RealView® Compilation Tools

Version 4.0

Libraries and Floating Point Support Guide

ARM®

ARM DUI 0349C (ID101213)

RealView Compilation Tools
Libraries and Floating Point Support Guide

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Confidentiality	Change
March 2007	A	Non-confidential	Release 3.1 for RealView Development Suite v3.1
September 2008	B	Non-Confidential	Release 4.0 for RealView Development Suite v4.0
23 January 2009	B	Non-Confidential	Update 1 for RealView Development Suite v4.0
10 December 2010	C	Non-Confidential	Update 2 for RealView Development Suite v4.0

Proprietary Notice

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Unrestricted Access is an ARM internal classification.

Product Status

The information in this document is final, that is for a developed product.

Web Address

`http://www.arm.com`

Contents
RealView Compilation Tools Libraries and Floating Point Support Guide

Preface

This preface introduces the *RealView Compilation Tools Libraries and Floating Point Support Guide*. It contains the following sections:

About this book

This book describes the ARM C and C++ libraries, compliance with the ISO standard, tailoring to target-dependent functions and application-specific requirements. It also describes the ARM C micro-library and the ARM support for floating-point computations.

Intended audience

This book is written for all developers who are producing applications using *RealView Compilation Tools* (RVCT). It assumes that you are an experienced software developer. See the *RealView Compilation Tools Essentials Guide* for an overview of the ARM development tools provided with RVCT.

Using this book

This book is organized into the following chapters:

Chapter 1 ***Introduction***

Read this chapter for an introduction to the ARM C and C++ libraries and the ARM floating-point environment.

Chapter 2 ***The C and C++ Libraries***

Read this chapter for a description of the ARM C and C++ libraries and instructions on re-implementing individual library functions. The ARM C library provides additional components to enable support for C++ and to compile code for different architectures and processors.

For a description of the Rogue Wave Standard C++ Library, see the Rogue Wave HTML documentation supplied with RVCT.

Chapter 3 ***The C Micro-library***

Read this chapter for a description of the ARM C micro-library and instructions on reimplementing individual micro-library functions.

Chapter 4 ***Floating-point Support***

Read this chapter for a description of floating-point support in the ARM compiler and libraries.

This book assumes that the ARM software is installed in the default location. For example, on Windows this might be *volume*:\Program Files\ARM. This is assumed to be the location of *install_directory* when referring to path names. For example *install_directory*\Documentation\.... You might have to change this if you have installed your ARM software in a different location.

Typographical conventions

The following typographical conventions are used in this book:

`monospace` Denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.

`mono`space Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.

`monospace italic`

Denotes arguments to commands and functions where the argument is to be replaced by a specific value.

`monospace bold`

Denotes language keywords when used outside example code.

italic Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.

bold Highlights interface elements, such as menu names. Also used for emphasis in descriptive lists, where appropriate, and for ARM processor signal names.

Further reading

This section lists publications from both ARM and third parties that provide additional information on developing code for the ARM family of processors.

ARM periodically provides updates and corrections to its documentation. See `http://infocenter.arm.com/help/index.jsp` for current errata sheets and addenda, and the ARM *Frequently Asked Questions* (FAQs).

ARM publications

This book contains reference information that is specific to development tools supplied with RVCT. Other publications included in the suite are:

- *RVDS Getting Started Guide* (ARM DUI 0255)
- *RVCT Essentials Guide* (ARM DUI 0202)
- *RVCT Compiler User Guide* (ARM DUI 0205)
- *RVCT Compiler Reference Guide* (ARM DUI 0348)
- *RVCT Linker User Guide* (ARM DUI 0206)
- *RVCT Linker Reference Guide* (ARM DUI 0381)

- *RVCT Utilities Guide* (ARM DUI 0382)
- *RVCT Assembler Guide* (ARM DUI 0204)
- *RVCT Developer Guide* (ARM DUI 0203)

A glossary is provided. See the *RVDS Getting Started Guide.*

For full information about the base standard, software interfaces, and standards supported by ARM, see *install_directory*\Documentation\Specifications\....

In addition, see the following documentation for specific information relating to ARM products:

- *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition* (ARM DDI 0406)
- *ARM7-M Architecture Reference Manual* (ARM DDI 0403)
- *ARM6-M Architecture Reference Manual* (ARM DDI 0419)
- ARM datasheet or technical reference manual for your hardware device.

Other publications

The following publication is referenced in the text:

- IEEE 754 - 1985 IEEE Standard for Binary Floating-Point Arithmetic

Feedback

ARM welcomes feedback on both RVCT and the documentation.

Feedback on RealView Compilation Tools

If you have any problems with RVCT, contact your supplier. To help them provide a rapid and useful response, give:

- your name and company
- the serial number of the product
- details of the release you are using
- details of the platform you are running on, such as the hardware platform, operating system type and version
- a small standalone sample of code that reproduces the problem
- a clear explanation of what you expected to happen, and what actually happened
- the commands you used, including any command-line options
- sample output illustrating the problem
- the version string of the tools, including the version number and build numbers.

Feedback on this book

If you notice any errors or omissions in this book, send email to `errata@arm.com` giving:

- the document title
- the document number
- the page number(s) to which your comments apply
- a concise explanation of the problem.

General suggestions for additions and improvements are also welcome.

Chapter 1
Introduction

This chapter introduces the ARM C and C++ libraries and the ARM floating-point environment. It contains the following sections:

- *About the runtime libraries* on page 1-2
- *About floating-point support* on page 1-8.

1.1 About the runtime libraries

The following runtime libraries are provided to support compiled C and C++:

C standardlib

C library consisting of:

- Functions defined by the ISO C library standard.
- Target-dependent functions used to implement the C library functions in the semihosted execution environment. You can redefine these functions in your own application.
- Functions called implicitly by the compiler.
- ARM extensions that are not defined by the ISO C library standard, but are included in the library.

C microlib

C micro-library (microlib) consisting of:

- Functions that are highly optimized to achieve the minimum code size.
- Functions that are not compliant with ISO C library standard.
- Functions that are not compliant with IEEE 754 standard for binary floating point arithmetic.

C++

C++ library containing the functions defined by the ISO C++ library standard.

The C++ libraries depend on the C library for target-specific support. There are no target dependencies in the C++ libraries.

The C++ libraries consist of:

- the Rogue Wave Standard C++ Library
- functions called implicitly by the compiler
- additional C++ functions not supported by the Rogue Wave library.

Note

Some library functions are called implicitly by the compiler, even though calls to such library functions might not exist in your source code. The compiler does this as a way of improving your application, where appropriate.

Note

If you are upgrading to RVCT from a previous release or are new to RVCT, ensure that you read the *RealView Compilation Tools Essentials Guide* for the latest information.

1.1.1 ABI for the ARM Architecture compliance

The *Application Binary Interface* for the ARM Architecture (ABI) is a family of specifications that describe the processor-specific aspects of the translation of a source program into object files. Object files produced by any toolchain that conforms to the relevant aspects of the ABI can be linked together to produce a final executable image or library. Each document in the specification covers a specific area of compatibility. For example, the *C Library ABI for the ARM Architecture* (CLIBABI) describes the parts of the C library that are expected to be common to all conforming implementations. The ABI documents contain several areas that are marked as *platform specific*. To define a complete execution environment these platform-specific details have to be provided. This gives rise to a number of supplemental specifications, for example the *ARM GNU/Linux ABI supplement*. The *Base Standard ABI for the ARM Architecture* (BSABI) enables you to use ARM, Thumb, and Thumb-2 objects and libraries from different producers that support the ABI for the ARM Architecture. RVCT fully supports the BSABI, including support for DWARF 3 debug tables (DWARF Debugging Standard Version 3).

For full information about the base standard, other *ARM Embedded ABIs* (AEABIs), software interfaces, and other standards supported by ARM, see *install_directory*\Documentation\Specifications\....

See http://www.arm.com for more information on the latest published versions.

The ARM C and C++ libraries conform to the standards described in the BSABI and in the following AEABIs:

- *C Library ABI for the ARM Architecture*
- *C++ ABI for the ARM Architecture* (CPPABI).

Testing conformance

To request full CLIBABI portability, specify `#define _AEABI_PORTABILITY_LEVEL 1` before you `#include` any library headers, for example, `<stdlib.h>`. You can also do this by using `-D_AEABI_PORTABILITY_LEVEL=1` on the command line. This increases the portability of your object files to other implementations of the *CLIBABI*, but reduces the performance of some library operations.

For more information see the CLIBABI specification, `clibabi.pdf`, in *install_directory*\Documentation\Specifications\...

1.1.2 Library directory structure

The libraries are installed in two subdirectories within the RVCT library directory `...\lib`:

`armlib` Contains the variants of the ARM C library, the floating-point arithmetic library (fplib), and the math library (mathlib). The accompanying header files are in `...\include`.

`cpplib` Contains the variants of the Rogue Wave C++ library (`cpp_*`) and supporting ARM C++ functions (`cpprt_*`), referred to collectively as the ARM C++ Libraries. The accompanying header files are installed in `...\include`.

The environment variable `RVCT40LIB` must be set to point to the `lib` directory, or if this variable is not set, `ARMLIB`. Alternatively, use the `--libpath` argument to the linker to identify the directory holding the library subdirectories. You must not identify the `armlib` and `cpplib` directories separately because this directory structure might change in future releases. The linker finds them from the location of `lib`.

Note

- The ARM C libraries are supplied in binary form only.
- The ARM libraries must not be modified. If you want to create a new implementation of a library function, place the new function in an object file, or your own library, and include it when you link the application. Your version of the function is used instead of the standard library version.
- Normally, only a few functions in the ISO C library require reimplementation to create a target-dependent application.
- The source for the Rogue Wave Standard C++ Library is not freely distributable. It can be obtained from Rogue Wave Software Inc., or through ARM, for an additional license fee. See the Rogue Wave online documentation in *`install_directory`*`\Documentation\RogueWave` for more information.

1.1.3 Build options and library variants

When you build your application, you must make certain choices. For example:

Target Architecture and instruction set

ARM, Thumb®, or Thumb-2.

Byte order Big-endian or little-endian.

Floating-point support

Software (SoftVFP), hardware (VFP), software or hardware with half-precision or double-precision extensions, or no floating-point support.

Position independence

Data can be read/write position independent or position dependent.

Code can be read-only position independent or position dependent.

Note

Position independence is not supported in microlib.

When you link your assembler code, C or C++ code, the linker selects appropriate C and C++ library variants compatible with the build options you specified. There is a variant of the ISO C library for each combination of major build options.

For more information see:

- Chapter 3 *Using armar* in the *Utilities Guide*
- *Specifying the procedure call standard (AAPCS)* on page 2-24 in the *Compiler User Guide*
- the *Assembler Guide*.

1.1.4 Using the VFP support libraries

The VFP support libraries are used by the VFP Support Code. The VFP Support Code is executed from an undefined instruction trap that is triggered when an exceptional floating point condition occurs.

Example code is provided in the main examples directory, in `...\vfpsupport`. This shows you how to set up the undefined instruction handler to invoke the VFP support code.

1.1.5 Thumb C libraries

The linker automatically links in the Thumb C library if it detects that one or more of the objects to be linked have been built for:

- Thumb or Thumb-2, either using the `--thumb` option or `#pragma thumb`
- interworking, using the `--apcs /interwork` option on architecture ARMv4T

- an ARMv6-M architecture target or processor, for example, Cortex-M1 or Cortex-M0
- an ARMv7-M architecture target or processor, for example, Cortex-M3.

Despite its name, the Thumb C library might not contain exclusively Thumb code. If ARM instructions are available, the Thumb library might use them to improve the performance of critical functions such as `memcpy()`, `memset()`, and `memclr()`. The bulk of the Thumb C library, however, is coded in Thumb for the best code density.

For an ARM-only build, compile with the `--arm_only` option.

Note

The Thumb C library used for ARMv7-M targets contains only Thumb-2 code.

See Chapter 2 *The C and C++ Libraries* for more information.

1.2 About floating-point support

The ARM floating-point environment is an implementation of the IEEE 754 standard for binary floating-point arithmetic. See *IEEE 754 arithmetic* on page 4-36 for more information on the ARM implementation of the standard.

An ARM system might have:

- a VFP coprocessor
- no floating-point hardware.

If you compile for a system with a hardware VFP coprocessor, the ARM compiler makes use of it. If you compile for a system without a coprocessor, the compiler implements the computations in software.

For example, the compiler option `--fpu=vfp` selects a hardware VFP coprocessor and the option `--fpu=softvfp` specifies that arithmetic operations are to be performed in software, without the use of any coprocessor instructions.

See Chapter 4 *Floating-point Support* for more information.

Chapter 2
The C and C++ Libraries

This chapter describes the C and C++ libraries. The libraries support programs written in C or C++. This chapter contains the following sections:

2.1 About the C and C++ libraries

This section contains information about the C and C++ libraries. If you are writing deeply embedded applications required to fit into small amounts of memory, see Chapter 3 *The C Micro-library*.

2.1.1 Features of the C and C++ libraries

In RVCT, the C library uses the standard ARM semihosted environment to provide facilities such as file input/output. This environment is supported by *RealView ARMulator® ISS*, RealView ICE, *Real Time Simulator Model* (RTSM), and *Instruction Set System Model* (ISSM). See the description in Chapter 8 *Semihosting* in the *Developer Guide* for more information on the debug environment.

You can reimplement any of the target-dependent functions of the C library as part of your application. This enables you to tailor the C library and, therefore, the C++ library, to your own execution environment.

You can also tailor many of the target-independent functions to your own application-specific requirements, for example:

- the `malloc` family
- the `ctype` family
- all the locale-specific functions.

Many of the C library functions are independent of any other function and contain no target dependencies. You can easily exploit these functions from assembler code.

2.1.2 Namespaces

All C++ standard library names, including the C library names, if you include them, are defined in the namespace `std` using the following C++ syntax:

```
#include <cstdlib> // instead of stdlib.h
```

This means that you must qualify all the library names using one of the following methods:

- specify the standard namespace, for example:

  ```
  std::printf("example\n");
  ```

- use the C++ keyword **`using`** to import a name to the global namespace:

  ```
  using namespace std;
  printf("example\n");
  ```

- use the compiler option `--using_std`.

Note

`errno` is a macro, so it is not necessary to qualify it with a namespace.

2.2 Writing reentrant and thread-safe code

The ARM C libraries support multithreading, for example, where you are using a *Real Time Operating System* (RTOS). When describing multithreading in the rest of this section, the following definitions are used:

Threads Mean multiple streams of execution sharing global data between them.

Process Means a collection of all the threads that share a particular set of global data.

If there are multiple processes on a machine, they can be entirely separate and do not share any data (except under unusual circumstances). Similarly, each process might be a single-threaded process or might be divided into multiple threads.

Where you have single-threaded processes, there is only one flow of control. In multithreaded applications, however, several flows of control might try to access the same functions, and the same resources, concurrently. To protect the integrity of resources, any code you write for multithreaded applications must be *reentrant* and *thread-safe*.

2.2.1 Introduction to reentrancy and thread-safety

Reentrancy and thread-safety are both related to the way functions handle resources. However, they are different:

- A reentrant function does not hold static data over successive calls, nor does it return a pointer to static data. For this type of function, the caller provides all the data that the function requires, such as pointers to any workspace. This means that multiple concurrent invocations of the function do not interfere with each other.

  Note

  A reentrant function must not call non reentrant functions.

- A thread-safe function protects shared resources from concurrent access using *locks*. Thread-safety concerns only how a function is implemented and not its external interface. In C, local variables are dynamically allocated on the stack. Therefore, any function that does not use static data, or other shared resources, is usually thread-safe.

2.2.2 Use of static data in the C libraries

Static data refers to persistent read/write data that is not stored on the stack or the heap. This persistent data can be external or internal in scope, and is:

- at a fixed address, when compiled with `--apcs /norwpi`
- at a fixed address relative to the static base, register `r9`, when compiled with `--apcs /rwpi`.

Libraries that use static data might be reentrant, but this depends on their use of the `__user_libspace` static data area, and on the build options you choose:

- When compiled with `--apcs /norwpi`, read/write static data is addressed in a position-dependent fashion. This is the default. Code from these variants is single-threaded because it uses read/write static data.
- When compiled with `--apcs /rwpi`, read/write static data is addressed in a position-independent fashion using offsets from the static base register `sb`. Code from these variants is reentrant and can be multiple threaded if each thread uses a different static base value.

The following describes how the C libraries use static data:

- The default floating-point arithmetic libraries `fz_*` and `fj_*` do not use static data and are always reentrant. However, the `f_*` and `g_*` libraries do use static data.
- All statically-initialized data in the C libraries is read-only.
- All writable static data is zero initialized.
- Most C library functions use no writable static data and are reentrant whether built with default build options, `--apcs /norwpi` or reentrant build options, `--apcs /rwpi`.
- Some functions have static data implicit in their definitions. You must not use these in a reentrant application unless you build it with `--apcs /rwpi` and the caller uses different values in `sb`.

See *Position independence qualifiers* on page 2-24 in the *Compiler User Guide* for information on the `--apcs` build options described here.

Note

Exactly which functions use static data in their definitions might change in future releases.

2.2.3 The __user_libspace static data area

The __user_libspace static data area holds the static data for the C libraries. This is a block of 96 bytes of zero-initialized data, supplied by the C library. It is also used as a temporary stack during C library initialization.

The default ARM C libraries use the __user_libspace area to hold:

- errno, used by any function that is capable of setting errno. By default, __rt_errno_addr() returns a pointer to errno.
- The FP status word for software floating-point (exception flags, rounding mode). It is unused in hardware floating-point. By default, __rt_fp_status_addr() returns a pointer to the FP status word.
- A pointer to the base of the heap (that is, the __Heap_Descriptor), used by all the malloc-related functions.
- The alloca state, used by alloca() and its supporting functions.
- The current locale settings, used by functions such as setlocale(), but also used by all other library functions that depend on them. For example, the ctype.h functions have to access the LC_CTYPE setting.

The C++ libraries use the __user_libspace area to hold:

- The new_handler field and ddtor_pointer:
  - the new_handler field is used to keep track of the value passed to std::set_new_handler()
  - the ddtor_pointer is used by __cxa_atexit() and __aeabi_atexit().
- C++ exception handling information for functions such as std::set_terminate() and std::set_unexpected().

See the following specifications: *CPPABI* and the *Exception Handling ABI for the ARM Architecture* for more information on __aeabi_atexit(), std::set_terminate() and std::set_unexpected().

Note

How the C and C++ libraries use the __user_libspace area might change in future releases.

Addressing __user_libspace

Two wrapper functions are provided to return a subsection of the `__user_libspace` static data area:

`__user_perproc_libspace()`

Returns a pointer to 96 bytes used to store data that is global to an entire process, that is data shared between all threads.

`__user_perthread_libspace()`

Returns a pointer to 96 bytes used to store data that is local to a particular thread. This means that `__user_perthread_libspace()` returns a different address depending on the thread it is called from.

Reimplementing __user_libspace

The `__user_libspace()` function does not normally have to be redefined. However, if you are writing an operating system or a process switcher, you must reimplement this function. See *Tailoring static data access* on page 2-41 for more information.

Where you are porting single-threaded processes to RVCT that reimplement this function, you can continue to do this without having to change your code. The change in behavior is, however, important if you are using RVCT to write multithreaded applications.

2.2.4 Managing locks in multithreaded applications

A thread-safe function protects shared resources from concurrent access using locks. There are functions you can reimplement that enable you to manage the locking mechanisms and so prevent the corruption of shared data such as the heap. The function prototypes for these functions are:

`_mutex_initialize()`

```
int _mutex_initialize(mutex *m);
```

This function accepts a pointer to a 32-bit word and initializes it as a valid mutex.

By default, `_mutex_initialize()` returns zero for a nonthreaded application. Therefore, in a multithreaded application, `_mutex_initialize()` must return a nonzero value on success so that, at runtime, the library knows that it is being used in a multithreaded environment.

This function must be supplied if you are using mutexes.

`_mutex_acquire()`

```
void _mutex_acquire(mutex *m);
```

This function causes the calling thread to obtain a lock on the supplied mutex.

`_mutex_acquire()` returns immediately if the mutex has no owner. If the mutex is owned by another thread, `_mutex_acquire()` must block it until it becomes available.

This function must be supplied if you are using mutexes.

`_mutex_release()`

```
void _mutex_release(mutex *m);
```

This function causes the calling thread to release the lock on a mutex acquired by `_mutex_acquire()`.

The mutex remains in existence, and can be relocked by a subsequent call to `mutex_acquire()`.

`_mutex_release()` assumes that the mutex is owned by the calling thread.

This function must be supplied if you are using mutexes.

`_mutex_free()`

```
void _mutex_free(mutex *m);
```

This function causes the calling thread to free the supplied mutex. Any operating system resources associated with the mutex are freed. The mutex is destroyed and cannot be reused.

`_mutex_free()` assumes that the mutex is owned by the calling thread.

This function is optional. If you do not supply this function, the C library does not attempt to call it.

For the C library, a mutex is specified as a single 32-bit word of memory that can be placed anywhere. However, if your mutex implementation requires more space than this, or demands that the mutex be in special memory area, then you must treat the default mutex as a pointer to a real mutex.

You can ensure that your mutex functions work correctly by adopting one of the following approaches:

- Place your mutex functions in a nonlibrary object file. This helps to ensure that they are resolved at link time.
- Place your mutex functions in a library object file, and arrange a nonweak reference to something in the object.

- Place your mutex functions in a library object file, and have the linker explicitly extract the specific object from the library on the command line by writing *libraryname*.a(*objectfilename*.o) when you invoke the linker.

2.2.5 Using the ARM C libraries with a multithreaded application

To use the ARM library in a multithreaded environment, you must provide:

- An implementation of `__user_perthread_libspace()` that returns a different block of memory for each thread. This can be achieved by either:
  - — returning a different address depending on the thread it is called from
  - — having a single `__user_perthread_libspace` block at a fixed address and swapping its contents when switching threads.

  You can use either approach to suit your environment.

  You do not have to reimplement `__user_perproc_libspace` unless there is a specific reason to do so. In the majority of cases, there is no requirement to reimplement this function.

- A way to manage multiple stacks.

  A simple way to do this is to use the ARM two-region memory model (see *The memory models* on page 2-69). Using this means that you keep the stack that belongs to the primary thread entirely separate from the heap. Then you must allocate more memory for additional stacks from the heap itself.

- Thread management functions, for example, to create or destroy threads, to handle thread synchronization, and to retrieve exit codes.

  Note

  The ARM C libraries supply no thread management functions of their own so you must supply any that are required.

- A thread-switching mechanism.

  Note

  The ARM C libraries supply no thread-switching mechanisms of their own. This is because there are many different ways to do this and the libraries are designed to work with all of them.

You only have to provide implementations of the mutex functions if you require them to be called. See *Managing locks in multithreaded applications* on page 2-7.

In some applications, the mutex functions might not be useful. For example, a co-operatively threaded program does not have to take steps to ensure data integrity, provided it avoids calling its yield function during a critical section. However, in other types of application, for example where you are implementing pre-emptive scheduling, or in a *Symmetric Multi-Processor* (SMP) model, these functions play an important part in handling locks.

If all of these requirements have been met, you can use the ARM C libraries in your multithreaded environment. Here:

- some functions work independently in each thread
- some functions automatically use the mutex functions to mediate multiple accesses to a shared resource
- some functions are still non reentrant so a reentrant equivalent is supplied
- a few functions remain non reentrant and no alternative is available.

See *Thread-safety in the ARM C libraries* and *Thread-safety in the ARM C++ libraries* on page 2-17 for more information.

2.2.6 Thread-safety in the ARM C libraries

In the ARM libraries, functions can be thread-safe as follows:

- some functions are never thread-safe, for example `setlocale()`
- some functions are inherently thread-safe, for example `memcpy()`
- some functions, such as `malloc()`, can be made thread-safe by implementing the `_mutex_*` functions
- other functions are only thread-safe if you pass the appropriate arguments, for example `tmpnam()`.

Threading problems might occur when your application makes use of the ARM libraries in a way that is hidden, for example, if the compiler implicitly calls functions that you have not explicitly called in your source code.

Functions that are thread-safe

Table 2-1 shows those C library functions that are thread-safe.

Table 2-1 Functions that are thread-safe

Functions	Description
`calloc()`, `free()`, `malloc()`, `realloc()`	The heap functions are thread-safe, if the `_mutex_*` functions are implemented. A single heap is shared between all threads, and mutexes are used to avoid data corruption when there is concurrent access. Each heap implementation is responsible for doing its own locking. If you supply your own allocator, it must also do its own locking. This enables it to do fine-grained locking, if required, rather than protecting the entire heap with a single mutex (coarse-grained locking).
`__alloca()`, `__alloca_finish()`, `__alloca_init()`, `__alloca_initialize()`	The `alloca` functions are thread-safe if the `_mutex_*` functions are implemented. The `alloca` state for each thread is contained in the `__user_perthread_libspace` block. This means that multiple threads do not conflict. **Note** Be aware that the `alloca` functions also use the heap. However, heap functions are all thread-safe.
`abort()`, `raise()`, `signal()`, `fenv.h`	The ARM signal handling functions and FP exception traps are thread-safe. The settings for signal handlers and FP traps are global across the entire process and are protected by locks. Data corruption does not occur if multiple threads call `signal()` or an `fenv.h` function at the same time. However, be aware that the effects of the call act on all threads and not only on the calling thread.

Table 2-1 Functions that are thread-safe (continued)

Functions	Description
`clearerr(), fclose(), feof(),ferror(), fflush(), fgetc(),fgetpos(), fgets(), fopen(),fputc(), fputs(), fread(),freopen(), fseek(), fsetpos(),ftell(), fwrite(), getc(),getchar(), gets(), perror(),putc(), putchar(), puts(),rewind(), setbuf(), setvbuf(),tmpfile(), tmpnam(), ungetc()`	The `stdio` library is thread-safe if the `_mutex_*` functions are implemented. Each individual stream is protected by a lock, so two threads can each open their own `stdio` stream and use it, without interfering with one another. If two threads both want to read or write the same stream, locking at the `fgetc()` and `fputc()` level prevents data corruption, but it is possible that the individual characters output by each thread might be interleaved in a confusing way. **Note** Be aware that `tmpnam()` also contains a static buffer but this is only used if the argument is NULL. To ensure that your use of `tmpnam()` is thread-safe, supply your own buffer space.
`fprintf(), printf(), vfprintf(), vprintf(), fscanf(), scanf()`	When using these functions: • The standard C `printf()` and `scanf()` functions use `stdio`, so they are thread-safe. • The standard C `printf()` is susceptible to changes in the locale settings if called in a multithreaded program.
`clock()`	`clock()` contains static data that is written once at program startup and then only ever read. Therefore, `clock()` is thread-safe provided no extra threads are already running at the time that the library is initialized.
`errno()`	`errno` is thread-safe. Each thread has its own `errno` stored in a `__user_perthread_libspace` block. This means that each thread can call `errno`-setting functions independently and then check `errno` afterwards without interference from other threads.
`atexit()`	The list of exit functions maintained by `atexit()` is process-global and protected by a lock. In the worst case, if more than one thread calls `atexit()`, the order that exit functions are called cannot be guaranteed.

Table 2-1 Functions that are thread-safe (continued)

Functions	Description
`abs(), acos(), asin(),atan(), atan2(), atof(),atol(), atoi(), bsearch(),ceil(), cos(), cosh(),difftime(), div(), exp(),fabs(), floor(), fmod(),frexp(), labs(), ldexp(),ldiv(), log(), log10(),memchr(), memcmp(), memcpy(),memmove(), memset(), mktime(),modf(), pow(), qsort(),sin(), sinh(), sqrt(),strcat(), strchr(), strcmp(),strcpy(), strcspn(), strlcat(),strlcpy(), strlen(), strncat(),strncmp(), strncpy(), strpbrk(),strrchr(), strspn(), strstr(),strxfrm(), tan(), tanh()`	These functions are inherently thread-safe.
`longjmp(), setjmp()`	Although `setjmp()` and `longjmp()` keep data in `__user_libspace`, they call the `__alloca_*` functions, that are thread-safe.
`remove(), rename(), time()`	These use interrupts that communicate with the ARM debugging environments. Typically, you have to reimplement these for a real-world application.
`snprintf(), sprintf(), vsnprintf(),vsprintf(), sscanf(), isalnum(),isalpha(), iscntrl(), isdigit(),isgraph(), islower(), isprint(),ispunct(), isspace(), isupper(),isxdigit(), tolower(), toupper(),strcoll(), strtod(), strtol(),strtoul(), strftime()`	When using these functions, the string-based functions read the locale settings. Typically, they are thread-safe. However, if you change locale in mid-session, you must ensure that these functions are not affected. The string-based functions, such as `sprintf()` and `sscanf()`, do not depend on the `stdio` library.
`stdin, stdout, stderr`	These are thread-safe.

FP status word

The FP status word is safe to use in a multithreaded environment, even in software floating-point. Here, a status word for each thread is stored in its own `__user_perthread_libspace` block.

Note

Be aware that, in hardware floating-point, the FP status word is stored in a VFP register. In this case, your thread-switching mechanism must keep a separate copy of this register for each thread.

Functions that are not thread-safe

Table 2-2 shows the C library functions that are not thread-safe.

Table 2-2 Functions that are not thread-safe

Functions	Description
`setlocale()`	The locale settings are global across all threads, and are not protected by a lock. If two threads call `setlocale()`, there might be data corruption. Also, many other functions, for example `strtod()` and `sprintf()`, read the current locale settings. So, if one thread calls `setlocale()` concurrently with another thread calling such a function then there might be unexpected results. ARM recommends that you choose the locale you want and call `setlocale()` once to initialize it. Do this before creating any additional threads in your program so that any number of threads can read the locale settings concurrently without interfering with one another. Be aware that `localeconv()` is not thread-safe. Call the ARM function `_get_lconv()` with a pointer to a user-supplied buffer instead.
`asctime()`, `localtime()`, `strtok()`	These functions are all thread-unsafe. Each contains a static buffer that might be overwritten by another thread between a call to the function and the subsequent use of its return value. ARM supplies reentrant versions, `_asctime_r()`, `_localtime_r()`, and `_strtok_r()`. ARM recommends that you use these functions instead to ensure safety. **Note** These reentrant versions take additional parameters. `_asctime_r()` takes an additional parameter that is a pointer to a buffer that the output string is written into. `_localtime_r()` takes an additional parameter that is a pointer to a `struct tm`, that the result is written into. `_strtok_r()` takes an additional parameter that is a pointer to a **`char`** pointer to the next token.
`gamma()`[a], `lgamma()`	These extended mathlib functions use a global variable, `_signgam`, so are not thread-safe.

