ARM® Compiler

Version 5.04

ARM C and C++ Libraries and Floating-Point Support User Guide

ARM DUI0475J

ARM® Compiler

ARM C and C++ Libraries and Floating-Point Support User Guide

Release information

Document History

Issue	Date	Confidentiality	Change
A	28 May 2010	Non-Confidential	ARM Compiler v4.1 Release
B	30 September 2010	Non-Confidential	Update 1 for ARM Compiler v4.1
C	28 January 2011	Non-Confidential	Update 2 for ARM Compiler v4.1 Patch 3
D	30 April 2011	Non-Confidential	ARM Compiler v5.0 Release
E	29 July 2011	Non-Confidential	Update 1 for ARM Compiler v5.0
F	30 August 2011	Non-Confidential	ARM Compiler v5.01 Release
G	29 February 2012	Non-Confidential	Document update 1 for ARM Compiler v5.01 Release
H	27 July 2012	Non-Confidential	ARM Compiler v5.02 Release
I	31 January 2013	Non-Confidential	ARM Compiler v5.03 Release
J	27 November 2013	Non-Confidential	ARM Compiler v5.04 Release

Proprietary notice

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM® in the EU and other countries, except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Where the term ARM is used it means “ARM or any of its subsidiaries as appropriate”.

Some material in this document is based on IEEE 754 - 1985 IEEE Standard for Binary Floating-Point Arithmetic. The IEEE disclaims any responsibility or liability resulting from the placement and use in the described manner.

Confidentiality status

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

Unrestricted Access is an ARM internal classification.

Product status

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

Web address

www.arm.com

Contents

ARM® Compiler ARM C and C++ Libraries and Floating-Point Support User Guide

Preface

Chapter 1 The ARM C and C++ Libraries

Chapter 2

The ARM C Micro-library

Chapter 3 Floating-point Support

Chapter 4 The C and C++ Library Functions

Chapter 5 Floating-point Support Functions

Appendix A Libraries Document Revisions

List of Figures
ARM® Compiler ARM C and C++ Libraries and Floating-Point Support User Guide

List of Tables
ARM® Compiler ARM C and C++ Libraries and Floating-Point Support User Guide

Preface

This preface introduces the *ARM® Compiler ARM C and C++ Libraries and Floating-Point Support User Guide*.

It contains the following:

- *About this book on page 13.*

About this book

ARM Compiler ARM C and C++ Libraries and Floating-Point Support User Guide. This manual provides user information for the ARM libraries and floating-point support. Available as PDF.

Using this book

This book is organized into the following chapters:

Chapter 1 The ARM C and C++ Libraries

Describes the ARM C and C++ libraries.

Chapter 2 The ARM C Micro-library

Describes the C micro-library (microlib).

Chapter 3 Floating-point Support

Describes ARM support for floating-point computations.

Chapter 4 The C and C++ Library Functions

Documents standard C and C++ library functions that are extensions to the C Standard or that differ in some way to the standard.

Chapter 5 Floating-point Support Functions

Describes ARM support for floating-point computations.

Appendix A Libraries Document Revisions

Describes the technical changes that have been made to the ARM C and C++ Libraries and Floating-Point Support User Guide.

Glossary

The ARM Glossary is a list of terms used in ARM documentation, together with definitions for those terms. The ARM Glossary does not contain terms that are industry standard unless the ARM meaning differs from the generally accepted meaning.

See the *ARM Glossary* for more information.

Typographic conventions

italic	Introduces special terminology, denotes cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.
`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace` (underlined: mono)	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`monospace italic`	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
`monospace bold`	Denotes language keywords when used outside example code.

