with traditional C++ libraries. Table 1-18 describes these files.

Table 1-18. Header Files for C++ Library Compatibility

Header	Description
<code>fstream.h</code>	Defines several <code>iostream</code> template classes that manipulate external files
<code>iomanip.h</code>	Declares several <code>iostreams</code> manipulators that take a single argument
<code>iostream.h</code>	Declares the <code>iostream</code> objects that manipulate the standard streams
<code>new.h</code>	Declares several functions that allocate and free storage

### Using Thread-Safe C/C++ Run-Time Libraries with VDK

When developing for VDK, the thread-safe variants of the run-time libraries are linked with user applications. These libraries may add an overhead to the VDK resources required by some applications.

The run-time libraries make use of VDK synchronicity functions to ensure thread safety.

### Measuring Cycle Counts

The common basis for benchmarking some arbitrary C-written source is to measure the number of processor cycles that the code uses. Once this figure is known, it can be used to calculate the actual time taken by multiplying the number of processor cycles by the clock rate of the processor. The run-time library provides three alternative methods for measuring processor cycles, as described in the following sections:

Each of these methods is described in:

- “Basic Cycle Counting Facility” on page 1-48
- “Cycle Counting Facility with Statistics” on page 1-50
- “Using `time.h` to Measure Cycle Counts” on page 1-53

## C and C++ Run-Time Libraries Guide

- “Determining the Processor Clock Rate” on page 1-54
- “Considerations When Measuring Cycle Counts” on page 1-55

### Basic Cycle Counting Facility

The fundamental approach to measuring the performance of a section of code is to record the current value of the cycle count register before executing the section of code, and then reading the register again after the code has been executed. This process is represented by two macros that are defined in the `cycle_count.h` header file:

```
START_CYCLE_COUNT(S)
```

```
STOP_CYCLE_COUNT(T,S)
```

The parameter `S` is set by the macro `START_CYCLE_COUNT` to the current value of the cycle count register; this value should then be passed to the macro `STOP_CYCLE_COUNT`, which will calculate the difference between the parameter and current value of the cycle count register. Reading the cycle count register incurs an overhead of a small number of cycles and the macro ensures that the difference returned (in the parameter `T`) will be adjusted to allow for this additional cost. The parameters `S` and `T` should be separate variables; they should be declared as a `cycle_t` data type which the header file `cycle_count.h` defines as:

```
typedef volatile unsigned long cycle_t;
```

The header file also defines the macro:

```
PRINT_CYCLES(String,T)
```

which is provided mainly as an example of how to print a value of type `cycle_t`; the macro outputs the text `String` on `stdout` followed by the number of cycles `T`.

The instrumentation represented by the macros defined in this section is activated only if the program is compiled with the `-DDO_CYCLE_COUNTS` switch. If this switch is not specified, then the macros are replaced by empty statements and have no effect on the program.

The following example demonstrates how the basic cycle counting facility may be used to monitor the performance of a section of code:

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

extern int
main(void)
{
    cycle_t start_count;
    cycle_t final_count;

    START_CYCLE_COUNT(start_count);
    Some_Function_Or_Code_To_Measure();
    STOP_CYCLE_COUNT(final_count,start_count);

    PRINT_CYCLES("Number of cycles: ",final_count);
}
```

The run-time libraries provide alternative facilities for measuring the performance of C source (see “Cycle Counting Facility with Statistics” on page 1-50 and “Using `time.h` to Measure Cycle Counts” on page 1-53); the relative benefits of this facility are outlined in “Considerations When Measuring Cycle Counts” on page 1-55.

The basic cycle counting facility is based upon macros; it may therefore be customized for a particular application (if required), without the need for rebuilding the run-time libraries.

## C and C++ Run-Time Libraries Guide

### Cycle Counting Facility with Statistics

The `cycles.h` header file defines a set of macros for measuring the performance of compiled C source. In addition to providing the basic facility for reading the `EMUCLK` cycle count register of the SHARC architecture, the macros can also accumulate statistics suited to recording the performance of a section of code that is executed repeatedly.

If the switch `-DDO_CYCLE_COUNTS` is specified at compile-time, the `cycles.h` header file defines the following macros:

- `CYCLES_INIT(S)`  
This macro initializes the system timing mechanism and clears the parameter `S`; an application must contain one reference to this macro.
- `CYCLES_START(S)`  
This macro extracts the current value of the cycle count register and saves it in the parameter `S`.
- `CYCLES_STOP(S)`  
This macro extracts the current value of the cycle count register and accumulates statistics in the parameter `S`, based on the previous reference to the `CYCLES_START` macro.
- `CYCLES_PRINT(S)`  
This macro prints a summary of the accumulated statistics recorded in the parameter `S`.
- `CYCLES_RESET(S)`  
This macro re-zeros the accumulated statistics that are recorded in the parameter `S`.

The parameter `S` that is passed to the macros must be declared to be of the type `cycle_stats_t`; this is a structured data type that is defined in the `cycles.h` header file. The data type can record the number of times that an instrumented part of the source has been executed, as well as the mini-

imum, maximum, and average number of cycles that have been used. For example, if an instrumented piece of code has been executed 4 times, the `CYCLES_PRINT` macro would generate output on the standard stream `stdout` in the form:

```
AVG   : 95
MIN   : 92
MAX   : 100
CALLS : 4
```

If an instrumented piece of code had only been executed once, then the `CYCLES_PRINT` macro would print a message of the form:

```
CYCLES : 95
```

If the switch `-DDO_CYCLE_COUNTS` is not specified, then the macros described above are defined as null macros and no cycle count information is gathered. Therefore, to switch between development and release mode only requires a re-compilation and will not require any changes to the source of an application.

The macros defined in the `cycles.h` header file may be customized for a particular application without having to rebuild the run-time libraries.

The following example demonstrates how this facility may be used.

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

extern void foo(void);
extern void bar(void);

extern int
main(void)
{
    cycle_stats_t stats;
    int i;
```

## C and C++ Run-Time Libraries Guide

```
CYCLES_INIT(stats);

for (i = 0; i < LIMIT; i++) {
    CYCLES_START(stats);
    foo();
    CYCLES_STOP(stats);
}
printf("Cycles used by foo\n");
CYCLES_PRINT(stats);
CYCLES_RESET(stats);

    CYCLES_START(stats);
    bar();
    CYCLES_STOP(stats);
}
printf("Cycles used by bar\n");
CYCLES_PRINT(stats);
}
```

**This example might output:**

```
Cycles used by foo
AVG   : 25454
MIN   : 23003
MAX   : 26295
CALLS : 16
```

```
Cycles used by bar
AVG   : 8727
MIN   : 7653
MAX   : 8912
CALLS : 16
```

Alternative methods of measuring the performance of compiled C source are described in the sections “Basic Cycle Counting Facility” on page 1-48 and “Using `time.h` to Measure Cycle Counts” on page 1-53. Also refer to “Considerations When Measuring Cycle Counts” on page 1-55 which provides some useful tips with regards to performance measurements.

### Using `time.h` to Measure Cycle Counts

The `time.h` header file defines the data type `clock_t`, the `clock` function, and the macro `CLOCKS_PER_SEC`, which together may be used to calculate the number of seconds spent in a program.

In the ANSI C standard, the `clock` function is defined to return the number of implementation dependent clock “ticks” that have elapsed since the program began. In this version of the C/C++ compiler, the function returns the number of processor cycles that an application has used.

The conventional way of using the facilities of the `time.h` header file to measure the time spent in a program is to call the `clock` function at the start of a program, and then subtract this value from the value returned by a subsequent call to the function. The computed difference is usually cast to a floating-point type, and is then divided by the macro `CLOCKS_PER_SEC` to determine the time in seconds that has occurred between the two calls.

If this method of timing is used by an application, note that:

- The value assigned to the macro `CLOCKS_PER_SEC` should be independently verified to ensure that it is correct for the particular processor being used (see “Determining the Processor Clock Rate” on page 1-54),
- The result returned by the `clock` function does not include the overhead of calling the library function.

## C and C++ Run-Time Libraries Guide

A typical example that demonstrates the use of the `time.h` header file to measure the amount of time that an application takes is shown below.

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

extern int
main(void)
{
    volatile clock_t clock_start;
    volatile clock_t clock_stop;

    double secs;

    clock_start = clock();
    Some_Function_Or_Code_To_Measure();
    clock_stop = clock();