Table 2-2 Functions that are not thread-safe (continued)

Functions	Description
`mbrlen(), mbsrtowcs(), mbrtowc(),wcrtomb(), wcsrtombs()`	The C89 multibyte conversion functions (defined in `stdlib.h`) are not thread-safe, for example `mblen()` and `mbtowc()`, because they contain internal static state that is shared between all threads without locking. However, the extended restartable versions (defined in `wchar.h`) are thread-safe, for example `mbrtowc()` and `wcrtomb()`, provided you pass in a pointer to your own `mbstate_t` object. You must exclusively use these functions with non-NULL `mbstate_t *` parameters if you want to ensure thread-safety when handling multibyte strings.
`exit()`	Do not call `exit()` in a multithreaded program even if you have provided an implementation of the underlying `_sys_exit()` that actually terminates all threads. In this case, `exit()` cleans up *before* calling `_sys_exit()` so disrupts other threads.
`rand(), srand()`	These functions keep internal state that is both global and unprotected. This means that calls to `rand()` are never thread-safe ARM recommends that you use your own locking to ensure that only one thread ever calls `rand()` at a time, for example, by defining `$Sub$$rand()` if you want to avoid changing your code. Alternatively, do one of the following: • supply your own random number generator that can have multiple independent instances • arrange that only one thread ever needs to generate random numbers.

a. `gamma()` is deprecated.

2.2.7 Thread-safety in the ARM C++ libraries

The following summarizes thread-safety in the C++ libraries:

- The function `std::set_new_handler()` is not thread-safe. This means that some forms of `::operator new` and `::operator delete` are not thread-safe with respect to `std::set_new_handler()`:
  - The default C++ runtime library implementations of the following use `malloc()` and `free()` and are thread-safe with respect to each other. They are not thread-safe with respect to `std::set_new_handler()`. You are permitted to replace them:

    ```
    ::operator new(std::size_t)
    ::operator new[](std::size_t)
    ::operator new(std::size_t, const std::nothrow_t&)
    ::operator new[](std::size_t, const std::nothrow_t)
    ::operator delete(void*)
    ::operator delete[](void*)
    ::operator delete(void*, const std::nothrow_t&)
    ::operator delete[](void*, const std::nothrow_t&)
    ```

  - The following placement forms are also thread-safe. You are not permitted to replace them:

    ```
    ::operator new(std::size_t, void*)
    ::operator new[](std::size_t, void*)
    ::operator delete(void*, void*)
    ::operator delete[](void*, void*)
    ```

- Construction and destruction of global objects is not thread-safe.
- Construction of local static objects can be made thread-safe, if you reimplement the functions `__cxa_guard_acquire()`, `__cxa_guard_release()`, `__cxa_guard_abort()`, `__cxa_atexit()` and `__aeabi_atexit()` appropriately. For example, with appropriate reimplementation, the following construction of `lsobj` can be made thread-safe:

  ```
  struct T { T(); };
  void f() { static T lsobj; }
  ```

  See the CPPABI for information on the `__cxa_` and `__aeabi_` functions.
- Throwing an exception is thread-safe if any user constructors and destructors that get called are also thread-safe. See the *Exception Handling ABI for the ARM Architecture* for more information.

- The ARM C++ library uses the ARM C library. To use the ARM C++ library in a multithreaded environment, you must provide the functions described in *Using the ARM C libraries with a multithreaded application* on page 2-9.

2.3 Building an application with the C library

This section covers creating an application that links with functions from the C or C++ libraries. Functions in the C library are responsible for:

- Creating an environment in which a C or C++ program can execute. This includes:
  - creating a stack
  - creating a heap, if required
  - initializing the parts of the library the program uses.
- Starting execution by calling `main()`.
- Supporting use of ISO-defined functions by the program.
- Catching runtime errors and signals and, if required, terminating execution on error or program exit.

2.3.1 Using the libraries with an application

There are three ways to use the libraries with an application:

- Build a semihosted application that can be debugged in a semihosted environment such as with RealView ISS, ISSM, RealView ICE or RealMonitor. See *Building an application for a semihosted environment*.
- Build a non hosted application that, for example, can be embedded into ROM. See *Building an application for a non semihosting environment* on page 2-21.
- Build an application that does not use `main()` and does not initialize the library. This application has restricted library functionality, unless you reimplement some functions. See *Building an application without the C library* on page 2-25.

2.3.2 Building an application for a semihosted environment

If you are developing an application to run in a semihosted environment for debugging, you must have an execution environment that supports ARM or Thumb semihosting, and has sufficient memory.

The execution environment can be provided by either:

- using the standard semihosting functionality that is present by default in, for example, RealView ISS, ISSM, RealView ICE and RealMonitor
- implementing your own handler for the semihosting calls. See Chapter 8 *Semihosting* in the *Developer Guide*.

See *Overview of semihosting dependencies* on page 2-22 for a list of functions that require semihosting.

It is not necessary to write any new functions or include files if you are using the default semihosting functionality of the library.

Using RealView ISS or ISSM

RealView ISS and ISSM support semihosting and have a memory map that enables the use of the library. RealView ISS and ISSM use memory in the host machine and this is normally adequate for your application.

Using RealView ICE

The ARM debug agents support semihosting but the memory map assumed by the library might require tailoring to match the hardware being debugged. However, it is easy to tailor the memory map assumed by the C library. See *Tailoring the runtime memory model* on page 2-69.

Using reimplemented functions in a semihosted environment

You can also mix the semihosting functionality with new input/output functions. For example, you can implement `fputc()` to output directly to hardware such as a UART, in addition to the semihosted implementation. See *Building an application for a non semihosting environment* on page 2-21 for information on how to reimplement individual functions.

Converting a semihosted application to a standalone application

After an application has been developed in a semihosted debugging environment, you can move the application to a non hosted environment by one of the following methods:

- Remove all calls to semihosted functions. See *Avoiding semihosting* on page 2-23.
- Reimplement the lower-level functions, for example, `fputc()`. See *Building an application for a non semihosting environment* on page 2-21. You do not have to reimplement all semihosted functions. You must, however, reimplement the functions that you are using in your application.
- Implement a handler for all the semihosting calls.

2.3.3 Building an application for a non semihosting environment

If you do not want to use any semihosting functionality, you must remove all calls to semihosting functions or reimplement them with non semihosting functions.

To build an application that does not use semihosting functionality:

1. Create the source files to implement the target-dependent features. For example, functions that use the semihosting calls or that depend on the target memory map.
2. Add the `__use_no_semihosting` symbol to the source. See *Avoiding semihosting* on page 2-23.
3. Link the new objects with your application.
4. Use the new configuration when creating the target-dependent application.

You must reimplement functions that the C library uses to insulate itself from target dependencies. For example, if you use `printf()` you must reimplement `fputc()`. If you do not use the higher-level input/output functions like `printf()`, you do not have to reimplement the lower-level functions like `fputc()`.

If you are building an application for a different execution environment, you can reimplement the target-dependent functions. For example, those functions that use semihosting calls or depend on the target memory map. There are no target-dependent functions in the C++ library, although some C++ functions use underlying C library functions that are target-dependent.

Examples of embedded applications that do not use a hosted environment are included in the main examples directory, in `...\emb_sw_dev`.

See the *Developer Guide* for examples of creating embedded applications.

C++ exceptions in a non semihosted environment

The default C++ `std::terminate()` handler is required by the C++ Standard to call `abort()`. The default C library implementation of `abort()` uses functions that require semihosting support. Therefore, if you use exceptions in a non semihosted environment, you must provide an alternative implementation of `abort()`.

Overview of semihosting dependencies

Table 2-3 shows the functions that depend directly on semihosting.

Table 2-3 Direct semihosting dependencies

Function	Description
`__user_initial_stackheap()`	See *Tailoring the runtime memory model* on page 2-69. You might have to reimplement this function if you are using scatter-loading.
`_sys_exit()` `_ttywrch()`	See *Tailoring error signaling, error handling, and program exit* on page 2-59.
`_sys_command_string()`, `_sys_close()`, `_sys_ensure()`, `_sys_iserror()`, `_sys_istty()`, `_sys_flen()`, `_sys_open()`, `_sys_read()`, `_sys_seek()`, `_sys_write()`, `_sys_tmpnam()`	See *Tailoring the input/output functions* on page 2-78.
`clock()`, `_clock_init()`, `remove()`, `rename()`, `system()`, `time()`	See *Tailoring other C library functions* on page 2-92.

Table 2-4 shows those functions that depend indirectly on one or more of the functions listed in Table 2-3.

Table 2-4 Indirect semihosting dependencies

Function	Usage
`__raise()`	Catch, handle, or diagnose C library exceptions, without C signal support. See *Tailoring error signaling, error handling, and program exit* on page 2-59.
`__default_signal_handler()`	Catch, handle, or diagnose C library exceptions, with C signal support. See *Tailoring error signaling, error handling, and program exit* on page 2-59.
`__Heap_Initialize()`	Choosing or redefining memory allocation. See *Tailoring storage management* on page 2-65.

Table 2-4 Indirect semihosting dependencies (continued)

Function	Usage
`ferror()`, `fputc()`, `__stdout`	Reimplementing the printf family. See *Tailoring the input/output functions* on page 2-78.
`__backspace()`, `fgetc()`, `__stdin`	Reimplementing the scanf family. See *Tailoring the input/output functions* on page 2-78.
`fwrite()`, `fputs()`, `puts()`,`fread()`, `fgets()`, `gets()`, `ferror()`	Reimplementing the stream output family. See *Tailoring the input/output functions* on page 2-78.

Avoiding semihosting

If you write an application in C, you must link it with the C library even if it makes no direct use of C library functions. This is because the compiler might implicitly call functions in the C library, and because the C library also contains initialization code. Some C library functions use semihosting.

To avoid using semihosting, do either of the following:

- reimplement the functions in your own application
- write the application so that it does not call any semihosted function.

To guarantee that no functions using semihosting are included in your application, use either:

- `IMPORT __use_no_semihosting` from assembly language
- `#pragma import(__use_no_semihosting)` from C.

Note

`IMPORT __use_no_semihosting` is only required to be added to a single assembly source file. Similarly, `#pragma import(__use_no_semihosting)` is only required to be added to a single C source file. It is not necessary to add these inserts to every single source file.

If you include a library function that uses semihosting and also reference `__use_no_semihosting`, the library detects the conflicting symbols and the linker reports an error. To find out which objects are using semihosting, link with `--verbose --list=out.txt`, search the output for the symbol, and find out what object referenced it. See *--list=file* on page 2-56 and *--verbose* on page 2-98 in the *Linker Reference Guide* for more information.

API definitions

In addition to the semihosted functions listed in Table 2-3 on page 2-22 and Table 2-4 on page 2-22, Table 2-5 shows functions and files that might be useful when building for a different environment.

Table 2-5 Published API definitions

File or function	Description
`__main()` `__rt_entry()`	Initializes the runtime environment and executes the user application.
`__rt_lib_init()`, `__rt_exit()`, `__rt_lib_shutdown()`	Initializes or finalizes the runtime library.
`LC_CTYPE locale`	Defines the character properties for the local alphabet. See *Tailoring locale and CTYPE using assembler macros* on page 2-42.
`rt_sys.h`	A C header file describing all the functions whose default (semihosted) implementations use semihosting calls.
`rt_heap.h`	A C header file describing the storage management abstract data type.
`rt_locale.h`	A C header file describing the five locale category *filing systems*, and defining some macros that are useful for describing the contents of locale categories.
`rt_misc.h`	A C header file describing miscellaneous unrelated public interfaces to the C library.
`rt_memory.s`	An empty, but commented, prototype implementation of the memory model. See *Writing your own memory model* on page 2-71 for a description of this file.

If you are reimplementing a function that exists in the standard ARM library, the linker uses an object or library from your project rather than the standard ARM library. Any library you add to a project does not have to follow the ARM library naming convention.

Caution

Do not replace or delete libraries supplied by ARM. You must not overwrite the supplied library files. Place your reimplemented functions in separate object files or libraries instead.

2.4 Building an application without the C library

Creating an application that has a `main()` function causes the C library initialization functions to be included as part of `__rt_lib_init`.

If your application does not have a `main()` function, the C library is not initialized and the following features are not available in your application:

- low-level `stdio` functions that have the prefix `_sys_`
- signal-handling functions, `signal()` and `raise()` in `signal.h`
- other functions, such as `atexit()` and `alloca()`.

See *The standalone C library functions* on page 2-28 for more information on the functions that are not available without library initialization.

This section refers to creating applications without the library as *bare machine C*. These applications do not automatically use the full C runtime environment provided by the C library. Even though you are creating an application without the library, some functions from the library that are called implicitly by the compiler must be included. There are also many library functions that can be made available with only minor reimplementations.

2.4.1 Integer and floating point functions

There are several compiler functions that are used by the compiler to handle operations that do not have a short machine code equivalent. For example, integer divide uses a function that is implicitly called by the compiler if there is no divide instruction available in the target instruction set. (ARMv7-R and ARMv7-M architectures use the instructions `SDIV` and `UDIV` in Thumb state. Other versions of the ARM architecture also use compiler functions that are implicitly invoked.)

Integer divide, and all the floating-point functions, require `__rt_raise()` to handle math errors. Reimplementing `__rt_raise()` enables all the math functions, and it avoids having to link in all the signal-handling library code.

2.4.2 Bare machine integer C

If you are writing a program in C that does not use the library and is to run without any environment initialization, you must:

- Reimplement `__rt_raise()`, because this error-handling function can be called from numerous places within the compiled code.
- Not define `main()` to avoid linking in the library initialization code.

- Write an assembly language veneer that establishes the register state required to run C. This veneer must branch to the entry function in your application.
- Provide your own RW/ZI initialization code.
- Ensure that your initialization veneer is executed by, for example, placing it in your reset handler.
- Build your application using `--fpu=none` and link it normally. The linker uses the appropriate C library variant to find any required compiler functions that are implicitly called.

Many library facilities require `__user_libspace` for static data. Even without the initialization code activated by having a `main()` function, `__user_libspace` is created automatically and uses 96 bytes in the ZI segment. See *The __user_libspace static data area* on page 2-6 for a description of the `__user_libspace` area.

2.4.3 Bare machine C with floating-point

If you want to use floating-point processing in your application you must:

- Perform the steps necessary for integer C as described in *Bare machine integer C* on page 2-25. However, do not build your application with the `--fpu=none` option.
- Use the appropriate FPU option when you build your application.
- Call `_fp_init()` to initialize the floating-point status register before performing any floating-point operations.

If you are using software floating-point, you can also define the function `__rt_fp_status_addr()` to return the address of a writable data word to be used instead of the floating-point status register. If you do not do this, the `__user_libspace` area is created, occupying 96 bytes. See *The __user_libspace static data area* on page 2-6 for a description of the `__user_libspace` area.

2.4.4 Exploiting the C library

If you create an application that includes a `main()` function, the linker automatically includes the initialization code necessary for the execution environment. See *Building an application with the C library* on page 2-19 for instructions. There are situations though where this is not desirable or possible.

You can create an application that consists of customized startup code and still use many of the library functions. You must either:

- avoid functions that require initialization
- provide the initialization and low-level support functions.

Program design

The functions you must reimplement depend on how much of the library functionality you require:

- If you want only the compiler support functions for division, structure copy, and FP arithmetic, you must provide `__rt_raise()`. This also enables very simple library functions such as those in `errno.h`, `setjmp.h`, and most of `string.h` to work.
- If you call `setlocale()` explicitly, locale-dependent functions are activated. This enables you to use the `atoi` family, `sprintf()`, `sscanf()`, and the functions in `ctype.h`.
- Programs that use floating-point must call `_fp_init()`. If you select software floating-point, the program must also provide `__rt_fp_status_addr()`. If this function is not reimplemented, the default action is to create a `__user_libspace` area. See *The __user_libspace static data area* on page 2-6 for a description of the `__user_libspace` area.
- Implementing high-level input/output support is necessary for functions that use `fprintf()` or `fputs()`. The high-level output functions depend on `fputc()` and `ferror()`. The high-level input functions depend on `fgetc()` and `__backspace()`.

Implementing these functions and the heap enables you to use almost the entire library.

Using low-level functions

If you are using the libraries in an application that does not have a `main()` function, you must reimplement some functions in the library. See *The standalone C library functions* on page 2-28 for more information.

`__rt_raise()` is essential. It is required by all FP functions, by integer division (so that divide-by-zero can be reported), and by some other library routines. You probably cannot write a non trivial program without doing something that requires `__rt_raise()`.

Note

If `rand()` is called, `srand()` *must* be called first. This is done automatically during library initialization but not when you avoid the library initialization.

Using high-level functions

High-level I/O functions, `fprintf()` for example, can be used if the low-level functions, `fputc()` for example, are reimplemented. Most of the formatted output functions also require a call to `setlocale()`. See *Tailoring the input/output functions* on page 2-78 for instructions.

Anything that uses `locale` must not be called before first calling `setlocale()` to initialize it, for example, call `setlocale(LC_ALL, "C")`. Locale-using functions are described in *The standalone C library functions*. These include the functions in `ctype.h` and `locale.h`, the `printf()` family, the `scanf()` family, `ato*`, `strto*`, `strcoll/strxfrm`, and much of `time.h`.

Using malloc()

If heap support is required for bare machine C, `_init_alloc()` must be called first to supply initial heap bounds, and `__rt_heap_extend()` *must* be provided even if it only returns failure. Prototypes for both functions are in `rt_heap.h`.

2.4.5 The standalone C library functions

The rest of this section lists the include files, and the functions they contain, that are available with an uninitialized library. Some otherwise-unavailable functions can be used if the library functions they depend on are reimplemented.

alloca.h

Functions listed in this file are not available without library initialization. See *Building an application with the C library* on page 2-19 for instructions.

assert.h

Functions listed in this file require high-level `stdio`, `__rt_raise()`, and `_sys_exit()`. See *Tailoring error signaling, error handling, and program exit* on page 2-59 for instructions.

ctype.h

Functions listed in this file require the `locale` functions.

errno.h

Functions in this file work without the requirement for any library initialization or function reimplementation.

fenv.h

Functions in this file work without the requirement for any library initialization and only require the reimplementation of `__rt_raise()`.

float.h

This file does not contain any code. The definitions in the file do not require library initialization or function reimplementation.

inttypes.h

Functions listed in this file require the `locale` functions.

limits.h

Functions in this file work without the requirement for any library initialization or function reimplementation.

locale.h

Call `setlocale()` before calling any function that uses `locale` functions. For example:

```
setlocale(LC_ALL, "C")
```

See the contents of `locale.h` for more information on the following functions and data structures:

- `setlocale()` selects the appropriate locale as specified by the `category` and `locale` arguments.
- `lconv` is the structure used by `locale` functions for formatting numeric quantities according to the rules of the current locale.
- `localeconv()` creates an `lconv` structure and returns a pointer to it.
- `_get_lconv()` fills the `lconv` structure pointed to by the parameter. This ISO extension removes the requirement for static data within the library.

`locale.h` also contains constant declarations used with `locale` functions. See *Tailoring locale and CTYPE using assembler macros* on page 2-42 for more information.

math.h

For functions in this file to work, you must first call `_fp_init()` and reimplement `__rt_raise()`.

setjmp.h

Functions in this file work without any library initialization or function reimplementation.

signal.h

Functions listed in this file are not available without library initialization. See *Building an application with the C library* on page 2-19 for instructions on building an application that uses library initialization.

`__rt_raise()` can be reimplemented for error and exit handling. See *Tailoring error signaling, error handling, and program exit* on page 2-59 for instructions.

stdarg.h

Functions listed in this file work without any library initialization or function reimplementation.

stddef.h

This file does not contain any code. The definitions in the file do not require library initialization or function reimplementation.

stdint.h

This file does not contain any code. The definitions in the file do not require library initialization or function reimplementation.

stdio.h

The following dependencies or limitations apply to these functions:

- The high-level functions such as `printf()`, `scanf()`, `puts()`, `fgets()`, `fread()`, `fwrite()`, and `perror()` depend on lower-level `stdio` functions `fgetc()`, `fputc()`, and `__backspace()`. You must reimplement these lower-level functions when using the standalone C library.

  However, you cannot reimplement the `_sys_` prefixed functions (for example, `_sys_read()`) when using the standalone C library because they require library initialization.

  See *Tailoring the input/output functions* on page 2-78 for more information.

- The `printf()` and `scanf()` family of functions require `locale`.

- The `remove()` and `rename()` functions are system-specific and probably not usable in your application.

stdlib.h

Most functions in this file work without any library initialization or function reimplementation. The following functions depend on other functions being instantiated correctly:

- `ato*()` requires `locale`
- `strto*()` requires `locale`
- `malloc()`, `calloc()`, `realloc()`, and `free()` require heap functions
- `atexit()` is not available when building an application without the C library.

string.h

Functions in this file work without any library initialization, with the exception of `strcoll()` and `strxfrm()`, that require `locale`.

time.h

- `mktime()` and `localtime()` can be used immediately
- `time()` and `clock()` are system-specific and probably not usable unless reimplemented
- `asctime()`, `ctime()`, and `strftime()` require `locale`.

wchar.h

Wide character library functions added to ISO C by *Normative Addendum 1* in 1994.

- Support for wide-character output and format strings, `swprintf()`, `vswprintf()`, `swscanf()`, and `vswscanf()`
- All the conversion functions (for example, `btowc`, `wctob`, `mbrtowc`, and `wcrtomb`) require `locale`
- `wcscoll` and `wcsxfrm` require `locale`.

wctype.h

Wide character library functions added to ISO C by *Normative Addendum 1* in 1994. This requires `locale`.

2.5 Tailoring the C library to a new execution environment

This section describes how to reimplement functions to produce an application for a different execution environment, for example, embedded in ROM or used with an RTOS.

Symbols that start with a single or double underscore name functions that are used as part of the low-level implementation. You can reimplement some of these functions.

Additional information on these library functions is available in the `rt_heap.h`, `rt_locale.h`, `rt_misc.h`, and `rt_sys.h` include files and the `rt_memory.s` assembler file.

See the following specifications for descriptions of functions that have the prefix `__cxa` or `__aeabi`:

- *C Library ABI for the ARM Architecture*
- *Exception Handling ABI for the ARM Architecture*
- *C++ ABI for the ARM Architecture*.

2.5.1 How C and C++ programs use the library functions

This section describes:

- specific library functions that are used to initialize the execution environment and application
- library exit functions
- target-dependent library functions that the application itself might call during its execution.

Initializing the execution environment and executing the application

The entry point of a program is at `__main` in the C library where library code does the following:

1. Copies non root (RO and RW) execution regions from their load addresses to their execution addresses. Also, if any data sections are compressed, they are decompressed from the load address to the execution address. See the *Linker Reference Guide* for more information.
2. Zeroes ZI regions.
3. Branches to `__rt_entry`.

If you do not want the library to perform these actions, you can define your own `__main` that branches to `__rt_entry` as shown in Example 2-1 on page 2-33.

Example 2-1 __main and __rt_entry

```
    IMPORT __rt_entry
    EXPORT __main
    ENTRY
__main
    B     __rt_entry
    END
```

The library function `__rt_entry()` runs the program as follows:

1. Calls `__rt_stackheap_init()` to set up the stack and heap.
2. Calls `__rt_lib_init()` to initialize referenced library functions, initialize the locale and, if necessary, set up `argc` and `argv` for `main()`.

   For C++, calls the constructors for any top-level objects by way of `__cpp_initialize__aeabi_`. See *C++ initialization, construction, and destruction* for more information.
3. Calls `main()`, the user-level root of the application.

   From `main()`, your program might call, among other things, library functions. See *Library functions called from main()* on page 2-36 for more information.
4. Calls `exit()` with the value returned by `main()`.

C++ initialization, construction, and destruction

C++ places certain requirements on the construction and destruction of objects with static storage duration. See *section 3.6 of the C++ Standard*.

The ARM C++ compiler uses the `.init_array` area to achieve this. This is a const data array of self-relative pointers to functions. For example, you might have the following C++ translation unit, contained in the file `test.cpp`:

```
struct T
{
    T();
    ~T();
} t;
int f()
{
    return 4;
}
int i = f();
```

This translates into the following pseudocode:

```
        AREA ||.text||, CODE, READONLY
     int f()
    {
        return 4;
    }
     static void __sti___8_test_cpp
     {
        // construct 't' and register its destruction
        __aeabi_atexit(T::T(&t), &T::~T, &__dso_handle);
        i = f();
     }
        AREA ||.init_array||, DATA, READONLY
        DCD __sti___8_test_cpp - {PC}
        AREA ||.data||, DATA
     t  % 4
     i  % 4
```

This pseudocode is for illustration only. To see the code that is generated, compile the C++ source code with `armcc -c --cpp -S`.

The linker collects each `.init_array` from the various translation units together. It is important that the `.init_array` is accumulated in the same order.

The library routine `__cpp_initialize__aeabi_` is called from the C library startup code, `__rt_lib_init`, before `main`. `__cpp_initialize__aeabi_` walks through the `.init_array` calling each function in turn. On exit, `__rt_lib_shutdown` calls `__cxa_finalize`.

Usually there is at most one function for `T::T()`, mangled name `_ZN1TC1Ev`, one function for `T::~T()`, mangled name `_ZN1TD1Ev`, one `__sti__` function, and four bytes of `.init_array` for each translation unit. The mangled name for the function `f()` is `_Z1fv`. There is no way to determine the initialization order between translation units.

Function-local static objects with destructors are also handled using `__aeabi_atexit`.

`.init_array` sections must be placed contiguously within the same region, for their base/limit symbols to be accessible. If they are not, the linker generates an error.

Legacy support

In RVCT v2.0 and earlier, `C$$pi_ctorvec` is used instead of `.init_array`. Objects with `C$$pi_ctorvec` are still supported. Therefore, if you have legacy objects, your scatter file is expected to contain:

```
LOAD_ROM 0x0000000
{
    EXEC_ROM 0x0000000
    {
        your_object.o(+RO)
        * (.init_array)
```

```
        * (C$$pi_ctorvec)        ; backwards compatibility
        ...
    }
    RAM 0x0100000
    {
        * (+RW,+ZI)
    }
}
```

Exceptions system initialization

The exceptions system can be initialized either on demand (that is, when first used), or before `main` is entered. Initialization on demand has the advantage of not allocating heap memory unless the exceptions system is used, but has the disadvantage that it becomes impossible to throw any exception (such as `std::bad_alloc`) if the heap is exhausted at the time of first use.

The default is to initialize on demand. To initialize the exceptions system before `main` is entered, include the following function in the link:

```
extern "C" void __cxa_get_globals(void);
extern "C" void __ARM_exceptions_init(void)
{
    __cxa_get_globals();
}
```

Although you can place the call to `__cxa_get_globals` directly in your code, placing it in `__ARM_exceptions_init` ensures that it is called as early as possible. That is, before any global variables are initialized and before `main` is entered.

`__ARM_exceptions_init` is weakly referenced by the library initialization mechanism, and is called if it is present as part of `__rt_lib_init`.

Note

The exception system is initialized by calls to various library functions, for example, `std::set_terminate()`. Therefore, you might not have to initialize before the entry to `main`.

Emergency buffer memory for exceptions

You can choose whether or not to allocate emergency memory that is to be used for throwing a `std::bad_alloc` exception when the heap is exhausted.

To allocate emergency memory, you must include the symbol `__ARM_exceptions_buffer_required` in the link. A call is then made to `__ARM_exceptions_buffer_init()` as part of the exceptions system initialization. The symbol is not included by default.

The following routines manage the exceptions emergency buffer:

`extern "C" void *__ARM_exceptions_buffer_init()`

Called once during runtime, to allocate the emergency buffer memory. It returns a pointer to the emergency buffer memory, or `NULL` if no memory is allocated.

`extern "C" void *__ARM_exceptions_buffer_allocate(void *buffer, size_t size)`

Called when an exception is about to be thrown, but there is not enough heap memory available to allocate the exceptions object. *`buffer`* is the value previously returned by `__ARM_exceptions_buffer_init()`, or `NULL` if that routine was not called. `__ARM_exceptions_buffer_allocate()` returns a pointer to *`size`* bytes of memory that is aligned on an eight-byte boundary, or `NULL` if the allocation is not possible.

`extern "C" void *__ARM_exceptions_buffer_free(void *buffer, void *addr)`

Called to free memory possibly allocated by `__ARM_exceptions_buffer_allocate()`. *`buffer`* is the value previously returned by `__ARM_exceptions_buffer_init()`, or `NULL` if that routine was not called. The routine determines whether the passed address has been allocated from the emergency memory buffer, and if so, frees it appropriately, then returns a non-`NULL` value. If the memory at *`addr`* was not allocated by `__ARM_exceptions_buffer_allocate()`, the routine must return `NULL`.

Default definitions of these routines are present in the image, but you can supply your own versions to override the defaults supplied by the library. The default routines reserve enough space for a single `std::bad_alloc` exceptions object. If you do not require an emergency buffer, it is safe to redefine all these routines to return only `NULL`.

Library functions called from main()

The function `main()` is the user-level root of the application. It requires that the execution environment is initialized, and that input/output functions can be called. While in `main()` the program might perform one of the following actions that calls user-customizable functions in the C library:

- Extend the stack or heap. See *Tailoring the runtime memory model* on page 2-69.

- Call library functions that require a callout to a user-defined function, for example `__rt_fp_status_addr()` or `clock()`. See *Tailoring other C library functions* on page 2-92.
- Call library functions that use `locale` or `CTYPE`. See *Tailoring locale and CTYPE using assembler macros* on page 2-42.
- Perform floating-point calculations that require the fpu or floating-point library.
- Input or output directly through low-level functions, for example `putc()`, or indirectly through high-level input/output functions and input/output support functions, for example, `fprintf()` or `sys_open()`. See *Tailoring the input/output functions* on page 2-78.
- Raise an error or other signal, for example `ferror`. See *Tailoring error signaling, error handling, and program exit* on page 2-59.

2.5.2 __rt_entry

The symbol `__rt_entry` is the starting point for a program using the ARM C library. Control passes to `__rt_entry` after all scatter-load regions have been relocated to their execution addresses.

Usage

The default implementation of `__rt_entry`:

1. Sets up the heap and stack.
2. Initializes the C library, by calling `__rt_lib_init`.
3. Calls `main()`.
4. Shuts down the C library, by calling `__rt_lib_shutdown`.
5. Exits.

`__rt_entry` must end with a call to one of the following functions:

`exit()` Calls `atexit()`-registered functions and shuts down the library.

