ARM® Compiler toolchain

Version 5.02

Using ARM C and C++ Libraries and Floating-Point Support

ARM®

ARM DUI 0475H (ID070912)

ARM Compiler toolchain
Using ARM C and C++ Libraries and Floating-Point Support

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Confidentiality	Change
28 May 2010	A	Non-Confidential	ARM Compiler toolchain v4.1 Release
30 September 2010	B	Non-Confidential	Update 1 for ARM Compiler toolchain v4.1
28 January 2011	C	Non-Confidential	Update 2 for ARM Compiler toolchain v4.1 Patch 3
30 April 2011	D	Non-Confidential	ARM Compiler toolchain v5.0 Release
29 July 2011	E	Non-Confidential	Update 1 for ARM Compiler toolchain v5.0
30 August 2011	F	Non-Confidential	ARM Compiler toolchain v5.01 Release
29 February 2012	G	Non-Confidential	Document update 1 for ARM Compiler toolchain v5.01 Release
27 July 2012	H	Non-Confidential	ARM Compiler toolchain v5.02 Release

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.

Product Status

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

Web Address

`http://www.arm.com`

Contents
ARM Compiler toolchain Using ARM C and C++ Libraries and Floating-Point Support

Chapter 1
Conventions and feedback

The following describes the typographical conventions and how to give feedback:

Typographical conventions

The following typographical conventions are used:

`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.

Feedback on this product

If you have any comments and suggestions about this product, contact your supplier and give:

- your name and company

- the serial number of the product
- details of the release you are using
- details of the platform you are using, 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 documentation

If you have comments on the documentation, e-mail `errata@arm.com`. Give:

- the title
- the number, ARM DUI 0475H
- if viewing online, the topic names to which your comments apply
- if viewing a PDF version of a document, the page numbers to which your comments apply
- a concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

ARM periodically provides updates and corrections to its documentation on the ARM Information Center, together with knowledge articles and *Frequently Asked Questions* (FAQs).

Other information

- ARM Information Center, `http://infocenter.arm.com/help/index.jsp`
- ARM Technical Support Knowledge Articles, `http://infocenter.arm.com/help/topic/com.arm.doc.faqs/index.html`
- ARM Support and Maintenance, `http://www.arm.com/support/services/support-maintenance.php`
- ARM Glossary, `http://infocenter.arm.com/help/topic/com.arm.doc.aeg0014-/index.html`.

Chapter 2
The ARM C and C++ libraries

The following topics describe the ARM C and C++ libraries:

2.1 Mandatory linkage with the C library

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 generate calls to C library functions to improve your application, even though calls to such functions might not exist in your source code.

Even if your application does not have a `main()` function, meaning that the C library is not initialized, some C library functions are still legitimately available and the compiler might implicitly generate calls to these functions.

2.1.1 See also

Concepts

- *Building an application without the C library* on page 2-41.

2.2 C and C++ runtime libraries

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

C standardlib

This is a C library consisting of:

- All functions defined by the ISO C99 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

This is a C library that can be used as an alternative to C standardlib. It is a micro-library that is ideally suited for deeply embedded applications that have to fit within small-sized memory. The C micro-library, `microlib`, consists of:

- Functions that are highly optimized to achieve the minimum code size.
- Functions that are not compliant with the ISO C library standard.
- Functions that are not compliant with the 1985 IEEE 754 standard for binary floating-point arithmetic.

C++

This is a C++ library that can be used with C standardlib. It consists of:

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

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

2.2.1 See also

Concepts

- *Mandatory linkage with the C library* on page 2-5
- Chapter 2 *The ARM C and C++ libraries*
- Chapter 3 *The ARM C micro-library*.

Developing Software for ARM Processors:

- Chapter 8 *Semihosting*.

Introducing the ARM Compiler toolchain:

- *Rogue Wave documentation* on page 2-30.

Other information

- ISO C library standard, `http://www.iso.org`
- *IEEE Standard for Floating-Point Arithmetic* (IEEE 754), 1985 version, `http://ieeexplore.ieee.org`

2.3 C and C++ library features

The C library uses the standard ARM semihosted environment to provide facilities such as file input/output. This environment is supported by the ARM RVI™ debug unit and the *Real-Time Simulator Model* (RTSM).

You can re-implement 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.

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 See also

Concepts

Developing Software for ARM Processors:

- Chapter 8 *Semihosting*.

2.4 Library heap usage requirements of the ARM C and C++ libraries

Functions such as `malloc()` and other dynamic memory allocation functions explicitly allocate memory when used. However, some library functions and mechanisms *implicitly* allocate memory from the heap. If heap usage requirements are significant to your code development (for example, you might be developing code for an embedded system with a tiny memory footprint), you must be aware of both implicit and explicit heap requirements.

In C standardlib, implicit heap usage occurs as a result of:

- calling the library function `fopen()` and the first time that an I/O operation is applied to the resulting stream
- passing command-line arguments into the `main()` function.

The size of heap memory allocated for `fopen()` is 80 bytes for the `FILE` structure. When the first I/O operation occurs, and not until the operation occurs, an additional default of 512 bytes of heap memory is allocated for a buffer associated with the operation. You can reconfigure the size of this buffer using `setvbuf()`.

When `fclose()` is called, the default 80 bytes of memory is kept on a freelist for possible re-use. The 512-byte buffer is freed on `fclose()`.

Declaring `main()` to take arguments requires 256 bytes of implicitly allocated memory from the heap. This memory is never freed because it is required for the duration of `main()`. In microlib, `main()` must not be declared to take arguments, so this heap usage requirement only applies to standardlib. In the standardlib context, it only applies if you have a heap.

Note

The memory sizes quoted might change in future releases.

2.4.1 See also

- *Library heap usage requirements of the ARM C micro-library* on page 3-4
- *Modification of memory management functions in the C library* on page 2-81.

2.5 Compliance with the *Application Binary Interface* (ABI) for the ARM architecture

The ABI for the ARM Architecture is a family of specifications that describes 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, 16-bit Thumb®, and 32-bit Thumb objects and libraries from different producers that support the ABI for the ARM Architecture. The ARM compilation tools fully support the BSABI, including support for *Debug With Arbitrary Record Format* (DWARF) 3 debug tables (DWARF Debugging Standard Version 3).

The ARM C and C++ libraries conform to the standards described in the BSABI, the CLIBABI, and the *C++ ABI* (CPPABI) for the ARM Architecture.

2.5.1 See also

Concepts

- *Increasing portability of object files to other CLIBABI implementations* on page 2-10.

Other information

- ABI for the ARM Architecture suite of specifications, `http://infocenter.arm.com/help/topic/com.arm.doc.subset.swdev.abi`
- DWARF Debugging Standard, `http://dwarfstd.org/`

2.6 Increasing portability of object files to other CLIBABI implementations

You can request full CLIBABI portability to increase the portability of your object files to other implementations of the CLIBABI. To do this, either:

- specify `#define _AEABI_PORTABILITY_LEVEL 1` before you `#include` any library headers, such as `<stdlib.h>`
- specify `-D_AEABI_PORTABILITY_LEVEL=1` on the compiler command line.

Note

This reduces the performance of some library operations.

2.6.1 See also

Concepts

- *Compliance with the Application Binary Interface (ABI) for the ARM architecture* on page 2-9.

Reference

Compiler Reference:

- *-Dname[(parm-list)][=def]* on page 3-54.

Other information

- Application Binary Interface (ABI) for the ARM Architecture, `http://infocenter.arm.com/help/topic/com.arm.doc.subset.swdev.abi`

2.7 ARM C and C++ library directory structure

The libraries are installed in two subdirectories within *install_directory*\lib:

armlib — Contains the variants of the ARM C library, the floating-point arithmetic library (fplib), and the math library (mathlib).

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 for these libraries are installed in:

install_directory\include.

The environment variable ARMCC*n*LIB 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 re-implementation 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.

2.7.1 See also

Concepts

Introducing the ARM Compiler toolchain:

- *Rogue Wave documentation* on page 2-30.

2.8 Selection of ARM C and C++ library variants based on build options

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

Target Architecture and instruction set

ARM, 16-bit Thumb, or 32-bit Thumb.

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

Different ways to access your data are as follows:

- by absolute address
- relative to `sb` (read/write position-independent)
- relative to `pc` (fpic).

Different ways to access your code are as follows:

- by absolute address when appropriate
- relative to `pc` (read-only position independent).

The standard C libraries provide variants to support all of these options.

Position-independent C++ code can only be achieved with `--apcs=/fpic`.

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.

2.8.1 See also

Tasks

Creating Static Software Libraries with armar:

- *Creating a new object library* on page 3-2.

Concepts

Using the Compiler:

- *Code compatibility between separately compiled and assembled modules* on page 3-22.

Reference

Compiler Reference:

- *--apcs=qualifer...qualifier* on page 3-11
- *--arm* on page 3-15
- *--bigend* on page 3-27
- *--fpu=name* on page 3-100
- *--littleend* on page 3-137
- *--thumb* on page 3-197.

Linker Reference:

- *--fpu=name* on page 2-76

- *--ropi* on page 2-136
- *--rwpi* on page 2-140.

Assembler Reference:

- *--arm* on page 2-9
- *--bigend* on page 2-12
- *--fpu=name* on page 2-39
- *--littleend* on page 2-51
- *--thumb* on page 2-79.

2.9 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:

- 16-bit Thumb or 32-bit Thumb, 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 instruction-only build, compile with the `--arm_only` option.

Note

- The Thumb C library used for ARMv6-M targets contains only 16-bit Thumb code.
- The Thumb C library used for ARMv7-M targets contains only 16-bit and 32-bit Thumb code.

2.9.1 See also

Concepts

- Chapter 2 *The ARM C and C++ libraries*.

Reference

Compiler Reference:

- *--arm_only* on page 3-23
- *--thumb* on page 3-197
- *#pragma thumb* on page 5-112.

Other information

- Cortex processors, `http://infocenter.arm.com/help/topic/com.arm.doc.set.cortex/index.html`

2.10 C++ and C libraries and the `std` namespace

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.

2.10.1 See also

Reference

Compiler Reference:

- *--using_std, --no_using_std* on page 3-212.

2.11 ARM C libraries and multithreading

The ARM C libraries support multithreading, for example, where you are using a *Real-Time Operating System* (RTOS). In this context, 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). 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*.

Reentrancy and thread safety are both related to the way functions in a multithreaded application handle resources.

2.11.1 See also

Concepts

- *ARM C libraries and reentrant functions* on page 2-17
- *ARM C libraries and thread-safe functions* on page 2-18
- *Using the ARM C library in a multithreaded environment* on page 2-27.

2.12 ARM C libraries and reentrant functions

A reentrant function does not hold static data over successive calls, and does not 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.

2.12.1 See also

Concepts

- *ARM C libraries and multithreading* on page 2-16
- *ARM C libraries and thread-safe functions* on page 2-18.

2.13 ARM C libraries and thread-safe 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 held in processor registers, or if the compiler runs out of registers, are dynamically allocated on the stack. Therefore, any function that does not use static data, or other shared resources, is thread-safe.

2.13.1 See also

Tasks

- *Management of locks in multithreaded applications* on page 2-24
- *Using the ARM C library in a multithreaded environment* on page 2-27.

Concepts

- *ARM C libraries and multithreading* on page 2-16
- *ARM C libraries and reentrant functions* on page 2-17
- *Management of locks in multithreaded applications* on page 2-24
- *Thread safety in the ARM C library* on page 2-29
- *Thread safety in the ARM C++ library* on page 2-30.

2.14 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 multithreaded 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 callers use different values in `sb`.

Note

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

Callouts from the C library enable access to static data. C library functions that use static data can be categorized as:

- 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 2-1. They do not use semihosting.

Table 2-1 C library callouts

Function	Description
`__rt_errno_addr()`	Called to get the address of the variable `errno`
`__rt_fp_status_addr()`	Called by the floating-point support code to get the address of the floating-point status word
`locale` functions	The function `__user_libspace()` creates a block of private static data for the library

The default implementation of `__user_libspace` creates a 96-byte block in the ZI region. Even if your application does not have a `main()` function, the `__user_libspace()` function does not normally have to be redefined.

Note

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

2.14.1 See also

Concepts

- *Re-implementation of legacy function __user_libspace() in the C library* on page 2-23.

Using the Compiler:

- *Code compatibility between separately compiled and assembled modules* on page 3-22.

Reference

- *Assembler macros that tailor locale functions in the C library* on page 2-63
- *ARM C libraries and multithreading* on page 2-16.

Compiler Reference:

- *--apcs=qualifer...qualifier* on page 3-11.

ARM C and C++ Libraries and Floating-Point Support Reference:

- *__rt_errno_addr()* on page 2-29
- *__rt_fp_status_addr()* on page 2-31.

2.15 Use of the __user_libspace static data area by the C libraries

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 *Floating-Point* (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 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, that points to a list of destructions to be performed on program exit. For example, objects constructed outside function scope exist for the duration of the program, but require destruction on program exit. 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().

Note

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

2.15.1 See also

Other information

- __aeabi_atexit() in *C++ ABI for the ARM Architecture*, http://infocenter.arm.com/help/topic/com.arm.doc.ihi0041-/index.html.
- std::set_terminate(), std::set_unexpected(), in *Exception Handling ABI for the ARM Architecture*, http://infocenter.arm.com/help/topic/com.arm.doc.ihi0038-/index.html

2.16 C library functions to access subsections of the `__user_libspace` static data area

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. This data is 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.

2.16.1 See also

Concepts

- *Use of the __user_libspace static data area by the C libraries* on page 2-21
- *Re-implementation of legacy function __user_libspace() in the C library* on page 2-23.

2.17 Re-implementation of legacy function `__user_libspace()` in the C library

The `__user_libspace()` function returns a pointer to a block of private static data for the C library. This function does not normally have to be redefined.

If you are writing an operating system or a process switcher, then typically you use the `__user_perproc_libspace()` and `__user_perthread_libspace()` functions (which are always available) rather than re-implement `__user_libspace()`.