    secs = ((double) (clock_stop - clock_start))
           / CLOCKS_PER_SEC;
    printf("Time taken is %e seconds\n",secs);
}
```

The `cycles.h` and `cycle_count.h` header files define other methods for benchmarking an application—these header files are described in the sections “Basic Cycle Counting Facility” on page 1-48 and “Cycle Counting Facility with Statistics” on page 1-50, respectively. Also refer to “Considerations When Measuring Cycle Counts” on page 1-55 which provides some guidelines that may be useful.

### Determining the Processor Clock Rate

Applications may be benchmarked with respect to how many processor cycles they use. However, applications are typically benchmarked with respect to how much time (for example, in seconds) that they take.

Measuring the amount of time that an application takes to run on a SHARC processor usually involves first determining the number of cycles that the processor takes, and then dividing this value by the processor's clock rate. The `time.h` header file defines the macro `CLOCKS_PER_SEC` as the number of processor "ticks" per second.

On an ADSP-21xxx (SHARC) architecture, this parameter is set by the run-time library to one of the following values in descending order of precedence:

- By way of the compile-time switch  
`-DCLOCKS_PER_SEC=<definition>`.
- By way of the **Processor speed** box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Processor** category
- From the `cycles.h` header file

If the value of the macro `CLOCKS_PER_SEC` is taken from the `cycles.h` header file, then be aware that the clock rate of the processor will usually be taken to be the maximum speed of the processor, which is not necessarily the speed of the processor at `RESET`.

### Considerations When Measuring Cycle Counts

This section summarizes cycle-counting techniques for benchmarking C-compiled code. Each of these alternatives are described below.

- "Basic Cycle Counting Facility" on page 1-48  
The basic cycle counting facility represents an inexpensive and relatively unobtrusive method for benchmarking C-written source using cycle counts. The facility is based on macros that factor in the overhead incurred by the instrumentation. The macros may be customized and can be switched either on or off, and so no source

## C and C++ Run-Time Libraries Guide

changes are required when moving between development and release mode. The same set of macros is available on other platforms provided by Analog Devices.

- “Cycle Counting Facility with Statistics” on page 1-50  
This cycle-counting facility has more features than the basic cycle counting facility described above. It is more expensive in terms of program memory, data memory, and cycles consumed. However, it can record the number of times that the instrumented code has been executed and can calculate the maximum, minimum, and average cost of each iteration. The provided macros take into account the overhead involved in reading the cycle count register. By default, the macros are switched off, but they can be switched on by specifying the `-DDO_CYCLE_COUNTS` compile-time switch. The macros may be customized for a specific application. This cycle counting facility is also available on other Analog Devices architectures.
- “Using `time.h` to Measure Cycle Counts” on page 1-53  
The facilities of the `time.h` header file represent a simple method for measuring the performance of an application that is portable across many different architectures and systems. These facilities are based on the `clock` function.

The `clock` function however does not account for the cost involved in invoking the function. In addition, references to the function may affect the optimizer-generated code in the vicinity of the function call. This benchmarking method may not accurately reflect the true cost of the code being measured.

This method is best suited for benchmarking applications rather than smaller sections of code that run for a much shorter time span.

When benchmarking code, some thought is required when adding instrumentation to C source that will be optimized. If the sequence of statements to be measured is not selected carefully, the optimizer may move instructions into (and out of) the code region and/or it may re-site the instrumentation itself, leading to distorted measurements. Therefore, it is generally considered more reliable to measure the cycle count of calling (and returning from) a function rather than a sequence of statements within a function.

It is recommended that variables used directly in benchmarking are simple scalars that are allocated in internal memory (either assigned the result of a reference to the `clock` function, or used as arguments to the cycle counting macros). In the case of variables that are assigned the result of the `clock` function, it is also recommended that they be defined with the `volatile` keyword.

The different methods presented here to obtain the performance metrics of an application are based on the `EMUCLK` register. This is a 32-bit register that is incremented at every processor cycle; once the counter reaches the value `0xffffffff` it will wrap back to zero and will also increment the `EMUCLK2` register. However, to save memory and execution time, the `EMUCLK2` register is not used by either the `clock` function or the cycle counting macros. The performance metrics therefore will wrap back to zero after approximately every 71 seconds on a 60 MHz processor.

## File I/O Support

The VisualDSP++ environment provides access to files on a host system by using `stdio` functions. File I/O support is provided through a set of low-level primitives that implement the `open`, `close`, `read`, `write`, and `seek` operations. The functions defined in the `stdio.h` header file make use of these primitives to provide conventional C input and output facilities. The source files for the I/O primitives are available under the ADSP-21xxx installation of VisualDSP++ in the subdirectory `.. \lib\src\libio_src`.

## C and C++ Run-Time Libraries Guide

This section describes:

- “Extending I/O Support To New Devices” on page 1-58
- “Default Device Driver Interface” on page 1-67

Refer to “stdio.h” on page 1-30 for information about the conventional C input and output facilities that are provided by the compiler.

### Extending I/O Support To New Devices

The I/O primitives are implemented using an extensible device driver mechanism. The default start-up code includes a device driver that can perform I/O through the VisualDSP++ simulator and EZ-KIT Lite evaluation systems. Other device drivers may be registered and then used through the normal `stdio` functions.

This section describes:

- “DevEntry Structure”
- “Registering New Devices” on page 1-63
- “Pre-Registering Devices” on page 1-64
- “Default Device” on page 1-66
- “Remove and Rename Functions” on page 1-67

#### DevEntry Structure

A device driver is a set of primitive functions grouped together into a `DevEntry` structure. This structure is defined in `device.h`.

```
struct DevEntry {
    int    DeviceID;
    void  *data;
```

## C/C++ Run-Time Library

```
int (*init)(struct DevEntry *entry);
int (*open)(const char *name, int mode);
int (*close)(int fd);
int (*write)(int fd, unsigned char *buf, int size);
int (*read)(int fd, unsigned char *buf, int size);
long (*seek)(long fd, int offset, int whence);
int stdinfd;
int stdoutfd;
int stderrfd;
}
```

```
typedef struct DevEntry DevEntry;
typedef struct DevEntry *DevEntry_t;
```

The fields within the `DevEntry` structure have the following meanings.

### **DeviceID:**

The `DeviceID` field is a unique identifier for the device, known to the user. Device IDs are used globally across an application.

### **data:**

The `data` field is a pointer for any private data the device may need; it is not used by the run-time libraries.

### **init:**

The `init` field is a pointer to an initialization function. The run-time library calls this function when the device is first registered, passing in the address of this structure (and thus giving the `init` function access to `DeviceID` and the field `data`). If the `init` function encounters an error, it must return -1. Otherwise, it must return a positive value to indicate success.

### **open:**

The `open` field is a pointer to a function performs the "*open file*" operation upon the device; the run-time library will call this function in

## C and C++ Run-Time Libraries Guide

response to requests such as `fopen()`, when the device is the currently-selected default device. The `name` parameter is the path name to the file to be opened, and the `mode` parameter is a bitmask that indicates how the file is to be opened:

```
0x0001  Open file for reading
0x0002  Open file for writing
0x0004  Open file for appending
0x0008  Truncate the file to zero length, if it already exists
0x00010 Create the file, if it does not already exist
```

By default, files are opened as text streams (in which the character sequence `\r\n` is converted to `\n` when reading, and the character `\n` is written to the file as `\r\n`). A file is opened as a binary stream if the following bit value is set in the `mode` parameter:

```
0x0020  Open the file as a binary stream (raw mode).
```

The `open` function must return a positive “*file descriptor*” if it succeeds in opening the file; this file descriptor is used to identify the file to the device in subsequent operations. The file descriptor must be unique for all files currently open for the device, but need not be distinct from file descriptors returned by other devices—the run-time library identifies the file by the combination of device and file descriptor.

If the `open` function fails, it must return `-1` to indicate failure.

### **close:**

The `close` field is a pointer to a function that performs the “*close file*” operation on the device. The run-time library calls the `close` function in response to requests such as `fclose()` on a stream that was opened on the device. The `fd` parameter is a file descriptor previously returned by a call to the `open` function. The `close` function must return a zero value for success, and a non-zero value for failure.

### **write:**

The `write` field is a pointer to a function that performs the “*write to file*” operation on the device. The run-time library calls the `write` function in response to requests, such as `fwrite()`, `fprintf()` and so on, that act on streams that were opened on the device. The `write` function takes three parameters:

- `fd` – this is a file descriptor that identifies the file to be written to; it will be a value that was returned from a previous call to the `open` function.
- `buf` – a pointer to the data to be written to the file
- `size` – the number of bytes to be written to the file

The `write` function must return one of the following values:

- A positive value from 1 to `size` inclusive, indicating how many bytes from `buf` were successfully written to the file
- Zero, indicating that the file has been closed, for some reason (for example, network connection dropped)
- A negative value, indicating an error

### **read:**

The `read` field is a pointer to a function that performs the “*read from file*” operation on the device. The run-time library calls the `read` function in response to requests, such as `fread()`, `fscanf()` and so on, that act on streams that were opened on the device. The `read` function’s parameters are:

- `fd` – this is the file descriptor for the file to be read
- `buf` – this is a pointer to the buffer where the retrieved data must be stored

## C and C++ Run-Time Libraries Guide

- `size` – this is the number of (8-bit) bytes to read from the file. This must not exceed the space available in the buffer pointed to by `buf`

The `read` function must return one of the following values:

- A positive value from 1 to `size` inclusive, indicating how many bytes were read from the file into `buf`
- Zero, indicating end-of-file
- A negative value, indicating an error



The run-time library expects the `read` function to return `0xa` (10) as the newline character.

**seek:**

The `seek` field is a pointer to a function that performs dynamic access on the file. The run-time library calls the `seek` function in response to requests such as `rewind()`, `fseek()`, and so on, that act on streams that were opened on the device.

The `seek` function takes the following parameters:

- `fd` – this is the file descriptor for the file which will have its read/write position altered
- `offset` – this is a value that is used to determine the new read/write pointer position within the file; it is in (8-bit) bytes
- `whence` – this is a value that indicates how the `offset` parameter is interpreted:
  - 0: `offset` is an absolute value, giving the new read/write position in the file
  - 1: `offset` is a value relative to the current position within the file
  - 2: `offset` is a value relative to the end of the file

The `seek` function returns a positive value that is the new (absolute) position of the read/write pointer within the file, unless an error is encountered, in which case the `seek` function must return a negative value.

If a device does not support the functionality required by one of these functions (such as read-only devices, or stream devices that do not support seeking), the `DevEntry` structure must still have a pointer to a valid function; the function must arrange to return an error for attempted operations.

### **stdinfd:**

The `stdinfd` field is set to the device file descriptor for `stdin` if the device is expecting to claim the `stdin` stream, or to the enumeration value `dev_not_claimed` otherwise.

### **stdoutfd:**

The `stdoutfd` field is set to the device file descriptor for `stdout` if the device is expecting to claim the `stdout` stream, or to the enumeration value `dev_not_claimed` otherwise.

### **stderrfd:**

The `stderrfd` field is set to the device file descriptor for `stderr` if the device is expecting to claim the `stderr` stream, or to the enumeration value `dev_not_claimed` otherwise.

## **Registering New Devices**

A new device can be registered with the following function:

```
int add_devtab_entry(DevEntry_t entry);
```

If the device is successfully registered, the `init()` routine of the device is called, with `entry` as its parameter. The `add_devtab_entry()` function returns the `DeviceID` of the device registered.

## C and C++ Run-Time Libraries Guide

If the device is not successfully registered, a negative value is returned. Reasons for failure include (but are not limited to):

- The `DeviceID` is the same as another device, already registered
- There are no more slots left in the device registry table
- The `DeviceID` is less than zero
- Some of the function pointers are `NULL`
- The device's `init()` routine returned a failure result
- The device has attempted to claim a standard stream that is already claimed by another device

### Pre-Registering Devices

The library source file `devtab.c` (which can be found under a VisualDSP++ installation in the subdirectory `... \lib\src\libio_src`) declares the array:

```
DevEntry_t DevDrvTable[];
```

This array contains pointers to `DevEntry` structures for each device that is pre-registered, that is, devices that are available as soon as `main()` is entered, and that do not need to be registered at run-time by calling `add_devtab_entry()`. By default, the “*PrimIO*” device is registered. The `PrimIO` device provides support for target/host communication when using the simulators and the Analog Devices emulators and debug agents. This device is pre-registered, so that `printf()` and similar functions operate as expected without additional setup.

Additional devices can be pre-registered by the following process:

1. Take a copy of the `devtab.c` source file and add it to your project.
2. Declare your new device's `DevEntry` structure within the `devtab.c` file, for example,

```
extern DevEntry myDevice;
```

3. Include the address of the `DevEntry` structure within the `DevDrvTable[]` array. Ensure that the table is null-terminated. For example,

```
DevEntry_t DevDrvTable[MAXDEV] = {  
#ifdef PRIMIO  
    &primio_deventry,  
#endif  
    &myDevice, /* new pre-registered device */  
    0,  
};
```

All pre-registered devices are initialized by the run-time library when it calls the `init` function of each of the pre-registered devices in turn.

The normal behavior of the `PrimIO` device when it is registered is to claim the first three files as `stdin`, `stdout` and `stderr`. These standard streams may be re-opened on other devices at run-time by using `freopen()` to close the `PrimIO`-based streams and reopen the streams on the current default device.

## C and C++ Run-Time Libraries Guide

To allow an alternative device (either pre-registered or registered by `add_devtab_entry()`) to claim one or all of the standard streams:

1. Take a copy of the `primiolib.c` source file, and add it to your project.
2. Edit the appropriate `stdinfd`, `stdoutfd`, and `stderrfd` file descriptors in the `primio_deventry` structure to have the value `dev_not_claimed`.
3. Ensure the alternative device's `DevEntry` structure has set the standard stream file descriptors appropriately.

Both the device initialization routines, called from the startup code and `add_devtab_entry()`, return with an error if a device attempts to claim a standard stream that is already claimed.

### Default Device

Once a device is registered, it can be made the default device using the following function:

```
void set_default_io_device(int);
```

The function should be passed the `DeviceID` of the device. There is a corresponding function for retrieving the current default device:

```
int get_default_io_device(void);
```

The default device is used by `fopen()` when a file is first opened. The `fopen()` function passes the open request to the `open()` function of the device indicated by `get_default_io_device()`. The device's file identifier (`fd`) returned by the `open()` function is private to the device; other devices may simultaneously have other open files that use the same identifier. An open file is uniquely identified by the combination of `DeviceID` and `fd`.

The `fopen()` function records the `DeviceID` and `fd` in the global open file table, and allocates its own internal `fid` to this combination. All future operations on the file use this `fid` to retrieve the `DeviceID` and thus direct the request to the appropriate device's primitive functions, passing the `fd` along with other parameters. Once a file has been opened by `fopen()`, the current value of `get_default_io_device()` is irrelevant to that file.

### Remove and Rename Functions

The `PrimIO` device provides support for the `remove()` and `rename()` functions. These functions are not currently part of the extensible File I/O interface, since they deal purely with path names, and not with file descriptors. All calls to `remove()` and `rename()` in the run-time library are passed directly to the `PrimIO` device.

### Default Device Driver Interface

The `stdio` functions provide access to the files on a host system through a device driver that supports a set of low-level I/O primitives. These low-level primitives are described under “Extending I/O Support To New Devices” on page 1-58. The default device driver implements these primitives based on a simple interface provided by the VisualDSP++ simulator and EZ-KIT Lite systems.

All the I/O requests submitted through the default device driver 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.

## C and C++ Run-Time Libraries Guide

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 resides 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 processor 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 causes `__primIO` to assume that a host environment is present and able to communicate with the process.

If `__primIO` sees that `__lone_SHARC` is cleared, then upon entry (for example, when an I/O request is made) it sets a non-zero value into the word labeled `__Godot`. The `__primIO` routine then busy-waits 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 implementation of the `__primIO` interface 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.

Data packing is set to four bytes per word for the SHARC architecture. Data packing can be changed to accommodate other architectures by modifying the constant `BITS_PER_WORD`, defined in `_wordsize.h`. (For example, a processor with 16-bit addressable words would change this value to 16).

Note that the file name provided in an `OPEN` request uses the processor'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`, as follows.

```
typedef struct
{
    enum
    {
        PRIM_OPEN = 100,
        PRIM_READ,
        PRIM_WRITE,
        PRIM_CLOSE,
        PRIM_SEEK,
        PRIM_REMOVE,
        PRIM_RENAME
    } 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 seven currently-supported operations is being requested.

## C and C++ Run-Time Libraries Guide

The file ID for an open file is a non-negative integer assigned by the debugger or other “host” mechanism. The `fileID` values 0, 1, and 2 are pre-assigned to `stdin`, `stdout`, and `stderr`, respectively. No open request is required for these file IDs.

Before “activating” the debugger or other host environment, an `OPEN` or `REMOVE` request may set the `fileID` field to the length of the filename to open or delete; a `RENAME` request may also set the field to the length of the old filename. If the `fileID` field does contain a string length, then this will be indicated in the `flags` field (see below), and the debugger or other host environment will be able to use the information to perform a batch memory read to extract the filename. If the information is not provided, then the file name has to be extracted one character at a time.

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) */
M_STRLEN_PROVIDED = 0x8000 /* length of file name(s) available */
```

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 is 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 processor word boundary by writing padding NULs to the file before the buffer contents are transferred.

For an `OPEN`, `REMOVE`, and `RENAME` operation, the debugger (or other host mechanism) has to extract the filename(s) one character at a time from the memory of the target. However, if the bit corresponding to the value `M_STRLLEN_PROVIDED` is set, then the I/O control block contains the length of the filename(s) and the debugger is able to use this information to perform a batch read of the target memory (see the description of the fields `fileID` and `nCompleted`).

For a `SEEK` request, the `flags` field indicates the seek mode (*whence*) as follows:

```
enum
{
    M_SEEK_SET = 0x0001,    /* seek origin is the start of
                           the file */
    M_SEEK_CUR = 0x0002,   /* seek origin is the current
                           position within the file */
    M_SEEK_END = 0x0004,   /* seek origin is the end of
                           the file */
};
```

The `flags` field is unused for a `CLOSE` request.

The `buf` field contains a pointer to the file name for an `OPEN` or `REMOVE` request, or a pointer to the data buffer for a `READ` or `WRITE` request. For a `RENAME` operation, this field contains a pointer to the old file name.

The `nDesired` field is set to the number of bytes that should be transferred for a `READ` or `WRITE` request. This field is also used by a `RENAME` request, and is set to a pointer to the new file name.

## Documented Library Functions

For a `SEEK` request, the `nDesired` field contains the offset at which the file should be positioned, relative to the origin specified by the `flags` field. (On architectures that only support 16-bit `ints`, the 32-bit offset at which the file should be positioned is stored in the combined fields `[buf, nDesired]`).

The `nCompleted` field is set by `__primIO` to the number of bytes actually transferred by a `READ` or `WRITE` operation. For a `SEEK` operation, `__primIO` sets this field to the new value of the file pointer. (On architectures that only support 16-bit `ints`, `__primIO` sets the new value of the file pointer in the combined fields `[nCompleted, more]`).

The `RENAME` operation may also make use of the `nCompleted` field. If the operation can determine the lengths of the old and new filenames, then it should store these sizes in the fields `fileID` and `nCompleted`, respectively, and also set the bit-field `flags` to `M_STRLLEN_PROVIDED`. The debugger (or other host mechanism) can then use this information to perform a batch read of the target memory to extract the filenames. If this information is not provided, then each character of the file names will have to be read individually.

The `more` field is reserved for future use and currently is always set to `NULL` before calling `_primIO`.

## Documented Library Functions

The C run-time library has several categories of functions and macros defined by the ANSI C standard, plus extensions provided by Analog Devices.

The following tables list the library functions documented in this chapter. Note that the tables list the functions for each header file separately; however, the reference pages for these library functions present the functions in alphabetical order.

## C/C++ Run-Time Library

Table 1-19 lists the library functions in the `ctype.h` header file. Refer to “`ctype.h`” on page 1-22 for more information on this header file.

Table 1-19. Library Functions in the `ctype.h` Header File

<code>isalnum</code>	<code>isalpha</code>	<code>isctrl</code>
<code>isdigit</code>	<code>isgraph</code>	<code>islower</code>
<code>isprint</code>	<code>ispunct</code>	<code>isspace</code>
<code>isupper</code>	<code>isxdigit</code>	<code>tolower</code>
<code>toupper</code>		

Table 1-20 lists the library functions in the `locale.h` header file. Refer to “`locale.h`” on page 1-25 for more information on this header file.

Table 1-20. Library Functions in the `locale.h` Header File

<code>localeconv</code>	<code>setlocale</code>	
-------------------------	------------------------	--

Table 1-21 lists the library functions in the `math.h` header file. Refer to “`math.h`” on page 1-26 for more information on this header file.

Table 1-21. Library Functions in the `math.h` Header File

<code>acos</code>	<code>asin</code>	<code>atan</code>
<code>atan2</code>	<code>ceil</code>	<code>cos</code>
<code>cosh</code>	<code>exp</code>	<code>fabs</code>
<code>floor</code>	<code>fmod</code>	<code>frexp</code>
<code>isinf</code>	<code>isnan</code>	<code>ldexp</code>
<code>log</code>	<code>log10</code>	<code>modf</code>
<code>pow</code>	<code>sin</code>	<code>sinh</code>
<code>sqrt</code>	<code>tan</code>	<code>tanh</code>

## Documented Library Functions

Table 1-22 lists the library functions in the `setjmp.h` header file. Refer to “`setjmp.h`” on page 1-27 for more information on this header file.

Table 1-22. Library Functions in the `setjmp.h` Header File

<code>longjmp</code>	<code>setjmp</code>	
----------------------	---------------------	--

Table 1-23 lists the library functions in the `signal.h` header file. Refer to “`signal.h`” on page 1-27 for more information on this header file.

Table 1-23. Library Functions in the `signal.h` Header File

<code>clear_interrupt</code>	<code>interrupt</code>	<code>raise</code>
<code>signal</code>		

Table 1-24 lists the library functions in the `stdarg.h` header file. Refer to “`stdarg.h`” on page 1-28 for more information on this header file.

Table 1-24. Library Functions in the `stdarg.h` Header File

<code>va_arg</code>	<code>va_end</code>	<code>va_start</code>
---------------------	---------------------	-----------------------

Table 1-25 lists the library functions in the `stdio.h` header file. Refer to “`stdio.h`” on page 1-30 for more information on this header file.

Table 1-25. Library Functions in the `stdio.h` Header File

<code>clearerr</code>	<code>fclose</code>	<code>feof</code>
<code>ferror</code>	<code>fflush</code>	<code>fgetc</code>
<code>fgetpos</code>	<code>fgets</code>	<code>fopen</code>
<code>fprintf</code>	<code>fputc</code>	<code>fputs</code>
<code>fread</code>	<code>freopen</code>	<code>fscanf</code>
<code>fseek</code>	<code>fsetpos</code>	<code>ftell</code>
<code>fwrite</code>	<code>getc</code>	<code>getchar</code>

## C/C++ Run-Time Library

Table 1-25. Library Functions in the `stdio.h` Header File (Cont'd)

<code>gets</code>	<code>perror</code>	<code>printf</code>
<code>putc</code>	<code>putchar</code>	<code>puts</code>
<code>remove</code>	<code>rename</code>	<code>rewind</code>
<code>scanf</code>	<code>setbuf</code>	<code>setvbuf</code>
<code>snprintf</code>	<code>sprintf</code>	<code>sscanf</code>
<code>ungetc</code>	<code>vfprintf</code>	<code>vprintf</code>
<code>vsnprintf</code>	<code>vsprintf</code>	

Table 1-26 lists the library functions in the `stdlib.h` header file. Refer to “`stdlib.h`” on page 1-33 for more information on this header file.

Table 1-26. Library Functions in the `stdlib.h` Header File

<code>abort</code>	<code>abs</code>	<code>atexit</code>
<code>atof</code>	<code>atoi</code>	<code>atol</code>
<code>atold</code>	<code>avg</code>	<code>bsearch</code>
<code>calloc</code>	<code>clip</code>	<code>count_ones</code>
<code>div</code>	<code>exit</code>	<code>free</code>
<code>getenv</code>	<code>heap_calloc</code>	<code>heap_free</code>
<code>heap_install</code>	<code>heap_lookup_name</code>	<code>heap_malloc</code>
<code>heap_realloc</code>	<code>heap_switch</code>	<code>labs</code>
<code>lavg</code>	<code>lclip</code>	<code>lcount_ones</code>
<code>ldiv</code>	<code>lmax</code>	<code>lmin</code>
<code>malloc</code>	<code>max</code>	<code>min</code>
<code>qsort</code>	<code>rand</code>	<code>realloc</code>
<code>set_alloc_type</code>	<code>srand</code>	<code>strtod</code>
<code>strtol</code>	<code>strtold</code>	<code>strtoul</code>
<code>system</code>		

## Documented Library Functions

Table 1-27 lists the library functions in the `string.h` header file. Refer to “string.h” on page 1-34 for more information on this header file.

Table 1-27. Library Functions in the `string.h` Header File

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


Table 1-28 lists the library functions in the `time.h` header file. Refer to “time.h” on page 1-35 for more information on this header file.

Table 1-28. Library Functions in the `time.h` Header File

<code>asctime</code>	<code>clock</code>	<code>ctime</code>
<code>difftime</code>	<code>gmtime</code>	<code>localtime</code>
<code>mktime</code>	<code>strftime</code>	<code>time</code>

## C Run-Time Library Reference

The C run-time library is a collection of functions that you can call from your C/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** – Method that the functions use to indicate an error

**Example** – Typical function usage

**See Also** – Related functions

## C Run-Time Library Reference

### **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`, `labs`

## C Run-Time Library Reference

### acos

Arc cosine

#### Synopsis