`__rt_exit()` Shuts down the library but does not call `atexit()` functions.

`_sys_exit()` Exits directly to the execution environment. It does not shut down the library and does not call `atexit()` functions. See *_sys_exit()* on page 2-60.

2.5.3 Exiting from the program

The program can exit normally at the end of `main()` or it can exit prematurely because of an error.

Exiting from an assert

The behavior of the `assert` macro depends on the conditions in operation at the most recent occurrence of `#include <assert.h>`:

1. If the `NDEBUG` macro is defined (on the command line or as part of a source file), the `assert` macro has no effect.
2. If the `NDEBUG` macro is not defined, the `assert` expression (the expression given to the `assert` macro) is evaluated. If the result is `TRUE`, that is `!= 0`, the `assert` macro has no more effect.
3. If the `assert` expression evaluates to `FALSE`, the `assert` macro calls the `__aeabi_assert()` function if any of the following are true:
   - you are compiling with `--strict`
   - you are using `-O0` or `-O1`
   - you are compiling with `--library_interface=aeabi_clib` or `--library_interface=aeabi_glibc`
   - `__ASSERT_MSG` is defined
   - `_AEABI_PORTABILITY_LEVEL` is defined and not 0.
4. Otherwise, if the `assert` expression evaluates to `FALSE` and the conditions specified in point 3 do not apply, the `assert` macro calls `abort()`. Then:
   a. `abort()` calls `__rt_raise()`.
   b. If `__rt_raise()` returns, `abort()` tries to finalize the library.

If you are creating an application that does not use the library, `__aeabi_assert()` works if you reimplement `abort()` and the `stdio` functions.

Another solution for retargeting is to reimplement the `__aeabi_assert()` function itself. The function prototype is:

```
void __aeabi_assert(const char *expr, const char *file, int line);
```

where:

- *expr* points to the string representation of the expression that was not `TRUE`
- *file* and *line* identify the source location of the assertion.

The behavior for `__aeabi_assert()` supplied in the ARM C library is to print a message on `stderr` and call `abort()`.

You can restore the default behavior for `__aeabi_assert()` at higher optimization levels by defining `__ASSERT_MSG`.

2.5.4 __rt_exit()

This function shuts down the library but does not call functions registered with `atexit()`.

Syntax

```
void __rt_exit(int code)
```

Where *code* is not used by the standard function.

Usage

Shuts down the C library by calling `__rt_lib_shutdown`, and then calls `_sys_exit` to terminate the application. Reimplement `_sys_exit` rather than `__rt_exit`. See *_sys_exit()* on page 2-60 for more information.

Return

The function does not return.

2.5.5 __rt_lib_init()

This is the library initialization function and is the companion to `__rt_lib_shutdown()`.

Syntax

```
extern value_in_regs struct __argc_argv __rt_lib_init(unsigned heapbase,
unsigned heaptop)
```

where:

heapbase The start of the heap memory block.

heaptop The end of the heap memory block.

Usage

This is the library initialization function. It is called immediately after `__rt_stackheap_init()` and passed an initial chunk of memory to use as a heap. This function is the standard ARM library initialization function and must not be reimplemented.

Return

The function returns `argc` and `argv` ready to be passed to `main()`. The structure is returned in the registers as:

```
struct __argc_argv
{   int argc;
    char **argv;
    int r2, r3;
};
```

2.5.6 __rt_lib_shutdown()

This is the library shutdown function and is the companion to `__rt_lib_init()`.

Syntax

```
void __rt_lib_shutdown(void)
```

Usage

This is the library shutdown function and is provided in case a user must call it directly. This is the standard ARM library shutdown function and must not be reimplemented.

2.6 Tailoring static data access

This section describes using callouts from the C library to access static data. C library functions that use static data can be categorized as follows:

- functions that do not use any static data of any kind, for example `fprintf()`
- functions that manage a static state, such as `malloc()`, `rand()`, and `strtok()`
- functions that do not manage a static state, but use static data in a way that is specific to the implementation in the ARM compiler, for example `isalpha()`.

When the C library does something that requires implicit static data, it uses a callout to a function you can replace. These functions are shown in Table 2-6. They do not use semihosting.

Table 2-6 C library callouts

Function	Description
`__rt_errno_addr()`	Called to get the address of the variable `errno`. See *__rt_errno_addr()* on page 2-61.
`__rt_fp_status_addr()`	Called by the floating-point support code to get the address of the floating-point status word. See *__rt_fp_status_addr()* on page 2-63.
`locale` functions	The function `__user_libspace()` creates a block of private static data for the library. See *Tailoring locale and CTYPE using assembler macros* on page 2-42, and *Writing reentrant and thread-safe code* on page 2-4.

See also *Tailoring the runtime memory model* on page 2-69 for more information about memory use.

The default implementation of `__user_libspace` creates a 96-byte block in the ZI segment. Even if your application does not have a `main()` function, the `__user_libspace()` function does not normally have to be redefined. However, if you are writing an operating system or a process switcher, you must reimplement this function (see *Writing reentrant and thread-safe code* on page 2-4).

Note

Exactly which functions use static data in their definitions might change in future releases.

2.7 Tailoring locale and CTYPE using assembler macros

This section describes the use of assembler macros for tailoring locale functions. Applications use locales when they display or process data that is dependent on the local language or region, for example character set, monetary symbols, decimal point, time, and date.

See the `rt_locale.s` include file for more information on locale-related functions.

2.7.1 Selecting locale at link time

The locale subsystem of the C library can be selected at link time or extended to be selectable at runtime. The following describes the use of locale categories by the library:

- The default implementation of each locale category is for the C locale. The library also provides an alternative, ISO8859-1 (Latin-1 alphabet) implementation of each locale category that you can select at link time.
- Both the C and ISO8859-1 default implementations usually provide one locale for each category to select at runtime.
- You can replace each locale category individually.
- You can include as many locales in each category as you choose and you can name your locales as you choose.
- Each locale category uses one word in the private static data of the library.
- The locale category data is read-only and position independent.
- `scanf()` forces the inclusion of the `LC_CTYPE` locale category, but in either of the default locales this adds only 260 bytes of read-only data to several kilobytes of code.

ISO8859-1 Implementation

Table 2-7 shows the ISO8859-1 (Latin-1 alphabet) locale categories.

Table 2-7 Default ISO8859-1 locales

Symbol	Description
`__use_iso8859_ctype`	Selects the ISO8859-1 (Latin-1) classification of characters. This is essentially 7-bit ASCII, except that the character codes 160-255 represent a selection of useful European punctuation characters, letters, and accented letters.
`__use_iso8859_collate`	Selects the `strcoll`/`strxfrm` collation table appropriate to the Latin-1 alphabet. The default C locale does not require a collation table.
`__use_iso8859_monetary`	Selects the Sterling monetary category using Latin-1 coding.
`__use_iso8859_numeric`	Selects separating thousands with commas in the printing of numeric values.
`__use_iso8859_locale`	Selects all the ISO8859-1 selections described in this table.

There is no ISO8859-1 version of the `LC_TIME` category.

Shift-JIS and UTF-8 Implementation

Table 2-8 shows the Shift-JIS (Japanese characters) or UTF-8 (Unicode characters) locale categories.

Table 2-8 Default Shift-JIS and UTF-8 locales

Function	Description
`__use_sjis_ctype`	Sets the character set to the Shift-JIS multibyte encoding of Japanese characters
`__use_utf8_ctype`	Sets the character set to the UTF-8 multibyte encoding of all Unicode characters

The following describes the effects of Shift-JIS encoding:

- The ordinary `ctype` functions behave correctly on any byte value that is a self-contained character in Shift-JIS. For example, half-width katakana characters, that Shift-JIS encodes as single bytes between `0xA6` and `0xDF`, are treated as alphabetic by `isalpha()`.

- The multibyte conversion functions, such as `mbrtowc()`, `mbsrtowcs()`, and `wcrtomb()`, all convert between wide strings in Unicode and multibyte character strings in Shift-JIS.
- `printf("%ls")` converts a Unicode wide string into Shift-JIS output, and `scanf("%ls")` converts Shift-JIS input into a Unicode wide string.

You can arbitrarily switch between multibyte locales and single-byte locales at runtime if you include more than one in your application. By default, only one locale at a time is included.

2.7.2 Selecting locale at runtime

The C library function `setlocale()` selects a locale at runtime for the locale category, or categories, specified in its arguments. It does this by selecting the requested locale separately in each locale category. In effect, each locale category is a small filing system containing an entry for each locale.

C header files describing what must be implemented, and providing some useful support macros, are given in `rt_locale.h` and `rt_locale.s`.

2.7.3 Defining a locale block

The locale data blocks are defined using a set of assembly language macros provided in `rt_locale.s`. Therefore, the recommended way to define locale blocks is by writing an assembly language source file. RVCT provides a set of macros for each type of locale data block, for example `LC_CTYPE`, `LC_COLLATE`, `LC_MONETARY`, `LC_NUMERIC`, and `LC_TIME`. You define each locale block in the same way with a `_begin` macro, some data macros and an `_end` macro.

Specifying the beginning

To begin defining your locale block, you call the `_begin` macro. This macro takes two arguments, a prefix and the textual name. For example:

`LC_`*`TYPE`*`_begin` *`prefix`*`,` *`name`*

where:

`TYPE` is one of the following:

- `CTYPE`
- `COLLATE`
- `MONETARY`
- `NUMERIC`
- `TIME`

prefix is the prefix for the assembler symbols defined within the locale data

name is the textual name for the locale data.

Specifying the data

To specify the data for your locale block, you call the macros for that locale type in the order specified in the documentation. For example:

```
LC_TYPE_function
```

Where:

TYPE is one of the following:

- CTYPE
- COLLATE
- MONETARY
- NUMERIC
- TIME

function is a specific function related to your locale data.

When specifying locale data, you must call the macro repeatedly for each respective function.

Specifying the ending

To complete the definition of your locale data block, you call the _end macro. This macro takes no arguments. For example:

```
LC_TYPE_end
```

where:

TYPE is one of the following:

- CTYPE
- COLLATE
- MONETARY
- NUMERIC
- TIME

Specifying a fixed locale

To write a fixed function that always returns the same locale, you can use the _start symbol name defined by the macros. Example 2-2 on page 2-46 shows how this is implemented for the CTYPE locale.

Example 2-2 Fixed locale

```
    GET rt_locale.s
    AREA my_locales, DATA, READONLY
    LC_CTYPE_begin my_ctype_locale, "MyLocale"
    ...                          ; include other LC_CTYPE_xxx macros here
    LC_CTYPE_end
    AREA my_locale_func, CODE, READONLY
_get_lc_ctype FUNCTION
    LDR r0, =my_ctype_locale_start
    BX lr
    ENDFUNC
```

Specifying multiple locales

Contiguous locale blocks suitable for passing to the `_findlocale()` function must be declared in sequence. You must call the macro `LC_index_end` to end the sequence of locale blocks. Example 2-3 shows how this is implemented for the `CTYPE` locale.

Example 2-3 Multiple locales

```
    GET rt_locale.s
    AREA my_locales, DATA, READONLY
my_ctype_locales
    LC_CTYPE_begin my_first_ctype_locale, "MyLocale1"
    ...                          ; include other LC_CTYPE_xxx macros here
    LC_CTYPE_end
    LC_CTYPE_begin my_second_ctype_locale, "MyLocale2"
    ...                          ; include other LC_CTYPE_xxx macros here
    LC_CTYPE_end
    LC_index_end
    AREA my_locale_func, CODE, READONLY
    IMPORT _findlocale
_get_lc_ctype FUNCTION
    LDR r0, =my_ctype_locales
    B _findlocale
    ENDFUNC
```

2.7.4 LC_CTYPE data block

The macros that define an `LC_CTYPE` data block are as follows:

1. Call `LC_CTYPE_begin` with a symbol name and a locale name. See *Specifying the beginning* on page 2-44 for more information.

2. Call LC_CTYPE_table repeatedly to specify 256 table entries. LC_CTYPE_table takes a single argument in quotes. This must be a comma-separated list of table entries. Each table entry describes one of the 256 possible characters, and can be either an illegal character (IL) or a combination sum using any of the following flags:

   __S whitespace characters

   __P punctuation characters

   __B printable space characters

   __L lowercase letters

   __U uppercase letters

   __N decimal digits

   __C control characters

   __X hexadecimal digit letters A-F and a-f

   __A alphabetic but neither uppercase nor lowercase, such as Japanese katakana.

   Note

   A printable space character is defined as any character where the result of both isprint() and isspace() is true.

   __A must not be specified for the same character as either __N or __X.

   See *Specifying the data* on page 2-45 for more information.

3. If required, call one or both of the following optional macros:

   - LC_CTYPE_full_wctype. Calling this macro without arguments causes the C99 wide-character ctype functions (iswalpha(), iswupper(), ...) to return useful values across the full range of Unicode when this LC_CTYPE locale is active. If this macro is not specified, the wide ctype functions treat the first 256 wchar_t values as the same as the 256 char values, and the rest of the wchar_t range as containing illegal characters.
   - LC_CTYPE_multibyte defines this locale to be a multi-byte character set. Call this macro with three arguments. The first two arguments are the names of functions that perform conversion between the multi-byte character set and Unicode wide characters. The last argument is the value that must be taken by the C macro MB_CUR_MAX for the respective character set. The two function arguments have the following prototypes:

```
size_t internal_mbrtowc(wchar_t *pwc, char c, mbstate_t *pstate);
size_t internal_wcrtomb(char *s, wchar_t w, mbstate_t *pstate);
```

`internal_mbrtowc()`

takes one byte `c` as input, and updates the `mbstate_t` pointed to by `pstate` as a result of reading that byte. If the byte completes the encoding of a multibyte character, it writes the corresponding wide character into the location pointed to by `pwc`, and returns 1 to indicate that it has done so. If not, it returns -2 to indicate the state change of `mbstate_t` and that no character is output. Otherwise, it returns -1 to indicate that the encoded input is invalid.

`internal_wcrtomb()`

takes one wide character `w` as input, and writes some number of bytes into the memory pointed to by `s`. It returns the number of bytes output, or -1 to indicate that the input character has no valid representation in the multibyte character set.

4. Call `LC_CTYPE_end`, without arguments, to finish the locale block definition. See *Specifying the ending* on page 2-45 for more information.

Example 2-4 shows an `LC_CTYPE` data block.

Example 2-4 Defining the CTYPE locale

```
LC_CTYPE_begin utf8_ctype, "UTF-8"
;
; Single-byte characters in the low half of UTF-8 are exactly
; the same as in the normal "C" locale.
LC_CTYPE_table "__C, __C, __C, __C, __C, __C, __C, __C, __C" ; 0x00-0x08
LC_CTYPE_table "__C+__S, __C+__S, __C+__S, __C+__S, __C+__S"
                                          ; 0x09-0x0D(BS,LF,VT,FF,CR)
LC_CTYPE_table "__C, __C, __C, __C, __C, __C, __C, __C, __C" ; 0x0E-0x16
LC_CTYPE_table "__C, __C, __C, __C, __C, __C, __C, __C, __C" ; 0x17-0x1F
LC_CTYPE_table "__B+__S" ; space
LC_CTYPE_table "__P, __P, __P, __P, __P, __P, __P, __P" ; !"#$%&'(
LC_CTYPE_table "__P, __P, __P, __P, __P, __P, __P" ; )*+,-./
LC_CTYPE_table "__N, __N, __N, __N, __N, __N, __N, __N, __N, __N" ; 0-9
LC_CTYPE_table "__P, __P, __P, __P, __P, __P, __P" ; :;<=>?@
LC_CTYPE_table "__U+__X, __U+__X, __U+__X, __U+__X, __U+__X, __U+__X" ; A-F
LC_CTYPE_table "__U, __U, __U, __U, __U, __U, __U, __U, __U, __U" ; G-P
LC_CTYPE_table "__U, __U, __U, __U, __U, __U, __U, __U, __U, __U" ; Q-Z
LC_CTYPE_table "__P, __P, __P, __P, __P, __P" ; [\]^_`
LC_CTYPE_table "__L+__X, __L+__X, __L+__X, __L+__X, __L+__X, __L+__X" ; a-f
LC_CTYPE_table "__L, __L, __L, __L, __L, __L, __L, __L, __L, __L" ; g-p
LC_CTYPE_table "__L, __L, __L, __L, __L, __L, __L, __L, __L, __L" ; q-z
```

```
LC_CTYPE_table "__P, __P, __P, __P" ; {|}~
LC_CTYPE_table "__C" ; 0x7F
;
; Nothing in the top half of UTF-8 is valid on its own as a
; single-byte character, so they are all illegal characters (IL).
LC_CTYPE_table "IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL"
LC_CTYPE_table "IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL"
LC_CTYPE_table "IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL"
LC_CTYPE_table "IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL"
LC_CTYPE_table "IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL"
LC_CTYPE_table "IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL"
LC_CTYPE_table "IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL"
LC_CTYPE_table "IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL"
;
; The UTF-8 ctype locale wants the full version of wctype.
LC_CTYPE_full_wctype
;
; UTF-8 is a multi-byte locale, so we must specify some
; conversion functions. MB_CUR_MAX is 6 for UTF-8 (the lead
; bytes 0xFC and 0xFD are each followed by five continuation
; bytes).
;
; The implementations of the conversion functions are not
; provided in this example.
;
IMPORT  utf8_mbrtowc
IMPORT  utf8_wcrtomb
LC_CTYPE_multibyte utf8_mbrtowc, utf8_wcrtomb, 6
LC_CTYPE_end
```

2.7.5 LC_COLLATE data block

The macros that define an LC_COLLATE data block are as follows:

1. Call LC_COLLATE_begin with a symbol name and a locale name. See *Specifying the beginning* on page 2-44 for more information.

2. Call one of the following alternative macros:

   - Call LC_COLLATE_table repeatedly to specify 256 table entries. LC_COLLATE_table takes a single argument in quotes. This must be a comma-separated list of table entries. Each table entry describes one of the 256 possible characters, and can be a number indicating its position in the sorting order. For example, if character A is intended to sort before B, then entry 65 (corresponding to A) in the table, must be smaller than entry 66 (corresponding to B).

- Call `LC_COLLATE_no_table` without arguments. This indicates that the collation order is the same as the string comparison order. Therefore, `strcoll()` and `strcmp()` are identical.

See *Specifying the data* on page 2-45 for more information.

3. Call `LC_COLLATE_end`, without arguments, to finish the locale block definition. See *Specifying the ending* on page 2-45 for more information.

Example 2-5 shows an `LC_COLLATE` data block.

Example 2-5 Defining the COLLATE locale

```
LC_COLLATE_begin iso88591_collate, "ISO8859-1"
LC_COLLATE_table "0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07"
LC_COLLATE_table "0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f"
LC_COLLATE_table "0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17"
LC_COLLATE_table "0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f"
LC_COLLATE_table "0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27"
LC_COLLATE_table "0x28, 0x29, 0x2a, 0x2b, 0x2c, 0x2d, 0x2e, 0x2f"
LC_COLLATE_table "0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37"
LC_COLLATE_table "0x38, 0x39, 0x3a, 0x3b, 0x3c, 0x3d, 0x3e, 0x3f"
LC_COLLATE_table "0x40, 0x41, 0x49, 0x4a, 0x4c, 0x4d, 0x52, 0x53"
LC_COLLATE_table "0x54, 0x55, 0x5a, 0x5b, 0x5c, 0x5d, 0x5e, 0x60"
LC_COLLATE_table "0x67, 0x68, 0x69, 0x6a, 0x6b, 0x6c, 0x71, 0x72"
LC_COLLATE_table "0x73, 0x74, 0x76, 0x79, 0x7a, 0x7b, 0x7c, 0x7d"
LC_COLLATE_table "0x7e, 0x7f, 0x87, 0x88, 0x8a, 0x8b, 0x90, 0x91"
LC_COLLATE_table "0x92, 0x93, 0x98, 0x99, 0x9a, 0x9b, 0x9c, 0x9e"
LC_COLLATE_table "0xa5, 0xa6, 0xa7, 0xa8, 0xaa, 0xab, 0xb0, 0xb1"
LC_COLLATE_table "0xb2, 0xb3, 0xb6, 0xb9, 0xba, 0xbb, 0xbc, 0xbd"
LC_COLLATE_table "0xbe, 0xbf, 0xc0, 0xc1, 0xc2, 0xc3, 0xc4, 0xc5"
LC_COLLATE_table "0xc6, 0xc7, 0xc8, 0xc9, 0xca, 0xcb, 0xcc, 0xcd"
LC_COLLATE_table "0xce, 0xcf, 0xd0, 0xd1, 0xd2, 0xd3, 0xd4, 0xd5"
LC_COLLATE_table "0xd6, 0xd7, 0xd8, 0xd9, 0xda, 0xdb, 0xdc, 0xdd"
LC_COLLATE_table "0xde, 0xdf, 0xe0, 0xe1, 0xe2, 0xe3, 0xe4, 0xe5"
LC_COLLATE_table "0xe6, 0xe7, 0xe8, 0xe9, 0xea, 0xeb, 0xec, 0xed"
LC_COLLATE_table "0xee, 0xef, 0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5"
LC_COLLATE_table "0xf6, 0xf7, 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd"
LC_COLLATE_table "0x42, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48, 0x4b"
LC_COLLATE_table "0x4e, 0x4f, 0x50, 0x51, 0x56, 0x57, 0x58, 0x59"
LC_COLLATE_table "0x77, 0x5f, 0x61, 0x62, 0x63, 0x64, 0x65, 0xfe"
LC_COLLATE_table "0x66, 0x6d, 0x6e, 0x6f, 0x70, 0x75, 0x78, 0xa9"
LC_COLLATE_table "0x80, 0x81, 0x82, 0x83, 0x84, 0x85, 0x86, 0x89"
LC_COLLATE_table "0x8c, 0x8d, 0x8e, 0x8f, 0x94, 0x95, 0x96, 0x97"
LC_COLLATE_table "0xb7, 0x9d, 0x9f, 0xa0, 0xa1, 0xa2, 0xa3, 0xff"
LC_COLLATE_table "0xa4, 0xac, 0xad, 0xae, 0xaf, 0xb4, 0xb8, 0xb5"
LC_COLLATE_end
```

2.7.6 LC_MONETARY data block

The macros that define an LC_MONETARY data block are as follows:

1. Call LC_MONETARY_begin with a symbol name and a locale name. See *Specifying the beginning* on page 2-44 for more information.
2. Call the LC_MONETARY data macros as follows:
   a. Call LC_MONETARY_fracdigits with two arguments: frac_digits and int_frac_digits from the lconv structure.
   b. Call LC_MONETARY_positive with four arguments: p_cs_precedes, p_sep_by_space, p_sign_posn and positive_sign.
   c. Call LC_MONETARY_negative with four arguments: n_cs_precedes, n_sep_by_space, n_sign_posn and negative_sign.
   d. Call LC_MONETARY_currsymbol with two arguments: currency_symbol and int_curr_symbol.
   e. Call LC_MONETARY_point with one argument: mon_decimal_point.
   f. Call LC_MONETARY_thousands with one argument: mon_thousands_sep.
   g. Call LC_MONETARY_grouping with one argument: mon_grouping.

   See *Specifying the data* on page 2-45 for more information.
3. Call LC_MONETARY_end, without arguments, to finish the locale block definition. See *Specifying the ending* on page 2-45 for more information.

Example 2-6 shows an LC_MONETARY data block.

Example 2-6 Defining the MONETARY locale

```
LC_MONETARY_begin c_monetary, "C"
LC_MONETARY_fracdigits 255, 255
LC_MONETARY_positive 255, 255, 255, ""
LC_MONETARY_negative 255, 255, 255, ""
LC_MONETARY_currsymbol "", ""
LC_MONETARY_point ""
LC_MONETARY_thousands ""
LC_MONETARY_grouping ""
LC_MONETARY_end
```

2.7.7 LC_NUMERIC data block

The macros that define an LC_NUMERIC data block are as follows:

1. Call LC_NUMERIC_begin with a symbol name and a locale name. See *Specifying the beginning* on page 2-44 for more information.
2. Call the LC_NUMERIC data macros as follows:
   a. Call LC_NUMERIC_point with one argument: decimal_point from lconv structure.
   b. Call LC_NUMERIC_thousands with one argument: thousands_sep.
   c. Call LC_NUMERIC_grouping with one argument: grouping.
3. Call LC_NUMERIC_end, without arguments, to finish the locale block definition. See *Specifying the ending* on page 2-45 for more information.

Example 2-7 shows an LC_NUMERIC data block.

Example 2-7 Defining the NUMERIC locale

```
LC_NUMERIC_begin c_numeric, “C”
LC_NUMERIC_point “.”
LC_NUMERIC_thousands ““
LC_NUMERIC_grouping ““
LC_NUMERIC_end
```

2.7.8 LC_TIME data block

The macros that define an LC_TIME data block are as follows:

1. Call LC_TIME_begin with a symbol name and a locale name. See *Specifying the beginning* on page 2-44 for more information.
2. Call the LC_TIME data macros as follows:
   a. Call LC_TIME_week_short seven times to provide the short names for the days of the week. Sunday being the first day. Then call LC_TIME_week_long and repeat the process for long names.
   b. Call LC_TIME_month_short twelve times to provide the short names for the days of the month. Then call LC_TIME_month_long and repeat the process for long names.
   c. Call LC_TIME_am_pm with two arguments that are respectively the strings representing morning and afternoon.

d. Call LC_TIME_formats with three arguments that are respectively the standard date/time format used in strftime("%c"), the standard date format strftime("%x"), and the standard time format strftime("%X"). These strings must define the standard formats in terms of other simpler strftime primitives. Example 2-8 shows that the standard date/time format is permitted to reference the other two formats.

e. Call LC_TIME_c99format with a single string that is the standard 12-hour time format used in strftime("%r") as defined in C99.

3. Call LC_TIME_end, without arguments, to finish the locale block definition. See *Specifying the ending* on page 2-45 for more information.

Example 2-8 shows an LC_TIME data block.

Example 2-8 Defining the TIME locale

```
LC_TIME_begin c_time, “C”
LC_TIME_week_short “Sun”
LC_TIME_week_short “Mon”
LC_TIME_week_short "Tue"
LC_TIME_week_short "Wed"
LC_TIME_week_short "Thu"
LC_TIME_week_short "Fri"
LC_TIME_week_short "Sat"
LC_TIME_week_long "Sunday"
LC_TIME_week_long "Monday"
LC_TIME_week_long "Tuesday"
LC_TIME_week_long "Wednesday"
LC_TIME_week_long "Thursday"
LC_TIME_week_long "Friday"
LC_TIME_week_long "Saturday"
LC_TIME_month_short "Jan"
LC_TIME_month_short "Feb"
LC_TIME_month_short "Mar"
LC_TIME_month_short "Apr"
LC_TIME_month_short "May"
LC_TIME_month_short "Jun"
LC_TIME_month_short "Jul"
LC_TIME_month_short "Aug"
LC_TIME_month_short "Sep"
LC_TIME_month_short "Oct"
LC_TIME_month_short "Nov"
LC_TIME_month_short "Dec"
LC_TIME_month_long "January"
LC_TIME_month_long "February"
LC_TIME_month_long "March"
LC_TIME_month_long "April"
```

```
LC_TIME_month_long "May"
LC_TIME_month_long "June"
LC_TIME_month_long "July"
LC_TIME_month_long "August"
LC_TIME_month_long "September"
LC_TIME_month_long "October"
LC_TIME_month_long "November"
LC_TIME_month_long "December"
LC_TIME_am_pm "AM", "PM"
LC_TIME_formats "%x %X", "%d %b %Y", "%H:%M:%S"
LC_TIME_c99format "%I:%M:%S %p"
LC_TIME_week_short “Sat”
LC_TIME_end
```

2.7.9 _get_lconv()

`_get_lconv()` sets the components of an `lconv` structure with values appropriate for the formatting of numeric quantities.

Syntax

```
void _get_lconv(struct lconv *lc);
```

Usage

This extension to ISO does not use any static data. If you are building an application that must conform strictly to the ISO C standard, use `localeconv()` instead.

Return

The existing `lconv` structure *lc* is filled with formatting data.

2.7.10 localeconv()

`localeconv()` creates and sets the components of an `lconv` structure with values appropriate for the formatting of numeric quantities according to the rules of the current locale.

Syntax

```
struct lconv *localeconv(void);
```

Usage

The members of the structure with type `char *` are strings. Any of these, except for `decimal_point`, can point to an empty string, `""`, to indicate that the value is not available in the current locale or is of zero length.

The members with type `char` are non-negative numbers. Any of the members can be `CHAR_MAX` to indicate that the value is not available in the current locale.

The members included in `lconv` are described in *The lconv structure* on page 2-56.

Return

The function returns a pointer to the filled-in object. The structure pointed to by the return value is not modified by the program, but might be overwritten by a subsequent call to the `localeconv()` function. In addition, calls to the `setlocale()` function with categories `LC_ALL`, `LC_MONETARY`, or `LC_NUMERIC` might overwrite the contents of the structure.

2.7.11 setlocale()

Selects the appropriate locale as specified by the *category* and *locale* arguments.

Syntax

```
char *setlocale(int category, const char *locale);
```

Usage

The `setlocale()` function is used to change or query part or all of the current locale. The effect of the category argument for each value is described in *Locale categories*. A value of `"C"` for *locale* specifies the minimal environment for C translation. An empty string, `""`, for *locale* specifies the implementation-defined native environment. At program startup, the equivalent of `setlocale(LC_ALL, "C")` is executed.

Locale categories

The values of *category* are:

`LC_COLLATE` Affects the behavior of `strcoll()`.

`LC_CTYPE` Affects the behavior of the character handling functions.

`LC_MONETARY` Affects the monetary formatting information returned by `localeconv()`.

LC_NUMERIC	Affects the decimal-point character for the formatted input/output functions and the string conversion functions and the numeric formatting information returned by localeconv().
LC_TIME	Can affect the behavior of strftime(). For currently supported locales, the option has no effect.
LC_ALL	Affects all locale categories. This is the bitwise OR of all the locale categories.

Return

If a pointer to a string is given for *locale* and the selection is valid, the string associated with the specified category for the new locale is returned. If the selection cannot be honored, a null pointer is returned and the locale is not changed.

A null pointer for *locale* causes the string associated with the category for the current locale to be returned and the locale is not changed.

If *category* is LC_ALL and the most recent successful locale-setting call uses a category other than LC_ALL, a composite string might be returned. The string returned when used in a subsequent call with its associated category restores that part of the program locale. The string returned is not modified by the program, but might be overwritten by a subsequent call to setlocale().