If you have legacy source code that re-implements `__user_libspace()`, you do not have to change the re-implementation for single-threaded processes. However, you are likely to be required to do so for multi-threaded applications. For multi-threaded applications, use either or both of `__user_perproc_libspace()` and `__user_perthread_libspace()`, instead of `__user_libspace()`.

2.17.1 See also

Concepts

- *C library functions to access subsections of the __user_libspace static data area* on page 2-22
- *Using the ARM C library in a multithreaded environment* on page 2-27
- *Use of the __user_libspace static data area by the C libraries* on page 2-21
- *Use of static data in the C libraries* on page 2-19.

2.18 Management of locks in multithreaded applications

A thread-safe function protects shared resources from concurrent access using locks. There are functions in the C library that you can re-implement, that enable you to manage the locking mechanisms and so prevent the corruption of shared data such as the heap. These functions are mutex functions, where the lifecycle of a mutex is one of initialization, iterative acquisition and releasing of the mutex as required, and then optionally freeing the mutex when it is never going to be required again. The mutex functions wrap onto your own *Real-Time Operating System* (RTOS) calls, and their function prototypes 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.

Ensure that `_mutex_initialize()` initializes the mutex to an unlocked state.

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.

`_mutex_acquire()` is not called by the thread that already owns the mutex.

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 re-locked 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.

The `mutex` data structure type that is used as the parameter to the `_mutex_*()` functions is not defined in any of the ARM compiler toolchain header files, but must be defined elsewhere. Typically, it is defined as part of RTOS code.

Functions that call `_mutex_*()` functions create 4 bytes of memory for holding the mutex data structure. `__Heap_Initialize()` is one such function.

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 a special memory area, then you must treat the default mutex as a pointer to a real mutex.

A typical example of a re-implementation framework for `_mutex_initialize()`, `_mutex_acquire()`, and `_mutex_release()` is as follows, where `SEMAPHORE_ID`, `CreateLock()`, `AcquireLock()`, and `ReleaseLock()` are defined in the RTOS, and `(...)` implies additional parameters:

```
int _mutex_initialize(SEMAPHORE_ID sid)
{
   /* Create a mutex semaphore */
    sid = CreateLock(...);
    return 1;
}

void _mutex_acquire(SEMAPHORE_ID sid)
{
    /* Task sleep until get semaphore */
    AcquireLock(sid, ...);
}

void _mutex_release(SEMAPHORE_ID sid)
{
    /* Release the semaphore. */
    ReleaseLock(sid);
}

void _mutex_free(SEMAPHORE_ID sid)
{
    /* Free the semaphore. */
    FreeLock(sid, ...);
}
```

Note

- `_mutex_release()` releases the lock on the mutex that was acquired by `_mutex_acquire()`, but the mutex still exists, and can be re-locked by a subsequent call to `_mutex_acquire()`.
- It is only when the optional wrapper function `_mutex_free()` is called that the mutex is destroyed. After the mutex is destroyed, it cannot be used without first calling `_mutex_initialize()` to set it up again.

2.18.1 See also

Tasks

- *How to ensure re-implemented mutex functions are called* on page 2-26
- *Using the ARM C library in a multithreaded environment* on page 2-27.

Concepts

- *Thread safety in the ARM C library* on page 2-29
- *Thread safety in the ARM C++ library* on page 2-30.

2.19 How to ensure re-implemented mutex functions are called

If your re-implemented_mutex_*() functions are within an object that is contained within a library file, the linker does not automatically include the object. This can result in the _mutex_*() functions being excluded from the image you have built. To avoid this problem, that is, to ensure that your _mutex_*() functions are called, you can either:

- Place your mutex functions in a non-library 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 non-weak 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.

2.19.1 See also

Tasks

- *Using the ARM C library in a multithreaded environment* on page 2-27.

Concepts

- *Management of locks in multithreaded applications* on page 2-24
- *Thread safety in the ARM C library* on page 2-29
- *Thread safety in the ARM C++ library* on page 2-30.

2.20 Using the ARM C library in a multithreaded environment

To use the ARM C 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 re-implement `__user_perproc_libspace()` unless there is a specific reason to do so. In the majority of cases, there is no requirement to re-implement this function.
- A way to manage multiple stacks.

  A simple way to do this is to use the ARM two-region memory model. 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.

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 preemptive scheduling, or in a *Symmetric Multi-Processor* (SMP) model, these functions play an important part in handling locks.

If all of these requirements are met, you can use the ARM C library in your multithreaded environment. The following behavior applies:

- 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 nonreentrant so a reentrant equivalent is supplied
- a few functions remain nonreentrant and no alternative is available.

2.20.1 See also

Concepts

- *ARM C libraries and multithreading* on page 2-16
- *Management of locks in multithreaded applications* on page 2-24
- *Thread safety in the ARM C library* on page 2-29
- *Thread safety in the ARM C++ library* on page 2-30.

Reference

ARM C and C++ Libraries and Floating-Point Support Reference:

- *Thread-safe C library functions* on page 2-65
- *C library functions that are not thread-safe* on page 2-68.

2.21 Thread safety in the ARM C library

In the ARM C library:

- 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 C library in a way that is hidden, for example, if the compiler implicitly calls functions that you have not explicitly called in your source code. Familiarity with the thread-safe C library functions and C library functions that are not thread-safe can help you to avoid this type of threading problem, although in general, it is unlikely to arise.

2.21.1 See also

Tasks

- *Using the ARM C library in a multithreaded environment* on page 2-27.

Concepts

- *Management of locks in multithreaded applications* on page 2-24
- *Thread safety in the ARM C++ library* on page 2-30.

Reference

ARM C and C++ Libraries and Floating-Point Support Reference:

- *Thread-safe C library functions* on page 2-65
- *C library functions that are not thread-safe* on page 2-68.

2.22 Thread safety in the ARM C++ library

The following points summarize thread safety in the C++ library:

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

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

- Throwing an exception is thread-safe if any user constructors and destructors that get called are also thread-safe.
- The ARM C++ library uses the ARM C library. To use the ARM C++ library in a multithreaded environment, you must provide the same functions that you would be required to provide when using the ARM C library in a multithreaded environment.

2.22.1 Rogue Wave Standard C++ library

The Rogue Wave Standard C++ library is a part of the ARM C++ library. What applies to the ARM C++ library applies to the Rogue Wave Standard C++ library too. In the Rogue Wave Standard C++ library, specifically:

- all containers and all functions are reentrant, making no use of internal, modifiable static data
  - except for the `std::random_shuffle` function, which uses static data to record the random number
- The `iostream` and `locale` classes are not thread safe.

You must protect shared objects while using the `iostream` and `locale` classes, and the `std::random_shuffle` function. To do this, you might use mutex functions, or co-operative threading. As an example, in a typical case of a pre-emptive multitasking environment, one that uses mutex functions with containers, this means that:

- reader threads can safely share a container if no thread writes to it during the reads
- while a thread writes to a shared container, you must apply locking around the use of the container
- writer threads can write to different containers safely
- you must apply locking around the use of `random_shuffle`
- multiple threads cannot use `iostream` and `locale` classes safely unless you apply locking around the use of their objects.

2.22.2 See also

Tasks

- *Using the ARM C library in a multithreaded environment* on page 2-27.

Concepts

- *C and C++ runtime libraries* on page 2-6
- *Management of locks in multithreaded applications* on page 2-24
- *Thread safety in the ARM C library* on page 2-29.

Introducing the ARM Compiler toolchain:

- *Rogue Wave documentation* on page 2-30.

Other information

- `__cxa_*`, `__aeabi_*` functions, *C++ ABI for the ARM Architecture*, `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0041-/index.html`
- *Exception Handling ABI for the ARM Architecture*, `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0038-/index.html`.

2.23 The floating-point status word in a multithreaded environment

Applicable to variants of the software floating-point libraries that require a status word (`--fpmode=ieee_fixed` or `--fpmode=ieee_full`), the floating-point status word is safe to use in a multithreaded environment, even with software floating-point. A status word for each thread is stored in its own `__user_perthread_libspace` block.

Note

In a hardware floating-point environment, the floating-point status word is stored in a *Vector Floating-Point* (VFP) register. In this case, your thread-switching mechanism must keep a separate copy of this register for each thread.

2.23.1 See also

Concepts

- *Thread safety in the ARM C library* on page 2-29.

Reference

- *--fpmode=model* on page 3-97.

2.24 Using the C library with an application

You can use the C and C ++ libraries with an application in the following ways:

- Build a semihosting application that can be debugged in a semihosted environment such as with RVI.
- Build a non-hosted application that, for example, can be embedded into ROM.
- Build an application that does not use `main()` and does not initialize the library. This application has restricted library functionality, unless you re-implement some functions.

2.24.1 See also

Tasks

- *Using the C and C++ libraries with an application in a semihosting environment* on page 2-34
- *Using the libraries in a nonsemihosting environment* on page 2-36
- *Building an application without the C library* on page 2-41.

2.25 Using the C and C++ libraries with an application in a semihosting 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, RVI
- implementing your own handler for the semihosting calls.

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

The ARM debug agents support semihosting, but the memory map assumed by the C library might require tailoring to match the hardware being debugged.

2.25.1 See also

Tasks

Developing Software for ARM Processors:

- *Tailoring the C library to your target hardware* on page 3-8
- *Tailoring the image memory map to your target hardware* on page 3-10.

Concepts

- *Using $Sub$$ to mix semihosted and nonsemihosted I/O functionality* on page 2-35.

Developing Software for ARM Processors:

- Chapter 8 *Semihosting*.

Reference

- *Direct semihosting C library function dependencies* on page 2-38.

2.26 Using $Sub$$ to mix semihosted and nonsemihosted I/O functionality

You can use $Sub$$ to provide a mixture of semihosted and nonsemihosted functionality. For example, given an implementation of fputc() that writes directly to a UART, and a semihosted implementation of fputc(), you can provide both of these depending on the nature of the FILE * pointer passed into the function.

Example 2-1 Using $Sub$$ to mix semihosting and nonsemihosting I/O functionality

```
int $Super$$fputc(int c, FILE *fp);
int $Sub$$fputc(int c, FILE *fp)
{
    if (fp == (FILE *)MAGIC_NUM) // where MAGIC_NUM is a special value that
    {                            // is different to all normal FILE * pointer
                                 // values.
        write_to_UART(c);
        return c;
    }
    else
    {
        return $Super$$fputc(c, fp);
    }
}
```

2.26.1 See also

Tasks

- *Using the libraries in a nonsemihosting environment* on page 2-36.

Concepts

- *Using the C and C++ libraries with an application in a semihosting environment* on page 2-34
- *Using the libraries in a nonsemihosting environment* on page 2-36.

Using the Linker:

- *Using $Super$$ and $Sub$$ to patch symbol definitions* on page 7-29.

Other information

- *ELF for the ARM Architecture*, http://infocenter.arm.com/help/topic/com.arm.doc.ihi0044c/IHI0044C_aaelf.pdf.

2.27 Using the libraries in a nonsemihosting environment

Some C library functions use semihosting. If you do not want to use semihosting, either:

- Remove all calls to semihosting functions.
- Re-implement the lower-level functions, for example, `fputc()`. You are not required to re-implement all semihosting functions. You must, however, re-implement the functions you are using in your application.

  (You must re-implement functions that the C library uses to isolate itself from target dependencies. For example, if you use `printf()` you must re-implement `fputc()`. If you do not use the higher-level input/output functions like `printf()`, you do not have to re-implement the lower-level functions like `fputc()`.)
- Implement a handler for all of the semihosting calls to be handled in your own specific way. One such example is for the handler to intercept the calls, redirecting them to your own nonsemihosted, that is, target-specific, functions.

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 unnecessary 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 determine which objects are using semihosting:

1. link with `--verbose --list=out.txt`
2. search the output for the symbol
3. determine what object referenced it.

There are no target-dependent functions in the C++ library, although some C++ functions use underlying C library functions that are target-dependent.

2.27.1 See also

Concepts

- *Mandatory linkage with the C library* on page 2-5.

Developing Software for ARM Processors:

- Chapter 8 *Semihosting*.

Reference

- *Direct semihosting C library function dependencies* on page 2-38.

Linker Reference:

- *--list=filename* on page 2-102
- *--verbose* on page 2-184.

2.28 C++ exceptions in a non-semihosting 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-semihosting environment, you must provide an alternative implementation of `abort()`.

2.28.1 See also

Tasks

- *Using the libraries in a nonsemihosting environment* on page 2-36.

2.29 Direct semihosting C library function dependencies

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

Table 2-2 Direct semihosting dependencies

Function	Description
`__user_setup_stackheap()`	Sets up and returns the locations of the stack and the heap. You might have to re-implement this function if you are using a scatter file at the link stage.
`_sys_exit()` `_ttywrch()`	Error signaling, error handling, and program exit.

`_sys_command_string()`
`_sys_close()`
`_sys_iserror()`
`_sys_istty()`
`_sys_flen()`
`_sys_open()`
`_sys_read()`
`_sys_seek()`
`_sys_write()`
`_sys_tmpnam()`
	Tailoring input/output functions in the C and C++ libraries.

`clock()`
`_clock_init()`
`remove()`
`rename()`
`system()`
`time()`
	Tailoring other C library functions.

2.29.1 See also

Tasks

- *Using the libraries in a nonsemihosting environment* on page 2-36.

Concepts

- *Modification of C library functions for error signaling, error handling, and program exit* on page 2-80
- *Tailoring input/output functions in the C and C++ libraries* on page 2-92
- *Initialization of the execution environment and execution of the application* on page 2-55.

Reference

- *Indirect semihosting C library function dependencies* on page 2-39
- *Tailoring non-input/output C library functions* on page 2-104.

ARM C and C++ Libraries and Floating-Point Support Reference:

- *__user_setup_stackheap()* on page 2-60.