```
#include <math.h>

float acosf (float x);
double acos (double x);
long double acosd (long double x);
```

#### Description

The arc cosine functions return the arc cosine of  $x$ . The input must be in the range  $[-1, 1]$ . The output, in radians, is in the range  $[0, \pi]$ .

#### Error Conditions

The arc cosine 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 x;
float y;

x = acos (0.0);      /* x =  $\pi/2$  */
y = acosf (0.0);    /* y =  $\pi/2$  */
```

#### See Also

cos

### **asctime**

convert broken-down time into a string

#### **Synopsis**

```
#include <time.h>
char *asctime(const struct tm *t);
```

#### **Description**

The `asctime` function converts a broken-down time, as generated by the functions `gmtime` and `localtime`, into an ASCII string that will contain the date and time in the form

```
DDD MMM dd hh:mm:ss YYYY\n
```

where

- `DDD` represents the day of the week (that is, Mon, Tue, Wed, etc.)
- `MMM` is the month and will be of the form Jan, Feb, Mar, etc
- `dd` is the day of the month, from 1 to 31
- `hh` is the number of hours after midnight, from 0 to 23
- `mm` is the minute of the day, from 0 to 59
- `ss` is the second of the day, from 0 to 61 (to allow for leap seconds)
- `YYYY` represents the year

The function returns a pointer to the ASCII string, which may be overwritten by a subsequent call to this function. Also note that the function `ctime` returns a string that is identical to

```
asctime(localtime(&t))
```

## C Run-Time Library Reference

### Error Conditions

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

### Example

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

struct tm tm_date;

printf("The date is %s",asctime(&tm_date));
```

### See Also

`ctime`, `gmtime`, `localtime`

### **asin**

Arc sine

#### **Synopsis**

```
#include <math.h>

float asinf (float x);
double asin (double x);
long double asind (long double x);
```

#### **Description**

The arc sine 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  $[-\pi/2, \pi/2]$ .

#### **Error Conditions**

The arc sine 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`

## C Run-Time Library Reference

### atan

Arc tangent

#### Synopsis

```
#include <math.h>

float atanf (float x);
double atan (double x);
long double atand (long double x);
```

#### Description

The arc tangent functions return the arc tangent of the first argument. The output, in radians, is in the range  $-\pi/2$  to  $\pi/2$ .

#### Error Conditions

The arc tangent 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, tan

**atan2**

Arc tangent of quotient

**Synopsis**

```
#include <math.h>

float atan2f (float y, float x);
double atan2 (double y, double x);
long double atan2d (long double y, long double x);
```

**Description**

The atan2 functions compute the arc tangent of the input value  $y$  divided by input value  $x$ . The output, in radians, is in the range  $-\pi$  to  $\pi$ .

**Error Conditions**

The atan2 functions return a zero if  $x=0$  and  $y=0$ .

**Example**

```
#include <math.h>

double a,d;
float b,c;

a = atan2 (0.0, 0.0);      /* the error condition: a = 0.0 */
b = atan2f (1.0, 1.0);   /* b =  $\pi/4$  */

c = atan2f (1.0, 0.0);   /* c =  $\pi/2$  */
d = atan2 (-1.0, 0.0);  /* d =  $-\pi/2$  */
```

**See Also**

atan, tan

## C Run-Time Library Reference

### **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 the `atexit` function.

#### **Error Conditions**

The `atexit` function returns a non-zero value if the function cannot be registered.

#### **Example**

```
#include <stdlib.h>

extern void goodbye(void);

if (atexit(goodbye))
    exit(1);
```

#### **See Also**

abort, exit

## atof

Convert string to a double

### Synopsis

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

### Description

The `atof` function converts a character string into a floating-point value of type `double`, and returns its value. The character string is pointed to by the argument `nptr` and may contain any number of leading whitespace characters (as determined by the function `isspace`) followed by a floating-point number. The floating-point number may either be a decimal floating-point number or a hexadecimal floating-point number.

A decimal floating-point number has the form:

```
[sign] [digits] [.digits] [{e|E} [sign] [digits]]
```

The `sign` token is optional and is either plus (+) or minus (-); and `digits` are one or more decimal digits. The sequence of digits may contain a decimal point (.).

The decimal digits can be followed by an exponent, which consists of an introductory letter (e or E) and an optionally signed integer. If neither an exponent part nor a decimal point appears, a decimal point is assumed to follow the last digit in the string.

The form of a hexadecimal floating-point number is:

```
[sign] [{0x}|{0X}] [hexdigs] [.hexdigs] [{p|P} [sign] [digits]]
```

## C Run-Time Library Reference

A hexadecimal floating-point number may start with an optional plus ( + ) or minus ( - ) followed by the hexadecimal prefix `0x` or `0X`. This character sequence must be followed by one or more hexadecimal characters that optionally contain a decimal point ( . ).

The hexadecimal digits are followed by a binary exponent that consists of the letter `p` or `P`, an optional sign, and a non-empty sequence of decimal digits. The exponent is interpreted as a power of two that is used to scale the fraction represented by the tokens `[hexdigs] [.hexdigs]`.

The first character that does not fit either form of number stops the scan.

### Error Conditions

The `atof` function returns a zero if no conversion could be made. If the correct value results in an overflow, a positive or negative (as appropriate) `HUGE_VAL` is returned. If the correct value results in an underflow, `0.0` is returned. The `ERANGE` value is stored in `errno` in the case of either an overflow or underflow.

### Notes

The `atof (pdata)` function reference is functionally equivalent to:

```
strtod (pdata, (char *) NULL);
```

and therefore, if the function returns zero, it is not possible to determine whether the character string contained a (valid) representation of `0.0` or some invalid numerical string.

### Example

```
#include <stdlib.h>

double x;

x = atof("5.5");      /* x = 5.5 */
```

**See Also**

atoi, atol, strtod

## C Run-Time Library Reference

### atoi

Convert string to integer

#### Synopsis

```
#include <stdlib.h>
int atoi (const char *nptr);
```

#### Description

The `atoi` function converts a character string to an integer value. The character string to be converted is pointed to by the input pointer, `nptr`. The function clears any leading characters for which `isspace` would return true. Conversion begins at the first digit (with an optional preceding sign) and terminates at the first non-digit.

#### Error Conditions

The `atoi` function returns -1 if no conversion can be made.

#### Example

```
#include <stdlib.h>

int i;

i = atoi ("5");    /* i = 5 */
```

#### See Also

`atof`, `atol`, `strtod`

### 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 -1 if no conversion can be made.

#### Example

```
#include <stdlib.h>

long int i;

i = atol ("5");    /* i = 5 */
```

#### See Also

`atof`, `atoi`, `strtod`, `strtoul`, `strtol`

## C Run-Time Library Reference

### atold

Convert string to a long double

#### Synopsis

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

#### Description

The `atold` function converts a character string into a floating-point value of type `long double`, and returns its value. The character string is pointed to by the argument `nptr` and may contain any number of leading whitespace characters (as determined by the function `isspace`) followed by a floating-point number. The floating-point number may either be a decimal floating-point number or a hexadecimal floating-point number.

A decimal floating-point number has the form:

```
[sign] [digits] [.digits] [{e|E} [sign] [digits]]
```

The `sign` token is optional and is either plus ( `+` ) or minus ( `-` ); and `digits` are one or more decimal digits. The sequence of digits may contain a decimal point ( `.` ).

The decimal digits can be followed by an exponent, which consists of an introductory letter ( `e` or `E` ) and an optionally signed integer. If neither an exponent part nor a decimal point appears, a decimal point is assumed to follow the last digit in the string.

The form of a hexadecimal floating-point number is:

```
[sign] [{0x}|{0X}] [hexdigs] [.hexdigs] [{p|P} [sign] [digits]]
```

A hexadecimal floating-point number may start with an optional plus ( + ) or minus ( - ) followed by the hexadecimal prefix `0x` or `0X` . This character sequence must be followed by one or more hexadecimal characters that optionally contain a decimal point ( . ).

The hexadecimal digits are followed by a binary exponent that consists of the letter `p` or `P` , an optional sign, and a non-empty sequence of decimal digits. The exponent is interpreted as a power of two that is used to scale the fraction represented by the tokens `[hexdigs] [.hexdigs]`.

The first character that does not fit either form of number stops the scan.

### Error Conditions

The `atold` function returns a zero if no conversion could be made. If the correct value results in an overflow, a positive or negative (as appropriate) `LDBL_MAX` is returned. If the correct value results in an underflow, `0.0` is returned. The `ERANGE` value is stored in `errno` in the case of either an overflow or underflow.

### Notes

The `atold (pdata)` function reference is functionally equivalent to:

```
strtold (pdata, (char *) NULL);
```

and therefore, if the function returns zero, it is not possible to determine whether the character string contained a (valid) representation of `0.0` or some invalid numerical string.

### Example

```
#include <stdlib.h>

long double x;

x = atold("5.5");      /* x = 5.5 */
```

## C Run-Time Library Reference

### See Also

atol, atoi, strtold

### **avg**

Mean of two values

### **Synopsis**

```
#include <stdlib.h>
int avg (int x, int y);
```

### **Description**

The `avg` 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 `Rn=(Rx+Ry)/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`

## C Run-Time Library Reference

### **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.

### Example

```
#include <stdlib.h>

char *answer;
char base[50][3];

answer = bsearch ("g", base, 50, 3, strcmp);
```

### See Also

qsort

## C Run-Time Library Reference

### **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` or `heap_switch` 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

## C Run-Time Library Reference

### ceil

Ceiling

#### Synopsis

```
#include <math.h>

float ceilf (float x);
double ceil (double x);
long double ceild (long double x);
```

#### Description

The ceiling functions return the smallest integral value that is not less than the argument *x*.

#### Error Conditions

The ceiling 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