2.7.12 _findlocale()

_findlocale() searches the locale database and returns a pointer to the data block for the requested category and locale.

Syntax

```
void const *_findlocale(void const *index, const char *name);
```

Return

Returns a pointer to the requested data block.

2.7.13 The lconv structure

The lconv structure contains numeric formatting information. The structure is filled by the functions _get_lconv() and localeconv().

The definition of lconv from locale.h is shown in Example 2-9 on page 2-57.

Example 2-9 lconv structure

```
struct lconv {
  char *decimal_point;
       /* The decimal point character used to format non monetary quantities */
  char *thousands_sep;
       /* The character used to separate groups of digits to the left of the */
       /* decimal point character in formatted non monetary quantities.      */
  char *grouping;
       /* A string whose elements indicate the size of each group of digits  */
       /* in formatted non monetary quantities. See below for more details.  */
  char *int_curr_symbol;
       /* The international currency symbol applicable to the current locale.*/
       /* The first three characters contain the alphabetic international    */
       /* currency symbol in accordance with those specified in ISO 4217.    */
       /* Codes for the representation of Currency and Funds. The fourth     */
       /* character (immediately preceding the null character) is the        */
       /* character used to separate the international currency symbol from  */
       /* the monetary quantity.                                             */
  char *currency_symbol;
       /* The local currency symbol applicable to the current locale.        */
  char *mon_decimal_point;
       /* The decimal-point used to format monetary quantities.              */
  char *mon_thousands_sep;
       /* The separator for groups of digits to the left of the decimal-point*/
       /* in formatted monetary quantities.                                  */
  char *mon_grouping;
       /* A string whose elements indicate the size of each group of digits  */
       /* in formatted monetary quantities. See below for more details.      */
  char *positive_sign;
       /* The string used to indicate a non negative-valued formatted         */
       /* monetary quantity.                                                 */
  char *negative_sign;
       /* The string used to indicate a negative-valued formatted monetary   */
       /* quantity.                                                          */
  char int_frac_digits;
       /* The number of fractional digits (those to the right of the         */
       /* decimal-point) to be displayed in an internationally formatted     */
       /* monetary quantities.                                               */
  char frac_digits;
       /* The number of fractional digits (those to the right of the         */
       /* decimal-point) to be displayed in a formatted monetary quantity.   */
  char p_cs_precedes;
       /* Set to 1 or 0 if the currency_symbol respectively precedes or      */
       /* succeeds the value for a non negative formatted monetary quantity. */
  char p_sep_by_space;
       /* Set to 1 or 0 if the currency_symbol respectively is or is not     */
       /* separated by a space from the value for a non negative formatted   */
       /* monetary quantity.                                                 */
```

```
  char n_cs_precedes;
       /* Set to 1 or 0 if the currency_symbol respectively precedes or      */
       /* succeeds the value for a negative formatted monetary quantity.     */
  char n_sep_by_space;
       /* Set to 1 or 0 if the currency_symbol respectively is or is not     */
       /* separated by a space from the value for a negative formatted       */
       /* monetary quantity.                                                 */
  char p_sign_posn;
       /* Set to a value indicating the position of the positive_sign for a  */
       /* non negative formatted monetary quantity. See below for more details*/
  char n_sign_posn;
       /* Set to a value indicating the position of the negative_sign for a  */
       /* negative formatted monetary quantity. */
};
```

The elements of `grouping` and `non_grouping` (shown in Example 2-9 on page 2-57) are interpreted as follows:

`CHAR_MAX` No additional grouping is to be performed.

`0` The previous element is repeated for the remainder of the digits.

`other` The value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits to the left of the current group.

The value of `p_sign_posn` and `n_sign_posn` (shown in Example 2-9 on page 2-57) are interpreted as follows:

`0` Parentheses surround the quantity and currency symbol.

`1` The sign string precedes the quantity and currency symbol.

`2` The sign string is after the quantity and currency symbol.

`3` The sign string immediately precedes the currency symbol.

`4` The sign string immediately succeeds the currency symbol.

2.8 Tailoring error signaling, error handling, and program exit

All trap or error signals raised by the C library go through the `__raise()` function. You can reimplement this function or the lower-level functions that it uses.

Caution

The IEEE 754 standard for floating-point processing states that the default response to an exception is to proceed without a trap. You can modify floating-point error handling by tailoring the functions and definitions in `fenv.h`. See also Chapter 4 *Floating-point Support*.

See the `rt_misc.h` include file for more information on error-related functions.

The trap and error-handling functions are shown in Table 2-9. See also *Tailoring the C library to a new execution environment* on page 2-32 for additional information about application initialization and shutdown.

Table 2-9 Trap and error handling

Function	Description
`_sys_exit()`	Called, eventually, by all exits from the library. See *_sys_exit()* on page 2-60.
`errno`	Is a static variable used with error handling. See *errno* on page 2-60.
`__rt_errno_addr()`	Is called to obtain the address of the variable `errno`. See *__rt_errno_addr()* on page 2-61.
`__raise()`	Raises a signal to indicate a runtime anomaly. See *__raise()* on page 2-61.
`__rt_raise()`	Raises a signal to indicate a runtime anomaly. See *__rt_raise()* on page 2-62.
`__default_signal_handler()`	Displays an error indication to the user. See *__default_signal_handler()* on page 2-62.
`_ttywrch()`	Writes a character to the console. The default implementation of `_ttywrch()` is semihosted and, therefore, uses semihosting calls. See *_ttywrch()* on page 2-63.
`__rt_fp_status_addr()`	This function is called to obtain the address of the fp status word. See *__rt_fp_status_addr()* on page 2-63.

2.8.1 _sys_exit()

The library exit function. All exits from the library eventually call `_sys_exit()`.

Syntax

```
void _sys_exit(int return_code);
```

Usage

This function must not return. You can intercept application exit at a higher level by either:

- Implementing the C library function `exit()` as part of your application. You lose `atexit()` processing and library shutdown if you do this.
- Implementing the function `__rt_exit(int n)` as part of your application. You lose library shutdown if you do this, but `atexit()` processing is still performed when `exit()` is called or `main()` returns.

Return

The return code is advisory. An implementation might attempt to pass it to the execution environment.

2.8.2 errno

The C library `errno` variable is defined in the implicit static data area of the library. This area is identified by `__user_libspace()`. It occupies part of initial stack space used by the functions that established the runtime stack. The definition of `errno` is:

```
(*(volatile int *) __rt_errno_addr())
```

You can define `__rt_errno_addr()` if you want to place `errno` at a user-defined location instead of the default location identified by `__user_libspace()`. See also *The __user_libspace static data area* on page 2-6.

Return

The default implementation is a veneer on `__user_libspace()` that returns the address of the status word. A suitable default definition is given in the C library standard headers.

2.8.3 __rt_errno_addr()

This function is called to get the address of the C library `errno` variable when the C library attempts to read or write `errno`. The library provides a default implementation. It is unlikely that you have to reimplement this function.

Syntax

```
volatile int *__rt_errno_addr(void);
```

2.8.4 __raise()

This function raises a signal to indicate a runtime anomaly.

Syntax

```
int __raise(int signal, int type);
```

where:

signal Is an integer that holds the signal number.

type Is an integer or string constant or variable.

Usage

This function calls the normal C signal mechanism or the default signal handler. See also *_ttywrch()* on page 2-63 for more information.

You can replace the `__raise()` function by defining:

```
int __raise(int signal, int type);
```

This enables you to bypass the C signal mechanism and its data-consuming signal handler vector, but otherwise gives essentially the same interface as:

```
void __default_signal_handler(int signal, int type);
```

See also *Thread-safety in the ARM C libraries* on page 2-10.

Return

There are three possibilities for a `__raise()` return condition:

no return The handler performs a long jump or restart.

0 The signal was handled.

nonzero The calling code must pass that return value to the exit code. The default library implementation calls `_sys_exit(rc)` if `__raise()` returns a nonzero return code *rc*.

2.8.5 __rt_raise()

This function raises a signal to indicate a runtime anomaly.

Syntax

```
void __rt_raise(int signal, int type);
```

where:

signal Is an integer that holds the signal number.

type Is an integer or string constant or variable.

Usage

Redefine this to replace the entire signal handling mechanism for the library. The default implementation calls `__raise()`. See *__raise()* on page 2-61 for more information.

Depending on the value returned from `__raise()`:

no return The handler performed a long jump or restart and `__rt_raise()` does not regain control.

0 The signal was handled and `__rt_raise()` exits.

nonzero The default library implementation calls `_sys_exit(rc)` if `__raise()` returns a nonzero return code *rc*.

2.8.6 __default_signal_handler()

This function handles a raised signal. The default action is to print an error message and exit.

Syntax

```
void __default_signal_handler(int signal, int type);
```

Usage

The default signal handler uses `_ttywrch()` to print a message and calls `_sys_exit()` to exit. You can replace the default signal handler by defining:

```
void __default_signal_handler(int signal, int type);
```

The interface is the same as `__raise()`, but this function is only called after the C signal handling mechanism has declined to process the signal.

A complete list of the defined signals is in `signal.h`. See Table 2-13 on page 2-101 for those signals that are used by the libraries.

Note

The signals used by the libraries might change in future releases of the product.

2.8.7 _ttywrch()

This function writes a character to the console. The console might have been redirected. You can use this function as a last resort error handling routine.

Syntax

```
void __ttywrch(int ch);
```

Usage

The default implementation of this function uses semihosting.

You can redefine this function, or `__raise()`, even if there is no other input/output. For example, it might write an error message to a log kept in nonvolatile memory.

2.8.8 __rt_fp_status_addr()

This function returns the address of the floating-point status word.

Syntax

```
unsigned *__rt_fp_status_addr(void);
```

Usage

If `__rt_fp_status_addr()` is not defined, the default implementation from the C library is used. The value is initialized when `__rt_lib_init()` calls `_fp_init()`. The constants for the status word are listed in `fenv.h`. The default fp status is 0.

See also *Thread-safety in the ARM C libraries* on page 2-10.

2.9 Tailoring storage management

This section describes the functions from rt_heap.h that you can define if you are tailoring memory management.

See the rt_heap.h and rt_memory.s include files for more information on memory-related functions.

2.9.1 Avoiding the ARM-supplied heap and heap-using functions

If you are developing embedded systems that have limited RAM or that provide their own heap management (for example, an operating system), you might require a system that does not define a heap area. To avoid using the heap you can either:

- reimplement the functions in your own application
- write the application so that it does not call any heap-using function.

You can reference the __use_no_heap or __use_no_heap_region symbols in your code to guarantee that no heap-using functions are linked in from the ARM library. You are only required to import these symbols once in your application, for example, using either:

- IMPORT __use_no_heap from assembly language
- #pragma import(__use_no_heap) from C.

If you include a heap-using function and also reference __use_no_heap or __use_no_heap_region, the linker reports an error. For example, the following sample code results in the linker error shown:

```
#include <stdio.h>
#include <stdlib.h>
#pragma import(__use_no_heap)

void main()
{
    char *p = malloc(256);
    ...
}
```

```
Error: L6915E: Library reports error: __use_no_heap was requested, but malloc
was referenced
```

To find out which objects are using the heap, link with --verbose --list=out.txt, search the output for the relevant symbol (in this case malloc), and find out what object referenced it.

__use_no_heap guards against the use of malloc(), realloc(), free(), and any function that uses those functions. For example, calloc() and other stdio functions.

__use_no_heap_region has the same properties as __use_no_heap, but in addition, guards against other things that use the heap memory region. For example, if you declare main() as a function taking arguments, the heap region is used for collecting argc and argv.

2.9.2 Support for malloc

malloc(), realloc(), calloc(), and free() are built on a heap abstract data type. You can choose between Heap1 or Heap2, the two provided heap implementations.

The default implementations of malloc(), realloc(), and calloc() maintain an eight-byte aligned heap.

Heap1: Standard heap implementation

Heap1, the default implementation, implements the smallest and simplest heap manager. The heap is managed as a singly-linked list of free blocks held in increasing address order. The allocation policy is first-fit by address.

This implementation has low overheads, but the cost of malloc() or free() grows linearly with the number of free blocks. The smallest block that can be allocated is four bytes and there is an additional overhead of four bytes. If you expect more than 100 unallocated blocks it is recommended that you use Heap2.

Heap2: Alternative heap implementation

Heap2 provides a compact implementation with the cost of malloc() or free() growing logarithmically with the number of free blocks. The allocation policy is first-fit by address. The smallest block that can be allocated is 12 bytes and there is an additional overhead of four bytes.

Heap2 is recommended when you require near constant-time performance in the presence of hundreds of free blocks. To select the alternative standard implementation, use either:

- IMPORT __use_realtime_heap from assembly language
- #pragma import(__use_realtime_heap) from C.

Using Heap2

The Heap2 real-time heap implementation must know the maximum address space the heap spans. The smaller the address range, the more efficient the algorithm is.

By default, the heap extent is taken to be 16MB starting at the beginning of the heap (defined as the start of the first chunk of memory given to the heap manager by `__rt_initial_stackheap()` or `__rt_heap_extend()`).

The heap bounds are given by:

```
struct __heap_extent {
    unsigned base, range;
};

__value_in_regs struct __heap_extent __user_heap_extent(
             unsigned defaultbase, unsigned defaultsize);
```

The function prototype for `__user_heap_extent()` is in `rt_misc.h`.

The Heap1 algorithm does not require the bounds on the heap extent, therefore it never calls this function.

You must redefine `__user_heap_extent()` if:

- you require a heap to span more than 16MB of address space
- your memory model can supply a block of memory at a lower address than the first one supplied.

If you know in advance that the address space bounds of your heap are small, you do not have to redefine `__user_heap_extent()`, but it does speed up the heap algorithms if you do.

The input parameters are the default values that are used if this routine is not defined. You can, for example, leave the default base value unchanged and only adjust the size.

Note

The size field returned must be a power of two. The library does not check this and fails in unexpected ways if this requirement is not met. If you return a size of zero, the extent of the heap is set to 4GB.

Using a heap implementation from bare machine C

To use a heap implementation in an application that does not define `main()` and does not initialize the C library:

1. Call `_init_alloc(`*base*`, `*top*`)` to define the base and top of the memory you want to manage as a heap.
2. Define the function `unsigned __rt_heap_extend(unsigned size, void **block)` to handle calls to extend the heap when it becomes full.

alloca()

`alloca()` behaves identically to `malloc()` except that `alloca()` has automatic garbage collection (see *alloca()* on page 2-111).

2.10 Tailoring the runtime memory model

This section describes:

- the management of writable memory by the C library as static data, heap, and stack
- functions that can be redefined to change how writable memory is managed.

See *Placing the stack and heap* on page 3-13 in the *Developer Guide* for more information.

2.10.1 The memory models

You can select either of the following memory models:

Single memory region

The stack grows downward from the top of the memory region. The heap grows upwards from the bottom of the region. This is the default. The memory managed by the heap never shrinks. Heap memory that is freed by calling `free()` cannot be reused for other purposes.

Two memory regions

One memory region is for the stack and the other is for the heap. The size of the heap region can be zero. The stack region can be in allocated memory or inherited from the execution environment.

To use the two-region model rather than the default one-region model, use either:

- `IMPORT __use_two_region_memory` from assembly language
- `#pragma import(__use_two_region_memory)` from C.

Note

If you use the two-region memory model and do not provide any heap memory, you cannot call `malloc()`, use `stdio`, or get command-line arguments for `main()`.

If you set the size of the heap region to zero and define `__user_heap_extend()` as a function that can extend the heap, the heap is created when it is required.

See the description of `__use_no_heap` in *Tailoring storage management* on page 2-65, for more information on how to issue a warning message if the heap or heap region is used.

2.10.2 Controlling the runtime memory model

To modify the behavior of the heap and stack manager, you can use any of the following methods:

- use a scatter-loading description file
- redefine `__user_setup_stackheap()` and `__user_heap_extend()`
- define symbols to specify the intial stack pointer and the start and end of the heap.

For information about scatter-loading description files, see Chapter 5 *Using Scatter-loading Description Files* in the *Linker User Guide*.

`__user_setup_stackheap()` sets up and returns the locations of the initial stack and heap. `__user_heap_extend()` returns extra blocks of memory for the heap to use. You only have to redefine these functions if you:

- are not using a scatter-loading description file
- do not want the values used by the default implementation of these functions
- are not defining symbols `__initial_sp`, `__heap_base`, and `__heap_limit` to specify the initial stack pointer and the start and end of the heap.

Note

- You might see `__user_initial_stackheap()` instead of `__user_setup_stackheap()`. This is most likely to occur if you are maintaining legacy source code, because `__user_initial_stackheap()` is an older implementation.
- You are advised to replace any occurrences of `__user_initial_stackheap()` in your source code with `__user_setup_stackheap()`.

The hidden static data for the library is provided by `__user_libspace`. The static data area is also used as a stack during the library initialization process. This function does not normally require reimplementation. See *Tailoring static data access* on page 2-41.

For more information about redefining `__user_setup_stackheap()`, see *__user_setup_stackheap()* on page 2-71.

For more information about redefining `__user_heap_extend()`, see *__user_heap_extend()* on page 2-74.

For information about defining the symbols `__initial_sp`, `__heap_base`, and `__heap_limit`, see *Creating the stack* on page 3-6 and *Creating the heap* on page 3-6. This method of controlling the runtime memory model is supported by both standardlib and microlib.

2.10.3 Writing your own memory model

If the provided memory models do not meet your requirements, you can write your own. A memory model must define the functions described in Table 2-10. All functions are ARM state functions. The library takes care of entry from Thumb state if this is required. An incomplete prototype implementation for the model is provided in `rt_memory.s` located in the `...\include` directory.

Use the prototype as a starting point for your own implementation.

Table 2-10 Memory model functions

Function	Description
`__rt_stackheap_init()`	Creates and initializes the stack pointer. Returns a region of memory for use as the initial heap. An assembler-level function.
`__rt_heap_extend()`	Returns a new block of memory to add to the heap. See *__rt_heap_extend()* on page 2-76.

2.10.4 __user_setup_stackheap()

`__user_setup_stackheap()` sets up and returns the locations of the initial stack and heap.

Unlike `__user_initial_stackheap()`, `__user_setup_stackheap()` works with systems where the application starts with a value of `sp` (`r13`) that is already correct, for example, Cortex-M3. To make use of `sp`, implement `__user_setup_stackheap()` to set up `r0`, `r2`, and `r3` (for a two-region model) and return. For a one-region model, set only `r0`.

When `__user_setup_stackheap()` is called, `sp` has the same value it had on entry to the application. If this value is not valid then `__user_setup_stackheap()` must change the value of `sp` before using any stack.

Using `__user_setup_stackheap()` rather than `__user_initial_stackheap()` improves code size because there is no requirement for a temporary stack.

`__user_setup_stackheap()` returns the:

- heap base in `r0`
- stack base in `sp`
- heap limit in `r2`
- stack limit in `r3`.

Note

`__user_setup_stackheap()` must be reimplemented in assembler.

2.10.5 __user_initial_stackheap()

If you have legacy source code you might see `__user_initial_stackheap()`. This is an old function that is only supported for backwards compatibility with legacy source code. The modern equivalent is `__user_setup_stackheap()`.

Migrating to RVCT v4.0 from RVCT v2.x and earlier

In RVCT v2.x and earlier, the default implementation of `__user_initial_stackheap()` used the value of the symbol `Image$$ZI$$Limit`. This symbol is not defined if the linker uses a scatter-loading description file (specified with the `--scatter` command-line option) so `__user_initial_stackheap()` must be reimplemented if you are using scatter-loading description files, otherwise your link step fails.

Alternatively, you can upgrade your source code to use `__user_setup_stackheap()` instead of `__user_initial_stackheap()`.

Migrating to RVCT v4.0 from RVCT v3.x

In RVCT v3.x and later, the library includes more implementations of `__user_initial_stackheap()`, and can select the correct implementation for you automatically from information given in a scatter-loading description file. This means that it is not necessary to reimplement this function if you are using scatter-loading files. See *Using a scatter-loading description file* on page 2-73 for more information.

Syntax

```
extern __value_in_regs struct __user_initial_stackheap
__user_initial_stackheap(unsigned R0, unsigned SP, unsigned R2, unsigned SL);
```

Usage

`__user_initial_stackheap()` returns the:

- heap base in `r0`
- stack base in `r1`, that is, the highest address in the stack region
- heap limit in `r2`
- stack limit in `r3`, that is, the lowest address in the stack region.

If this function is reimplemented, it must:

- use no more than 88 bytes of stack
- not corrupt registers other than `r12` (`ip`)
- maintain eight-byte alignment of the heap.

For the default one-region model, the values in r2 and r3 are ignored and all memory between r0 and r1 is available for the heap. For a two-region model, the heap limit is set by r2 and the stack limit is set by r3.

The value of sp (r13) at the time __main() is called is passed as an argument in r1. The default implementation of __user_initial_stackheap(), using the semihosting SYS_HEAPINFO, is given by the library in module sys_stackheap.o.

To create a version of __user_initial_stackheap() that inherits sp from the execution environment and does not have a heap, set r0 and r2 to the value of r3 and return. See *__user_setup_stackheap()* on page 2-71) for more information.

The definition of __initial_stackheap in rt_misc.h is:

```
struct __initial_stackheap{
    unsigned heap_base, stack_base, heap_limit, stack_limit;
};
```

Note

The value of stack_base is 0x1 greater than the highest address used by the stack because a full-descending stack is used.

See the examples directory for example reimplementations of this function.

Return

The values returned in r0 to r3 depend on whether you are using the one- or two-region memory model:

One-region (r0,r1) is the single stack and heap region. r1 is greater than r0. r2 and r3 are ignored.

Two-region (r0, r2) is the initial heap and (r3, r1) is the initial stack. r2 is greater than or equal to r0. r3 is less than r1.

Using a scatter-loading description file

The default implementation of __user_initial_stackheap() uses the value of the symbol Image$$ZI$$Limit. This symbol is not defined if the linker uses a scatter-loading description file. However, the C library provides alternative implementations that you can make use of through information in scatter-loading description files.

Selecting the one-region model automatically

Define one special execution region in your scatter-loading description file, ARM_LIB_STACKHEAP. This region has the EMPTY attribute.

This causes the library to select an implementation of __user_initial_stackheap() that uses this as the combined heap/stack region. This uses the value of the symbols Image$$ARM_LIB_STACKHEAP$$Base and Image$$ARM_LIB_STACKHEAP$$ZI$$Limit.

Selecting the two-region model automatically

Define two special execution regions in your scatter-loading description file, ARM_LIB_HEAP and ARM_LIB_STACK. Both regions have the EMPTY attribute.

This causes the library to select an implementation of __user_initial_stackheap() that uses the value of the symbols Image$$ARM_LIB_HEAP$$Base, Image$$ARM_LIB_HEAP$$ZI$$Limit, Image$$ARM_LIB_STACK$$Base, and Image$$ARM_LIB_STACK$$ZI$$Limit.

An example scatter-loading description file to define both ARM_LIB_HEAP and ARM_LIB_STACK is supplied as scatter.scat. See *install_directory*\RVDS\Examples\...\Cortex-M3\Example2.

sp is initialized appropriately from either ARM_LIB_STACKHEAP (for the one-region model) or ARM_LIB_STACK (for the two-region model).

Error messages

If you use a scatter file but do not specify any special region names and do not reimplement __user_initial_stackheap(), the library generates an error message.

2.10.6 __user_heap_extend()

This function can be defined to return extra blocks of memory, separate from the initial one, to be used by the heap. If defined, this function must return the size and base address of an eight-byte aligned heap extension block.

Syntax

```
extern unsigned __user_heap_extend(int 0, void **base, unsigned requested_size);
```

Usage

There is no default implementation of this function. If you define this function, it must have the following characteristics:

- The returned size must be either:
  - a multiple of eight bytes of at least the requested size
  - 0, denoting that the request cannot be honored.
- Size is measured in bytes.
- The function is subject only to AAPCS constraints.
- The first argument is always zero on entry and can be ignored. The base is returned in the register holding this argument.
- The returned base address must be aligned on an eight-byte boundary.

Return

This function places a pointer to a block of at least the requested size in `*base` and returns the size of the block. `0` is returned if no such block can be returned, in which case the value stored at `*base` is never used.

2.10.7 __user_heap_extent()

If defined, this function returns the base address and maximum range of the heap.

Syntax

```
extern __value_in_regs struct __heap_extent __user_heap_extent(unsigned ignore1,
unsigned ignore2);
```

See also *Support for malloc* on page 2-66.

Usage

There is no default implementation of this function. The values of the parameters *ignore1* and *ignore2* are not used by the function.

2.10.8 __rt_stackheap_init()

This function sets up the stack pointer and returns a region of memory for use as the initial heap. It is called from the library initialization code.

On return from this function, SP must point to the top of the stack region, r0 must point to the base of the heap region, and r1 must point to the limit of the heap region.

A user-defined memory model (that is, __rt_stackheap_init() and __rt_heap_extend()) is allocated 16 bytes of storage from the __user_perproc_libspace area if wanted. It accesses this storage by calling __rt_stackheap_storage() to return a pointer to its 16-byte region.

2.10.9 __rt_heap_extend()

This function returns a new eight-byte aligned block of memory to add to the heap, if possible. If you reimplement __rt_stackheap_init(), you must reimplement this function. An incomplete prototype implementation is in rt_memory.s.

Syntax

```
extern unsigned __rt_heap_extend(unsigned size, void **block);
```

Usage

The calling convention is ordinary AAPCS. On entry, r0 is the minimum size of the block to add, and r1 holds a pointer to a location to store the base address.

The default implementation has the following characteristics:

- The returned size is either:
  - a multiple of eight bytes of at least the requested size
  - 0, denoting that the request cannot be honored.
- The returned base address is aligned on an eight-byte boundary.
- Size is measured in bytes.
- The function is subject only to AAPCS constraints.

Return

The default implementation extends the heap if there is sufficient free heap memory. If it cannot, it calls __user_heap_extend() if it is implemented (see *__user_heap_extend()* on page 2-74). On exit, r0 is the size of the block acquired, or 0 if nothing could be obtained, and the memory location r1 pointed to on entry contains the base address of the block.

2.10.10 __vectab_stack_and_reset

__vectab_stack_and_reset is a library section that provides a way for the initial values of sp and pc to be placed in the vector table, starting at address 0 for Cortex-M3 embedded applications.

__vectab_stack_and_reset requires the existence of a main() function in your source code. If a main() function does not exist and you place the __vectab_stack_and_reset section in a scatter-loading description file, an error is generated to the following effect:

```
Error: L6236E: No section matches selector - no section to be FIRST/LAST
```

If the normal start-up code is bypassed, that is, if there is intentionally no main() function, you are responsible for setting up the vector table without __vectab_stack_and_reset.

An example scatter-loading description file is supplied as cortex-m3.scat. See *install_directory*\RVDS\Examples. It includes a minimal vector table illustrating the use of __vectab_stack_and_reset to place the initial sp and pc values at addresses 0x0 and 0x4 in the vector table.

2.11 Tailoring the input/output functions

The higher-level input/output, such as the functions `fscanf()` and `fprintf()`, and the C++ object `std::cout`, are not target-dependent. However, the higher-level functions perform input/output by calling lower-level functions that are target-dependent. To retarget input/output, you can either avoid these higher-level functions or redefine the lower-level functions.

See `rt_sys.h` for more information on I/O functions.

See also *Writing reentrant and thread-safe code* on page 2-4.

2.11.1 Dependencies on low-level functions

Table 2-11 on page 2-79 shows the dependencies of the higher-level functions on lower-level functions. If you define your own versions of the lower-level functions, you can use the library versions of the higher-level functions directly. `fgetc()` uses `__FILE`, but `fputc()` uses `__FILE` and `ferror()`.

Note

You *must* provide definitions of `__stdin` and `__stdout` if you use any of their associated high-level functions. This applies even if your re-implementations of other functions such as `fgetc()` and `fputc()` do not reference any data stored in those objects.

Table key:

1. `__FILE`[1]
2. `__stdin`[2]
3. `__stdout`[3]
4. `fputc()`[4]
5. `ferror()`[5]
6. `fgetc()`[6]
7. `fgetwc()`
8. `fputwc()`
9. `__backspace()`[7]

1. The file structure.
2. The standard input object of type `__FILE`.
3. The standard output object of type `__FILE`.
4. Outputs a character to a file.
5. Returns the error status accumulated during file I/O.
6. Gets a character from a file.

10. `__backspacewc()`.

Table 2-11 Input/output dependencies

High-level function	Low-level object									
	1	2	3	4	5	6	7	8	9	10
fgets	x	-	-	-	x	x	-	-	-	-
fgetws	x	-	-	-	-	-	x	-	-	-
fprintf	x	-	-	x	x	-	-	-	-	-
fputs	x	-	-	x	-	-	-	-	-	-
fputws	x	-	-	-	-	-	-	x	-	-
fread	x	-	-	-	-	x	-	-	-	-
fscanf	x	-	-	-	-	x	-	-	x	-
fwprintf	x	-	-	-	x	-	-	x	-	-
fwrite	x	-	-	x	-	-	-	-	-	-
fwscanf	x	-	-	-	-	-	x	-	-	x
getchar	x	x	-	-	-	x	-	-	-	-
gets	x	x	-	-	x	x	-	-	-	-
getwchar	x	x	-	-	-	-	x	-	-	-
perror	x	-	x	x	-	-	-	-	-	-
printf	x	-	x	x	x	-	-	-	-	-
putchar	x	-	x	x	-	-	-	-	-	-
puts	x	-	x	x	-	-	-	-	-	-
putwchar	x	-	x	-	-	-	-	x	-	-
scanf	x	x	-	-	-	x	-	-	x	-
vfprintf	x	-	-	x	x	-	-	-	-	-

7. Moves the file pointer to the previous character. See *Reimplementing __backspace()* on page 2-84.

Table 2-11 Input/output dependencies (continued)

High-level function	Low-level object									
vfscanf	x	-	-	-	-	x	-	-	x	-
vfwprintf	x	-	-	-	x	-	-	x	-	-
vfwscanf	x	-	-	-	-	-	x	-	-	x
vprintf	x	-	x	x	x	-	-	-	-	-
vscanf	x	x	-	-	-	x	-	-	x	-
vwprintf	x	-	x	-	x	-	-	x	-	-
vwscanf	x	x	-	-	-	-	x	-	-	x
wprintf	x	-	x	-	x	-	-	x	-	-
wscanf	x	x	-	-	-	-	x	-	-	x

See the ISO C Reference for the syntax of the low-level functions.