<and>

Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>
```

SMALL CAPITALS

Used in body text for a few terms that have specific technical meanings, that are defined in the *ARM glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Feedback

Feedback on this product

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

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to *errata@arm.com*. Give:

- The title.
- The number ARM DUI0475J.
- The page number(s) to which your comments refer.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Other information

- *ARM Information Center*.
- *ARM Technical Support Knowledge Articles*.
- *Support and Maintenance*.
- *ARM Glossary*.

Chapter 1
The ARM C and C++ Libraries

Describes the ARM C and C++ libraries.

It contains the following:

1.1 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.

Related concepts

1.2 C and C++ runtime libraries on page 1-19.

1.27 Using the libraries in a nonsemihosting environment on page 1-48.

Related references

1.32 Building an application without the C library on page 1-54.

1.2 C and C++ runtime libraries

ARM provides the C standardlib, C microlib, and C++ runtime libraries 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 that 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.

Related concepts

1.1 Mandatory linkage with the C library on page 1-18.

Related references

1.32 Building an application without the C library on page 1-54.

1 The ARM C and C++ Libraries on page 1-15.

2 The ARM C Micro-library on page 2-132.

Related information

ISO C library standard.

IEEE Standard for Floating-Point Arithmetic (IEEE 754), 1985 version.

What is Semihosting?.

Rogue Wave documentation.

1.3 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.

Related information

What is Semihosting?.

1.4 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.

Related concepts

2.3 Library heap usage requirements of the ARM C micro-library on page 2-135.

1.63 Heap1, standard heap implementation on page 1-91.

1.64 Heap2, alternative heap implementation on page 1-92.

1.65 Using a heap implementation from bare machine C on page 1-93.

Related references

1.61 Avoiding the heap and heap-using library functions supplied by ARM on page 1-89.

1.62 C library support for memory allocation functions on page 1-90.

1.5 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 and 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.

Related tasks

1.6 Increasing portability of object files to other CLIBABI implementations on page 1-23.

Related information

Application Binary Interface (ABI) for the ARM Architecture.

DWARF Debugging Standard.

1.6 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.

Note

This reduces the performance of some library operations.

There are a number of methods you can use to request full CLIBABI portability.

Procedure

- 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.

Related concepts

1.5 Compliance with the Application Binary Interface (ABI) for the ARM architecture on page 1-22.

Related information

Application Binary Interface (ABI) for the ARM Architecture.

-Dname[(parm-list)][=def] compiler option.

1.7 ARM C and C++ library directory structure

The libraries are installed in the `armlib` and `cpplib` 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 `ARMCC5LIB` 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.

Related information

Rogue Wave documentation.

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

When you build your application, you must make certain choices such as the target architecture, instruction set, and byte order. You communicate these choices to the compiler using build options. The linker then selects appropriate C and C++ library variants compatible with these build options.

Choices that influence the ARM C and C++ library variant include the following:

Target Architecture and instruction set
ARM or Thumb instruction sets.

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.

Related information

Code compatibility between separately compiled and assembled modules.

--apcs=qualifier...qualifier compiler option.

--arm compiler option.

--bigend compiler option.

--fpu=name compiler option.

--littleend compiler option.

--thumb compiler option.

--fpu=name linker option.

--ropi linker option.

--rwpi linker option.

--arm assembler option.

--bigend assembler option.

--fpu assembler option.

--littleend assembler option.

--thumb assembler option.

1.9 Thumb C libraries

The linker automatically links in the Thumb C library if the objects to be linked contain Thumb instructions.

Objects use Thumb instructions if they have been built for:

- Thumb code, 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 both 16-bit and 32-bit Thumb code.

Related information

Cortex-A series processors.

Cortex-R series processors.

Cortex-M series processors.

--arm compiler option.

--thumb compiler option.

--arm_only compiler option.

#pragma thumb.

1.10 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`.

Standard library names are defined 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.

Related information

--using_std, --no_using_std compiler option.

1.11 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.

Related concepts

1.12 ARM C libraries and reentrant functions on page 1-29.

1.13 ARM C libraries and thread-safe functions on page 1-30.

Related references

1.20 Using the ARM C library in a multithreaded environment on page 1-39.

1.12 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.

Related concepts

1.13 ARM C libraries and thread-safe functions on page 1-30.

1.11 ARM C libraries and multithreading on page 1-28.

1.13 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.

Related concepts

1.12 ARM C libraries and reentrant functions on page 1-29.

1.11 ARM C libraries and multithreading on page 1-28.

1.11 ARM C libraries and multithreading on page 1-28.

1.12 ARM C libraries and reentrant functions on page 1-29.

1.21 Thread safety in the ARM C library on page 1-41.

1.22 Thread safety in the ARM C++ library on page 1-42.

Related references

1.18 Management of locks in multithreaded applications on page 1-36.