2.30 Indirect semihosting C library function dependencies

Table 2-3 shows functions that depend indirectly on one or more of the functions listed in Table 2-2 on page 2-38. You can use Table 2-3 as an initial guide, but it is recommended that you use either of the following to identify any other functions with indirect or direct dependencies on semihosting at link time:

- `#pragma import(__use_no_semihosting)` in C source code
- `IMPORT __use_no_semihosting` in assembly language source code.

Table 2-3 Indirect semihosting dependencies

Function	Usage
`__raise()`	Catching, handling, or diagnosing C library exceptions, without C signal support. (Tailoring error signaling, error handling, and program exit.)
`__default_signal_handler()`	Catching, handling, or diagnosing C library exceptions, with C signal support. (Tailoring error signaling, error handling, and program exit.)
`__Heap_Initialize()`	Choosing or redefining memory allocation. Avoiding the heap and heap-using C library functions supplied by ARM.
`ferror()`, `fputc()`, `__stdout`	Re-implementing the printf family. (Tailoring input/output functions in the C and C++ libraries.).
`__backspace()`, `fgetc()`, `__stdin`	Re-implementing the scanf family. (Tailoring input/output functions in the C and C++ libraries.).
`fwrite()`, `fputs()`, `puts()`,`fread()`, `fgets()`, `gets()`, `ferror()`	Re-implementing the stream output family. (Tailoring input/output functions in the C and C++ libraries.).

2.30.1 See also

Tasks

- *Using the libraries in a nonsemihosting environment* on page 2-36
- *Avoiding the heap and heap-using library functions supplied by ARM* on page 2-82.

Concepts

- *Modification of C library functions for error signaling, error handling, and program exit* on page 2-80
- *Tailoring input/output functions in the C and C++ libraries* on page 2-92.

Reference

- *Tailoring non-input/output C library functions* on page 2-104.

2.31 C library API definitions for targeting a different environment

In addition to the semihosting functions listed in Table 2-2 on page 2-38 and Table 2-3 on page 2-39, Table 2-4 shows functions and files that might be useful when building for a different environment.

Table 2-4 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
`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

If you are re-implementing 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.

Caution

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

2.31.1 See also

Tasks

- *Using the libraries in a nonsemihosting environment* on page 2-36.

Concepts

- *Assembler macros that tailor locale functions in the C library* on page 2-63.

Reference

Linker Reference:

- *--list=filename* on page 2-102
- *--verbose* on page 2-184.

2.32 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 functions 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()`.

Table 2-5 shows header 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 re-implemented.

(ARM Linux header files are not included in this table because their use serves no purpose with an uninitialized library.)

Table 2-5 Standalone C library functions

Function	Description
`alloca.h`	Functions in this file work without any library initialization or function re-implementation. You must know how to build an application with the C library to use this header file.
`assert.h`	Functions listed in this file require high-level stdio, `__rt_raise()`, and `_sys_exit()`. You must be familiar with tailoring error signaling, error handling, and program exit to use this header file.
`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 re-implementation.
`fenv.h`	Functions in this file work without the requirement for any library initialization and only require the re-implementation of `__rt_raise()`.
`float.h`	This file does not contain any code. The definitions in the file do not require library initialization or function re-implementation.
`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 re-implementation.
`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.

Table 2-5 Standalone C library functions (continued)

Function	Description
`math.h`	For functions in this file to work, you must first call `_fp_init()` and re-implement `__rt_raise()`.
`setjmp.h`	Functions in this file work without any library initialization or function re-implementation.
`signal.h`	Functions listed in this file are not available without library initialization. You must know how to build an application with the C library to use this header file. `__rt_raise()` can be re-implemented for error and exit handling. You must be familiar with tailoring error signaling, error handling, and program exit.
`stdarg.h`	Functions listed in this file work without any library initialization or function re-implementation.
`stddef.h`	This file does not contain any code. The definitions in the file do not require library initialization or function re-implementation.
`stdint.h`	This file does not contain any code. The definitions in the file do not require library initialization or function re-implementation.
`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 re-implement these lower-level functions when using the standalone C library. However, you cannot re-implement the `_sys_` prefixed functions (for example, `_sys_read()`) when using the standalone C library because they require library initialization. You must be familiar with tailoring the input/output functions in the C and C++ libraries. • 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 re-implementation. 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.

Table 2-5 Standalone C library functions (continued)

Function	Description
`time.h`	`mktime()` and `localtime()` can be used immediately `time()` and `clock()` are system-specific and are probably not usable unless re-implemented `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.

2.32.1 See also

Tasks

- *Creating an application as bare machine C without the C library* on page 2-44
- *Assembler macros that tailor locale functions in the C library* on page 2-63
- *Tailoring input/output functions in the C and C++ libraries* on page 2-92.

Concepts

- *Modification of C library functions for error signaling, error handling, and program exit* on page 2-80.

2.33 Creating an application as bare machine C without the C library

The following topics refer to creating applications as *bare machine C* without the library. 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 re-implementations.

2.34 Integer and floating-point compiler functions and building an application without the C library

There are several compiler helper functions that the compiler uses 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 if you use a floating-point mode that involves throwing exceptions, require `__rt_raise()` to handle math errors. Re-implementing `__rt_raise()` enables all the math functions, and it avoids having to link in all the signal-handling library code.

2.34.1 See also

Tasks

- *Building an application without the C library* on page 2-41.

Concepts

Compiler generated and library-resident helper functions on page 2-119.

Other information

- *Application Binary Interface (ABI) for the ARM Architecture*, `http://infocenter.arm.com/help/topic/com.arm.doc.subset.swdev.abi`

2.35 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:

- Re-implement `__rt_raise()` if you are using the heap.
- 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`.

When you have met these requirements, link your application 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.

2.35.1 See also

Tasks

- *Building an application without the C library* on page 2-41.

Concepts

- *Bare machine C with floating-point processing* on page 2-47
- *Use of the __user_libspace static data area by the C libraries* on page 2-21.

Reference

- *--fpu=name* on page 3-100.

2.36 Bare machine C with floating-point processing

If you want to use floating-point processing in an application without the C library you must:

- Re-implement `__rt_raise()` if you are using the heap.
- 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. The register state required to run C primarily comprises the stack pointer. It also consists of `sb`, the stack base, if *Read/Write Position-Independent* (RWPI) code applies.
- Provide your own RW/ZI initialization code.
- Ensure that your initialization veneer is executed by, for example, placing it in your reset handler.
- 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.

Do not build your application with the `--fpu=none` option.

If you are using software floating-point support and a floating-point mode that requires a floating-point status word (`--fpmode=ieee_fixed` or `--fpmode=ieee_full`), 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 rely on the default library definition of `__rt_fp_status_addr()`, this word resides in the program data section, unless you define `__user_perthread_libspace()` (or in the case of legacy code that does not yet use `__user_perthread_libspace()`, `__user_libspace()`).

2.36.1 See also

Tasks

- *Building an application without the C library* on page 2-41.

Concepts

- *Bare machine integer C* on page 2-46
- *Use of static data in the C libraries* on page 2-19
- *Use of the __user_libspace static data area by the C libraries* on page 2-21.

2.37 Customized C library startup code and access to C library functions

When building an application without 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. There are situations where this is not desirable or possible. For example, a system running a *Real-Time Operating System* (RTOS) might have its execution environment configured by the RTOS startup code.

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.

2.37.1 See also

Tasks

- *Building an application without the C library* on page 2-41
- *Using the C library with an application* on page 2-33.

Concepts

- *Building an application without the C library* on page 2-41
- *Use of the __user_libspace static data area by the C libraries* on page 2-21
- *Bare machine integer C* on page 2-46
- *Tailoring input/output functions in the C and C++ libraries* on page 2-92.

2.38 Program design when exploiting the C library

The functions you must re-implement depend on how much of the library functionality you require:

- If you want only the compiler support functions for division, structure copy, and floating-point 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 in `--fpmode=ieee_fixed` or `--fpmode=ieee_full` mode, the program must also provide `__rt_fp_status_addr()`.
- 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.

2.38.1 See also

Tasks

- *Customized C library startup code and access to C library functions* on page 2-48.

Concepts

- *Use of the __user_libspace static data area by the C libraries* on page 2-21.

Reference

ARM C and C++ Libraries and Floating-Point Support Reference:

- *__rt_fp_status_addr()* on page 2-31
- *__rt_raise()* on page 2-35.

2.39 Using low-level functions when exploiting the C library

If you are using the libraries in an application that does not have a `main()` function, you must re-implement some functions in the library.

Caution

`__rt_raise()` is essential if you are using the heap.

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.

2.39.1 See also

Tasks

- *Customized C library startup code and access to C library functions* on page 2-48.

Reference

- *Building an application without the C library* on page 2-41.

2.40 Using high-level functions when exploiting the C library

High-level I/O functions can be used if the low-level functions are re-implemented. High-level I/O functions are those such as `fprintf()`, `printf()`, `scanf()`, `puts()`, `fgets()`, `fread()`, `fwrite()`, and `perror()`. Low-level functions are those such as `fputc()`, `fgetc()`, and `__backspace()`. Most of the formatted output functions also require a call to `setlocale()`.

Anything that uses `locale` must not be called before first calling `setlocale()`.`setlocale()` selects the appropriate locale. For example, `setlocale(LC_ALL, "C")`, where `LC_ALL` means that the call to `setlocale()` affects all locale categories, and `"C"` specifies the minimal environment for C translation. Locale-using functions include the functions in `ctype.h` and `locale.h`, the `printf()` family, the `scanf()` family, `ato*`, `strto*`, `strcoll/strxfrm`, and most of `time.h`.

2.40.1 See also

Tasks

- *Tailoring input/output functions in the C and C++ libraries* on page 2-92.

Concepts

- *Customized C library startup code and access to C library functions* on page 2-48
- *Building an application without the C library* on page 2-41.

2.41 Using `malloc()` when exploiting the C library

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. Without `__rt_heap_extend()`, certain library functionality is included that causes problems when you are writing bare machine C.

Prototypes for both `_init_alloc()` and `__rt_heap_extend()` are in `rt_heap.h`.

2.41.1 See also

Reference

- *Customized C library startup code and access to C library functions* on page 2-48.

2.42 Tailoring the C library to a new execution environment

Tailoring the C library to a new execution environment involves re-implementing functions to produce an application for a new 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 re-implement 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.

2.42.1 See also

Concepts

- *How C and C++ programs use the library functions* on page 2-54
- *Initialization of the execution environment and execution of the application* on page 2-55
- *C++ initialization, construction and destruction* on page 2-56
- *Legacy support for C$$pi_ctorvec instead of .init_array* on page 2-58
- *Exceptions system initialization* on page 2-59
- *Emergency buffer memory for exceptions* on page 2-60
- *Library functions called from main()* on page 2-61
- *Program exit and the assert macro* on page 2-62.

Reference

C and C++ Libraries and Floating-Point Support Reference:

- *__rt_entry* on page 2-28
- *__rt_exit()* on page 2-30
- *__rt_lib_init()* on page 2-33
- *__rt_lib_shutdown()* on page 2-34.

Other information

The following specifications include 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*.

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

The following topics describe specific functions used to initialize the execution environment and application, library exit functions, and target-dependent library functions that the application might call during its execution:

2.44 Initialization of the execution environment and execution of the application

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

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.
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`. For example:

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

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

1. Sets up the stack and the heap by one of a number of means that include calling `__user_setup_stackheap()`, calling `__rt_stackheap_init()`, or loading the absolute addresses of scatter-loaded regions.
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_`.
3. Calls `main()`, the user-level root of the application.

   From `main()`, your program might call, among other things, library functions.
4. Calls `exit()` with the value returned by `main()`.

2.44.1 See also

Concepts

- *C++ initialization, construction and destruction* on page 2-56
- *Library functions called from main()* on page 2-61
- *How C and C++ programs use the library functions* on page 2-54
- *Tailoring the C library to a new execution environment* on page 2-53.

Reference

ARM C and C++ Libraries and Support Reference:

- *__rt_entry* on page 2-28
- *__rt_lib_init()* on page 2-33.

2.45 C++ initialization, construction and destruction

The C++ Standard places certain requirements on the construction and destruction of objects with static storage duration.

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 and limit symbols to be accessible. If they are not, the linker generates an error.

2.45.1 See also

Concepts

- *Legacy support for C$$pi_ctorvec instead of .init_array* on page 2-58
- *How C and C++ programs use the library functions* on page 2-54
- *Tailoring the C library to a new execution environment* on page 2-53.

2.46 Legacy support for C$$pi_ctorvec instead of .init_array

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 something similar to:

```
LOAD_ROM 0x0000000
{
    EXEC_ROM 0x0000000
    {
        your_object.o(+RO)
        * (.init_array)
        * (C$$pi_ctorvec)        ; backwards compatibility
        ...
    }
    RAM 0x0100000
    {
        * (+RW,+ZI)
    }
}
```

2.46.1 See also

Concepts

- *C++ initialization, construction and destruction* on page 2-56
- *How C and C++ programs use the library functions* on page 2-54
- *Tailoring the C library to a new execution environment* on page 2-53.

2.47 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 behavior 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`.

2.47.1 See also

Concepts

- *How C and C++ programs use the library functions* on page 2-54
- *Tailoring the C library to a new execution environment* on page 2-53.

2.48 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`.

2.48.1 See also

Concepts

- *How C and C++ programs use the library functions* on page 2-54
- *Tailoring the C library to a new execution environment* on page 2-53.

2.49 Library functions called from `main()`

The function `main()` is the user-level root of the application. It requires the execution environment to be initialized and input/output functions to be capable of being 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.
- Call library functions that require a callout to a user-defined function, for example `__rt_fp_status_addr()` or `clock()`.
- Call library functions that use `locale` or `CTYPE`.
- Perform floating-point calculations that require the floating-point unit 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()`.
- Raise an error or other signal, for example `ferror`.

2.49.1 See also

Tasks

- *Tailoring the C library to a new execution environment* on page 2-53
- *Tailoring input/output functions in the C and C++ libraries* on page 2-92
- *Tailoring non-input/output C library functions* on page 2-104.