Note

If you choose to reimplement `fgetc()`, `fputc()`, and `__backspace()`, be aware that `fopen()` and related functions use the ARM layout for the `__FILE` structure. You might also have to reimplement `fopen()` and related functions if you define your own version of `__FILE`.

printf family

The `printf` family consists of `_printf()`, `printf()`, `_fprintf()`, `fprintf()`, `vprintf()`, and `vfprintf()`. All these functions use `__FILE` opaquely and depend only on the functions `fputc()` and `ferror()`. The functions `_printf()` and `_fprintf()` are identical to `printf()` and `fprintf()` except that they cannot format floating-point values.

The standard output functions of the form `_printf(...)` are equivalent to:

```
fprintf(& __stdout, ...)
```

where `__stdout` has type `__FILE`.

scanf family

The scanf() family consists of scanf() and fscanf(). These functions depend only on the functions fgetc(), __FILE, and __backspace(). See *Reimplementing __backspace()* on page 2-84.

The standard input function of the form scanf(...) is equivalent to:

```
fscanf(& __stdin, ...)
```

where __stdin has type __FILE.

fwrite(), fputs(), and puts()

If you define your own version of __FILE, and your own fputc() and ferror() functions and the __stdout object, you can use all of the printf() family, fwrite(), fputs(), puts() and the C++ object std::cout unchanged from the library. Example 2-10 and Example 2-11 on page 2-82 show you how to do this. Consider modifying the system routines if you require real file handling.

You are not required to reimplement every function shown in these examples. Only reimplement the functions that require reimplementation.

Example 2-10 Retargeting printf()

```
#include <stdio.h>

struct __FILE
{
  int handle;

  /* Whatever you require here. If the only file you are using is */
  /* standard output using printf() for debugging, no file handling */
  /* is required. */
};

/* FILE is typedef'd in stdio.h. */

FILE __stdout;

int fputc(int ch, FILE *f)
{
  /* Your implementation of fputc(). */
  return ch;
}
int ferror(FILE *f)
{
```

```
  /* Your implementation of ferror(). */
  return 0;
}
void test(void)
{
  printf("Hello world\n");
}
```

Be aware of endianness with `fputc()`. `fputc()` takes an `int` parameter, but contains only a character. Whether the character is in the top or the bottom byte of the integer variable depends on the endianness. The following code sample avoids problems with endianness:

```
extern void sendchar(char *ch);

int fputc(int ch, FILE *f)
{
  /* example: write a character to an LCD */
  char tempch = ch; // temp char avoids endianness issue
  sendchar(&tempch);
  return ch;
}
```

See also the implementation in the main examples directory, in `...\emb_sw_dev\source\retarget.c`.

Example 2-11 Retargeting cout

```
File 1: Reimplement any functions that require reimplementation.

#include <stdio.h>

namespace std {

  struct __FILE
  {
    int handle;

    /* Whatever you require here. If the only file you are using is */
    /* standard output using printf() for debugging, no file handling */
    /* is required. */
  };

  FILE __stdout;
  FILE __stdin;
  FILE __stderr;
```

```
  int fgetc(FILE *f)
  {
    /* Your implementation of fgetc(). */
    return 0;
  };
  int fputc(int c, FILE *stream)
  {
    /* Your implementation of fputc(). */
  }
  int ferror(FILE *stream)
  {
    /* Your implementation of ferror(). */
  }
  long int ftell(FILE *stream)
  {
    /* Your implementation of ftell(). */
  }
  int fclose(FILE *f)
  {
    /* Your implementation of fclose(). */
    return 0;
  }
  int fseek(FILE *f, long nPos, int nMode)
  {
    /* Your implementation of fseek(). */
    return 0;
  }
  int fflush(FILE *f)
  {
    /* Your implementation of fflush(). */
    return 0;
  }
}
```

File 2: Print “Hello world” using your reimplemented functions.

```
#include <stdio.h>
#include <iostream>
using namespace std;

int main()
{
  cout << “Hello world\n”;
  return 0;
}
```

By default, fread() and fwrite() call fast block input/output functions that are part of the ARM stream implementation. If you define your own __FILE structure instead of using the ARM stream implementation, fread() and fwrite() call fgetc() instead of calling the block input/output functions.

fread(), fgets(), and gets()

The functions fread(), fgets(), and gets() are implemented as a loop over fgetc() and ferror(). Each uses the FILE argument opaquely.

If you provide your own implementation of __FILE, __stdin (for gets()), fgetc(), and ferror(), you can use these functions, and the C++ object std::cin directly from the library.

Reimplementing __backspace()

The function __backspace() is used by the scanf family of functions. It must never be called directly, but reimplemented if you are retargeting the stdio arrangements at the fgetc() level.

The semantics are:

```
int __backspace(FILE *stream);
```

__backspace(stream) must only be called after reading a character from the stream. You must not call it after a write, a seek, or immediately after opening the file, for example. It returns to the stream the last character that was read from the stream, so that the same character can be read from the stream again by the next read operation. This means that a character that was read from the stream by scanf but that is not required (that is, it terminates the scanf operation) is read correctly by the next function that reads from the stream.

__backspace is separate from ungetc(). This is to guarantee that a single character can be pushed back after the scanf family of functions has finished.

The value returned by __backspace() is either 0 (success) or EOF (failure). It returns EOF only if used incorrectly, for example, if no characters have been read from the stream. When used correctly, __backspace() must always return 0, because the scanf family of functions do not check the error return.

The interaction between __backspace() and ungetc() is:

- If you apply __backspace() to a stream and then ungetc() a character into the same stream, subsequent calls to fgetc() must return first the character returned by ungetc(), and then the character returned by __backspace().

- If you ungetc() a character back to a stream, then read it with fgetc(), and then backspace it, the next character read by fgetc() must be the same character that was returned to the stream. That is the __backspace() operation must cancel the effect of the fgetc() operation. However, another call to ungetc() after the call to __backspace() is not required to succeed.
- The situation where you ungetc() a character into a stream and then __backspace() another one immediately, with no intervening read, never arises. __backspace() must only be called after fgetc(), so this sequence of calls is illegal. If you are writing __backspace() implementations, you can assume that the unget() of a character into a stream followed immediately by a __backspace() with no intervening read, never occurs.

Reimplementing __backspacewc()

__backspacewc() is the wide-character equivalent of __backspace(). __backspacewc() behaves in the same way as __backspace() except that it pushes back the last wide character instead of a narrow character.

2.11.2 Target-dependent input and output support functions

rt_sys.h defines the type FILEHANDLE. The value of FILEHANDLE is returned by _sys_open() and identifies an open file on the host system.

Target-dependent input and output functions use semihosting. If any function is redefined, all stream-support functions must be redefined.

If the _sys_* functions are redefined, both normal character I/O and wide character I/O works. That is, you are not required to do anything extra with these functions for wide character I/O to work.

2.11.3 _sys_open()

This function opens a file.

Syntax

```
FILEHANDLE _sys_open(const char *name, int openmode)
```

Usage

The _sys_open function is required by fopen() and freopen().These functions, in turn, are required if any file input/output function is to be used.

The *openmode* parameter is a bitmap, whose bits mostly correspond directly to the ISO mode specification. See `rt_sys.h` for more information. Target-dependent extensions are possible, but `freopen()` must also be extended.

Return

The return value is `-1` if an error occurs.

2.11.4 _sys_close()

This function closes a file previously opened with `_sys_open()`.

Syntax

```
int _sys_close(FILEHANDLE fh)
```

Usage

This function must be defined if any input/output function is to be used.

Return

The return value is `0` if successful. A nonzero value indicates an error.

2.11.5 _sys_read()

This function reads the contents of a file into a buffer.

Syntax

```
int _sys_read(FILEHANDLE fh, unsigned char *buf, unsigned len, int mode)
```

Note

The mode parameter is here for historical reasons. It contains nothing useful and must be ignored.

Return

The return value is one of the following:

- The number of characters *not* read (that is, *len* - *result* were read).
- An error indication.

- An EOF indicator. The EOF indication involves the setting of 0x80000000 in the normal result. The target-independent code is capable of handling either:

  Early EOF The last read from a file returns some characters plus an EOF indicator.

  Late EOF The last read returns only EOF.

2.11.6 _sys_write()

Writes the contents of a buffer to a file previously opened with _sys_open().

Syntax

```
int _sys_write(FILEHANDLE fh, const unsigned char *buf, unsigned len, int mode)
```

Note

The mode parameter is here for historical reasons. It contains nothing useful and must be ignored.

Return

The return value is either:

- a positive number representing the number of characters *not* written (so any nonzero return value denotes a failure of some sort)
- a negative number indicating an error.

2.11.7 _sys_ensure()

This function flushes buffers associated with a file handle.

Syntax

```
int _sys_ensure(FILEHANDLE fh)
```

Usage

A call to _sys_ensure() flushes any buffers associated with file handle *fh*, and ensures that the file is up to date on the backing store medium.

Return

If an error occurs, the result is negative.

2.11.8 _sys_flen()

This function returns the current length of a file.

Syntax

```
long _sys_flen(FILEHANDLE fh)
```

Usage

The function is required to convert `fseek(..., SEEK_END)` into `(..., SEEK_SET)` as required by `_sys_seek()`.

If `fseek()` is used with an underlying system that does not directly support seeking relative to the end of a file, then `_sys_flen()` must be defined. If the underlying system can seek relative to the end of a file, you can define `fseek()` so that `_sys_flen()` is not required.

Return

This function returns the current length of the file *fh*, or a negative error indicator.

2.11.9 _sys_seek()

This function puts the file pointer at offset *pos* from the beginning of the file.

Syntax

```
int _sys_seek(FILEHANDLE fh, long pos)
```

Usage

This function sets the current read or write position to the new location *pos* relative to the start of the current file *fh*.

Return

The result is non-negative if no error occurs or is negative if an error occurs.

2.11.10 _sys_istty()

This function determines if a file handle identifies a terminal.

Syntax

```
int _sys_istty(FILEHANDLE fh)
```

Usage

When a file is connected to a terminal device, this function is used to provide unbuffered behavior by default (in the absence of a call to `set(v)buf`) and to prohibit seeking.

Return

The return value is:

0 There is no interactive device.

1 There is an interactive device.

other An error occurred.

2.11.11 _sys_tmpnam()

This function converts the file number *fileno* for a temporary file to a unique filename, for example, `tmp0001`.

Syntax

```
void _sys_tmpnam(char *name, int fileno, unsigned maxlength)
```

Usage

The function must be defined if `tmpnam()` or `tmpfile()` is used.

Return

Returns the filename in *name*.

2.11.12 _sys_command_string()

This function retrieves the command line used to invoke the current application from the environment that called the application.

Syntax

```
char *_sys_command_string(char *cmd, int len)
```

where:

cmd Is a pointer to a buffer that can be used to store the command line. It is not required that the command line is stored in *cmd*.

len Is the length of the buffer.

Usage

This function is called by the library startup code to set up `argv` and `argc` to pass to `main()`.

Note

You must not assume that the C library is fully initialized when this function is called. For example, you must not call `malloc()` from within this function. This is because the C library startup sequence calls this function before the heap is fully configured.

Return

The function must return either:

- A pointer to the command line, if successful. This can be either a pointer to the *cmd* buffer if it is used, or a pointer to wherever else the command line is stored.
- `NULL`, if not successful.

2.11.13 #pragma import(_main_redirection)

This pragma must be defined when redirecting standard input, output and error streams at runtime. See *Environment* on page B-3 in the *Compiler Reference Guide* for more information.

Syntax

```
#pragma import(_main_redirection)
```

2.12 Tailoring other C library functions

This section describes target-dependent ISO C library functions. Implementation of these ISO standard functions depends entirely on the target operating system.

The default implementation of these functions is semihosted. That is, each function uses semihosting.

2.12.1 clock()

This is the standard C library clock function from `time.h`.

Syntax

```
clock_t clock(void)
```

Usage

If the units of `clock_t` differ from the default of centiseconds, you must define `__CLK_TCK` on the compiler command line or in your own header file. The value in the definition is used for `CLK_TCK` and `CLOCKS_PER_SEC`. The default value is 100 for centiseconds.

Note

If you reimplement `clock()` you must also reimplement `_clock_init()`.

Return

The returned value is an unsigned integer.

2.12.2 _clock_init()

This is an optional initialization function for `clock()`.

Syntax

```
__weak void _clock_init(void)
```

Usage

You must provide a clock initialization function if `clock()` must work with a read-only timer. If implemented, `_clock_init()` is called from the library initialization code.

2.12.3 time()

This is the standard C library `time()` function from `time.h`.

Syntax

```
time_t time(time_t *timer)
```

The return value is an approximation of the current calendar time.

Return

The value `-1` is returned if the calendar time is not available. If *timer* is not a `NULL` pointer, the return value is also stored in *timer*.

2.12.4 remove()

This is the standard C library `remove()` function from `stdio.h`.

Syntax

```
int remove(const char *filename)
```

Usage

`remove()` causes the file whose name is the string pointed to by *filename* to be removed. Subsequent attempts to open the file result in failure, unless it is created again. If the file is open, the behavior of the `remove()` function is implementation-defined.

Return

Returns zero if the operation succeeds or nonzero if it fails.

2.12.5 rename()

This is the standard C library `rename()` function from `stdio.h`.

Syntax

```
int rename(const char *old, const char *new)
```

Usage

`rename()` causes the file whose name is the string pointed to by *old* to be subsequently known by the name given by the string pointed to by *new*. The file named *old* is effectively removed. If a file named by the string pointed to by *new* exists prior to the call of the `rename()` function, the behavior is implementation-defined.

Return

Returns zero if the operation succeeds or nonzero if it fails. If the operation returns nonzero and the file existed previously it is still known by its original name.

2.12.6 system()

This is the standard C library `system()` function from `stdlib.h`.

Syntax

```
int system(const char *string)
```

Usage

`system()` passes the string pointed to by *string* to the host environment to be executed by a command processor in an implementation-defined manner. A null pointer can be used for *string*, to inquire whether a command processor exists.

Return

If the argument is a null pointer, the system function returns nonzero only if a command processor is available.

If the argument is not a null pointer, the `system()` function returns an implementation-defined value.

2.12.7 getenv()

This is the standard C library `getenv()` function from `stdlib.h`.

Syntax

```
char *getenv(const char *name)
```

Usage

The default implementation returns `NULL` indicating that no environment information is available. You can reimplement `getenv()`. It depends on no other function and no other function depends on it.

If you redefine the function, you can also call a function `_getenv_init()`. The C library initialization code then calls this when the library is initialized, that is, before `main()` is entered.

Unless you redefine the behavior that is normally expected from `getenv()`, that is, the behavior of the standard C library `getenv()` function, `getenv()` searches the environment list, provided by the host environment, for a string that matches the string pointed to by *name*. The set of environment names and the method for altering the environment list are implementation-defined.

Return

The return value is a pointer to a string associated with the matched list member. The array pointed to must not be modified by the program, but might be overwritten by a subsequent call to `getenv()`.

2.12.8 _getenv_init()

This enables a user version of `getenv()` to initialize itself.

Syntax

```
void _getenv_init(void)
```

Usage

If this function is defined, the C library initialization code calls it when the library is initialized, that is, before `main()` is entered.

2.13 Selecting real-time division

The division routine supplied with the ARM libraries provides good overall performance. However, the amount of time required to perform a division depends on the input values. A 4-bit quotient requires only 12 cycles, but a 32-bit quotient requires 96 cycles. Depending on your target, some applications require a faster worst-case cycle count at the expense of lower average performance. For this reason, the ARM library provides two divide routines.

The real-time routine:

- always executes in fewer than 45 cycles
- is faster than the standard division routine for larger quotients
- is slower than the standard division routine for typical quotients
- returns the same results
- does not require any change in the surrounding code.

Select the real-time divide routine, instead of the generally more efficient routine, by using either:

- `IMPORT __use_realtime_division` from assembly language
- `#pragma import(__use_realtime_division)` from C.

Note

Real-time division is not available in the libraries for Cortex-M1 and Cortex-M0.

2.14 ISO implementation definition

This section describes how the libraries fulfill the requirements of the ISO specification.

2.14.1 ISO C library implementation definition

The ISO specification leaves some features to the implementors, but requires that implementation choices be documented. This section describes the ARM library implementation.

In the generic ARM C library:

- The macro `NULL` expands to the integer constant `0`.
- If a program redefines a reserved external identifier, an error might occur when the program is linked with the standard libraries. If it is not linked with standard libraries, no error is diagnosed.
- The `__aeabi_assert()` function prints information on the failing diagnostic on `stderr` and then calls the `abort()` function:

  `*** assertion failed: expression, file name, line number`

  Note

  The behavior of the `assert` macro depends on the conditions in operation at the most recent occurrence of `#include <assert.h>`. See *Exiting from an assert* on page 2-38 for more information.

The following functions test for character values in the range `EOF` (–1) to 255 (inclusive):

- `isalnum()`
- `isalpha()`
- `iscntrl()`
- `islower()`
- `isprint()`
- `isupper()`
- `ispunct()`.

The ISO C library variants are listed in *Library naming conventions* on page 2-117.

Mathematical functions

Table 2-12 shows the default response of mathematical functions when supplied with out-of-range arguments.

Table 2-12 Mathematical functions

Function	Condition	Returned value	Error number
acos(x)	abs(x) > 1	QNaN	EDOM
asin(x)	abs(x) > 1	QNaN	EDOM
atan2(x,y)	x =0, y = 0	QNaN	EDOM
atan2(x,y)	x = Inf, y = Inf	QNaN	EDOM
cos(x)	x=Inf	QNaN	EDOM
cosh(x)	Overflow	+Inf	ERANGE
exp(x)	Overflow	+Inf	ERANGE
exp(x)	Underflow	+0	ERANGE
fmod(x,y)	x=Inf	QNaN	EDOM
fmod(x,y)	y = 0	QNaN	EDOM
log(x)	x < 0	QNaN	EDOM
log(x)	x = 0	-Inf	EDOM
log10(x)	x < 0	QNaN	EDOM
log10(x)	x = 0	-Inf	EDOM
pow(x,y)	Overflow	+Inf	ERANGE
pow(x,y)	Underflow	0	ERANGE
pow(x,y)	x=0 or x=Inf, y=0	+1	EDOM
pow(x,y)	x=+0, y<0	-Inf	EDOM
pow(x,y)	x=-0, y<0 and y integer	-Inf	EDOM
pow(x,y)	x= -0, y<0 and y non-integer	QNaN	EDOM
pow(x,y)	x<0, y non-integer	QNaN	EDOM
pow(x,y)	x=1, y=Inf	QNaN	EDOM

Table 2-12 Mathematical functions (continued)

Function	Condition	Returned value	Error number
sqrt(x)	x < 0	QNaN	EDOM
sin(x)	x=Inf	QNaN	EDOM
sinh(x)	Overflow	+Inf	ERANGE
tan(x)	x=Inf	QNaN	EDOM
atan(x)	SNaN	SNaN	None
ceil(x)	SNaN	SNaN	None
floor(x)	SNaN	SNaN	None
frexp(x)	SNaN	SNaN	None
ldexp(x)	SNaN	SNaN	None
modf(x)	SNaN	SNaN	None
tanh(x)	SNaN	SNaN	None

HUGE_VAL is an alias for Inf. Consult the errno variable for the error number. Other than the cases shown in Table 2-12 on page 2-98, all functions return QNaN when passed QNaN and throw an invalid operation exception when passed SNaN.

The string passed to C99 nan() is ignored, and the same *Not a Number* (NaN) is returned always, namely the one with all fraction bits clear except the topmost one. The sign bit is clear as well. Passing strings of the form NAN(xxxx) to strtod has the same effect.

To enable C99 mathlib error-handling behavior that complies with POSIX and Annex F of the C99 standard, refer to the symbol __use_c99_matherr in your program by inserting:

- IMPORT __use_c99_matherr, in assembly language
- #pragma IMPORT __use_c99_matherr, in C.

When __use_c99_matherr is defined, the following behavior applies:

- In addition to setting errno, mathlib functions report errors by setting IEEE floating-point exception flags. IEEE floating-point exception flags can only be used in floating-point modes that support exceptions.

- Underflow and overflow range errors, in addition to setting `ERANGE`, also set the IEEE overflow or underflow flag as appropriate (in floating-point modes that support IEEE exceptions). The underflow flag is also set for partial underflow, that is, denormal return values, whereas `ERANGE` is only set in the case of total underflow (returning zero when the real mathematical answer is nonzero).
- Range errors, in addition to setting `ERANGE`, also set either the IEEE invalid operation flag or divide-by-zero flag, depending on the nature of the range error. Range errors that set divide-by-zero are:
  - atanh of -1 and +1
  - lgamma of zero and of negative integers
  - log of zero (and equivalently, log10(0), log2(0), and log1p(-1), but not log of negative numbers)
  - pow(0, negative number)
  - tgamma(0), but not tgamma of negative numbers.

  Note

  In legacy RVCT 4.0 behavior, these errors are classed as domain errors rather than range errors.

  All other range errors report Invalid Operation.

  The IEEE invalid operation flag or divide-by-zero flag is only set in floating-point modes that support IEEE exceptions.
- Functions with two input parameters ignore a quiet NaN in one input parameter if the other input parameter is sufficient to completely correct the output value. (When `__use_c99_matherr` is not defined, a QNaN result is returned if any NaN is present in the input parameters.) The list of cases that this applies to is as follows:
  - hypot(infinity, x) and hypot(x, infinity) return infinity, even when *x* is QNaN
  - pow(x,0) and pow(1,x) return 1, even when *x* is QNan.

  However, signalling NaNs (unspecified in C99) in these situations still cause an exception and a QNaN return value.
- pow(x,0) and pow(1,x) return 1 with no error, for all finite or infinite value of *x*. (When `__use_c99_matherr` is not defined, pow(infinity,0), pow(1,infinity) and pow(0,0) are domain errors.

- When *x* is negative and pow(x,y) is infinite in magnitude but has mathematically indeterminate sign, positive infinity is returned. (When `__use_c99_matherr` is not defined, NaN is returned.) The affected cases are:
  - `pow(-0,`*`fraction`*`)`
  - `pow(-0,-`*`infinity`*`)`
  - `pow(`*`negative number`*`,`*`infinity`*`)`
  - `pow(-`*`infinity`*`,+`*`infinity`*`)`
  - `pow(-`*`infinity`*`,`*`fraction`*`)`.

  All such cases are still domain errors.
- `pow(-1,`*`infinity`*`)` gives `+1` with no error. (When `__use_c99_matherr` is not defined, `pow(-1,`*`infinity`*`)` gives NaN with a domain error.)
- `atan2(`*`infinity`*`,`*`infinity`*`)` is treated as if it were `atan2(1,1)`, with the same signs, so it returns *pi*`/4`, `3*pi/4`, `-pi/4`, or `-3*`*pi*`/4`. (When `__use_c99_matherr` is not defined, these are domain errors.)
- `atan2(0,0)` is treated as if it were `atan2(0,1)`, with the same signs, so it returns `+0`, `-0`, *+pi*, or *-pi*. (When `__use_c99_matherr` is not defined, these are domain errors.)
- `tgamma(0)` returns an infinity of the same sign as the input. (When `__use_c99_matherr` is not defined, `tgamma(0)` returns NaN.) It remains a domain error.

Signal functions

Table 2-13 shows the signals supported by the `signal()` function.

Table 2-13 Signal functions

Signal	Number	Description	Additional argument
SIGABRT	1	This signal is only used if `abort()` or `assert()` are called by your application.	None
SIGFPE	2	Used to signal any arithmetic exception, for example, division by zero. Used by hard and soft floating-point and by integer division.	A set of bits from {FE_EX_INEXACT, FE_EX_UNDERFLOW, FE_EX_OVERFLOW, FE_EX_DIVBYZERO, FE_EX_INVALID, DIVBYZERO}
SIGILL	3	Illegal instruction.	None

Table 2-13 Signal functions (continued)

Signal	Number	Description	Additional argument
SIGINT	4	Attention request from user.	None
SIGSEGV	5	Bad memory access.	None
SIGTERM	6	Termination request.	None
SIGSTAK	7	Obsolete.	None
SIGRTRED	8	Redirection failed on a runtime library input/output stream.	Name of file or device being re-opened to redirect a standard stream
SIGRTMEM	9	Out of heap space during initialization or after corruption.	Size of failed request
SIGUSR1	10	User-defined.	User-defined
SIGUSR2	11	User-defined.	User-defined
SIGPVFN	12	A pure virtual function was called from C++.	-
SIGCPPL	13	Exception from C++.	-
SIGOUTOFHEAP	14	Returned by the C++ function `::operator new` when out of heap space.	Size of failed request
reserved	15-31	Reserved.	Reserved
other	> 31	User-defined.	User-defined

Although **SIGSTAK** exists in `signal.h`, this signal is no longer generated by the C library and is considered obsolete.

A signal number greater than **SIGUSR2** can be passed through `__raise()`, and caught by the default signal handler, but it cannot be caught by a handler registered using `signal()`.

`signal()` returns an error code if you try to register a handler for a signal number greater than **SIGUSR2**.

The default handling of all recognized signals is to print a diagnostic message and call `exit()`. This default behavior applies at program startup and until you change it.

Caution

The IEEE 754 standard for floating-point processing states that the default action to an exception is to proceed without a trap. A raised exception in floating-point calculations does not, by default, generate **SIGFPE**. You can modify floating-point error handling by tailoring the functions and definitions in `fenv.h`. See *Tailoring error signaling, error handling, and program exit* on page 2-59 and Chapter 4 *Floating-point Support* for more information.

For all the signals in Table 2-13 on page 2-101, when a signal occurs, if the handler points to a function, the equivalent of `signal(sig, SIG_DFL)` is executed before the call to the handler.

If the **SIGILL** signal is received by a handler specified to by the `signal()` function, the default handling is reset.

Input/output characteristics

The generic ARM C library has the following input/output characteristics:

- The last line of a text stream does not require a terminating newline character.
- Space characters written out to a text stream immediately before a newline character do appear when read back in.
- No null characters are appended to a binary output stream.
- The file position indicator of an append mode stream is initially placed at the end of the file.
- A write to a text stream causes the associated file to be truncated beyond the point where the write occurred if this is the behavior of the device category of the file.
- If semihosting is used, the maximum number of open files is limited by the available target memory.
- A zero-length file exists, that is, where no characters have been written by an output stream.
- A file can be opened many times for reading, but only once for writing or updating. A file cannot simultaneously be open for reading on one stream, and open for writing or updating on another.
- Local time zones and Daylight Saving Time are not implemented. The values returned indicate that the information is not available. For example, the `gmtime()` function always returns `NULL`.

- The status returned by `exit()` is the same value that was passed to it. For definitions of `EXIT_SUCCESS` and `EXIT_FAILURE`, see the header file `stdlib.h`. Semihosting, however, does not pass the status back to the execution environment.
- The error messages returned by the `strerror()` function are identical to those given by the `perror()` function.
- If the size of area requested is zero, `calloc()` and `realloc()` return `NULL`.
- If the size of area requested is zero, `malloc()` returns a pointer to a zero-size block.
- `abort()` closes all open files and deletes all temporary files.
- `fprintf()` prints `%p` arguments in lowercase hexadecimal format as if a precision of 8 had been specified. If the variant form (`%#p`) is used, the number is preceded by the character `@`.
- `fscanf()` treats `%p` arguments exactly the same as `%x` arguments.
- `fscanf()` always treats the character "-" in a `%...[...]` argument as a literal character.
- `ftell()`, `fsetpos()` and `fgetpos()` set `errno` to the value of `EDOM` on failure.
- `perror()` generates the messages shown in Table 2-14.

Table 2-14 perror() messages

Error	Message
`0`	`No error (errno = 0)`
`EDOM`	`EDOM - function argument out of range`
`ERANGE`	`ERANGE - function result not representable`
`ESIGNUM`	`ESIGNUM - illegal signal number`
Others	`Unknown error`

The following characteristics must be specified in an ISO-compliant implementation (these are unspecified in the ARM C library):

- the validity of a filename
- whether `remove()` can remove an open file
- the effect of calling the `rename()` function when the new name already exists
- the effect of calling `getenv()` (the default is to return `NULL`, no value available)

- the effect of calling `system()`
- the value returned by `clock()`.

2.14.2 Standard C++ library implementation definition

This section describes the implementation of the C++ libraries. The ARM C++ library provides all of the library defined in the *ISO/IEC 14822 :1998(E) C++ Standard*, aside from some limitations described in Table 2-15.

For information on implementation-defined behavior that is defined in the Rogue Wave C++ library, see the included Rogue Wave HTML documentation. By default, this is installed in *`install_directory`*`\Documentation\RogueWave`.

The standard C++ library is distributed in binary form only.

Table 2-15 describes the most important features missing from the current release.

Table 2-15 Standard C++ library differences

Standard	Implementation differences
locale	The locale message facet is not supported. It fails to open catalogs at runtime because the ARM C library does not support `catopen` and `catclose` through `nl_types.h`. One of two locale definitions can be selected at link time. Other locales can be created by user-redefinable functions.
Timezone	Not supported. The ARM C library does not support it.

2.15 C library extensions

This section describes C library extensions and functions. Some are defined by the *ISO/IEC 9899:1999* C standard and some are specific to the ARM compiler. These are summarized in Table 2-16.

Table 2-16 C library extensions

Function	Header file definition	Extension
atoll() on page 2-107	`stdlib.h`	C99 standard
strtoll() on page 2-107	`stdlib.h`	C99 standard
strtoull() on page 2-108	`stdlib.h`	C99 standard
printf() on page 2-108	`stdlib.h`	C99 standard
snprintf() on page 2-108	`stdio.h`	C99 standard
vsnprintf() on page 2-109	`stdio.h`	C99 standard
lldiv() on page 2-109	`stdlib.h`	C99 standard
llabs() on page 2-109	`stdlib.h`	C99 standard
wcscasecmp() on page 2-110	`wchar.h`	GNU extension supported by the ARM libraries.
wcsncasecmp() on page 2-110	`wchar.h`	GNU extension supported by the ARM libraries.
wcstombs() on page 2-110	`stdlib.h`	POSIX extended functionality
alloca() on page 2-111	`alloca.h`	Common nonstandard extension to many C libraries
strlcpy() on page 2-111	`string.h`	Common BSD-derived extension to many C libraries
strlcat() on page 2-111	`string.h`	Common BSD-derived extension to many C libraries
strcasecmp() on page 2-112	`string.h`	Standardized by POSIX.
strncasecmp() on page 2-112	`string.h`	Standardized by POSIX.
_fisatty() on page 2-112	`stdio.h`	Specific to ARM compiler

Table 2-16 C library extensions (continued)

Function	Header file definition	Extension
__heapstats() on page 2-113	`stdlib.h`	Specific to ARM compiler
__heapvalid() on page 2-114	`stdlib.h`	Specific to ARM compiler
_membitcpybl(), *_membitcpybb()*, *_membitcpyhl()*, *_membitcpyhb()*, *_membitcpywl()*, *_membitcpywb()*, *_membitmovebl()*, *_membitmovebb()*, *_membitmovehl()*, *_membitmovehb()*, *_membitmovewl()*, *_membitmovewb()* on page 2-115	`string.h`	Specific to ARM compiler

The headers `<stdint.h>` and `<inttypes.h>` from C99 are also available.

2.15.1 atoll()

The `atoll()` function converts a decimal string into an integer. This is similar to the ISO functions `atol()` and `atoi()`, but returns a **`long long`** result. Like `atoi()`, `atoll()` can accept octal or hexadecimal input if the string begins with `0` or `0x`.

Syntax

```
longlong atoll(const char *nptr)
```

2.15.2 strtoll()

The `strtoll()` function converts a string in an arbitrary base to an integer. This is similar to the ISO function `strtol()`, but returns a **`long long`** result. Like `strtol()`, the parameter *endptr* can hold the location where a pointer to the end of the translated string is to be stored, or can be `NULL`. The parameter *base* must contain the number base. Setting *base* to zero indicates that the base is to be selected in the same way as `atoll()`.

Syntax

```
longlong strtoll(const char *nptr, char **endptr, int base)
```

2.15.3 strtoull()

`strtoull()` is exactly the same as `strtoll()`, but returns an **`unsigned long long`**.

Syntax

```
unsigned long long strtoull(const char *nptr, char **endptr, int base)
```

2.15.4 printf()

`printf()` is fully conformant to the ISOC89 standard at all times. It also optionally supports the additional format directives defined in C99, that is, `%a` and `%A` for hexadecimal floating-point, and `%E`, `%F`, and `%G` as uppercase versions of `%e`, `%f`, and `%g`. These C99 format directives are not included by default.

To enable the C99 features in `printf()` you must specify `#pragma import(__use_c99_library)`. This affects all functions in the `printf()` and `scanf()` families.

Syntax

```
int printf(const char *format,  ...)
```

2.15.5 snprintf()

The `snprintf()` function works almost exactly like the ISO `sprintf()` function, except that the caller can specify the maximum size of the buffer. The return value is the length of the complete formatted string that would have been written if the buffer were big enough. Therefore, the string written into the buffer is complete only if the return value is at least zero and at most `n-1`.

The *bufsize* parameter specifies the number of characters of *buffer* that the function can write into, *including* the terminating null.

`stdio.h` is an ISO header file.

The ISO C90 standard prohibits this function from being defined in ISO header files. Code compiled with this function using the `--c90 --strict` options, generates an error.

The ISO C99 standard requires this function in ISO header files. Code compiled with this function using the `--c99 --strict` options does not generate an error.

Syntax

```
int snprintf(char *buffer, size_t bufsize, const char *format,  ...)
```

2.15.6 vsnprintf()

The `vsnprintf()` function works almost exactly like the ISO `vsprintf()` function, except that the caller can specify the maximum size of the buffer. The return value is the length of the complete formatted string that would have been written if the buffer were big enough. Therefore, the string written into the buffer is complete only if the return value is at least zero and at most `n-1`.

The *bufsize* parameter specifies the number of characters of *buffer* that the function can write into, *including* the terminating null.

`stdio.h` is an ISO header file.

The ISO C90 standard prohibits this function from being defined in ISO header files. Code compiled with this function using the `--c90 --strict` options, generates an error.

The ISO C99 standard requires this function in ISO header files. Code compiled with this function using the `--c99 --strict` options, does not generate an error.

Syntax

```
int vsnprintf(char *buffer,  size_t bufsize, const  char *format,  va_list ap)
```

2.15.7 lldiv()

The `lldiv` function divides two **`long long`** integers and returns both the quotient and the remainder. It is the **`long long`** equivalent of the ISO function `ldiv`. The return type `lldiv_t` is a structure containing two **`long long`** members, called `quot` and `rem`.

`stdlib.h` is an ISO header file.

The ISO C90 standard prohibits this function from being defined in ISO header files. Code compiled with this function using the `--c90 --strict` options, generates an error.

The ISO C99 standard requires this function in ISO header files. Code compiled with this function using the `--c99 --strict` options, does not generate an error.

Syntax

```
lldiv_t lldiv(long long num, long long denom)
```

2.15.8 llabs()

The `llabs()` function returns the absolute value of its input. It is the **`long long`** equivalent of the ISO function `labs`.

`stdlib.h` is an ISO header file.

The ISO C90 standard prohibits this function from being defined in ISO header files. Code compiled with this function using the `--c90 --strict` options, generates an error.

The ISO C99 standard requires this function in ISO header files. Code compiled with this function using the `--c99 --strict` options, does not generate an error.

Syntax

```
long long llabs(long long num)
```

2.15.9 wcscasecmp()

The `wcscasecmp()` function performs a case-insensitive string comparison test on wide characters. It is a GNU extension to the libraries. It is not POSIX-standardized.

Syntax

```
int wcscasecmp(const wchar_t * __restrict s1, const wchar_t * __restrict s2)
__attribute__((__nonnull__(1,2)));
```

2.15.10 wcsncasecmp()

The `wcsncasecmp()` function performs a case-insensitive string comparison test of not more than a specified number of wide characters. It is a GNU extension to the libraries. It is not POSIX-standardized.

Syntax

```
int wcsncasecmp(const wchar_t * __restrict s1, const wchar_t * __restrict s2,
size_t n) __attribute__((__nonnull__(1,2)));
```

2.15.11 wcstombs()

This function works as described in the ISO C standard, with extended functionality as specified by POSIX, that is, if *s* is a null pointer, `wcstombs()` returns the length required to convert the entire array regardless of the value of *n*, but no values are stored.

Syntax

```
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n)
```

2.15.12 alloca()

The `alloca()` function allocates local storage in a function. It returns a pointer to *size* bytes of memory, or `NULL` if not enough memory was available. The default implementation returns an eight-byte aligned block of memory.

Memory returned from `alloca()` must never be passed to `free()`. Instead, the memory is deallocated automatically when the function that called `alloca()` returns.

`alloca()` must not be called through a function pointer. You must take care when using `alloca()` and `setjmp()` in the same function, because memory allocated by `alloca()` between calling `setjmp()` and `longjmp()` is deallocated by the call to `longjmp()`.

This function is a common non standard extension to many C libraries.

Syntax

```
void* alloca(size_t size)
```

2.15.13 strlcpy()

The `strlcpy()` function copies up to *size*-1 characters from the `NUL` terminated string *src* to *dst*. It takes the full size of the buffer, not only the length, and terminates the result with `NUL` as long as *size* is greater than 0. Include a byte for the `NUL` in your *size* value.

The `strlcpy()` function returns the total length of the string that *would* have been copied if there was unlimited space. This might or might not be equal to the length of the string *actually* copied, depending on whether there was enough space. This means that you can call `strlcpy()` once to find out how much space is required, then allocate it if you do not have enough, and finally call `strlcpy()` a second time to do the required copy.

This function is a common BSD-derived extension to many C libraries.

Syntax

```
extern size_t strlcpy(char *dst, const char *src, size_t size)
```

2.15.14 strlcat()

The `strlcat()` function concatenates two strings. It appends up to *size*-`strlen`(*dst*)-1 bytes from the `NUL` terminated string *src* to the end of *dst*. It takes the full size of the buffer, not only the length, and terminates the result with `NUL` as long as *size* is greater than 0. Include a byte for the `NUL` in your *size* value.

The strlcat() function returns the total length of the string that *would* have been created if there was unlimited space. This might or might not be equal to the length of the string *actually* created, depending on whether there was enough space. This means that you can call strlcat() once to find out how much space is required, then allocate it if you do not have enough, and finally call strlcat() a second time to create the required string.

This function is a common BSD-derived extension to many C libraries.

Syntax

```
extern size_t strlcat(char *dst, *src, size_t size)
```

2.15.15 strcasecmp()

The strcasecmp() function performs a case-insensitive string comparison test.

Syntax

```
extern _ARMABI int strcasecmp(const char * s1, const char * s2)
__attribute__((__nonnull__(1,2)));
```

2.15.16 strncasecmp()

The strncasecmp() function performs a case-insensitive string comparison test of not more than a specified number of characters.

Syntax

```
extern _ARMABI int strncasecmp(const char * s1, const char * s2, size_t n)
__attribute__((__nonnull__(1,2)));
```

2.15.17 _fisatty()

The _fisatty() function determines whether the given stdio stream is attached to a terminal device or a normal file. It calls the _sys_istty() low-level function on the underlying file handle. See *Tailoring the input/output functions* on page 2-78 for more information.

This function is an extension that is specific to the ARM library.

Syntax

```
int _fisatty(FILE *stream)
```

The return value indicates the stream destination:

0	A file.
1	A terminal.
Negative	An error.

2.15.18 __heapstats()

The `__heapstats()` function displays statistics on the state of the storage allocation heap. The default implementation in the ARM compiler gives information on how many free blocks exist, and estimates their size ranges.

Example 2-12 shows an example of the output from `__heapstats()`.

Example 2-12 Output from __heapstats()

```
32272 bytes in 2 free blocks (avge size 16136)
1 blocks 2^12+1 to 2^13
1 blocks 2^13+1 to 2^14
```

Line 1 of the output displays the total number of bytes, the number of free blocks, and the average size. The following lines give an estimate of the size of each block in bytes, expressed as a range. `__heapstats()` does not give information on the number of used blocks.

The function outputs its results by calling the output function *dprint*, that must work like `fprintf()`. The first parameter passed to *dprint* is the supplied pointer *param*. You can pass `fprintf()` itself, provided you cast it to the right function pointer type. This type is defined as a **`typedef`** for convenience. It is called `__heapprt`. For example:

```
__heapstats((__heapprt)fprintf, stderr);
```

Note

If you call `fprintf()` on a stream that you have not already sent output to, the library calls `malloc()` internally to create a buffer for the stream. If this happens in the middle of a call to `__heapstats()`, the heap might be corrupted. Therefore, you must ensure you have already sent some output to `stderr`.

If you are using the default one-region memory model, heap memory is allocated only as it is required. This means that the amount of free heap changes as you allocate and deallocate memory. For example, the sequence:

```
int *ip;
__heapstats((__heapprt)fprintf,stderr);   // print initial free heap size
ip = malloc(200000);
free(ip);
__heapstats((__heapprt)fprintf,stderr);    // print heap size after freeing
```

gives output such as:

```
4076 bytes in 1 free blocks (avge size 4076)
1 blocks 2^10+1 to 2^11
2008180 bytes in 1 free blocks (avge size 2008180)
1 blocks 2^19+1 to 2^20
```

This function is an extension that is specific to the ARM library.

Syntax

```
void __heapstats(int (*dprint)( void *param, char const *format,...),
                 void* param)
```

2.15.19 __heapvalid()

The `__heapvalid()` function performs a consistency check on the heap. It outputs full information about every free block if the *verbose* parameter is nonzero. Otherwise, it only outputs errors.

The function outputs its results by calling the output function *dprint*, that must work like `fprintf()`. The first parameter passed to *dprint* is the supplied pointer *param*. You can pass `fprintf()` itself, provided you cast it to the right function pointer type. This type is defined as a **typedef** for convenience. It is called `__heapprt`. For example:

Example 2-13 Calling __heapvalid() with fprintf()

```
__heapvalid((__heapprt) fprintf, stderr, 0);
```

Note

If you call `fprintf()` on a stream that you have not already sent output to, the library calls `malloc()` internally to create a buffer for the stream. If this happens in the middle of a call to `__heapvalid()`, the heap might be corrupted. You must therefore ensure you have already sent some output to `stderr`. The code in Example 2-13 fails if you have not already written to the stream.

This function is an extension that is specific to the ARM library.

Syntax

```
int __heapvalid(int (*dprint)( void *param, char const *format,...), void*
                param, int verbose)
```

2.15.20 _membitcpybl(), _membitcpybb(), _membitcpyhl(), _membitcpyhb(), _membitcpywl(), _membitcpywb(), _membitmovebl(), _membitmovebb(), _membitmovehl(), _membitmovehb(), _membitmovewl(), _membitmovewb()

Similar to the standard C library `memcpy()` and `memmove()` functions, these functions provide efficient bit-aligned memory operations.

Syntax

```
void _membitcpy[b|h|w][b|l](void *dest, const void *src, int dest_offset, int
src_offset, size_t nbits);
void _membitmove[b|h|w][b|l](void *dest, const void *src, int dest_offset, int
src_offset, size_t nbits);
```

Usage

The number of contiguous bits specified by *nbits* is copied, or moved (depending on the function being used), from a memory location starting *src_offset* bits after (or before if a negative offset) the address pointed to by *src*, to a location starting *dest_offset* bits after (or before if a negative offset) the address pointed to by *dest*.

To define a contiguous sequence of bits, a form of ordering is required. The variants of each function define this order, as follows:

- Functions whose second-last character is `b`, for example `_membitcpybl()`, are byte-oriented. Byte-oriented functions consider all of the bits in one byte to come before the bits in the next byte.
- Functions whose second-last character is `h` are halfword-oriented.
- Functions whose second-last character is `w` are word-oriented.

Within each byte, halfword, or word, the bits can be considered to go in different order depending on the endianness. Functions ending in `b`, for example `_membitmovewb()`, are bitwise big-endian. This means that the *Most Significant Bit* (MSB) of each byte, halfword, or word (as appropriate) is considered to be the first bit in the word, and the *Least Significant Bit* (LSB) is considered to be the last. Functions ending in `l` are bitwise little-endian. They consider the LSB to come first and the MSB to come last.

As with `memcpy()` and `memmove()`, the bitwise memory copying functions copy as fast as they can in their assumption that source and destination memory regions do not overlap, whereas the bitwise memory move functions ensure that source data in overlapping regions is copied before being overwritten.

On a little-endian platform, the big-endian functions are distinct, but the little-endian functions use the same bit ordering, so they are synonymous symbols that refer to the same function. On a big-endian platform, the big-endian functions are all effectively the same, but the little-endian functions are distinct.

2.16 Library naming conventions

The library naming convention described in this section applies to the current release of RVCT. Do not rely on the library names because they might change in future releases.

2.16.1 Placing ARM libraries

Normally, you do not have to list any of these libraries explicitly on the linker command line. The ARM linker automatically selects the correct C or C++ libraries to use, and it might use several, based on the accumulation of the object attributes.

If library names are explicitly named in a makefile, you must rebuild your project as follows:

1. Remove the explicit references to the old library names from the linker command-line.
2. Add `--info libraries` to the linker command-line and rebuild the project. This produces a list of all the libraries in use.
3. Add the new list of libraries to the linker command-line.

To include a specific library in a scatter-loading description file, see *Placing ARM C library code* on page 5-43 in the *Linker User Guide.*

2.16.2 Helper functions

Compiler support, or *helper*, functions specific to RVCT are typically used when the compiler cannot easily produce a suitable code sequence itself.

In RVCT v4.0 and later, the helper functions are generated by the compiler in the resulting object files.

In RVCT v3.1 and earlier, the helper functions reside in libraries. Because these libraries are specific to the ARM C compiler, they are intended to be redistributed as necessary with your own code. For example, if you are distributing a library to a third party, they might also require the appropriate helper library to link their final application successfully. For more information about the redistribution rights of the libraries, see your End User License Agreement.

2.16.3 Identifying library variants

The library filename identifies how the variant was built. The values for the fields of the filename, and the relevant build options are:

**root/prefix_arch[fpu][entrant].endian*

root	armlib	An ARM C library.
	cpplib	An ARM C++ library.
prefix	c	ISO C and C++ basic runtime support.
	cpp	Rogue Wave C++ library.
	cpprt	The ARM C++ runtime libraries.
	f	IEEE compliant library with a fixed rounding mode (Round to nearest) and no inexact exceptions.
	fj	IEEE compliant library with a fixed rounding mode (Round to nearest) and no exceptions.
	fz	Behaves like the fj library, but additionally flushes denormals and infinities to zero. This library behaves like the ARM VFP in Fast mode. This is the default.
	g	IEEE compliant library with configurable rounding mode and all IEEE exceptions.
	h	Compiler support (helper) library. See *Helper functions* on page 2-117.
	m	Transcendental math functions.
	mc	Non compliant ISO C micro-library basic runtime support.
	mf	Non compliant IEEE 754 micro-library support.
arch	4	An ARM only library for use with ARMv4.
	t	An ARM/Thumb interworking library for use with ARMv4T.
	5	An ARM/Thumb interworking library for use with ARMv5T and later. **Note** Architectures that the 5 variant is used for contain subvariants of function implementations that lend themselves to optimization for a particular core. memcpy() is one such example. The subvariant that is selected is based on the build attributes of input object files at link time.
	w	A Thumb-2 only library for use with Cortex-M3.
	p	A Thumb-1 only library for use with Cortex-M1 and Cortex-M0.

fpu	v	Uses VFP instruction set.
	s	Soft VFP.

Note

If neither v nor s is present in a library name, the library uses no floating point.

entrant	e	Position-independent access to static data.
	f	FPIC addressing is enabled.

Note

If neither e nor f is present in a library name, the library uses position-dependent access to static data.

endian	l	Little-endian.
	b	Big-endian.

For example:

```
*armlib/c_4.b
*cpplib/cpprt_5f.l
```

Note

Not all variant/name combinations are valid. See the armlib and cpplib directories for the libraries that are supplied with RVCT.

The linker command-line option --info libraries provides information on every library automatically selected for the link stage. See *--info=topic[,topic,...]* on page 2-44 in the *Linker Reference Guide* for more information.

See *Specifying the target processor or architecture* on page 2-23 in the *Compiler User Guide* for more information.

Chapter 3
The C Micro-library

This chapter describes the C *micro-library* (microlib). It contains the following sections:

See Chapter 2 *The C and C++ Libraries* for more information on the default libraries.

3.1 About microlib

Microlib is an alternative library to the default C library. It is intended for use with deeply embedded applications that must fit into extremely small amounts of memory. These applications do not run under an operating system.

Note

Microlib does not attempt to be a standards-compliant ISO C library.

Microlib is highly optimized for small code size. It has less functionality than the default C library and some ISO C features are completely missing. Some library functions are also slower.

3.1.1 Differences from the default C library

The main differences between microlib and the default C library are:

- Microlib is not compliant with the ISO C library standard. Some ISO features are not supported and others have less functionality.
- Microlib is not compliant with the IEEE 754 standard for binary floating point arithmetic.
- Microlib is highly optimized for small code size.
- Locales are not configurable. The default C locale is the only one available.
- `main()` must not be declared to take arguments and must not return.
- Microlib provides limited support for C99 functions.
- Microlib does not support C++.
- Microlib does not support operating system functions.
- Microlib does not support position-independent code.
- Microlib does not provide mutex locks to guard against code that is not thread safe.
- Microlib does not support wide characters or multibyte strings.
- Microlib does not support selectable one or two region memory models as the standard library (stdlib) does. Microlib provides only the two region memory model with separate stack and heap regions.

- Microlib does not support the bit-aligned memory functions `_membitcpy[b|h|w][b|l]()` and `membitmove[b|h|w][b|l]()`.
- Microlib can be used with either `--fpmode=std` or `--fpmode=fast`.
- The level of ANSI C `stdio` support that is provided can be controlled with `#pragma import(__use_full_stdio)`.
- `setvbuf()` and `setbuf()` always fail because all streams are unbuffered.
- `feof()` and `ferror()` always return `0` because the error and `EOF` indicators are not supported.

See *ISO C features missing from microlib* on page 3-10 for more information.

3.2 Building an application with microlib

This section describes how to link your application with microlib.

Functions in microlib are responsible for:

- Creating an environment that a C program can execute in. This includes:
  - creating a stack
  - creating a heap, if required
  - initializing the parts of the library the program uses.
- Starting execution by calling `main()`.

To build a program using microlib, you must use the command-line option `--library_type=microlib`. This option can be used by the compiler, assembler or linker. Use it with the linker to override all other options.

Example 3-1 shows `--library_type=microlib` being used by the compiler. Specifying `--library_type=microlib` when compiling `main.c` results in an object file containing an attribute that asks the linker to use microlib. Compiling `extra.c` with `--library_type=microlib` is unnecessary, because the request to link against microlib exists in the object file generated by compiling `main.c`.

Example 3-1 Compiler option

```
armcc --library_type=microlib -c main.c
armcc -c extra.c
armlink -o image.axf main.o extra.o
```

Example 3-2 shows this option being used by the assembler. The request to the linker to use microlib is made as a result of assembling `more.s` with `--library_type=microlib`.

Example 3-2 Assembler option

```
armcc -c main.c
armcc -c extra.c
armasm --library_type=microlib more.s
armlink -o image.axf main.o extra.o more.o
```

Example 3-3 on page 3-5 shows this option being used by the linker. Neither object file contains the attribute requesting that the linker link against microlib, so the linker selects microlib as a result of being explicitly asked to do so on the command line.

Example 3-3 Linker option

```
armcc -c main.c
armcc -c extra.c
armlink --library_type=microlib -o image.axf main.o extra.o
```

For more information see:

- *--library_type=lib* on page 2-81 in the *Compiler Reference Guide*
- *input_file_list* on page 2-48 in the *Linker Reference Guide*.

3.3 Using microlib

To begin, you must specify a starting pointer for the stack. To use the heap functions, for example, `malloc()`, `calloc()`, `realloc()` and `free()`, you must specify the location and size of the heap region.

3.3.1 Creating the stack

To specify the initial stack pointer you can use either of the following methods:

- use a scatter-loading description file
- define a symbol, `__initial_sp`, to be equal to the top of the stack.

For information on the scatter-loading description file method, see `ARM_LIB_STACK` and `ARM_LIB_STACKHEAP` in *Using a scatter-loading description file* on page 2-73.

Otherwise, specify the initial stack pointer by defining a symbol, `__initial_sp`, to be equal to the top of the stack. The initial stack pointer must be aligned to a multiple of eight bytes.

Example 3-4 shows how to set up the initial stack pointer using assembly language.

Example 3-4 Assembly language

```
    EXPORT __initial_sp
__initial_sp EQU 0x100000        ; equal to the top of the stack
```

Example 3-5 shows how to set up the initial stack pointer using embedded assembler in C.

Example 3-5 Embedded Assembler in C

```
__asm void dummy_function(void)
{
    EXPORT __initial_sp
__initial_sp EQU 0x100000        ; equal to the top of the stack
}
```

3.3.2 Creating the heap

To specify the start and end of the heap you can use either of the following methods:

- use a scatter-loading description file

- define symbols __heap_base and __heap_limit.

For information on the scatter-loading description file method, see ARM_LIB_HEAP, and ARM_LIB_STACKHEAP in *Using a scatter-loading description file* on page 2-73.

Otherwise, specify the start and end of the heap by defining symbols __heap_base and __heap_limit respectively. On completion, you can use the heap functions in the normal way.

Note

The __heap_limit must point to the byte beyond the last byte in the heap region.

Example 3-6 shows how to set up the heap pointers using assembly language.

Example 3-6 Assembly language

```
    EXPORT __heap_base
__heap_base EQU 0x400000        ; equal to the start of the heap
    EXPORT __heap_limit
__heap_limit EQU 0x800000       ; equal to the end of the heap
```

Example 3-7 shows how to set up the heap pointer using embedded assembler in C.

Example 3-7 Embedded Assembler in C

```
__asm void dummy_function(void)
{
    EXPORT __heap_base
__heap_base EQU 0x400000        ; equal to the start of the heap
    EXPORT __heap_limit
__heap_limit EQU 0x800000       ; equal to the end of the heap
}
```

3.3.3 Entering and exiting your program

Use main() to begin your program. Do not declare main() to take arguments.

Note

Your program must not return from main().

Microlib does not support:

- command-line arguments from an operating system
- programs that call `exit()`.

3.4 Tailoring the microlib input/output functions

Microlib provides a limited `stdio` subsystem that supports unbuffered `stdin`, `stdout` and `stderr` only. This enables you to use `printf()` for displaying diagnostic messages from your application.

To use high level I/O functions you must provide your own implementation of the following base functions so that they work with your own I/O device.

`fputc()` Implement this base function for all output functions. For example, `fprintf()`, `printf()`, `fwrite()`, `fputs()`, `puts()`, `putc()` and `putchar()`.

`fgetc()` Implement this base function for all input functions. For example, `fscanf()`, `scanf()`, `fread()`, `read()`, `fgets()`, `gets()`, `getc()` and `getchar()`.

`__backspace()`
Implement this base function if your input functions use `scanf()` or `fscanf()`.

Note

Conversions that are not supported in microlib are `%lc`, `%ls` and `%a`.

See *Tailoring the input/output functions* on page 2-78 for more information.

3.5 ISO C features missing from microlib

This section provides a list of the major ISO C90 features that are not supported by microlib.

Wide character and multibyte support

All functions dealing with wide characters or multibyte strings are not supported by microlib. A link error is generated if these are used. For example, `mbtowc()`, `wctomb()`, `mbstowcs()` and `wcstombs()`. All functions defined in Normative Addendum 1 are not supported by microlib.

Operating system interaction

All functions that interact with an operating system are not supported by microlib. For example, `abort()`, `exit()`, `atexit()`, `clock()`, `time()`, `system()` and `getenv()`.

File I/O By default, all the `stdio` functions that interact with a file pointer return an error if called. The only exceptions to this are the three standard streams `stdin`, `stdout` and `stderr`.

You can change this behavior using `#pragma import(__use_full_stdio)`. Use of this pragma provides a microlib version of `stdio` that supports ANSI C, with only the following exceptions:

- the error and `EOF` indicators are not supported, so `feof()` and `ferror()` return `0`
- all streams are unbuffered, so `setvbuf()` and `setbuf()` fail.

Configurable locale

The default C locale is the only one available.

Signals The functions `signal()` and `raise()` are provided but microlib does not generate signals. The only exception to this is if the program explicitly calls `raise()`.

Floating point support

Floating point support diverges from IEEE 754 in the following ways, but uses the same data formats and matches IEEE 754 in operations involving only normalized numbers:

- Operations involving NaNs, infinities or denormals can produce unpredictable results.
- IEEE exceptions cannot be flagged by microlib, and there is no `fp_status()` register in microlib.

- The sign of zero is not treated as significant by microlib, and zeroes that are output from microlib floating point arithmetic have an unpredictable sign bit.
- Only the default rounding mode is supported.

Position independent and thread safe code

Microlib has no reentrant variant. Microlib does not provide mutex locks to guard against code that is not thread safe. Use of microlib is not compatible with FPIC or RWPI compilation modes, and although ROPI code can be linked with microlib, the resulting binary is not ROPI-compliant overall.

Chapter 4
Floating-point Support

This chapter describes the ARM support for floating-point computations. It contains the following sections:

4.1 The software floating-point library, fplib

When programs are compiled to use a floating-point coprocessor, they perform basic floating-point arithmetic by means of floating-point machine instructions for the target coprocessor. When programs are compiled to use software floating-point, there is no floating-point instruction set available, so the ARM libraries must provide a set of procedure calls to do floating-point arithmetic. These procedures are in the software floating-point library, fplib.

4.1.1 Features of the floating-point library, fplib

Floating-point routines have names like `__aeabi_dadd` (add two **`double`**s) and `__aeabi_fdiv` (divide two **`float`**s). User programs can call these routines directly. Even in environments with a coprocessor, the routines are provided. However, they are typically only a few instructions long (because all they do is to execute the appropriate coprocessor instruction).

All the fplib routines are called using a software floating-point variant of the calling standard. This means that floating-point arguments are passed and returned in integer registers. By contrast, if the program is compiled for a coprocessor, floating-point data is passed in its floating-point registers.

So, for example, `__aeabi_dadd` takes a **`double`** in registers `r0` and `r1`, and another **`double`** in registers `r2` and `r3`, and returns the sum in `r0` and `r1`.

Note

For a **`double`** in registers `r0` and `r1`, the register that holds the high 32 bits of the **`double`** depends on whether your program is little-endian or big-endian.

All the fplib routines (except those listed in Table 4-5 on page 4-8) are declared in the header file `rt_fp.h`. You can include this file if you want to call an fplib routine directly. The routines shown in Table 4-5 on page 4-8 are declared in the standard header file `math.h`.

To call a function from assembler, the software floating-point function is called `__softfp_`*fn*. For example, to call the `cos()` function, do the following:

```
IMPORT __softfp_cos
BL __softfp_cos
```

4.1.2 Arithmetic on numbers in a particular format

Table 4-1 describes routines to perform arithmetic on numbers in a particular format. Arguments and return types are always in the same format.

Table 4-1 Arithmetic routines

Function	Argument types	Return type	Operation
`__aeabi_fadd`	2 `float`	`float`	Return x plus y
`__aeabi_fsub`	2 `float`	`float`	Return x minus y
`__aeabi_frsub`	2 `float`	`float`	Return y minus x
`__aeabi_fmul`	2 `float`	`float`	Return x times y
`__aeabi_fdiv`	2 `float`	`float`	Return x divided by y
`_frdiv`	2 `float`	`float`	Return y divided by x
`_frem`	2 `float`	`float`	Return remainder of x by y (see a in *Notes on arithmetic routines* on page 4-4)
`_frnd`	`float`	`float`	Return x rounded to an integer (see b in *Notes on arithmetic routines* on page 4-4)
`_fsqrt`	`float`	`float`	Return square root of x
`__aeabi_dadd`	2 `double`	`double`	Return x plus y
`__aeabi_dsub`	2 `double`	`double`	Return x minus y
`__aeabi_drsub`	2 `double`	`double`	Return y minus x
`__aeabi_dmul`	2 `double`	`double`	Return x times y
`__aeabi_ddiv`	2 `double`	`double`	Return x divided by y
`_drdiv`	2 `double`	`double`	Return y divided by x
`_drem`	2 `double`	`double`	Return remainder of x by y (see a in *Notes on arithmetic routines* on page 4-4)
`_drnd`	`double`	`double`	Return x rounded to an integer (see b in *Notes on arithmetic routines* on page 4-4)
`_dsqrt`	`double`	`double`	Return square root of x

Notes on arithmetic routines

a Functions that perform the IEEE 754 remainder operation. This is defined to take two numbers, *x* and *y*, and return a number *z* so that $z = x - n * y$, where *n* is an integer. To return an exactly correct result, *n* is chosen so that *z* is no bigger than half of *x* (so that *z* might be negative even if both *x* and *y* are positive). The IEEE 754 remainder function is not the same as the operation performed by the C library function `fmod`, where *z* always has the same sign as *x*. Where the IEEE 754 specification gives two acceptable choices of *n*, the even one is chosen. This behavior occurs independently of the current rounding mode.

b Functions that perform the IEEE 754 round-to-integer operation. This takes a number and rounds it to an integer (in accordance with the current rounding mode), but returns that integer in the floating-point number format rather than as a C **`int`** variable. To convert a number to an **`int`** variable, you must use the `_ffix` routines described in Table 4-2.