Concepts

- *Assembler macros that tailor locale functions in the C library* on page 2-63
- *Modification of C library functions for error signaling, error handling, and program exit* on page 2-80.

2.50 Program exit and the `assert` macro

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

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. 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 re-implement `abort()` and the `stdio` functions.

Another solution for retargeting is to re-implement 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`.

2.50.1 See also

Concepts

- *Tailoring the C library to a new execution environment* on page 2-53.

2.51 Assembler macros that tailor locale functions in the C library

Applications use locales when they display or process data that depends on the local language or region, for example, character set, monetary symbols, decimal point, time, and date. Locale-related functions are declared in the include file, `rt_locale.`

The following topics describe assembler macros that tailor locale functions:

Concepts

- *Link time selection of the locale subsystem in the C library* on page 2-64
- *Runtime selection of the locale subsystem in the C library* on page 2-67
- *Definition of locale data blocks in the C library* on page 2-68
- *LC_CTYPE data block* on page 2-71
- *LC_COLLATE data block* on page 2-74
- *LC_MONETARY data block* on page 2-76
- *LC_NUMERIC data block* on page 2-77
- *LC_TIME data block* on page 2-78.

Reference

ARM C and C++ Libraries and Floating-Point Support Reference:

- *lconv structure* on page 2-17
- *_get_lconv()* on page 2-12
- *localeconv()* on page 2-19
- *setlocale()* on page 2-37
- *_findlocale()* on page 2-10.

2.52 Link time selection of the locale subsystem in the C library

The locale subsystem of the C library can be selected at link time or can be extended to be selectable at runtime. The following list 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.

2.52.1 See also

Concepts

- *Assembler macros that tailor locale functions in the C library* on page 2-63
- *ISO8859-1 implementation* on page 2-65
- *Shift-JIS and UTF-8 implementation* on page 2-66.

2.53 ISO8859-1 implementation

Table 2-6 shows the ISO8859-1 (Latin-1 alphabet) locale categories.

Table 2-6 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 separation of 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.

2.53.1 See also

Concepts

- *Link time selection of the locale subsystem in the C library* on page 2-64.

2.54 Shift-JIS and UTF-8 implementation

Table 2-7 shows the Shift-JIS (Japanese characters) or UTF-8 (Unicode characters) locale categories.

Table 2-7 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 list describes the effects of Shift-JIS and UTF-8 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()`.

  UTF-8 encoding uses the same set of self-contained characters as the ASCII character set.

- The multibyte conversion functions such as `mbrtowc()`, `mbsrtowcs()`, and `wcrtomb()`, all convert between wide strings in Unicode and multibyte character strings in Shift-JIS or UTF-8.

- `printf("%ls")` converts a Unicode wide string into Shift-JIS or UTF-8 output, and `scanf("%ls")` converts Shift-JIS or UTF-8 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.

2.54.1 See also

Concepts

- *Link time selection of the locale subsystem in the C library* on page 2-64.

2.55 Runtime selection of the locale subsystem in the C library

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`.

2.55.1 See also

Concepts

- *Assembler macros that tailor locale functions in the C library* on page 2-63.

Reference

ARM C and C++ Libraries and Floating-Point Support Reference:

- *setlocale()* on page 2-37.

2.56 Definition of locale data blocks in the C library

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. The ARM Compiler toolchain 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.

2.56.1 Beginning the definition of a locale block

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

`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.

2.56.2 Specifying the data for a locale block

To specify the data for your locale block, call the macros for that locale type in the order specified for that particular locale type. The syntax is as follows:

`LC_`*`TYPE`*`_`*`function`*

Where:

`TYPE` is one of the following:
- `CTYPE`
- `COLLATE`
- `MONETARY`
- `NUMERIC`
- `TIME`.

`function` is a specific function, `table()`, `full_wctype()`, or `multibyte()`, related to your locale data.

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

2.56.3 Ending the definition of a locale block

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

`LC_`*`TYPE`*`_end`

where:

`TYPE` is one of the following:
- `CTYPE`

- COLLATE
- MONETARY
- NUMERIC
- TIME.

2.56.4 Example of a fixed locale block

To write a fixed function that always returns the same locale, you can use the _start symbol name defined by the macros. The following 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
```

2.56.5 Example of multiple contiguous locale blocks

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. The following 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
```

2.56.6 See also

Concepts

- *Definition of locale data blocks in the C library* on page 2-68
- *Assembler macros that tailor locale functions in the C library* on page 2-63

- *LC_CTYPE data block* on page 2-71
- *LC_COLLATE data block* on page 2-74
- *LC_MONETARY data block* on page 2-76
- *LC_NUMERIC data block* on page 2-77
- *LC_TIME data block* on page 2-78.

2.57 LC_CTYPE data block

When defining a locale data block in the C library, 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.
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 the bitwise OR of one or more 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.

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 multibyte character set. Call this macro with three arguments. The first two arguments are the names of functions that perform conversion between the multibyte 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.

The following example 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 multibyte 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
```

2.57.1 See also

Concepts

- *Definition of locale data blocks in the C library* on page 2-68
- *Assembler macros that tailor locale functions in the C library* on page 2-63.

2.58 LC_COLLATE data block

When defining a locale data block in the C library, 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.
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.
3. Call LC_COLLATE_end, without arguments, to finish the locale block definition.

The following example 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
```

2.58.1 See also

Concepts

- *ISO8859-1 implementation* on page 2-65
- *Definition of locale data blocks in the C library* on page 2-68
- *Assembler macros that tailor locale functions in the C library* on page 2-63.

2.59 LC_MONETARY data block

When defining a locale data block in the C library, 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.
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.
3. Call LC_MONETARY_end, without arguments, to finish the locale block definition.

The following example 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
```

2.59.1 See also

Concepts

- *Definition of locale data blocks in the C library* on page 2-68
- *Assembler macros that tailor locale functions in the C library* on page 2-63.

2.60 LC_NUMERIC data block

When defining a locale data block in the C library, 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.
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.

The following example 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
```

2.60.1 See also

Concepts

- *Definition of locale data blocks in the C library* on page 2-68
- *Assembler macros that tailor locale functions in the C library* on page 2-63.

2.61 LC_TIME data block

When defining a locale data block in the C library, 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.
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.

The following example 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
```

2.61.1 See also

Concepts

- *Definition of locale data blocks in the C library* on page 2-68
- *Assembler macros that tailor locale functions in the C library* on page 2-63.

2.62 Modification of C library functions for error signaling, error handling, and program exit

All trap or error signals raised by the C library go through the `__raise()` function. You can re-implement 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`.

The `rt_misc.h` header file contains more information on error-related functions.

The trap and error-handling functions are shown in Table 2-8.

Table 2-8 Trap and error handling

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

2.62.1 See also

Reference

- Chapter 4 *Floating-point support*
- *Tailoring the C library to a new execution environment* on page 2-53.

ARM C and C++ Libraries and Floating-Point Support Reference:

- *_sys_exit()* on page 2-47
- *errno* on page 2-9
- *__rt_errno_addr()* on page 2-29
- *__raise()* on page 2-23
- *__rt_raise()* on page 2-35
- *__default_signal_handler()* on page 2-8
- *_ttywrch()* on page 2-57
- *__rt_fp_status_addr()* on page 2-31.

2.63 Modification of memory management functions in the C library

The following topics describe the functions in `rt_heap.h` that you can define if you are tailoring memory management:

- *Avoiding the heap and heap-using library functions supplied by ARM* on page 2-82
- *C library support for memory allocation functions* on page 2-83
- *Heap1, standard heap implementation* on page 2-84
- *Heap2, alternative heap implementation* on page 2-85
- *Using a heap implementation from bare machine C* on page 2-86.

ARM C and C++ Libraries and Floating-Point Support Reference:

- *alloca()* on page 2-5.

The `rt_heap.h` and `rt_memory.s` include files contain more information on memory-related functions.

2.64 Avoiding the heap and heap-using library functions supplied by ARM

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:

- re-implement 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`.

2.64.1 See also

Concepts

- *Modification of memory management functions in the C library* on page 2-81.

Reference

Linker Reference:

- *--list=filename* on page 2-102
- *--verbose* on page 2-184.

2.65 C library support for memory allocation functions

`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.

Concepts

- *Heap1, standard heap implementation* on page 2-84
- *Heap2, alternative heap implementation* on page 2-85.

Tasks

- *Using a heap implementation from bare machine C* on page 2-86.

Reference

ARM C and C++ Libraries and Floating-Point Support Reference:

- *alloca()* on page 2-5.

2.66 Heap1, standard heap implementation

Heap1, the default implementation, implements the smallest and simplest heap manager. The heap is managed as a single-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 performance 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.

2.66.1 See also

Concepts

- *C library support for memory allocation functions* on page 2-83.

2.67 Heap2, alternative heap implementation

Heap2 provides a compact implementation with the performance 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 of the following:

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

The Heap2 real-time heap implementation must know the maximum address space that 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.

2.67.1 See also

Concepts

- *C library support for memory allocation functions* on page 2-83.

2.68 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.

   Note

   The parameters of `_init_alloc(`*`base`*`, `*`top`*`)` must be eight-byte aligned.

2. Define the function `unsigned __rt_heap_extend(unsigned size, void **block)` to handle calls to extend the heap when it becomes full.

2.68.1 See also

Concepts

- *C library support for memory allocation functions* on page 2-83.

2.69 Stack pointer initialization and heap bounds

The C library requires you to specify where the stack pointer begins. If you intend to use ARM library functions that use the heap, for example, `malloc()`, `calloc()`, or if you define `argc` and `argv` command-line arguments for `main()`, the C library also requires you to specify which region of memory the heap is initially expected to use.

The region of memory used by the heap can be extended at a later stage of program execution, if required.

You can specify where the stack pointer begins, and which region of memory the heap is initially expected to use, with any of the following methods:

- Define the symbol `__intial_sp` to point to the top of the stack. If using the heap, also define symbols `__heap_base` and `__heap_limit`.
- In a scatter file, either:
  - define `ARM_LIB_STACK` and `ARM_LIB_HEAP` regions
  - if you do not intend to use the heap, only define an `ARM_LIB_STACK` region
  - define an `ARM_LIB_STACKHEAP` region.

  If you define an `ARM_LIB_STACKHEAP` region, the stack starts at the top of that region. The heap starts at the bottom.

  Note

  The above two methods are the only methods that microlib supports, of defining where the stack pointer starts and of defining the heap bounds.
- Implement `__user_setup_stackheap()` to set up the stack pointer and return the bounds of the initial heap region.
- If you are using legacy code that uses `__user_initial_stackheap()`, and you do not want to replace `__user_initial_stackheap()` with `__user_setup_stackheap()`, continue to use `__user_initial_stackheap()`.

  Note

  ARM recommends that you switch to using `__user_setup_stackheap()` if you are still using `__user_initial_stackheap()`, unless your implementation of `__user_initial_stackheap()` is:
  - specialized in some way such that it is complex enough to require its own temporary stack to run on before it has created the proper stack
  - has some user-specific special requirement that means it has to be implemented in C rather than in assembly language.

The initial stack pointer must be aligned to a multiple of eight bytes.

By default, if memory allocated for the heap is destined to overlap with memory that lies in close proximity with the stack, the potential collision of heap and stack is automatically detected and the requested heap allocation fails. If you do not require this automatic collision detection, you can save a small amount of code size by disabling it with `#pragma import __use_two_region_memory`.

Note

The memory allocation functions (`malloc()`, `realloc()`, `calloc()`, `posix_memalign()`) *attempt* to detect allocations that collide with the current stack pointer. Such detection cannot be guaranteed to always be successful.

Although it is possible to automatically detect expansion of the heap into the stack, it is not possible to automatically detect expansion of the stack into heap memory.

For legacy purposes, it is possible for you to bypass all of these methods and behavior. You can do this by defining the following functions to perform your own stack and heap memory management:

- `__rt_stackheap_init()`
- `__rt_heap_extend()`.

2.69.1 See also

Tasks

- *Defining __initial_sp, __heap_base and __heap_limit* on page 2-89
- *Creating an initial stack pointer for use with microlib* on page 3-8
- *Creating the heap for use with microlib* on page 3-9
- *Extending heap size at runtime* on page 2-90.

ARM C and C++ Libraries and Floating-Point Support Reference:

- *__rt_heap_extend()* on page 2-32
- *__rt_stackheap_init()* on page 2-36.

Using the Linker:

- *Specifying stack and heap using the scatter file* on page 8-12.

Developing Software for ARM Processors:

- *Placing the stack and heap* on page 3-13.

Concepts

- *Legacy support for __user_initial_stackheap()* on page 2-91.

Reference

ARM C and C++ Libraries and Floating-Point Support Reference:

- *__user_heap_extend()* on page 2-58
- *__user_heap_extent()* on page 2-59
- *Legacy function __user_initial_stackheap()* on page 2-71
- *__rt_heap_extend()* on page 2-32
- *__rt_stackheap_init()* on page 2-36
- *__user_setup_stackheap()* on page 2-60
- *__vectab_stack_and_reset* on page 2-61.

2.70 Defining __initial_sp, __heap_base and __heap_limit

One of several methods you can use to specify the initial stack pointer and heap bounds is to define the following symbols:

- __initial_sp
- __heap_base
- __heap_limit.

You can define these symbols in an assembly language file, or by using the embedded assembler in C.

For example:

```
__asm void dummy_function(void)
{
    EXPORT __initial_sp
    EXPORT __heap_base
    EXPORT __heap_limit

__initial_sp EQU STACK_BASE
__heap_base EQU HEAP_BASE
__heap_limit EQU (HEAP_BASE + HEAP_SIZE)
}
```

The constants STACK_BASE, HEAP_BASE and HEAP_SIZE can be defined in a header file, for example stack.h, as follows:

```
/* stack.h */
#define HEAP_BASE 0x20100000  /* Example memory addresses */
#define STACK_BASE 0x20200000
#define HEAP_SIZE ((STACK_BASE-HEAP_BASE)/2)
#define STACK_SIZE ((STACK_BASE-HEAP_BASE)/2)
```

Note

This method of specifying the initial stack pointer and heap bounds is supported by both the standard C library (standardlib) and the micro C library (microlib).

2.70.1 See also

Concepts

- *Stack pointer initialization and heap bounds* on page 2-87.

2.71 Extending heap size at runtime

To enable the heap to extend into areas of memory other than the region of memory that is specified when the program starts, you can redefine the function `__user_heap_extend()`.

`__user_heap_extend()` returns blocks of memory for heap usage in extending the size of the heap.

2.71.1 See also

Concepts

- *Stack pointer initialization and heap bounds* on page 2-87.

Reference

Using the ARM C and C++ Libraries and Floating-Point Support:

- *__rt_heap_extend()* on page 2-32.

2.72 Legacy support for __user_initial_stackheap()

Defined in rt_misc.h, __user_initial_stackheap() is supported for backwards compatibility with earlier versions of the ARM C and C++ libraries.

Note

ARM recommends that you use __user_setup_stackheap() in preference to __user_initial_stackheap().

The differences between __user_initial_stackheap() and __user_setup_stackheap() are:

- __user_initial_stackheap() receives the stack pointer (containing the same value it had on entry to __main()) in r1, and is expected to return the new stack base in r1.

  __user_setup_stackheap() receives the stack pointer in sp, and returns the stack base in sp.

- __user_initial_stackheap() is provided with a small temporary stack to run on. This temporary stack enables __user_initial_stackheap() to be implemented in C, providing that it uses no more than 88 bytes of stack space.

  __user_setup_stackheap() has no temporary stack and cannot usually be implemented in C.

Using __user_setup_stackheap() instead of __user_initial_steakheap() reduces code size, because __user_setup_stackheap() has no requirement for a temporary stack.

In the following circumstances you cannot use the provided __user_setup_stackheap() function, but you can use the __user_initial_stackheap() function:

- your implementation is sufficiently complex that it warrants the use of a temporary stack when setting up the initial heap and stack
- you have a requirement to implement the heap and stack creation code in C rather than in assembly language.

2.72.1 See also

Reference

ARM C and C++ Libraries and Floating-Point Support Reference:

- *Legacy function __user_initial_stackheap()* on page 2-71
- *__user_setup_stackheap()* on page 2-60.

2.73 Tailoring input/output functions in the C and C++ libraries

The input/output library functions, such as the high-level `fscanf()` and `fprintf()`, and the low-level `fputc()` and `ferror()`, and the C++ object `std::cout`, are not target-dependent. However, the high-level library functions perform input/output by calling the low-level ones, which themselves call the system I/O functions, which are target-dependent. To retarget input/output, you can either avoid the high-level library functions, or redefine the lower-level functions, either the low-level library functions or, below them, the system I/O functions, and then use them through the high-level library functions.

Whether redefining the low-level library functions or redefining the system I/O functions is a better solution depends on your use. For example, UARTs write a single character at a time and the default `fputc()` uses buffering, so redefining this function without a buffer might suit a UART. However, where buffer operations are possible, redefining the system I/O functions would probably be more appropriate.

2.73.1 See also

Concepts

- *Target dependencies on low-level functions in the C and C++ libraries* on page 2-93
- *The C library printf family of functions* on page 2-95
- *The C library scanf family of functions* on page 2-96
- *The C library functions fread(), fgets() and gets()* on page 2-100
- *ARM C libraries and multithreading* on page 2-16.

Tasks

- *Re-implementing __backspace() in the C library* on page 2-101
- *Re-implementing __backspacewc() in the C library* on page 2-102
- *Redefining low-level library functions to enable direct use of high-level library functions in the C library* on page 2-97
- *Redefining target-dependent system I/O functions in the C library* on page 2-103.

Reference

ARM C and C++ Libraries and Floating-Point Support Reference:

- *_sys_open()* on page 2-50
- *_sys_close()* on page 2-44
- *_sys_read()* on page 2-51
- *_sys_write()* on page 2-54
- *_sys_ensure()* on page 2-46
- *_sys_flen()* on page 2-48
- *_sys_seek()* on page 2-52
- *_sys_istty()* on page 2-49
- *_sys_tmpnam()* on page 2-53
- *_sys_command_string()* on page 2-45
- *#pragma import(_main_redirection)* on page 2-22.

install_directory\include\rt_sys.h:

- system I/O function declarations.

2.74 Target dependencies on low-level functions in the C and C++ libraries

Table 2-9 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 `__stdin` and `__stdout`.

Table key:

1. `__FILE`, the file structure.
2. `__stdin`, the standard input object of type `__FILE`.
3. `__stdout`, the standard output object of type `__FILE`.
4. `fputc()`, outputs a character to a file.
5. `ferror()`, returns the error status accumulated during file I/O.
6. `fgetc()`, gets a character from a file.
7. `fgetwc()`
8. `fputwc()`
9. `__backspace()`, moves the file pointer to the previous character.
10. `__backspacewc()`.

Table 2-9 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	-	-	-	-	-	-

Table 2-9 Input/output dependencies (continued)

High-level function	Low-level object									
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	-	-	-	-	-
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

Note

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

2.74.1 See also

Tasks

- *Re-implementing __backspace() in the C library* on page 2-101
- *Redefining low-level library functions to enable direct use of high-level library functions in the C library* on page 2-97
- *Tailoring input/output functions in the C and C++ libraries* on page 2-92.

Concepts

- *The C library printf family of functions* on page 2-95
- *The C library scanf family of functions* on page 2-96.

Other information

- ISO C Reference, `http://www.open-std.org/`

2.75 The C library printf family of functions

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`.

2.75.1 See also

Tasks

- *Tailoring input/output functions in the C and C++ libraries* on page 2-92.

2.76 The C library scanf family of functions

The `scanf()` family consists of `scanf()` and `fscanf()`. These functions depend only on the functions `fgetc()`, `__FILE`, and `__backspace()`.

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

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

where `__stdin` is of type `__FILE`.

2.76.1 See also

Tasks

- *Re-implementing __backspace() in the C library* on page 2-101
- *Tailoring input/output functions in the C and C++ libraries* on page 2-92.

2.77 Redefining low-level library functions to enable direct use of high-level library functions in the C library

If you define your own version of `__FILE`, 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 2-9 and Example 2-10 on page 2-98 show you how to do this. However, consider modifying the system I/O functions instead of these low-level library functions if you require real file handling.

You are not required to re-implement every function shown in these examples. Only re-implement the functions that are used in your application.

Example 2-9 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");
}
```

Note

Be aware of endianness with `fputc()`. `fputc()` takes an `int` parameter, but contains only a character. Whether the character is in the first or the last 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;
}
```

Example 2-10 Retargeting cout

```
File 1: Re-implement any functions that require re-implementation.

#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 re-implemented 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.

2.77.1 See also

Tasks

- *Tailoring input/output functions in the C and C++ libraries* on page 2-92
- *Redefining target-dependent system I/O functions in the C library* on page 2-103.

2.78 The C library functions fread(), fgets() and gets()

The functions `fread()`, `fgets()`, and `gets()` are implemented as fast block input/output functions where possible. These fast implementations are part of the ARM stream implementation and they bypass `fgetc()`. Where the fast implementation is not possible, they 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.

2.78.1 See also

Tasks

- *Tailoring input/output functions in the C and C++ libraries* on page 2-92.

2.79 Re-implementing __backspace() in the C library

The function `__backspace()` is used by the `scanf` family of functions, and must be re-implemented if you retarget the `stdio` arrangements at the `fgetc()` level.

Note

Normally, you are not required to call `__backspace()` directly, unless you are implementing your own `scanf`-like function.

The syntax is:

```
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.

2.79.1 See also

Tasks

- *Re-implementing __backspacewc() in the C library* on page 2-102
- *Tailoring input/output functions in the C and C++ libraries* on page 2-92.

2.80 Re-implementing __backspacewc() in the C library

`__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.

2.80.1 See also

Tasks

- *Tailoring input/output functions in the C and C++ libraries* on page 2-92.

Reference

- *Re-implementing __backspace() in the C library* on page 2-101.

2.81 Redefining target-dependent system I/O functions in the C library

`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.

The default target-dependent I/O functions use semihosting. If any of these functions is redefined, then they must all be redefined. Example 2-11 shows you how to do this, for a device that supports writing but not reading.

Example 2-11 Retargeting the system I/O functions

```
FILEHANDLE _sys_open(const char *name, int openmode)
{
  return 1; /* everything goes to the same output */
}

int _sys_close(FILEHANDLE fh)
{
  return 0;
}
int _sys_write(FILEHANDLE fh, const unsigned char *buf,
               unsigned len, int mode)
{
  your_device_write(buf, len);
  return 0;
}
int _sys_read(FILEHANDLE fh, unsigned char *buf,
              unsigned len, int mode)
{
  return -1; /* not supported */
}
void _ttywrch(int ch)
{
  char c = ch;
  your_device_write(&c, 1);
}
int _sys_istty(FILEHANDLE fh)
{
  return 0; /* buffered output */
}
int _sys_seek(FILEHANDLE fh, long pos)
{
  return -1; /* not supported */
}
long _sys_flen(FILEHANDLE fh)
{
  return -1; /* not supported */
}
```

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

2.81.1 See also

Tasks

- *Tailoring input/output functions in the C and C++ libraries* on page 2-92.

2.82 Tailoring non-input/output C library functions

In addition to tailoring input/output C library functions, many C library functions that are not input/output functions can also be tailored. 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.

2.82.1 See also

Reference

ARM C and C++ Libraries and Floating-Point Support Reference:

- *clock()* on page 2-6
- *_clock_init()* on page 2-7
- *time()* on page 2-56
- *remove()* on page 2-26
- *rename()* on page 2-27
- *system()* on page 2-55
- *getenv()* on page 2-13
- *_getenv_init()* on page 2-14.

2.83 Real-time integer division in the ARM libraries

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. For example, a division that generates a four-bit quotient might require only 12 cycles while a 32-bit quotient might require 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.

Note

- Real-time division is not available in the libraries for Cortex-M1 or Cortex-M0.
- The Cortex-R4 and Cortex-M3 processors support hardware floating-point divide, so they do not require the library divide routines.

2.83.1 See also

Tasks

- *Selecting real-time division in the ARM libraries* on page 2-106.

2.84 Selecting real-time division in the ARM libraries

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

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

2.84.1 See also

Concepts

- *Real-time integer division in the ARM libraries* on page 2-105.

2.85 How the ARM C library fulfills ISO C specification requirements

The ISO specification leaves some features to implementors, but requires that implementation choices be documented. The implementation of the generic ARM C library in this respect is as follows:

- 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>`.

- The following functions test for character values in the range `EOF` (-1) to 255 inclusive:
  - `isalnum()`
  - `isalpha()`
  - `iscntrl()`
  - `islower()`
  - `isprint()`
  - `isupper()`
  - `ispunct()`.
- The fully POSIX-compliant functions `remquo()`, `remquof()` and `remquol()` return the remainder of the division of x by y and store the quotient of the division in the pointer `*quo`. An implementation-defined integer value defines the number of bits of the quotient that are stored. In the ARM C library, this value is set to `4`.
- C99 behavior, with respect to mathlib error handling, is enabled by default.

  Note

  In RVCT 4.0, this behavior is not enabled by default, but is enabled through the use of `IMPORT __use_c99_matherr` in assembly language, or `#pragma import __use_c99_matherr` in C.

2.85.1 See also

Concepts

- *mathlib error handling* on page 2-108
- *ISO-compliant implementation of signals supported by the signal() function in the C library and additional type arguments* on page 2-110
- *ISO-compliant C library input/output characteristics* on page 2-112
- *Standard C++ library implementation definition* on page 2-114
- *Program exit and the assert macro* on page 2-62
- *C and C++ library naming conventions* on page 2-120.

2.86 mathlib error handling

In ARM Compiler 4.1 and later, the error handling of mathematical functions is consistent with Annex F of the ISO/IEC C99 standard. In RVCT 4.0 and earlier, it is not.

2.86.1 mathlib error handling in RVCT 4.0 and earlier

To invoke RVCT 4.0 and earlier behavior, you can define `__use_rvct_matherr`. Table 2-10 shows how the math functions respond when supplied with out-of-range arguments.

Table 2-10 Mathematical functions in RVCT 4.0 and earlier

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`
`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

Table 2-10 Mathematical functions in RVCT 4.0 and earlier (continued)

Function	Condition	Returned value	Error number
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 2-10 on page 2-108, 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 always returned, namely the one with all fraction bits clear except the topmost one. The sign bit is also clear. Passing strings of the form NAN(xxxx) to strtod has the same effect.

2.86.2 See also

Concepts

- *How the ARM C library fulfills ISO C specification requirements* on page 2-107
- *ISO-compliant implementation of signals supported by the signal() function in the C library and additional type arguments* on page 2-110
- *ISO-compliant C library input/output characteristics* on page 2-112
- *Standard C++ library implementation definition* on page 2-114.

Other info

- WG14/N1256 Committee Draft, *ISO/IEC 9899:TC3*, September 7, 2007.

2.87 ISO-compliant implementation of signals supported by the `signal()` function in the C library and additional type arguments

Table 2-11 shows the signals supported by the `signal()` function. It also shows which signals use an additional argument to give more information about the circumstance in which the signal was raised. The additional argument is given in the *type* parameter of `__raise()`. For example, division by floating-point zero results in a `SIGFPE` signal with a corresponding additional argument of `FE_EX_DIVBYZERO`.

Table 2-11 Signals supported by the `signal()` function

Signal	Number	Description	Additional argument
SIGABRT	1	Returned when any untrapped exception is thrown, such as: • a negative array size is allocated through the **new** operator • an invalid dynamic cast. This signal is only used if `abort()` or `assert()` are called by your C++ application, and `--exceptions` is specified.	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 [a]
SIGILL [b]	3	Illegal instruction.	None
SIGINT [b]	4	Attention request from user.	None
SIGSEGV [b]	5	Bad memory access.	None
SIGTERM [b]	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	Not normally used.	-
SIGOUTOFHEAP [c]	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

a. These constants are defined in `fenv.h`.

b. This signal is never generated by the library. It is available for you to raise manually, if required.

c. Not used in RVCT 2.1, and later.

Although **SIGSTAK** exists in `signal.h`, this signal is not 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`. However, you must compile these functions with a non-default FP model, such as `--fpmode=ieee_fixed` and upwards.

For all the signals in Table 2-11 on page 2-110, 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.

2.87.1 See also

Concepts

- *How the ARM C library fulfills ISO C specification requirements* on page 2-107
- *mathlib error handling* on page 2-108
- *ISO-compliant C library input/output characteristics* on page 2-112
- *Standard C++ library implementation definition* on page 2-114
- *Modification of C library functions for error signaling, error handling, and program exit* on page 2-80
- *Exception types recognized by the ARM floating-point environment* on page 4-45.

Reference

ARM C and C++ Libraries and Floating-Point Support Reference:

- *__raise()* on page 2-23
- *__rt_raise()* on page 2-35.

Compiler Reference:

- *--exceptions, --no_exceptions* on page 3-87
- *--fpmode=model* on page 3-97.

Other information

- *IEEE Standard for Floating-Point Arithmetic* (IEEE 754), 1985 version, `http://ieeexplore.ieee.org`

2.88 ISO-compliant C library 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 `NUL` 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 2-12.

Table 2-12 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 are unspecified in the ARM C library. They must be specified in an ISO-compliant implementation:

- 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()`.

2.88.1 See also

Concepts

- *How the ARM C library fulfills ISO C specification requirements* on page 2-107
- *ISO-compliant implementation of signals supported by the signal() function in the C library and additional type arguments* on page 2-110
- *mathlib error handling* on page 2-108
- *Standard C++ library implementation definition* on page 2-114.

2.89 Standard C++ library implementation definition

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 2-13.

For information on implementation-defined behavior that is defined in the Rogue Wave C++ library, see the included Rogue Wave HTML documentation.

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

Table 2-13 describes the most important features missing from the current release.

Table 2-13 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 by the ARM C library.

2.89.1 See also

Concepts

Introducing the ARM Compiler toolchain:

- *Rogue Wave documentation* on page 2-30.

2.90 C library functions and extensions

The ARM C library is fully compliant with the ISO C99 library standard. See Table 2-14 for GNU, POSIX, BSD-derived, and ARM compiler-specific extensions.

Table 2-14 C library extensions

Function	Header file definition	Extension
wcscasecmp()	`wchar.h`	GNU extension supported by the ARM libraries
wcsncasecmp()	`wchar.h`	GNU extension supported by the ARM libraries
wcstombs()	`stdlib.h`	POSIX extended functionality
posix_memalign()	`stdlib.h`	POSIX extended functionality
alloca()	`alloca.h`	Common non standard extension to many C libraries
strlcpy()	`string.h`	Common BSD-derived extension to many C libraries
strlcat()	`string.h`	Common BSD-derived extension to many C libraries
strcasecmp()	`string.h`	Standardized by POSIX
strncasecmp()	`string.h`	Standardized by POSIX
_fisatty()	`stdio.h`	Specific to the ARM compiler
__heapstats()	`stdlib.h`	Specific to the ARM compiler
__heapvalid()	`stdlib.h`	Specific to the ARM compiler

2.90.1 See also

Reference

ARM C and C++ Libraries and Floating-Point Support Reference:

- *wcscasecmp()* on page 2-62
- *wcsncasecmp()* on page 2-63
- *wcstombs()* on page 2-64
- *alloca()* on page 2-5
- *strlcpy()* on page 2-43
- *strlcat()* on page 2-42
- *strcasecmp()* on page 2-40
- *strncasecmp()* on page 2-41
- *_fisatty()* on page 2-11
- *__heapstats()* on page 2-15
- *__heapvalid()* on page 2-16.

2.91 Persistence of C and C++ library names across releases of the ARM compilation tools

The library naming convention described in this documentation applies to the current release of the ARM compilation tools.

Note

Do not rely on C and C++ library names. They might change in future releases.

2.91.1 See also

Tasks

- *Managing projects that have explicit C or C++ library names in makefiles* on page 2-118.

Concepts

- *Link time selection of C and C++ libraries* on page 2-117
- *Compiler generated and library-resident helper functions* on page 2-119
- *C and C++ library naming conventions* on page 2-120.

2.92 Link time selection of C and C++ libraries

Normally, you do not have to list any of the C and C++ 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.

2.92.1 See also

Tasks

- *Managing projects that have explicit C or C++ library names in makefiles* on page 2-118.

Concepts

- *Persistence of C and C++ library names across releases of the ARM compilation tools* on page 2-116
- *Compiler generated and library-resident helper functions* on page 2-119
- *C and C++ library naming conventions* on page 2-120.

2.93 Managing projects that have explicit C or C++ library names in makefiles

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.

A specific library can be included in a scatter file.

2.93.1 See also

Tasks

Using the Linker:

- *About placing ARM C and C++ library code* on page 8-49.

Concepts

- *Persistence of C and C++ library names across releases of the ARM compilation tools* on page 2-116
- *Link time selection of C and C++ libraries* on page 2-117
- *Compiler generated and library-resident helper functions* on page 2-119
- *C and C++ library naming conventions* on page 2-120.

Reference

Linker Reference:

- *--info=topic[,topic,...]* on page 2-80.

2.94 Compiler generated and library-resident helper functions

Compiler support or *helper* functions specific to the compilation tools 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. Be aware of redistribution rights of the libraries, as specified in your End User License Agreement.

2.94.1 See also

Tasks

- *Managing projects that have explicit C or C++ library names in makefiles* on page 2-118.

Concepts

- *Persistence of C and C++ library names across releases of the ARM compilation tools* on page 2-116
- *Link time selection of C and C++ libraries* on page 2-117
- *C and C++ library naming conventions* on page 2-120.

2.95 C and C++ library naming conventions

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][enum][wchar].endian*

root	armlib	An ARM C library. The arm_linux subdirectory contains libraries used for building ARM Linux applications.
	cpplib	An ARM C++ library.
prefix	armlinux	Libraries used when building ARM Linux applications.
	c	ISO C and C++ basic runtime support.
	cpp	Rogue Wave C++ library.
	cpprt	The ARM C++ runtime libraries.
	f	--fpmode=ieee_fixed. IEEE-compliant library with a fixed rounding mode (round to nearest) and no inexact exceptions.
	fj	--fpmode=ieee_no_fenv. IEEE-compliant library with a fixed rounding mode (round to nearest) and no exceptions.
	fz	--fpmode=fast or --fpmode=std. 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	--fpmode=ieee_full. IEEE-compliant library with configurable rounding mode and all IEEE exceptions.
	h	Compiler support (helper) library.
	m	Transcendental math functions.
	mc	Non ISO C-compliant ISO C micro-library basic runtime support.
	mf	Non IEEE 754 floating-point compliant 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.
	w	A Thumb only library for use with Cortex-M3.
	p	A Thumb only library for use with Cortex-M1 and Cortex-M0.
	2	A combined ARM and Thumb library for use with Cortex-A and Cortex-R series processor cores. You can prevent this library being selected using the linker option --no_thumb2_library.
fpu	m	A variant of the library for processors that have single-precision hardware floating-point only, such as Cortex-M4.
	v	Uses VFP instruction set.
	s	Soft VFP.

Note

If neither `v` nor `s` is present in a library name, the library provides no floating-point support.

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.

enum	`n`	Compatible with the compiler option, `--enum_is_int`.
wchar	`u`	Indicates the size of `wchar_t`.When present, the library is compatible with the compiler option, `--wchar32`. Otherwise, it is compatible with `--wchar16`
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 the ARM compilation tools.

The linker command-line option `--info libraries` provides information on every library that is automatically selected for the link stage.

2.95.1 See also

Concepts

Introducing the ARM Compiler toolchain:

- *ARM architectures supported by the toolchain* on page 2-17.

Reference

- *Compiler generated and library-resident helper functions* on page 2-119.

Linker Reference:

- *--info=topic[,topic,...]* on page 2-80
- *--thumb2_library, --no_thumb2_library* on page 2-172.

Compiler Reference:

- *--enum_is_int* on page 3-85
- *--wchar16* on page 3-224
- *--wchar32* on page 3-225.

2.96 Using macro__ARM_WCHAR_NO_IO to disable FILE declaration and wide I/O function prototypes

In strict C/C++ mode, the header files `wchar.h` and `cwchar` do not declare the `FILE` type. You can also define the macro `__ARM_WCHAR_NO_IO` to cause these header files not to declare `FILE` or the wide I/O function prototypes.

(Declaring the `FILE` type can lead to better consistency in debug information.)

Chapter 3
The ARM C micro-library

The following topics describe the C *micro-library* (microlib):

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 memory footprints. These applications do not run under an operating system.

Note

Microlib does not attempt to be an ISO C-compliant 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.

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()`.

3.1.1 See also

Concepts

- *Differences between microlib and the default C library* on page 3-3
- *ISO C features missing from microlib* on page 3-5.

3.2 Differences between microlib and 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)`.
- `#pragma import(__use_smaller_memcpy)` selects a smaller, but slower, version of `memcpy()`.
- `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.

3.2.1 See also

Concepts

- *ISO C features missing from microlib* on page 3-5.

Reference

Compiler Reference:

- *--fpmode=model* on page 3-97
- *#pragma import(__use_full_stdio)* on page 5-99
- *#pragma import(__use_smaller_memcpy)* on page 5-100.

3.3 Library heap usage requirements of the ARM C micro-library

Library heap usage requirements for microlib differ to those of standardlib as follows:

- the size of heap memory allocated for `fopen()` is 20 bytes for the `FILE` structure
- no buffer is ever allocated.

You must not declare `main()` to take arguments if you are using microlib.

Note

The size of heap memory allocated for `fopen()` might change in future releases.

3.3.1 See also

Concepts

- *Library heap usage requirements of the ARM C and C++ libraries* on page 2-8.

3.4 ISO C features missing from microlib

Major ISO C90 features not supported by microlib are:

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

Almost all functions that interact with an operating system are not supported by microlib. For example, `abort()`, `exit()`, `atexit()`, `assert()`, `time()`, `system()` and `getenv()`. An exception is `clock()`. A minimal implementation of `clock()` has been provided, which returns only –1, not the elapsed time. You may reimplement `clock()` (and `_clock_init()`, which it needs), if required.

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 input denormals produce indeterminate results. Operations that produce a result that is nonzero but very small in value, return zero.
- 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 UNKNOWN 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.

3.4.1 See also

Concepts

- *Differences between microlib and the default C library* on page 3-3
- *C library API definitions for targeting a different environment* on page 2-40
- *Building an application without the C library* on page 2-41.

Reference

ARM C and C++ Libraries and Floating-Point Support Reference:

- *clock()* on page 2-6
- *_clock_init()* on page 2-7.

Compiler Reference:

- *#pragma import(__use_full_stdio)* on page 5-99.

3.5 Building an application with microlib

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 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
```

3.5.1 See also

Tasks

- *Creating an initial stack pointer for use with microlib* on page 3-8
- *Creating the heap for use with microlib* on page 3-9
- *Entering and exiting programs linked with microlib* on page 3-10.

Reference

Compiler Reference:

- *--library_type=lib* on page 3-130.

Linker Reference:

- *input-file-list* on page 2-87.

3.6 Creating an initial stack pointer for use with microlib

To use microlib, you must specify an initial pointer for the stack. You can use either of the following methods to do this:

- use a scatter file
- define a symbol, `__initial_sp`, to be equal to the top of the stack.

The scatter file method uses `ARM_LIB_STACK` and `ARM_LIB_STACKHEAP`.

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
}
```

3.6.1 See also

Tasks

- *Defining __initial_sp, __heap_base and __heap_limit* on page 2-89
- *Creating the heap for use with microlib* on page 3-9
- *Entering and exiting programs linked with microlib* on page 3-10.

Using the Linker:

- Chapter 8 *Using scatter files*.

3.7 Creating the heap for use with microlib

To use the heap functions, for example, `malloc()`, `calloc()`, `realloc()` and `free()`, you must specify the location and size of the heap region.

To specify the start and end of the heap you can use either of the following methods:

- use a scatter file
- define symbols `__heap_base` and `__heap_limit`.

The scatter file method uses `ARM_LIB_HEAP` and `ARM_LIB_STACKHEAP`.

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
}
```

3.7.1 See also

Tasks

- *Creating an initial stack pointer for use with microlib* on page 3-8
- *Entering and exiting programs linked with microlib* on page 3-10
- *Defining __initial_sp, __heap_base and __heap_limit* on page 2-89.

Using the Linker:

- Chapter 8 *Using scatter files*.

3.8 Entering and exiting programs linked with microlib

Use `main()` to begin your program. Do not declare `main()` to take arguments.

Your program must not return from `main()`. This is because microlib does not contain any code to handle exit from `main()`. You can ensure that your `main()` function does not return, by inserting an endless loop at the end of the function. For example:

```
void main()
{
  ...
  while (1); // endless loop to prevent return from main()
}
```

Microlib does not support:

- command-line arguments from an operating system
- programs that call `exit()`.

3.9 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`.

3.9.1 See also

Reference

- *Tailoring input/output functions in the C and C++ libraries* on page 2-92.

Chapter 4
Floating-point support

The following topics describe the ARM support for floating-point computations:

4.1 About floating-point support

The ARM floating-point environment is an implementation of the IEEE 754-1985 standard for binary floating-point arithmetic.

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.

4.1.1 See also

Concepts

- *IEEE 754 arithmetic* on page 4-32
- *The software floating-point library, fplib* on page 4-4.