4.1.3 Conversions between floats, doubles, and ints

Table 4-2 describes routines to perform conversions between number formats, excluding **`long long`**s.

Table 4-2 Number format conversion routines

Function	Argument types	Return type
`__aeabi_f2d`	**`float`**	**`double`**
`__aeabi_d2f`	**`double`**	**`float`**
`_fflt`	**`int`**	**`float`**
`_ffltu`	**`unsigned int`**	**`float`**
`_dflt`	**`int`**	**`double`**
`_dfltu`	**`unsigned int`**	**`double`**
`_ffix`	**`float`**	**`int`** (see *Notes on rounding* on page 4-5)
`_ffix_r`	**`float`**	**`int`**
`_ffixu`	**`float`**	**`unsigned int`** (see *Notes on rounding* on page 4-5)

Table 4-2 Number format conversion routines (continued)

Function	Argument types	Return type
_ffixu_r	**float**	**unsigned int**
_dfix	**double**	**int** (see *Notes on rounding*)
_dfix_r	**double**	**int**
_dfixu	**double**	**unsigned int** (see *Notes on rounding*)
_dfixu_r	**double**	**unsigned int**

Notes on rounding

Rounded toward zero, independently of the current rounding mode. This is because the C standard requires implicit conversions to integers to round this way, so it is convenient not to have to change the rounding mode to do so. Each function has a corresponding function with _r on the end of its name, that performs the same operation but rounds according to the current mode.

4.1.4 Conversions between long longs and other number formats

Table 4-3 describes routines to perform conversions between **long long**s and other number formats.

Table 4-3 Conversion routines involving long long format

Function	Argument types	Return type
_ll_sto_f	**long long**	**float**
_ll_uto_f	**unsigned long long**	**float**
_ll_sto_d	**long long**	**double**
_ll_uto_d	**unsigned long long**	**double**
_ll_sfrom_f	**float**	**long long** (see *Notes on rounding* on page 4-6)
_ll_sfrom_f_r	**float**	**long long**

Table 4-3 Conversion routines involving long long format (continued)

Function	Argument types	Return type
`_ll_ufrom_f`	**`float`**	**`unsigned long long`** (see *Notes on rounding* on page 4-6)
`_ll_ufrom_f_r`	**`float`**	**`unsigned long long`**
`_ll_sfrom_d`	**`double`**	**`long long`** (see *Notes on rounding*)
`_ll_sfrom_d_r`	**`double`**	**`long long`**
`_ll_ufrom_d`	**`double`**	**`unsigned long long`** (see *Notes on rounding*)
`_ll_ufrom_d_r`	**`double`**	**`unsigned long long`**

Notes on rounding

Rounded toward zero, independently of the current rounding mode. This is because the C standard requires implicit conversions to integers to round this way, so it is convenient not to have to change the rounding mode to do so. This function has a corresponding function with `_r` on the end of its name. This function performs the same operation but rounds according to the current mode.

4.1.5 Floating-point comparisons

Table 4-4 describes routines to perform comparisons between floating-point numbers. See *Notes on floating-point comparison routines* on page 4-7 for more information as indicated in the **Notes** column.

Table 4-4 Floating-point comparison routines

Function	Argument types	Return type	Condition tested	Notes
`_fcmpeq`	2 **`float`**	Flags, EQ/NE	x equal to y	a
`_fcmpge`	2 **`float`**	Flags, HS/LO	x greater than or equal to y	a, b
`_fcmple`	2 **`float`**	Flags, HI/LS	x less than or equal to y	a, b
`_feq`	2 **`float`**	Boolean	x equal to y	-
`_fneq`	2 **`float`**	Boolean	x not equal to y	-

Table 4-4 Floating-point comparison routines (continued)

Function	Argument types	Return type	Condition tested	Notes
_fgeq	2 **float**	Boolean	x greater than or equal to y	b
_fgr	2 **float**	Boolean	x greater than y	b
_fleq	2 **float**	Boolean	x less than or equal to y	b
_fls	2 **float**	Boolean	x less than y	b
_dcmpeq	2 **double**	Flags, EQ/NE	x equal to y	a
_dcmpge	2 **double**	Flags, HS/LO	x greater than or equal to y	a, b
_dcmple	2 **double**	Flags, HI/LS	x less than or equal to y	a, b
_deq	2 **double**	Boolean	x equal to y	-
_dneq	2 **double**	Boolean	x not equal to y	-
_dgeq	2 **double**	Boolean	x greater than or equal to y	b
_dgr	2 **double**	Boolean	x greater than y	b
_dleq	2 **double**	Boolean	x less than or equal to y	b
_dls	2 **double**	Boolean	x less than y	b
_fcmp4	2 **float**	Flags, VFP	x less than or equal to y	c
_fcmp4e	2 **float**	Flags, VFP	x less than or equal to y	b, c
_fdcmp4	**float**, **double**	Flags, VFP	x less than or equal to y	c
_fdcmp4e	**float**, **double**	Flags, VFP	x less than or equal to y	b, c
_dcmp4	2 **double**	Flags, VFP	x less than or equal to y	c
_dcmp4e	2 **double**	Flags, VFP	x less than or equal to y	b, c
_dfcmp4	**double**, **float**	Flags, VFP	x less than or equal to y	c
_dfcmp4e	**double**, **float**	Flags, VFP	x less than or equal to y	b, c

Notes on floating-point comparison routines

a Returns results in the ARM condition flags. This is efficient in assembly language, because you can directly follow a call to the function with a conditional instruction, but it means there is no way to use this function from C. This function is not declared in `rt_fp.h`.

b Causes an Invalid Operation exception if either argument is a NaN, even a quiet NaN. Other functions only cause Invalid Operation if an argument is an SNaN. QNaNs return *not equal* when compared to anything, including other QNaNs (so comparing a QNaN to the same QNaN still returns *not equal*).

c Returns VFP-type status flags in the `CPSR`. Also returns VFP-type status flags in the top four bits of `r0`, meaning that it is possible to use this function from C. This function is declared in `rt_fp.h`.

4.1.6 C99 functions

Table 4-5 describes routines that implement C99 functionality.

Table 4-5 C99 routines

Function	Argument types	Return type	Returns section	Standard
`ilogb`	**`double`**	**`int`**	Exponent of argument x	7.12.6.5
`ilogbf`	**`float`**	**`int`**	Exponent of argument x	7.12.6.5
`ilogbl`	**`long double`**	**`int`**	Exponent of argument x	7.12.6.5
`logb`	**`double`**	**`double`**	Exponent of argument x	7.12.6.11
`logbf`	**`float`**	**`float`**	Exponent of argument x	7.12.6.11
`logbl`	**`long double`**	**`long double`**	Exponent of argument x	7.12.6.11
`scalbn`	**`double`**, **`int`**	**`double`**	x * (`FLT_RADIX` ** n)	7.12.6.13
`scalbnf`	**`float`**, **`int`**	**`float`**	x * (`FLT_RADIX` ** n)	7.12.6.13
`scalbnl`	**`long double`**, **`int`**	**`long double`**	x * (`FLT_RADIX` ** n)	7.12.6.13
`scalbln`	**`double`**, **`long int`**	**`double`**	x * (`FLT_RADIX` ** n)	7.12.6.13
`scalblnf`	**`float`**, **`long int`**	**`float`**	x * (`FLT_RADIX` ** n)	7.12.6.13
`scalblnl`	**`long double`**, **`long int`**	**`long double`**	x * (`FLT_RADIX` ** n)	7.12.6.13
`nextafter`	2 **`double`**	**`double`**	Next representable value after x towards y	7.12.11.3
`nextafterf`	2 **`float`**	**`float`**	Next representable value after x towards y	7.12.11.3

Table 4-5 C99 routines (continued)

Function	Argument types	Return type	Returns section	Standard
nextafterl	2 **long double**	**long double**	Next representable value after x towards y	7.12.11.3
nexttoward	**double**, **long double**	**double**	Next representable value after x towards y	7.12.11.4
nexttowardf	**float**, **long double**	**float**	Next representable value after x towards y	7.12.11.4
nexttowardl	2 **long double**	**long double**	Next representable value after x towards y	7.12.11.4

4.2 Controlling the floating-point environment

This section describes the functions you can use to control the ARM floating-point environment. RVCT supplies several different interfaces to the floating-point environment, for compatibility and porting ease. With these functions, you can change the rounding mode, enable and disable trapping of exceptions, and install your own custom exception trap handlers.

4.2.1 __ieee_status()

RVCT supports an interface to the status word in the floating-point environment. This is called `__ieee_status` and it is generally the most efficient function to use for modifying the status word for VFP. `__ieee_status` is defined in `fenv.h`.

`__ieee_status` has the prototype:

```
unsigned int __ieee_status(unsigned int mask, unsigned int flags);
```

The function modifies the writable parts of the status word according to the parameters, and returns the previous value of the whole word.

The writable bits are modified by setting them to:

```
new = (old & ~mask) ^ flags;
```

Four different operations can be performed on each bit of the status word, depending on the corresponding bits in mask and flags (see Table 4-6).

Table 4-6 Status word bit modification

Bit of mask	Bit of flags	Effect
0	0	Leave alone
0	1	Toggle
1	0	Set to 0
1	1	Set to 1

The layout of the status word as seen by `__ieee_status` is shown in Figure 4-1.

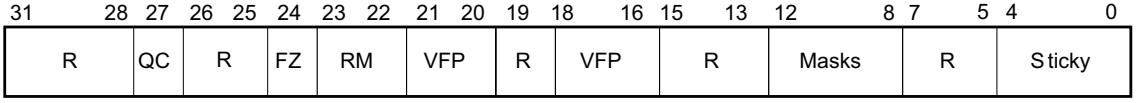

Figure 4-1 IEEE status word layout

The fields in Figure 4-1 on page 4-10 are as follows:

- Bits 0 to 4 (values `0x1` to `0x10`, respectively) are the sticky flags, or cumulative flags, for each exception. The sticky flag for an exception is set to 1 whenever that exception happens and is not trapped. Sticky flags are never cleared by the system, only by the user. The mapping of exceptions to bits is:
  - bit 0 (`0x01`) is for the Invalid Operation exception
  - bit 1 (`0x02`) is for the Divide by Zero exception
  - bit 2 (`0x04`) is for the Overflow exception
  - bit 3 (`0x08`) is for the Underflow exception
  - bit 4 (`0x10`) is for the Inexact Result exception.
- Bits 8 to 12 (values `0x100` to `0x1000`) are the exception masks. These control whether each exception is trapped or not. If a bit is set to 1, the corresponding exception is trapped. If a bit is set to 0, the corresponding exception sets its sticky flag and returns a plausible result, as described in *Exceptions* on page 4-41.
- Bits 16 to 18, and bits 20 and 21, are used by VFP hardware to control the VFP vector capability. The `__ieee_status` call does not let you modify these bits.
- Bits 22 and 23 control the rounding mode (Table 4-7).

Table 4-7 Rounding mode control

Bits	Rounding mode
`00`	Round to nearest
`01`	Round up
`10`	Round down
`11`	Round toward zero

Note

The standard fplib libraries `f*` support only the Round to nearest rounding mode. If you require support for the other rounding modes, you must use the full IEEE `g*` libraries. See *Library naming conventions* on page 2-117.

- Bit 24 enables FZ (Flush to Zero) mode if it is set. In FZ mode, denormals are forced to zero to speed up processing (because denormals can be difficult to work with and slow down floating-point systems). Setting this bit reduces accuracy but might increase speed.

Note

The standard fplib libraries do not support the FZ mode. Instead, each library either always flushes to zero or never flushes to zero, and you choose the library to use at build time. Only the VFP hardware honors the FZ mode if bit 24 is set.

However, this means that functions that are not provided in the hardware (and so are supplied in software fplib) do not support the FZ mode even when compiling for hard VFP. As a result, the behavior of fplib is not consistent across all functions when you change the FZ mode dynamically.

- Bit 27 indicates that saturation has occurred in an advanced SIMD saturating integer operation. This is accessible through the `__ieee_status` call.
- Bits marked R are reserved. They cannot be written to by the `__ieee_status` call, and you must ignore anything you find in them.

In addition to defining the `__ieee_status` call itself, `fenv.h` also defines some constants to be used for the arguments:

```
#define FE_IEEE_FLUSHZERO          (0x01000000)
#define FE_IEEE_ROUND_TONEAREST    (0x00000000)
#define FE_IEEE_ROUND_UPWARD       (0x00400000)
#define FE_IEEE_ROUND_DOWNWARD     (0x00800000)
#define FE_IEEE_ROUND_TOWARDZERO   (0x00C00000)
#define FE_IEEE_ROUND_MASK         (0x00C00000)
#define FE_IEEE_MASK_INVALID       (0x00000100)
#define FE_IEEE_MASK_DIVBYZERO     (0x00000200)
#define FE_IEEE_MASK_OVERFLOW      (0x00000400)
#define FE_IEEE_MASK_UNDERFLOW     (0x00000800)
#define FE_IEEE_MASK_INEXACT       (0x00001000)
#define FE_IEEE_MASK_ALL_EXCEPT    (0x00001F00)
#define FE_IEEE_INVALID            (0x00000001)
#define FE_IEEE_DIVBYZERO          (0x00000002)
#define FE_IEEE_OVERFLOW           (0x00000004)
#define FE_IEEE_UNDERFLOW          (0x00000008)
#define FE_IEEE_INEXACT            (0x00000010)
#define FE_IEEE_ALL_EXCEPT         (0x0000001F)
```

For example, to set the rounding mode to round down, you would do:

```
__ieee_status(FE_IEEE_ROUND_MASK, FE_IEEE_ROUND_DOWNWARD);
```

To trap the Invalid Operation exception and untrap all other exceptions:

```
__ieee_status(FE_IEEE_MASK_ALL_EXCEPT, FE_IEEE_MASK_INVALID);
```

To untrap the Inexact Result exception:

```
__ieee_status(FE_IEEE_MASK_INEXACT, 0);
```

To clear the Underflow sticky flag:

```
__ieee_status(FE_IEEE_UNDERFLOW, 0);
```

4.2.2 __fp_status()

Previous versions of the ARM libraries implemented a function called `__fp_status` that manipulated a status word in the floating-point environment. This is the same as `__ieee_status` but uses an older-style status word layout. The ARM compiler still supports the `__fp_status` function for backwards compatibility. `__fp_status` is defined in `stdlib.h`.

`__fp_status` has the following prototype:

```
unsigned int __fp_status(unsigned int mask, unsigned int flags);
```

The layout of the status word as seen by `__fp_status` isshown in Figure 4-2.

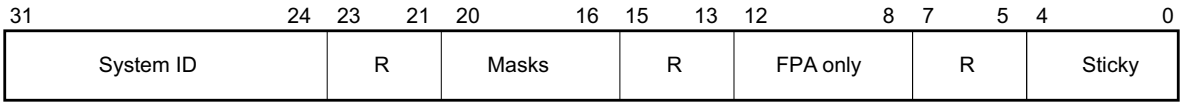

Figure 4-2 Floating-point status word layout

The fields in Figure 4-2 are as follows:

- Bits 0 to 4 are the sticky flags, as described in *__ieee_status()* on page 4-10.
- Bits 8 to 12 (values `0x100` to `0x1000`) control various aspects of the FPA floating-point. The FPA architecture is obsolete and is not supported by RVCT. Any attempt to write to these bits is ignored.
- Bits 16 to 20 (values `0x10000` to `0x100000`) are the exception masks, as described in *__ieee_status()* on page 4-10, but in a different place.
- Bits 24 to 31 contain the system ID that cannot be changed. It is set to `0x40` for software floating-point, to `0x80` or above for hardware floating-point, and to `0` or 1 if a hardware floating-point environment is being faked by an emulator.
- Bits marked `R` are reserved. They cannot be written to by the `__fp_status` call, and you must ignore anything you find in them.

The rounding mode cannot be changed with the `__fp_status` call.

In addition to defining the `__fp_status` call itself, `stdlib.h` also defines some constants to be used for the arguments:

```
#define __fpsr_IXE  0x100000
#define __fpsr_UFE  0x80000
#define __fpsr_OFE  0x40000
#define __fpsr_DZE  0x20000
#define __fpsr_IOE  0x10000
#define __fpsr_IXC  0x10
#define __fpsr_UFC  0x8
#define __fpsr_OFC  0x4
#define __fpsr_DZC  0x2
#define __fpsr_IOC  0x1
```

For example, to trap the Invalid Operation exception and untrap all other exceptions, you would do:

```
__fp_status(_fpsr_IXE | _fpsr_UFE | _fpsr_OFE |
            _fpsr_DZE | _fpsr_IOE, _fpsr_IOE);
```

To untrap the Inexact Result exception:

```
__fp_status(_fpsr_IXE, 0);
```

To clear the Underflow sticky flag:

```
__fp_status(_fpsr_UFC, 0);
```

4.2.3 Microsoft compatibility functions

The following functions are implemented for compatibility with Microsoft products, to ease porting of floating-point code to the ARM architecture. They are defined in `float.h`.

_controlfp()

The function `_controlfp()` enables you to control exception traps and rounding modes:

```
unsigned int _controlfp(unsigned int new, unsigned int mask);
```

This function also modifies a control word using a mask to isolate the bits to modify. For every bit of `mask` that is zero, the corresponding control word bit is unchanged. For every bit of `mask` that is nonzero, the corresponding control word bit is set to the value of the corresponding bit of `new`. The return value is the previous state of the control word.

Note

This is not quite the same as the behavior of `__ieee_status()` (or `__fp_status()`), where you can toggle a bit by setting a zero in the mask word and a one in the flags word.

Table 4-8 describes the macros you can use to form the arguments to `_controlfp()`.

Table 4-8 _controlfp argument macros

Macro	Description
`_MCW_EM`	Mask containing all exception bits
`_EM_INVALID`	Bit describing the Invalid Operation exception
`_EM_ZERODIVIDE`	Bit describing the Divide by Zero exception
`_EM_OVERFLOW`	Bit describing the Overflow exception
`_EM_UNDERFLOW`	Bit describing the Underflow exception
`_EM_INEXACT`	Bit describing the Inexact Result exception
`_MCW_RC`	Mask for the rounding mode field
`_RC_CHOP`	Rounding mode value describing Round Toward Zero
`_RC_UP`	Rounding mode value describing Round Up
`_RC_DOWN`	Rounding mode value describing Round Down
`_RC_NEAR`	Rounding mode value describing Round To Nearest

Note

The values of these macros are not guaranteed to remain the same in future versions of ARM products. To ensure that your code continues to work if the value changes in future releases, use the macro rather than its value.

For example, to set the rounding mode to round down, you would do:

```
_controlfp(_RC_DOWN, _MCW_RC);
```

To trap the Invalid Operation exception and untrap all other exceptions:

```
_controlfp(_EM_INVALID, _MCW_EM);
```

To untrap the Inexact Result exception:

```
_controlfp(0, _EM_INEXACT);
```

_clearfp()

The function `_clearfp()` clears all five exception sticky flags, and returns their previous value. The macros given in Table 4-8 on page 4-15, for example `_EM_INVALID`, `_EM_ZERODIVIDE`, can be used to test bits of the returned result.

`_clearfp()` has the following prototype:

```
unsigned _clearfp(void);
```

_statusfp()

The function `_statusfp()` returns the current value of the exception sticky flags. You can use the macros shown in Table 4-8 on page 4-15 to test bits of the returned result, for example, `_EM_INVALID`, `_EM_ZERODIVIDE`.

`_statusfp` has the following prototype:

```
unsigned _statusfp(void);
```

4.2.4 C99-compatible functions

The ARM compiler also supports a set of functions defined in the C99 standard, in addition to the functions described previously.

These C99-compatible functions are the only interface that enables you to install custom exception trap handlers with the ability to invent a return value. All the functions, types, and macros in this section are defined in `fenv.h`.

C99 defines two data types, `fenv_t` and `fexcept_t`. The C99 standard does not give information about these types, so for portable code you must treat them as opaque. The ARM compiler defines them to be structure types. See *ARM compiler extensions to the C99 interface* on page 4-19 for more information.

The type `fenv_t` is defined to hold all the information about the current floating-point environment:

- the rounding mode
- the exception sticky flags
- whether each exception is masked
- what handlers are installed, if any.

The type `fexcept_t` is defined to hold all the information relevant to a given set of exceptions.

C99 rounding mode and exception macros

C99 also defines a macro for each rounding mode and each exception. The macros are as follows:

```
FE_DIVBYZERO
FE_INEXACT
FE_INVALID
FE_OVERFLOW
FE_UNDERFLOW
FE_ALL_EXCEPT
FE_DOWNWARD
FE_TONEAREST
FE_TOWARDZERO
FE_UPWARD
```

The exception macros are bit fields. The macro `FE_ALL_EXCEPT` is the bitwise OR of all of them.

Handling exception flags

C99 provides three functions to clear, test and raise exceptions:

```
void feclearexcept(int excepts);
int fetestexcept(int excepts);
void feraiseexcept(int excepts);
```

The `feclearexcept()` function clears the sticky flags for the given exceptions. The `fetestexcept()` function returns the bitwise OR of the sticky flags for the given exceptions (so that if the Overflow flag was set but the Underflow flag was not, then calling `fetestexcept(FE_OVERFLOW|FE_UNDERFLOW)` would return `FE_OVERFLOW`).

The `feraiseexcept()` function raises the given exceptions, in unspecified order. If an exception trap is enabled for an exception raised this way, it is called.

C99 also provides functions to save and restore everything about a given exception. This includes the sticky flag, whether the exception is trapped, and the address of the trap handler, if any. These functions are:

```
void fegetexceptflag(fexcept_t *flagp, int excepts);
void fesetexceptflag(const fexcept_t *flagp, int excepts);
```

The `fegetexceptflag()` function copies all the information relating to the given exceptions into the `fexcept_t` variable provided. The `fesetexceptflag()` function copies all the information relating to the given exceptions from the `fexcept_t` variable into the current floating-point environment.

Note

`fesetexceptflag()` can be used to set the sticky flag of a trapped exception to 1 without calling the trap handler, whereas `feraiseexcept()` calls the trap handler for any trapped exception.

Handling rounding modes

C99 provides two functions for controlling rounding modes:

```
int fegetround(void);
int fesetround(int round);
```

The `fegetround()` function returns the current rounding mode, that has a value equal to that of one of the macros listed in *C99 rounding mode and exception macros* on page 4-17. The `fesetround()` function sets the current rounding mode to the value provided. `fesetround()` returns zero for success, or nonzero if its argument is not a valid rounding mode.

Saving the whole environment

C99 provides functions to save and restore the entire floating-point environment:

```
void fegetenv(fenv_t *envp);
void fesetenv(const fenv_t *envp);
```

The `fegetenv()` function stores the current state of the floating-point environment into the `fenv_t` variable provided. The `fesetenv()` function restores the environment from the variable provided.

Like `fesetexceptflag()`, `fesetenv()` does not call trap handlers when it sets the sticky flags for trapped exceptions.

Temporarily disabling exceptions

C99 provides two functions that enable you to avoid risking exception traps when executing code that might cause exceptions. This is useful when, for example, trapped exceptions are using the ARM default behavior. The default is to cause **SIGFPE** and terminate the application.

```
int feholdexcept(fenv_t *envp);
void feupdateenv(const fenv_t *envp);
```

The feholdexcept() function saves the current floating-point environment in the fenv_t variable provided, sets all exceptions to be untrapped, and clears all the exception sticky flags. You can then execute code that might cause unwanted exceptions, and make sure the sticky flags for those exceptions are cleared. Then you can call feupdateenv(). This restores any exception traps and calls them if necessary.

For example, suppose you have a function frob() that might cause the Underflow or Invalid Operation exceptions (assuming both exceptions are trapped). You are not interested in Underflow, but you want to know if an invalid operation is attempted. So you could do this:

```
fenv_t env;
feholdexcept(&env);
frob();
feclearexcept(FE_UNDERFLOW);
feupdateenv(&env);
```

Then, if the frob() function raises Underflow, it is cleared again by feclearexcept(), so no trap occurs when feupdateenv() is called. However, if frob() raises Invalid Operation, the sticky flag is set when feupdateenv() is called, so the trap handler is invoked.

This mechanism is provided by C99 because C99 specifies no way to change exception trapping for individual exceptions. A better method is to use __ieee_status() to disable the Underflow trap while leaving the Invalid Operation trap enabled. This has the advantage that the Invalid Operation trap handler is provided with all the information about the invalid operation (that is, what operation was being performed, and on what data), and can invent a result for the operation. Using the C99 method, the Invalid Operation trap handler is called after the fact, receives no information about the cause of the exception, and is called too late to provide a substitute result.

4.2.5 ARM compiler extensions to the C99 interface

The ARM compiler provides some extensions to the C99 interface, to enable it to do everything that the ARM floating-point environment is capable of. This includes trapping and untrapping individual exception types, and also installing custom trap handlers.

The types fenv_t and fexcept_t are not defined by C99 to be anything in particular. The ARM compiler defines them both to be the same structure type:

```
typedef struct{
    unsigned statusword;
    __ieee_handler_t __invalid_handler;
    __ieee_handler_t __divbyzero_handler;
    __ieee_handler_t __overflow_handler;
```

```
    __ieee_handler_t __underflow_handler;
    __ieee_handler_t __inexact_handler;
} fenv_t, fexcept_t;
```

The members of this structure are:

- `statusword` is the same status variable that the function `__ieee_status` sees, laid out in the same format (see *__ieee_status()* on page 4-10).
- Five function pointers giving the address of the trap handler for each exception. By default each is `NULL`. This means that if the exception is trapped then the default exception trap action happens. The default is to cause a **SIGFPE** signal.

Writing custom exception trap handlers

If you want to install a custom exception trap handler, declare it as a function like this:

```
__softfp__ieee_value_t myhandler(__ieee_value_t op1,
                                 __ieee_value_t op2,
                                 __ieee_edata_t edata);
```

The parameters to this function are:

- `op1` and `op2` are used to give the operands, or the intermediate result, for the operation that caused the exception:
  - For the Invalid Operation and Divide by Zero exceptions, the original operands are supplied.
  - For the Inexact Result exception, all that is supplied is the ordinary result that would have been returned anyway. This is provided in `op1`.
  - For the Overflow exception, an intermediate result is provided. This result is calculated by working out what the operation would have returned if the exponent range had been big enough, and then adjusting the exponent so that it fits in the format. The exponent is adjusted by 192 (`0xC0`) in single-precision, and by 1536 (`0x600`) in double-precision.

    If Overflow happens when converting a **`double`** to a **`float`**, the result is supplied in **`double`** format, rounded to single-precision, with the exponent biased by 192.
  - For the Underflow exception, a similar intermediate result is produced, but the bias value is added to the exponent instead of being subtracted. The `edata` parameter also contains a flag to show whether the intermediate result has had to be rounded up, down, or not at all.

  The type `__ieee_value_t` is defined as a union of all the possible types that an operand can be passed as:

```
typedef union{
    float __f;
    float __s;
    double __d;
    short __h;
    unsigned short __uh;
    int __i;
    unsigned int __ui;
    long long __l;
    unsigned long long __ul;
    ...
                                /* __STRICT_ANSI__ */
    struct { int __word1, __word2; } __str;
} __ieee_value_t;               /* in and out values passed to traps */
```

Note

If you do not compile with `--strict`, and you have code that used the older definition of `__ieee_value_t`, your older code still works. See the file `fenv.h` for more information.

- `edata` contains flags that give information about the exception that occurred, and what operation was being performed. (The type `__ieee_edata_t` is a synonym for **`unsigned int`**.)
- The return value from the function is used as the result of the operation that caused the exception.

The flags contained in `edata` are:

- `edata & FE_EX_RDIR` is nonzero if the intermediate result in Underflow was rounded down, and 0 if it was rounded up or not rounded. (The difference between the last two is given in the Inexact Result bit.) This bit is meaningless for any other type of exception.
- `edata & FE_EX_`*`exception`* is nonzero if the given *`exception`* (`INVALID`, `DIVBYZERO`, `OVERFLOW`, `UNDERFLOW`, or `INEXACT`) occurred. This enables you to:
  - use the same handler function for more than one exception type (the function can test these bits to tell what exception it is supposed to handle)
  - determine whether Overflow and Underflow intermediate results have been rounded or are exact.

  Because the `FE_EX_INEXACT` bit can be set in combination with either `FE_EX_OVERFLOW` or `FE_EX_UNDERFLOW`, you must determine the type of exception that actually occurred by testing Overflow and Underflow before testing Inexact.

- edata & FE_EX_FLUSHZERO is nonzero if the FZ bit was set when the operation was performed (see *__ieee_status()* on page 4-10).
- edata & FE_EX_ROUND_MASK gives the rounding mode that applies to the operation. This is normally the same as the current rounding mode, unless the operation that caused the exception was a routine such as _ffix, that always rounds toward zero. The available rounding mode values are FE_EX_ROUND_NEAREST, FE_EX_ROUND_PLUSINF, FE_EX_ROUND_MINUSINF and FE_EX_ROUND_ZERO.
- edata & FE_EX_INTYPE_*MASK* gives the type of the operands to the function, as one of the type values shown in Table 4-9.

Table 4-9 FE_EX_INTYPE_*MASK* operand type flags

Flag	Operand type
FE_EX_INTYPE_FLOAT	**float**
FE_EX_INTYPE_DOUBLE	**double**
FE_EX_INTYPE_FD	**float double**
FE_EX_INTYPE_DF	**double float**
FE_EX_INTYPE_HALF	**short**
FE_EX_INTYPE_INT	**int**
FE_EX_INTYPE_UINT	**unsigned int**
FE_EX_INTYPE_LONGLONG	**long long**
FE_EX_INTYPE_ULONGLONG	**unsigned long long**

- edata & FE_EX_OUTTYPE_*MASK* gives the type of the operands to the function, as one of the type values shown in Table 4-10.