Reference

Compiler Reference:

- *--fpu=name* on page 3-100.

4.2 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 provide a set of procedure calls to do floating-point arithmetic. These procedures are in the software floating-point library, fplib.

4.2.1 See also

Tasks

- *Calling fplib routines* on page 4-5.

Concepts

- *fplib arithmetic on numbers in a particular format* on page 4-6
- *fplib conversions between floats, doubles, and ints* on page 4-8
- *fplib conversion between long longs, floats, and doubles* on page 4-9
- *fplib comparisons between floats and doubles* on page 4-10
- *fplib C99 functions* on page 4-12.

4.3 Calling fplib routines

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. They are typically only a few instructions long because all they do is 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.

Except for the software floating-point library routines that implement C99 functionality, the fplib routines are declared in the header file `rt_fp.h`. You can include this file if you want to call an fplib routine directly. Software floating-point library routines that implement C99 functionality are declared in the standard header file `math.h`.

To call a function from assembler, the software floating-point function is named `__softfp_`*fn*. For example, to call the `cos()` function, implement the following code:

```
IMPORT __softfp_cos
BL __softfp_cos
```

4.3.1 See also

Concepts

- *fplib C99 functions* on page 4-12
- *The software floating-point library, fplib* on page 4-4.

Using the Compiler:

- *Compiler support for floating-point computations and linkage* on page 6-65.

Other information

- *Application Binary Interface* (ABI) for the ARM Architecture, `http://infocenter.arm.com/help/topic/com.arm.doc.subset.swdev.abi`

4.4 fplib 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*)
`_frnd`	**`float`**	**`float`**	Return x rounded to an integer (see b in *Notes on arithmetic routines*)
`_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 and c in *Notes on arithmetic routines*)
`_drnd`	**`double`**	**`double`**	Return x rounded to an integer (see b in *Notes on arithmetic routines*)
`_dsqrt`	**`double`**	**`double`**	Return square root of x

4.4.1 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 on page 4-8.

c The IEEE 754 `remainder()` function is a synonym for `_drem`. `remainder()` is defined in `math.h`.

4.4.2 See also

Concepts

- *fplib conversions between floats, doubles, and ints* on page 4-8
- *fplib conversion between long longs, floats, and doubles* on page 4-9
- *fplib comparisons between floats and doubles* on page 4-10
- *fplib C99 functions* on page 4-12
- *The software floating-point library, fplib* on page 4-4.

Other information

- *Application Binary Interface* (ABI) for the ARM Architecture, `http://infocenter.arm.com/help/topic/com.arm.doc.subset.swdev.abi`

4.5 fplib conversions between floats, doubles, and ints

Table 4-2 describes routines to perform conversions between number formats, excluding **long long** types.

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*)
_ffix_r	**float**	**int**
_ffixu	**float**	**unsigned int** (see *Notes on rounding*)
_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**

4.5.1 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.

4.5.2 See also

Concepts

- *fplib conversions between floats, doubles, and ints*
- *fplib conversion between long longs, floats, and doubles* on page 4-9
- *fplib comparisons between floats and doubles* on page 4-10
- *fplib C99 functions* on page 4-12
- *The software floating-point library, fplib* on page 4-4.

Other information

- *Application Binary Interface* (ABI) for the ARM Architecture, http://infocenter.arm.com/help/topic/com.arm.doc.subset.swdev.abi

4.6 fplib conversion between long longs, floats, and doubles

Table 4-3 describes routines to perform conversions between **long long**s, **float**s, and **double**s.

Table 4-3 Conversion routines between long longs, floats, and doubles

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*)
_ll_sfrom_f_r	**float**	**long long**
_ll_ufrom_f	**float**	**unsigned long long** (see *Notes on rounding*)
_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**

4.6.1 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.

4.6.2 See also

Concepts

- *fplib arithmetic on numbers in a particular format* on page 4-6
- *fplib conversions between floats, doubles, and ints* on page 4-8
- *fplib comparisons between floats and doubles* on page 4-10
- *fplib C99 functions* on page 4-12
- *The software floating-point library, fplib* on page 4-4.

4.7 fplib comparisons between floats and doubles

Table 4-4 describes routines to perform comparisons between floating-point numbers. See *Notes on floating-point comparison routines* on page 4-11 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	-
`_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

4.7.1 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`.

4.7.2 See also

Concepts

- *fplib arithmetic on numbers in a particular format* on page 4-6
- *fplib conversions between floats, doubles, and ints* on page 4-8
- *fplib conversion between long longs, floats, and doubles* on page 4-9
- *fplib comparisons between floats and doubles* on page 4-10
- *fplib C99 functions* on page 4-12
- *The software floating-point library, fplib* on page 4-4.

4.8 fplib C99 functions

Table 4-5 describes fplib functions that implement C99 functionality.

Table 4-5 fplib C99 functions

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
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

4.8.1 See also

Concepts

- *fplib arithmetic on numbers in a particular format* on page 4-6
- *fplib conversions between floats, doubles, and ints* on page 4-8
- *fplib conversion between long longs, floats, and doubles* on page 4-9
- *fplib comparisons between floats and doubles* on page 4-10
- *fplib C99 functions*
- *The software floating-point library, fplib* on page 4-4.

4.9 Controlling the ARM floating-point environment

The ARM compilation tools supply several different interfaces to the floating-point environment, for compatibility and porting ease. These interfaces enable you to change the rounding mode, enable and disable trapping of exceptions, and install your own custom exception trap handlers.

4.9.1 See also

Concepts

- *Floating-point functions for compatibility with Microsoft products* on page 4-14
- *C99-compatible functions for controlling the ARM floating-point environment* on page 4-15
- *ARM floating-point compiler extensions to the C99 interface* on page 4-21.

Reference

ARM C and C++ Libraries and Floating-Point Support Reference:

- *__ieee_status()* on page 3-8
- *__fp_status()* on page 3-5.

4.10 Floating-point functions for compatibility with Microsoft products

Some functions give compatibility with Microsoft products to ease porting of floating-point code to the ARM architecture. They are defined in `float.h`. These functions require you to select a floating-point model that supports exceptions. For example, `--fpmode=ieee_full` or `--fpmode=ieee_fixed`.

4.10.1 See also

Tasks

- *Controlling the ARM floating-point environment* on page 4-13.

Reference

- *_controlfp()* on page 3-3
- *_clearfp()* on page 3-2
- *_statusfp()* on page 3-13.

4.11 C99-compatible functions for controlling the ARM floating-point environment

Note

The following functionality requires you to select a floating-point model that supports exceptions, such as `--fpmode=ieee_full` or `--fpmode=ieee_fixed`.

The compiler supports all functions defined in the C99 standard, and functions that are not C99-standard. The C99-compatible functions are the only interface that enables you to install custom exception trap handlers with the ability to define your own return value. All the function prototypes, data types, and macros for this functionality 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 compiler defines them to be structure types.

The type `fenv_t` is defined to hold all the information about the current floating-point environment. This comprises:

- 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.

4.11.1 See also

Tasks

- *Controlling the ARM floating-point environment* on page 4-13.

Concepts

- *ARM floating-point compiler extensions to the C99 interface* on page 4-21
- *C99 rounding mode and floating-point exception macros* on page 4-16
- *Exception flag handling* on page 4-17
- *Functions for handling rounding modes* on page 4-18
- *Functions for saving and restoring the whole floating-point environment* on page 4-19
- *Functions for temporarily disabling exceptions* on page 4-20.

Reference

Compiler Reference:

- *--fpmode=model* on page 3-97.

4.12 C99 rounding mode and floating-point exception macros

Note

The following functionality requires you to select a floating-point model that supports exceptions, such as `--fpmode=ieee_full` or `--fpmode=ieee_fixed`.

C99 defines a macro for each rounding mode and each exception. The macros are:

- `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.

4.12.1 See also

Concepts

- *C99-compatible functions for controlling the ARM floating-point environment* on page 4-15.

Reference

Compiler Reference:

- *--fpmode=model* on page 3-97.

4.13 Exception flag handling

Note

The following functionality requires you to select a floating-point model that supports exceptions, such as `--fpmode=ieee_full` or `--fpmode=ieee_fixed`.

C99 provides the following 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.

4.13.1 See also

Concepts

- *C99-compatible functions for controlling the ARM floating-point environment* on page 4-15.

Reference

Compiler Reference:

- *--fpmode=model* on page 3-97.

4.14 Functions for handling rounding modes

Note

The following functionality requires you to select a floating-point model that supports exceptions., such as `--fpmode=ieee_full` or `--fpmode=ieee_fixed`.

C99 provides the following functions for handling rounding modes:

```
int fegetround(void); int fesetround(int round);
```

The `fegetround()` function returns the current rounding mode. The current rounding mode has a value equal to one of the C99 rounding mode macros or exceptions.

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.

4.14.1 See also

Concepts

- *C99 rounding mode and floating-point exception macros* on page 4-16
- *C99-compatible functions for controlling the ARM floating-point environment* on page 4-15.

Reference

Compiler Reference:

- *--fpmode=model* on page 3-97.

4.15 Functions for saving and restoring the whole floating-point environment

Note

The following functionality requires you to select a floating-point model that supports exceptions, such as `--fpmode=ieee_full` or `--fpmode=ieee_fixed`.

C99 provides the following 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.

4.15.1 See also

Concepts

- *C99-compatible functions for controlling the ARM floating-point environment* on page 4-15.

Reference

Compiler Reference:

- *--fpmode=model* on page 3-97.

4.16 Functions for temporarily disabling exceptions

Note

The following functionality requires you to select a floating-point model that supports exceptions, such as `--fpmode=ieee_full` or `--fpmode=ieee_fixed`.

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. You can implement the following code to 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.

4.16.1 See also

Concepts

- *C99-compatible functions for controlling the ARM floating-point environment* on page 4-15.

Reference

ARM C and C++ Libraries and Floating-Point Support Reference:

- *__ieee_status()* on page 3-8.

Compiler Reference:

- *--fpmode=model* on page 3-97.

4.17 ARM floating-point compiler extensions to the C99 interface

Note

The following functionality requires you to select a floating-point model that supports exceptions, such as `--fpmode=ieee_full` or `--fpmode=ieee_fixed`.

The ARM C library 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 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`, the same status variable that the function `__ieee_status()` sees, laid out in the same format.
- 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, the default exception trap action happens. The default is to cause a **SIGFPE** signal.

4.17.1 See also

Tasks

- *Writing a custom exception trap handler* on page 4-22
- *Controlling the ARM floating-point environment* on page 4-13.

Concepts

- *C99-compatible functions for controlling the ARM floating-point environment* on page 4-15
- *Example of a custom exception handler* on page 4-26.

Reference

ARM C and C++ Libraries and Floating-Point Support Reference:

- *__ieee_status()* on page 3-8.

Compiler Reference:

- *--fpmode=model* on page 3-97.

4.18 Writing a custom exception trap handler

Note

The following functionality requires you to select a floating-point model that supports exceptions, such as `--fpmode=ieee_full` or `--fpmode=ieee_fixed`.

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, op2` These 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` This 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`**.)

`__softfp__ieee_value_t myhandler`

The return value from the function is used as the result of the operation that caused the exception.

4.18.1 edata flags for exception trap handler

The flags contained in `edata` are:

`edata & FE_EX_RDIR`

This 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`*

This 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`

This is nonzero if the `FZ` bit was set when the operation was performed.

`edata & FE_EX_ROUND_MASK`

This 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`*

This gives the type of the operands to the function, as one of the type values shown in Table 4-6.

Table 4-6 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`**

Table 4-6 FE_EX_INTYPE_*MASK* operand type flags (continued)

Flag	Operand type
FE_EX_INTYPE_UINT	**unsigned int**
FE_EX_INTYPE_LONGLONG	**long long**
FE_EX_INTYPE_ULONGLONG	**unsigned long long**

edata & FE_EX_OUTTYPE_*MASK*

This gives the type of the operands to the function, as one of the type values shown in Table 4-7.

Table 4-7 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**
FE_EX_OUTTYPE_UINT	**unsigned int**
FE_EX_OUTTYPE_LONGLONG	**long long**
FE_EX_OUTTYPE_ULONGLONG	**unsigned long long**

edata & FE_EX_FN_*MASK*

This gives the nature of the operation that caused the exception, as one of the operation codes shown in Table 4-8.

Table 4-8 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 4-8 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 4-9.

Input and output types are the same for all operations except Compare and Convert.

Table 4-9 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

4.18.2 See also

Tasks

- *Controlling the ARM floating-point environment* on page 4-13.

Concepts

- *Example of a custom exception handler* on page 4-26.
- *ARM floating-point compiler extensions to the C99 interface* on page 4-21
- *C99-compatible functions for controlling the ARM floating-point environment* on page 4-15.

Reference

ARM C and C++ Libraries and Floating-Point Support Reference:

- *__ieee_status()* on page 3-8.

Compiler Reference:

- *--fpmode=model* on page 3-97
- *--strict, --no_strict* on page 3-194.

4.19 Example of a custom exception handler

Note

The following functionality requires you to select a floating-point model that supports exceptions, such as `--fpmode=ieee_full` or `--fpmode=ieee_fixed`.