Table 4-10 FE_EX_OUTTYPE_*MASK* operand type flags

Flag	Operand type
FE_EX_OUTTYPE_FLOAT	**float**
FE_EX_OUTTYPE_DOUBLE	**double**
FE_EX_OUTTYPE_HALF	**short**
FE_EX_OUTTYPE_INT	**int**

Table 4-10 FE_EX_OUTTYPE_*MASK* operand type flags (continued)

Flag	Operand type
FE_EX_OUTTYPE_UINT	**unsigned int**
FE_EX_OUTTYPE_LONGLONG	**long long**
FE_EX_OUTTYPE_ULONGLONG	**unsigned long long**

- edata & FE_EX_FN_*MASK* gives the nature of the operation that caused the exception, as one of the operation codes shown in Table 4-11.

Table 4-11 FE_EX_FN_*MASK* operation type flags

Flag	Operation type
FE_EX_FN_ADD	Addition.
FE_EX_FN_SUB	Subtraction.
FE_EX_FN_MUL	Multiplication.
FE_EX_FN_DIV	Division.
FE_EX_FN_REM	Remainder.
FE_EX_FN_RND	Round to integer.
FE_EX_FN_SQRT	Square root.
FE_EX_FN_CMP	Compare.
FE_EX_FN_CVT	Convert between formats.
FE_EX_FN_LOGB	Exponent fetching.

Table 4-11 FE_EX_FN_*MASK* operation type flags (continued)

Flag	Operation type
FE_EX_FN_SCALBN	Scaling. **Note** The FE_EX_INTYPE_MASK flag only specifies the type of the first operand. The second operand is always an **int**.
FE_EX_FN_NEXTAFTER	Next representable number. **Note** Both operands are the same type. Calls to nexttoward cause the value of the second operand to change to a value that is of the same type as the first operand. This does not affect the result.
FE_EX_FN_RAISE	The exception was raised explicitly, by feraiseexcept() or feupdateenv(). In this case, almost nothing in the edata word is valid.

When the operation is a comparison, the result must be returned as if it were an **int**, and must be one of the four values shown in Table 4-12.

Input and output types are the same for all operations except Compare and Convert.

Table 4-12 FE_EX_CMPRET_*MASK* comparison type flags

Flag	Comparison
FE_EX_CMPRET_LESS	op1 is less than op2
FE_EX_CMPRET_EQUAL	op1 is equal to op2
FE_EX_CMPRET_GREATER	op1 is greater than op2
FE_EX_CMPRET_UNORDERED	op1 and op2 are not comparable

Example exception handler

Example 4-1 on page 4-25 shows a custom exception handler. Suppose you are converting some Fortran code into C. The Fortran numerical standard requires 0 divided by 0 to be 1, whereas IEEE 754 defines 0 divided by 0 to be an Invalid Operation and so by default it returns a quiet NaN. The Fortran code is likely to rely on this behavior, and rather than modifying the code, it is probably easier to make 0 divided by 0 return 1.

When compiling, you must select a floating-point model that supports exceptions, for example `--fpmode=ieee_full` or `--fpmode=ieee_fixed`.

After the handler is installed, dividing 0.0 by 0.0 returns 1.0.

Example 4-1 Custom exception handler

```
#include <fenv.h>
#include <signal.h>
#include <stdio.h>
__softfp __ieee_value_t myhandler(__ieee_value_t op1, __ieee_value_t op2,
                                  __ieee_edata_t edata)
{
    __ieee_value_t ret;
    if ((edata & FE_EX_FN_MASK) == FE_EX_FN_DIV)
    {
        if ((edata & FE_EX_INTYPE_MASK) == FE_EX_INTYPE_FLOAT)
        {
            if (op1.f == 0.0 && op2.f == 0.0)
            {
                ret.f = 1.0;
                return ret;
            }
        }
        if ((edata & FE_EX_INTYPE_MASK) == FE_EX_INTYPE_DOUBLE)
        {
            if (op1.d == 0.0 && op2.d == 0.0)
            {
                ret.d = 1.0;
                return ret;
            }
        }
    }
    /* For all other invalid operations, raise SIGFPE as usual */
    raise(SIGFPE);
}
int main(void)
{
    float i, j, k;
    fenv_t env;
    fegetenv(&env);
    env.statusword |= FE_IEEE_MASK_INVALID;
    env.invalid_handler = myhandler;
    fesetenv(&env);
    i = 0.0;
    j = 0.0;
```

```
    k = i/j;
    printf("k is %f\n", k);
}
```

Exception trap handling by signals

If an exception is trapped but the trap handler address is set to `NULL`, a default trap handler is used.

The default trap handler raises a **SIGFPE** signal. The default handler for **SIGFPE** prints an error message and terminates the program.

If you trap **SIGFPE**, you can declare your signal handler function to have a second parameter that tells you the type of floating-point exception that occurred. This feature is provided for compatibility with Microsoft products. The values are `_FPE_INVALID`, `_FPE_ZERODIVIDE`, `_FPE_OVERFLOW`, `_FPE_UNDERFLOW` and `_FPE_INEXACT`. They are defined in `float.h`. For example:

```
void sigfpe(int sig, int etype){
    printf("SIGFPE (%s)\n",
           etype == _FPE_INVALID ? "Invalid Operation" :
           etype == _FPE_ZERODIVIDE ? "Divide by Zero" :
           etype == _FPE_OVERFLOW ? "Overflow" :
           etype == _FPE_UNDERFLOW ? "Underflow" :
           etype == _FPE_INEXACT ? "Inexact Result" :
           "Unknown");
}
signal(SIGFPE, (void(*)(int))sigfpe);
```

To generate your own **SIGFPE** signals with this extra information, you can call the function `__rt_raise()` instead of the ISO function `raise()`. In Example 4-1 on page 4-25, instead of:

```
    raise(SIGFPE);
```

it is better to code:

```
    __rt_raise(SIGFPE, _FPE_INVALID);
```

`__rt_raise()` is declared in `rt_misc.h`.

4.3 The math library, mathlib

In addition to the functions defined by the ISO C standard, mathlib provides the following C99 functions and macros. These are in addition to those provided by the floating-point library, fplib (see *The software floating-point library, fplib* on page 4-2):

- *Determine the type of a floating-point number (fpclassify)* on page 4-28
- *Determine if a number is finite (isfinite)* on page 4-28
- *Determine if a number is infinite (isinf)* on page 4-29
- *Determine if a number is a NaN (isnan)* on page 4-29
- *Determine if a number is normal (isnormal)* on page 4-29
- *Return the sign bit of a number (signbit)* on page 4-29
- *Copy sign functions (copysign, copysignf)* on page 4-30
- *Comparison macros (isgreater, isgreaterequal, isless, islessequal, islessgreater, isunordered)* on page 4-30
- *Inverse hyperbolic functions (acosh, asinh, atanh)* on page 4-31
- *Cube root (cbrt)* on page 4-31
- *Error functions (erf, erfc)* on page 4-31
- *One less than exp(x) (expm1)* on page 4-32
- *Hypotenuse function (hypot)* on page 4-32
- *The logarithm of the gamma function (lgamma)* on page 4-32
- *Logarithm of one more than x (log1p)* on page 4-33
- *IEEE 754 remainder function (remainder)* on page 4-33
- *IEEE round-to-integer operation (rint)* on page 4-33
- *Signalling NaNs (_WANT_SNAN)* on page 4-33
- *Complex math functions (cacos, casin, catan, ccos, csin, ctan, cacosh, casinh, catanh, ccosh, csinh, ctanh, cexp, clog, cabs, cpow, csqrt, carg, cproj)* on page 4-33.

Mathlib also provides the following nonstandard functions:

- *Gamma function (gamma, gamma_r)* on page 4-34
- *Bessel functions of the first kind (j0, j1, jn)* on page 4-34

- *Bessel functions of the second kind(y0, y1, yn)* on page 4-35.
- *Return the fraction part of a number(significand)* on page 4-35

4.3.1 Range reduction in mathlib

Trigonometric functions in mathlib use range reduction to bring large arguments within the range 0 to 2π. The ARM compiler provides two different range reduction functions. One is accurate to one unit in the last place for *any* input values, but is larger and slower than the other. The other is reliable enough for almost all purposes and is faster and smaller.

The fast and small range reducer is used by default. To select the more accurate one, use either:

- `#pragma import (__use_accurate_range_reduction)` from C
- `IMPORT __use_accurate_range_reduction` from assembly language.

4.3.2 Determine the type of a floating-point number (fpclassify)

This macro classifies its argument value based on type.

```
int fpclassify(real-floating x);
```

where *real-floating* can be type **`float`**, **`double`**,or **`long double`**.

Here, `fpclassify` returns one of the following values:

- `FP_INFINITE` if `x` is an infinity
- `FP_NAN` if `x` is a NaN
- `FP_NORMAL` if `x` is a normal
- `FP_SUBNORMAL` if `x` is a subnormal
- `FP_ZERO` if `x` is zero.

It does not cause any errors or exceptions.

4.3.3 Determine if a number is finite (isfinite)

This macro determines if its argument has a finite value.

```
int isfinite(real-floating x);
```

where *real-floating* can be type **`float`**, **`double`**,or **`long double`**.

Here, `isfinite` returns:

- nonzero if `x` is finite
- `0` if `x` is infinite or a NaN.

It does not cause any errors or exceptions.

4.3.4 Determine if a number is infinite (isinf)

This macro determines if its argument is an infinity.

```
int isinf(real-floating x);
```

where *real-floating* can be type **float**, **double**,or **long double**.

Here, `isinf` returns:

- nonzero if `x` is infinite
- `0` if `x` is finite or a NaN.

It does not cause any errors or exceptions.

4.3.5 Determine if a number is a NaN (isnan)

This macro determines if the value of its argument is a NaN.

```
int isnan(real-floating x);
```

where *real-floating* can be type **float**, **double**,or **long double**.

Here, `isnan` returns:

- nonzero if `x` is a NaN
- `0` otherwise.

It does not cause any errors or exceptions.

4.3.6 Determine if a number is normal (isnormal)

This macro determines if its argument value is normal.

```
int isnormal(real-floating x);
```

where *real-floating* can be type **float**, **double**,or **long double**.

Here, `isnormal` returns:

- nonzero if `x` is a normal
- `0` if `x` is infinite, a NaN, subnormal, or zero.

It does not cause any errors or exceptions.

4.3.7 Return the sign bit of a number (signbit)

This macro determines if the sign of its argument value is negative.

```
int signbit(real-floating x);
```

where *real-floating* can be type **float**, **double**,or **long double**.

Here, `signbit` returns:

- `1` if `x` is negative
- `0` if `x` is positive.

`signbit` returns a sign value for all values of `x` (including zeros and NaNs).

It does not cause any errors or exceptions.

4.3.8 Copy sign functions (copysign, copysignf)

These functions replace the sign bit of `x` with the sign bit of `y`.

```
double copysign(double x, double y);
float copysignf(float x, float y);
```

These functions treat all floating-point numbers as signed (including zeros and NaNs) and do not cause any errors or exceptions.

4.3.9 Comparison macros (isgreater, isgreaterequal, isless, islessequal, islessgreater, isunordered)

These macros compare `x` and `y` as shown in Table 4-13.

```
int isgreater(real-floating x, real-floating y);
int isgreaterequal(real-floating x, real-floating y);
int isless(real-floating x, real-floating y);
int islessequal(real-floating x, real-floating y);
int islessgreater(real-floating x, real-floating y);
int isunordered(real-floating x, real-floating y);
```

where *real-floating* can be type **float**, **double**,or **long double**.

These macros do not cause any errors or exceptions, unlike the equivalent relationship operators that do raise exceptions on all NaNs.

Table 4-13 Comparison macros

Macro	Comparison
`isgreater`	x is greater than y
`isgreaterequal`	x is greater than or equal to y
`isless`	x is less than y

Table 4-13 Comparison macros (continued)

Macro	Comparison
islessequal	x is less than or equal to y
islessgreater	x (is less than or greater than) y
isunordered	x is unordered with respect to y

4.3.10 Inverse hyperbolic functions (acosh, asinh, atanh)

These functions are the inverses of cosh, sinh and tanh.

```
double acosh(double x);
double asinh(double x);
double atanh(double x);
```

Here:

- acosh always has a choice of two return values, one positive and one negative, because cosh is a symmetric function (that is, it returns the same value when applied to *x* or –*x*). It chooses the positive result.
- acosh returns an EDOM error if called with an argument less than 1.0.
- atanh returns an EDOM error if called with an argument whose absolute value exceeds 1.0.

4.3.11 Cube root (cbrt)

This function returns the cube root of its argument.

```
double cbrt(double x);
```

4.3.12 Error functions (erf, erfc)

These functions compute the standard statistical error function, related to the Normal distribution:

```
double erf(double x);
double erfc(double x);
```

Here:

- erf computes the ordinary error function of *x*

- `erfc` computes one minus `erf(x)`. It is better to use `erfc(x)` than `1-erf(x)` when *x* is large, because the answer is more accurate.

4.3.13 One less than exp(*x*) (expm1)

This function computes e^x, minus one. It is better to use `expm1(x)` than `exp(x)-1` if *x* is very near to zero, because `expm1` returns a more accurate value.

```
double expm1(double x);
```

4.3.14 Hypotenuse function (hypot)

This function computes the length of the hypotenuse of a right-angled triangle whose other two sides have length *x* and *y*. Equivalently, it computes the length of the vector (*x,y*) in Cartesian coordinates. Using `hypot(x,y)` is better than `sqrt(x*x+y*y)` because some values of *x* and *y* might cause `x * x + y * y` to overflow even though its square root would not.

```
double hypot(double x, double y);
```

`hypot` returns an `ERANGE` error when the result does not fit in a **`double`**.

4.3.15 The logarithm of the gamma function (lgamma)

These functions compute the logarithm of the absolute value of the gamma function of *x*. The sign of the function is returned separately, so that the two can be used to compute the actual gamma function of *x*.

Both functions return an `ERANGE` error if the answer is too big to fit in a **`double`**.

Both functions return an `EDOM` error if *x* is zero or a negative integer.

lgamma

`double` lgamma(**`double`** x);

`lgamma` returns the sign of the gamma function of *x* in the global variable `_signgam`.

lgamma_r

```
double lgamma_r(double x, int *sign);
```

`lgamma_r` returns it in a user variable, whose address is passed in the `sign` parameter. The value, in either case, is either +1 or –1.

Note

`lgamma_r()` is deprecated.

4.3.16 Logarithm of one more than x (log1p)

This function computes the natural logarithm of $x + 1$. Like `expm1`, it is better to use this function than `log(x+1)` because this function is more accurate when x is near zero.

```
double log1p(double x);
```

4.3.17 IEEE 754 remainder function (remainder)

This function is the IEEE 754 remainder operation. It is a synonym for `_drem` (see *Arithmetic on numbers in a particular format* on page 4-3).

```
double remainder(double x, double y);
```

4.3.18 IEEE round-to-integer operation (rint)

This function is the IEEE 754 round-to-integer operation. It is a synonym for `_drnd` (see *Arithmetic on numbers in a particular format* on page 4-3).

```
double rint(double x);
```

4.3.19 Signalling NaNs (_WANT_SNAN)

If you want to use signalling NaNs, you must indicate this to the compiler by defining the macro `_WANT_SNAN` in your application. This macro must be defined before you include any standard C headers. If your application is comprised of two or more translation units, either all or none of them must define `_WANT_SNAN`. That is, the definition must be consistent for any given application.

You must also use the relevant command-line option when you compile your source code. This is associated with the predefined macro, `__SUPPORT_SNAN__`. See *Predefined macros* on page 4-198 in the *Compiler Reference Guide*.

4.3.20 Complex math functions (cacos, casin, catan, ccos, csin, ctan, cacosh, casinh, catanh, ccosh, csinh, ctanh, cexp, clog, cabs, cpow, csqrt, carg, cproj)

All of the complex math functions defined in section 7.3 *Complex arithmetic <complex.h>* of *ISO/IEC 9899:TC2* are supported.

In C99 mode, both imaginary and complex types are supported. In GNU mode, only complex types are supported.

The macro `I` expands to `_Imaginary_I` in C99 mode, or `_Complex_I` in GNU mode.

Branch cuts are supported. The sign of zero distinguishes one side of a cut from another.

The `CX_LIMITED_RANGE` pragma is supported, but has no effect because arithmetic is performed according to the usual mathematical formulas for complex multiplication, division, and absolute value, regardless of the state that the pragma is in.

All functions treat infinities, NaNs and zeroes in the manner specified in section G.6 of *ISO/IEC 9899:TC2*, except that errors are signalled by setting `errno` instead of raising floating point exceptions, as follows:

- instead of raising `invalid` floating-point exceptions, `errno` is set to `EDOM`
- instead of raising `divide-by-zero` exceptions, `errno` is set to `ERANGE`.

4.3.21 Gamma function (gamma, gamma_r)

These functions both compute the logarithm of the gamma function.They are synonyms for `lgamma` and `lgamma_r` (see *The logarithm of the gamma function (lgamma)* on page 4-32).

```
double gamma(double x);
double gamma_r(double x, int *);
```

Note

Despite their names, these functions compute the logarithm of the gamma function, not the gamma function itself.

Note

These functions are deprecated.

4.3.22 Bessel functions of the first kind (j0, j1, jn)

These functions compute Bessel functions of the first kind. `j0` and `j1` compute the functions of order 0 and 1 respectively. `jn` computes the function of order *n*.

```
double j0(double x);
double j1(double x);
double jn(int n, double x);
```

If the absolute value of *x* exceeds π times 2^{52}, these functions return an `ERANGE` error, denoting total loss of significance in the result.

Note

These functions are deprecated.

4.3.23 Bessel functions of the second kind(y0, y1, yn)

These functions compute Bessel functions of the second kind. `y0` and `y1` compute the functions of order 0 and 1 respectively. `yn` computes the function of order *n*.

```
double y0(double x);
double y1(double x);
double yn(int, double);
```

If *x* is positive and exceeds π times 2^{52}, these functions return an `ERANGE` error, denoting total loss of significance in the result.

Note

These functions are deprecated.

4.3.24 Return the fraction part of a number(significand)

This function returns the fraction part of *x*, as a number between 1.0 and 2.0 (not including 2.0).

```
double significand(double x);
```

Note

This function is deprecated.

4.4 IEEE 754 arithmetic

The ARM floating-point environment is an implementation of the IEEE 754 standard for binary floating-point arithmetic. This section contains a summary of the standard as it is implemented by the ARM compiler.

This section includes:

- *Basic data types*
- *Arithmetic and rounding* on page 4-40
- *Exceptions* on page 4-41.

4.4.1 Basic data types

ARM floating-point values are stored in one of two data types, *single-precision* and *double-precision*. In this document these are called **`float`** and **`double`**. These are the corresponding C types.

Single precision

A **`float`** value is 32 bits wide. The structure is shown in Figure 4-3.

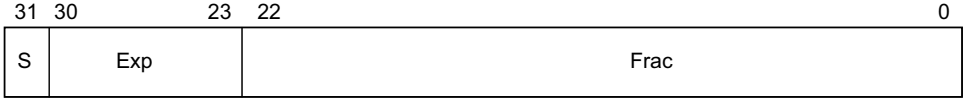

Figure 4-3 IEEE 754 single-precision floating-point format

The `S` field gives the sign of the number. It is 0 for positive, or 1 for negative.

The `Exp` field gives the exponent of the number, as a power of two. It is *biased* by `0x7F` (127), so that very small numbers have exponents near zero and very large numbers have exponents near `0xFF` (255).

So, for example:

- if *Exp* = `0x7D` (125), the number is between 0.25 and 0.5 (not including 0.5)
- if *Exp* = `0x7E` (126), the number is between 0.5 and 1.0 (not including 1.0)
- if *Exp* = `0x7F` (127), the number is between 1.0 and 2.0 (not including 2.0)
- if *Exp* = `0x80` (128), the number is between 2.0 and 4.0 (not including 4.0)
- if *Exp* = `0x81` (129), the number is between 4.0 and 8.0 (not including 8.0).

The `Frac` field gives the fractional part of the number. It usually has an implicit 1 bit on the front that is not stored to save space.

So if *Exp* is `0x7F`, for example:

- if *Frac* = `00000000000000000000000` (binary), the number is 1.0
- if *Frac* = `10000000000000000000000` (binary), the number is 1.5
- if *Frac* = `01000000000000000000000` (binary), the number is 1.25
- if *Frac* = `11000000000000000000000` (binary), the number is 1.75.

So in general, the numeric value of a bit pattern in this format is given by the formula:

$(-1)^{S} * 2^{(Exp-0x7F)} * (1 + Frac * 2^{-23})$

Numbers stored in this form are called *normalized* numbers.

The maximum and minimum exponent values, 0 and 255, are special cases. Exponent 255 is used to represent infinity, and store NaN values. Infinity can occur as a result of dividing by zero, or as a result of computing a value that is too large to store in this format. NaN values are used for special purposes. Infinity is stored by setting Exp to 255 and Frac to all zeros. If Exp is 255 and Frac is nonzero, the bit pattern represents a NaN.

Exponent 0 is used to represent very small numbers in a special way. If *Exp* is zero, then the `Frac` field has no implicit 1 on the front. This means that the format can store 0.0, by setting both `Exp` and `Frac` to all 0 bits. It also means that numbers that are too small to store using `Exp >= 1` are stored with less precision than the ordinary 23 bits. These are called *denormals*.

Double precision

A **`double`** value is 64 bits wide. Figure 4-4 shows its structure.

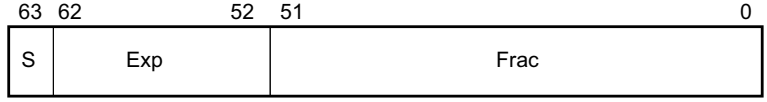

Figure 4-4 IEEE 754 double-precision floating-point format

As before, `S` is the sign, `Exp` the exponent, and `Frac` the fraction. Most of the discussion of **`float`** values remains true, except that:

- The `Exp` field is biased by `0x3FF` (1023) instead of `0x7F`, so numbers between 1.0 and 2.0 have an `Exp` field of `0x3FF`.
- The `Exp` value used to represent infinity and NaNs is `0x7FF` (2047) instead of `0xFF`.

Sample values

This section provides sample floating-point values:

- *Single-precision floating-point values*
- *Double-precision floating-point values* on page 4-39.

Single-precision floating-point values

Some sample `float` bit patterns, together with their mathematical values, are given in Table 4-14.

Table 4-14 Sample single-precision floating-point values

Float value	S	Exp	Frac	Mathematical value	Notes[a]
0x3F800000	0	0x7F	000...000	1.0	-
0xBF800000	1	0x7F	000...000	-1.0	-
0x3F800001	0	0x7F	000...001	1.000 000 119	a
0x3F400000	0	0x7E	100...000	0.75	-
0x00800000	0	0x01	000...000	$1.18 * 10^{-38}$	b
0x00000001	0	0x00	000...001	$1.40 * 10^{-45}$	c
0x7F7FFFFF	0	0xFE	111...111	$3.40 * 10^{38}$	d
0x7F800000	0	0xFF	000...000	Plus infinity	-
0xFF800000	1	0xFF	000...000	Minus infinity	-
0x00000000	0	0x00	000...000	0.0	e
0x7F800001	0	0xFF	000...001	Signaling NaN	f
0x7FC00000	0	0xFF	100...000	Quiet NaN	f

a. See *Notes on sample floating-point values* on page 4-39 for more information.

Double-precision floating-point values

Some sample **double** bit patterns, together with their mathematical values, are given in Table 4-15.

Table 4-15 Sample double-precision floating-point values

Double value	S	Exp	Frac	Mathematical value	Notes[a]
0x3FF00000 00000000	0	0x3FF	000...000	1.0	-
0xBFF00000 00000000	1	0x3FF	000...000	-1.0	-
0x3FF00000 00000001	0	0x3FF	000...001	1.000 000 000 000 000 222	a
0x3FE80000 00000000	0	0x3FE	100...000	0.75	-
0x00100000 00000000	0	0x001	000...000	$2.23 * 10^{-308}$	b
0x00000000 00000001	0	0x000	000...001	$4.94 * 10^{-324}$	c
0x7FEFFFFF FFFFFFFF	0	0x7FE	111...111	$1.80 * 10^{308}$	d
0x7FF00000 00000000	0	0x7FF	000...000	Plus infinity	-
0xFFF00000 00000000	1	0x7FF	000...000	Minus infinity	-
0x00000000 00000000	0	0x000	000...000	0.0	e
0x7FF00000 00000001	0	0x7FF	000...001	Signaling NaN	f
0x7FF80000 00000000	0	0x7FF	100...000	Quiet NaN	f

a. See *Notes on sample floating-point values* for more information.

Notes on sample floating-point values

a The smallest representable number that can be seen to be greater than 1.0. The amount that it differs from 1.0 is known as the *machine epsilon*. This is 0.000 000 119 in **float**, and 0.000 000 000 000 000 222 in **double**. The machine epsilon gives a rough idea of the number of significant figures the format can keep track of. **float** can do six or seven places. **double** can do fifteen or sixteen.

b The smallest value that can be represented as a normalized number in each format. Numbers smaller than this can be stored as denormals, but are not held with as much precision.

c The smallest positive number that can be distinguished from zero. This is the absolute lower limit of the format.

d The largest finite number that can be stored. Attempting to increase this number by addition or multiplication causes overflow and generates infinity (in general).

e Zero. Strictly speaking, they show plus zero. Zero with a sign bit of 1, minus zero, is treated differently by some operations, although the comparison operations (for example == and !=) report that the two types of zero are equal.

f There are two types of NaNs, signaling NaNs and quiet NaNs. Quiet NaNs have a 1 in the first bit of Frac, and signaling NaNs have a zero there. The difference is that signaling NaNs cause an exception (see *Exceptions* on page 4-41) when used, whereas quiet NaNs do not.

4.4.2 Arithmetic and rounding

Arithmetic is generally performed by computing the result of an operation as if it were stored exactly (to infinite precision), and then rounding it to fit in the format. Apart from operations whose result already fits exactly into the format (such as adding 1.0 to 1.0), the correct answer is generally somewhere between two representable numbers in the format. The system then chooses one of these two numbers as the rounded result. It uses one of the following methods:

Round to nearest

The system chooses the nearer of the two possible outputs. If the correct answer is exactly halfway between the two, the system chooses the one where the least significant bit of `Frac` is zero. This behavior (round-to-even) prevents various undesirable effects.

This is the default mode when an application starts up. It is the only mode supported by the ordinary floating-point libraries. Hardware floating-point environments and the enhanced floating-point libraries support all four rounding modes. See *Library naming conventions* on page 2-117.

Round up, or round toward plus infinity

The system chooses the larger of the two possible outputs (that is, the one further from zero if they are positive, and the one closer to zero if they are negative).

Round down, or round toward minus infinity

The system chooses the smaller of the two possible outputs (that is, the one closer to zero if they are positive, and the one further from zero if they are negative).

Round toward zero, or chop, or truncate

The system chooses the output that is closer to zero, in all cases.

4.4.3 Exceptions

Floating-point arithmetic operations can run into various problems. For example, the result computed might be either too big or too small to fit into the format, or there might be no way to calculate the result (as in trying to take the square root of a negative number, or trying to divide zero by zero). These are known as exceptions, because they indicate unusual or exceptional situations.

The ARM floating-point environment can handle exceptions in more than one way.

Ignoring exceptions

The system invents a plausible result for the operation and returns that. For example, the square root of a negative number can produce a NaN, and trying to compute a value too big to fit in the format can produce infinity. If an exception occurs and is ignored, a flag is set in the floating-point status word to tell you that something went wrong at some point in the past.

Trapping exceptions

This means that when an exception occurs, a piece of code called a trap handler is run. The system provides a default trap handler that prints an error message and terminates the application. However, you can supply your own trap handlers to clean up the exceptional condition in whatever way you choose. Trap handlers can even supply a result to be returned from the operation.

For example, if you had an algorithm where it was convenient to assume that 0 divided by 0 was 1, you could supply a custom trap handler for the Invalid Operation exception to identify that particular case and substitute the answer you required.

Types of exception

The ARM floating-point environment recognizes the following types of exception:

- The Invalid Operation exception happens when there is no sensible result for an operation. This can happen for any of the following reasons:
  - performing any operation on a signaling NaN, except the simplest operations (copying and changing the sign)
  - adding plus infinity to minus infinity, or subtracting an infinity from itself
  - multiplying infinity by zero

— dividing 0 by 0, or dividing infinity by infinity

— taking the remainder from dividing anything by 0, or infinity by anything

— taking the square root of a negative number (not including minus zero)

— converting a floating-point number to an integer if the result does not fit

— comparing two numbers if one of them is a NaN.

If the Invalid Operation exception is not trapped, all these operations return a quiet NaN, except for conversion to an integer, that returns zero (as there are no quiet NaNs in integers).

- The Divide by Zero exception happens if you divide a finite nonzero number by zero. (Dividing zero by zero gives an Invalid Operation exception. Dividing infinity by zero is valid and returns infinity.)

  If Divide by Zero is not trapped, the operation returns infinity.

- The Overflow exception happens when the result of an operation is too big to fit into the format. This happens, for example, if you add the largest representable number (marked d in Table 4-14 on page 4-38) to itself.

  If Overflow is not trapped, the operation returns infinity, or the largest finite number, depending on the rounding mode.

- The Underflow exception can happen when the result of an operation is too small to be represented as a normalized number (with `Exp` at least 1).

  The situations that cause Underflow depend on whether it is trapped or not:

  — If Underflow is trapped, it occurs whenever a result is too small to be represented as a normalized number.

  — If Underflow is not trapped, it only occurs if the result requires rounding. So, for example, dividing the **`float`** number `0x00800000` by 2 does not signal Underflow, because the result (`0x00400000`) is exact. However, trying to multiply the float number `0x00000001` by 1.5 does signal Underflow

  (For readers familiar with the IEEE 754 specification, the chosen implementation options in the ARM compiler are to detect tininess before rounding, and to detect loss of accuracy as an inexact result.)

  If Underflow is not trapped, the result is rounded to one of the two nearest representable denormal numbers, according to the current rounding mode. The loss of precision is ignored and the system returns the best result it can.

  — The Inexact Result exception happens whenever the result of an operation requires rounding. This would cause significant loss of speed if it had to be detected on every operation in software, so the ordinary floating-point

libraries do not support the Inexact Result exception. The enhanced floating-point libraries, and hardware floating-point systems, all support Inexact Result.

If Inexact Result is not trapped, the system rounds the result in the usual way.

The flag for Inexact Result is also set by Overflow and Underflow if either one of those is not trapped.

All exceptions are untrapped by default.