Example 4-1 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);
}
```

4.19.1 See also

Tasks

- *Writing a custom exception trap handler* on page 4-22
- *Controlling the ARM floating-point environment* on page 4-13.

Concepts

- *ARM floating-point compiler extensions to the C99 interface* on page 4-21
- *C99-compatible functions for controlling the ARM floating-point environment* on page 4-15.

Reference

Compiler Reference:

- *--fpmode=model* on page 3-97.

4.20 Exception trap handling by signals

Note

The following functionality requires you to select a floating-point model that supports exceptions, such as `--fpmode=ieee_full` or `--fpmode=ieee_fixed`.

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()`. For example:

```
    __rt_raise(SIGFPE, _FPE_INVALID);
```

`__rt_raise()` is declared in `rt_misc.h`.

4.20.1 See also

Tasks

- *Writing a custom exception trap handler* on page 4-22
- *Controlling the ARM floating-point environment* on page 4-13.

Concepts

- *ARM floating-point compiler extensions to the C99 interface* on page 4-21
- *C99-compatible functions for controlling the ARM floating-point environment* on page 4-15.

Reference

ARM C and C++ Libraries and Floating-Point Support Reference:

- *__rt_raise()* on page 2-35.

Compiler Reference:

- *--fpmode=model* on page 3-97.

4.21 Using C99 signalling NaNs provided by mathlib (_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__`.

4.21.1 See also

Reference

Compiler Reference:

- *Predefined macros* on page 5-183.

Other information

- WG14 - C N965, *Optional support for Signaling NaNs*, http://www.open-std.org/

4.22 mathlib double and single-precision floating-point functions

The math library, `mathlib`, provides double and single-precision functions for mathematical calculations. For example, to calculate a cube root, you can use `cbrt()` (double-precision) or `cbrtf()` (single-precision).

ISO/IEC 14882 specifies that in addition to the **`double`** versions of the math functions in `<cmath>`, C++ adds **`float`** (and **`long double`**) overloaded versions of these functions. The ARM implementation extends this in scope to include the additional math functions that do not exist in C89, but that do exist in C99.

In C++, `std::cbrt()` on a **`float`** argument selects the single-precision version of the function, and the same type of selection applies to other floating-point functions in C++.

4.23 Nonstandard functions in mathlib

See the following topics for nonstandard mathlib functions:

- *gamma(), gamma_r()* on page 3-7
- *j0(), j1(), jn(), Bessel functions of the first kind* on page 3-11
- *y0(), y1(), yn(), Bessel functions of the second kind* on page 3-14.
- *significand(), fractional part of a number* on page 3-12.

4.24 IEEE 754 arithmetic

The ARM floating-point environment is an implementation of the IEEE 754 standard for binary floating-point arithmetic. See the following topics for a summary of the standard as it is implemented by the ARM compiler:

- *Basic data types for IEEE 754 arithmetic* on page 4-33
- *IEEE 754 arithmetic and rounding* on page 4-41
- *Exceptions arising from IEEE 754 floating-point arithmetic* on page 4-42.

4.25 Basic data types for IEEE 754 arithmetic

ARM floating-point values are stored in one of two data types, *single-precision* and *double-precision*. In this documentation, they are called **`float`** and **`double`**, these being the corresponding C data types.

4.25.1 See also

Concepts

- *Single precision data type for IEEE 754 arithmetic* on page 4-34
- *Double precision data type for IEEE 754 arithmetic* on page 4-36
- *Sample single precision floating-point values for IEEE 754 arithmetic* on page 4-37
- *Sample double precision floating-point values for IEEE 754 arithmetic* on page 4-39.

Other information

- *IEEE Standard for Floating-Point Arithmetic* (IEEE 754), 1985 version, `http://ieeexplore.ieee.org`

4.26 Single precision data type for IEEE 754 arithmetic

A **float** value is 32 bits wide. The structure is shown in Figure 4-1.

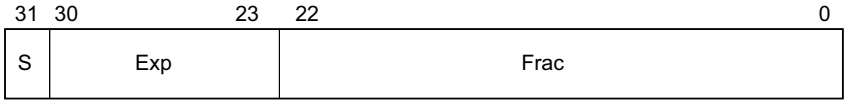

Figure 4-1 IEEE 754 single-precision floating-point format

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 *Not a Number* (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*.

4.26.1 See also

Concepts

- *Basic data types for IEEE 754 arithmetic* on page 4-33.

Reference

- *Sample single precision floating-point values for IEEE 754 arithmetic* on page 4-37.

Other information

- *IEEE Standard for Floating-Point Arithmetic* (IEEE 754), 1985 version, http://ieeexplore.ieee.org

4.27 Double precision data type for IEEE 754 arithmetic

A **double** value is 64 bits wide. Figure 4-2 shows its structure.

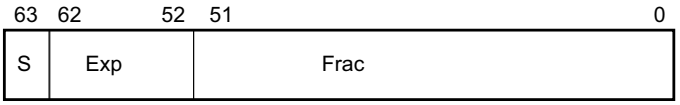

Figure 4-2 IEEE 754 double-precision floating-point format

As before, S is the sign, Exp the exponent, and Frac the fraction. Most of the detail of **float** values remains true for double values, 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.

4.27.1 See also

Concepts

- *Single precision data type for IEEE 754 arithmetic* on page 4-34
- *Basic data types for IEEE 754 arithmetic* on page 4-33.

Reference

- *Sample double precision floating-point values for IEEE 754 arithmetic* on page 4-39.

Other information

- *IEEE Standard for Floating-Point Arithmetic* (IEEE 754), 1985 version, http://ieeexplore.ieee.org

4.28 Sample single precision floating-point values for IEEE 754 arithmetic

Some sample **float** bit patterns, together with their mathematical values, are given in Table 4-10.

Table 4-10 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 single precision floating-point values* for more information.

4.28.1 Notes on sample single precision 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 when used, whereas quiet NaNs do not.

4.28.2 See also

Concepts

- *Single precision data type for IEEE 754 arithmetic* on page 4-34
- *Basic data types for IEEE 754 arithmetic* on page 4-33
- *Exceptions arising from IEEE 754 floating-point arithmetic* on page 4-42.

Other information

- *IEEE Standard for Floating-Point Arithmetic* (IEEE 754), 1985 version, http://ieeexplore.ieee.org

4.29 Sample double precision floating-point values for IEEE 754 arithmetic

Some sample **double** bit patterns, together with their mathematical values, are given in Table 4-11.

Table 4-11 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 double precision floating-point values* for more information.

4.29.1 Notes on sample double precision 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 when used, whereas quiet NaNs do not.

4.29.2 See also

Concepts

- *Exceptions arising from IEEE 754 floating-point arithmetic* on page 4-42
- *Double precision data type for IEEE 754 arithmetic* on page 4-36
- *Basic data types for IEEE 754 arithmetic* on page 4-33.

Other information

- *IEEE Standard for Floating-Point Arithmetic* (IEEE 754), 1985 version, http://ieeexplore.ieee.org

4.30 IEEE 754 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 output 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.

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.

4.30.1 See also

Concepts

- *IEEE 754 arithmetic* on page 4-32.

Reference

- *C and C++ library naming conventions* on page 2-120.

Other information

- *IEEE Standard for Floating-Point Arithmetic* (IEEE 754), 1985 version, `http://ieeexplore.ieee.org`

4.31 Exceptions arising from IEEE 754 floating-point arithmetic

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 an exception by inventing a plausible result for the operation and returning that result, or by trapping the exception.

4.31.1 See also

Concepts

- *Ignoring exceptions from IEEE 754 floating-point arithmetic operations* on page 4-43
- *Trapping exceptions from IEEE 754 floating-point arithmetic operations* on page 4-44
- *Exception types recognized by the ARM floating-point environment* on page 4-45
- *IEEE 754 arithmetic* on page 4-32.

Other information

- *IEEE Standard for Floating-Point Arithmetic* (IEEE 754), 1985 version, `http://ieeexplore.ieee.org`

4.32 Ignoring exceptions from IEEE 754 floating-point arithmetic operations

The system invents a plausible result for the operation and returns that result. 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 time in the past.

4.32.1 See also

Concepts

- *Exceptions arising from IEEE 754 floating-point arithmetic* on page 4-42
- *IEEE 754 arithmetic* on page 4-32.

Other information

- *IEEE Standard for Floating-Point Arithmetic* (IEEE 754), 1985 version, http://ieeexplore.ieee.org

4.33 Trapping exceptions from IEEE 754 floating-point arithmetic operations

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.

4.33.1 See also

Concepts

- *Exceptions arising from IEEE 754 floating-point arithmetic* on page 4-42
- *IEEE 754 arithmetic* on page 4-32.

Other information

- *IEEE Standard for Floating-Point Arithmetic* (IEEE 754), 1985 version, http://ieeexplore.ieee.org

4.34 Exception types recognized by the ARM floating-point environment

The ARM floating-point environment recognizes the following types of exception:

Invalid Operation exception

This occurs 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, these operations return a quiet NaN. The exception is conversion to an integer. This returns zero because there are no quiet NaNs in integers.

Divide by Zero exception

This occurs if you divide a finite nonzero number by zero. Be aware that:

- 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.

Overflow exception

This occurs 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 to itself. The largest float value is `0x7F7FFFFF`.

If Overflow is not trapped, the operation returns infinity, or the largest finite number, depending on the rounding mode.

Underflow exception

This can occur 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.

  Note

  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.

4.34.1 See also

Tasks

- *Writing a custom exception trap handler* on page 4-22.

Concepts

- *Exception flag handling* on page 4-17
- *Example of a custom exception handler* on page 4-26
- *Exception trap handling by signals* on page 4-28
- *IEEE 754 arithmetic* on page 4-32
- *Exceptions arising from IEEE 754 floating-point arithmetic* on page 4-42
- *Ignoring exceptions from IEEE 754 floating-point arithmetic operations* on page 4-43
- *Trapping exceptions from IEEE 754 floating-point arithmetic operations* on page 4-44.

Reference

- *ISO-compliant implementation of signals supported by the signal() function in the C library and additional type arguments* on page 2-110
- *Sample single precision floating-point values for IEEE 754 arithmetic* on page 4-37.

Other information

- *IEEE Standard for Floating-Point Arithmetic* (IEEE 754), 1985 version, `http://ieeexplore.ieee.org`

4.35 Using the *Vector Floating-Point* (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.

4.35.1 See also

Concepts

Using the Compiler:

- *Limitations on hardware handling of floating-point arithmetic* on page 6-61
- *Implementation of Vector Floating-Point (VFP) support code* on page 6-62.

Other information

- *Using VFP with RVDS, Application Note 133*, http://infocenter.arm.com/help/topic/com.arm.doc.dai0133-/index.html.

Appendix A
Revisions for Using ARM C and C++ Libraries and Floating-Point Support

This appendix describes the technical changes between released issues of this book.

Table A-1 Differences between issue G and issue H

Change	Topics affected
Noted the correct use of `_init_alloc(base, top)`.	*Using a heap implementation from bare machine C* on page 2-86
Added thread safety information about the Rogue Wave Standard C++ library.	*Thread safety in the ARM C++ library* on page 2-30

Table A-2 Differences between issue F and issue G

Change	Topics affected
Where appropriate, rather than 16-bit Thumb or 32-bit Thumb, referred instead to Thumb or Thumb-2 technology.	Various topics

Table A-2 Differences between issue F and issue G (continued)

Change	Topics affected
Explicitly stated that there are two ways to tailor I/O functions, with guidance about use.	*Tailoring input/output functions in the C and C++ libraries* on page 2-92
Made the terms for system I/O functions more consistent, and gave an example of how to redefine the functions.	• *Tailoring input/output functions in the C and C++ libraries* on page 2-92 • *Redefining target-dependent system I/O functions in the C library* on page 2-103
Clarified the type of division by zero in the example. Removed the incorrect floating point error values for `type`, and gave a reference for the correct ones.	*ISO-compliant implementation of signals supported by the signal() function in the C library and additional type arguments* on page 2-110

Table A-3 Differences between issue E and issue F

Change	Topics affected
Changed the ARMCC*nn*LIB environment variable to ARMCC*n*LIB.	*ARM C and C++ library directory structure* on page 2-11
Added the values for the arguments to the **SIGFPE** signal.	*ISO-compliant implementation of signals supported by the signal() function in the C library and additional type arguments* on page 2-110
Removed `clock()` from the list of unsupported ISO C features.	*ISO C features missing from microlib* on page 3-5
Where appropriate: • prefixed Thumb with 16-bit • changed Thumb-1 to 16-bit Thumb • changed Thumb-2 to 32-bit Thumb.	Various topics

Table A-4 Differences between issue D and issue E

Change	Topics affected
Corrected the library naming conventions.	*C and C++ library naming conventions* on page 2-120
Added `#pragma import(__use_smaller_memcpy)`.	*Differences between microlib and the default C library* on page 3-3

Table A-5 Differences between issue C and issue D

Change	Topics affected
Added a table footnote for signals **SIGILL**, **SIGINT**, **SIGSEGV**, and **SIGTERM**. Changed the description for **SIGCPPL**. Also updated the Caution.	*ISO-compliant implementation of signals supported by the signal() function in the C library and additional type arguments* on page 2-110
Added subtitles to the topic.	*Definition of locale data blocks in the C library* on page 2-68

Table A-6 Differences between issue B and issue C

Change	Topics affected
Removed `alloca()` from the first sentence.	*Library heap usage requirements of the ARM C and C++ libraries* on page 2-8
Removed the item describing `alloca` state from the list.	*Use of the __user_libspace static data area by the C libraries* on page 2-21
Reworded the `alloca.h` table entry.	*Building an application without the C library* on page 2-41

Table A-7 Differences between issue A and issue B

Change	Topics affected
New topic.	*Stack pointer initialization and heap bounds* on page 2-87
New topic.	*Defining __initial_sp, __heap_base and __heap_limit* on page 2-89
New topic.	*Extending heap size at runtime* on page 2-90
New topic.	*mathlib double and single-precision floating-point functions* on page 4-30
Textual clarification.	*Entering and exiting programs linked with microlib* on page 3-10
`__user_setup_stackheap()` clarification.	*Direct semihosting C library function dependencies* on page 2-38
`__user_libspace()` is a legacy function.	*Re-implementation of legacy function __user_libspace() in the C library* on page 2-23
Reliance of `remquo()`, `remquof()`, and `remquol()`, on implementation-defined integer value.	*How the ARM C library fulfills ISO C specification requirements* on page 2-107
`__user_initial_stackheap()` downgraded to legacy support only.	*Legacy support for __user_initial_stackheap()* on page 2-91
Topic obsolete.	Memory models and the C library
Topic obsolete.	Methods of modifying the runtime memory model with the C library
Topic obsolete.	User-defined C library memory models