1.20 Using the ARM C library in a multithreaded environment on page 1-39.

1.14 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. Callouts from the C library enable access to static data.

Static 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 the following table. They do not use semihosting.

Table 1-1 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.

Related concepts

1.17 Re-implementation of legacy function __user_libspace() in the C library on page 1-35.

1.49 Assembler macros that tailor locale functions in the C library on page 1-73.

1.11 ARM C libraries and multithreading on page 1-28.

Related references

4.24 __rt_errno_addr() on page 4-216.

4.26 __rt_fp_status_addr() on page 4-218.

Related information

Code compatibility between separately compiled and assembled modules.

--apcs=qualifier...qualifier compiler option.

1.15 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. The C libraries use the `__user_libspace` area to store a number of different types of data.

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 keeps 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.

Related concepts

1.16 C library functions to access subsections of the __user_libspace static data area on page 1-34.

Related information

__aeabi_atexit() in C++ ABI for the ARM Architecture.

std::set_terminate(), std::set_unexpected(), in Exception Handling ABI for the ARM Architecture.

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

The `__user_perproc_libspace()` and `__user_perthread_libspace()` functions return subsections of the `__user_libspace` static data area.

`__user_perproc_libspace()`

Returns a pointer to 96 bytes for storing data that is global to an entire process. This data is shared between all threads.

`__user_perthread_libspace()`

Returns a pointer to 96 bytes for storing 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.

Related concepts

1.17 Re-implementation of legacy function __user_libspace() in the C library on page 1-35.

Related references

1.15 Use of the __user_libspace static data area by the C libraries on page 1-33.

1.17 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()`.

Related concepts

1.16 C library functions to access subsections of the __user_libspace static data area on page 1-34.

1.16 C library functions to access subsections of the __user_libspace static data area on page 1-34.

1.14 Use of static data in the C libraries on page 1-31.

Related references

1.20 Using the ARM C library in a multithreaded environment on page 1-39.

1.15 Use of the __user_libspace static data area by the C libraries on page 1-33.

1.18 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.

Related concepts

1.19 How to ensure re-implemented mutex functions are called on page 1-38.

1.21 Thread safety in the ARM C library on page 1-41.

1.22 Thread safety in the ARM C++ library on page 1-42.

Related references

1.20 Using the ARM C library in a multithreaded environment on page 1-39.

1.19 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.

Related concepts

1.21 Thread safety in the ARM C library on page 1-41.

1.22 Thread safety in the ARM C++ library on page 1-42.

Related references

1.20 Using the ARM C library in a multithreaded environment on page 1-39.

1.18 Management of locks in multithreaded applications on page 1-36.

1.20 Using the ARM C library in a multithreaded environment

There are a number of requirements you must fulfil before you can use 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.

Related concepts

1.11 ARM C libraries and multithreading on page 1-28.

1.19 How to ensure re-implemented mutex functions are called on page 1-38.

1.21 Thread safety in the ARM C library on page 1-41.

1.22 Thread safety in the ARM C++ library on page 1-42.

Related references

1.18 Management of locks in multithreaded applications on page 1-36.

4.59 Thread-safe C library functions on page 4-252.

4.60 C library functions that are not thread-safe on page 4-255.

1.21 Thread safety in the ARM C library

ARM C library functions are either always thread-safe, never thread-safe, or thread-safe in certain circumstances.

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.

Related concepts

1.19 How to ensure re-implemented mutex functions are called on page 1-38.

1.22 Thread safety in the ARM C++ library on page 1-42.

Related references

1.20 Using the ARM C library in a multithreaded environment on page 1-39.

1.18 Management of locks in multithreaded applications on page 1-36.

4.59 Thread-safe C library functions on page 4-252.

4.60 C library functions that are not thread-safe on page 4-255.

Related information

Rogue Wave documentation.

1.22 Thread safety in the ARM C++ library

ARM C++ library functions are either always thread-safe, never thread-safe, or thread-safe in certain circumstances.

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.

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.

Related concepts

1.19 How to ensure re-implemented mutex functions are called on page 1-38.

1.21 Thread safety in the ARM C library on page 1-41.

Related references

1.20 Using the ARM C library in a multithreaded environment on page 1-39.

1.18 Management of locks in multithreaded applications on page 1-36.

Related information

__cxa_*, __aeabi_* functions, C++ ABI for the ARM Architecture.

Exception Handling ABI for the ARM Architecture.

1.23 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.

Related concepts

1.21 Thread safety in the ARM C library on page 1-41.

Related information

--fpmode=model compiler option.

1.24 Using the C library with an application

Depending on how you use the C and C ++ libraries with your application, you may need to re-implement particular functions.

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.

Related concepts

1.25 Using the C and C++ libraries with an application in a semihosting environment on page 1-46.

1.27 Using the libraries in a nonsemihosting environment on page 1-48.

Related references

1.32 Building an application without the C library on page 1-54.

1.25 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.

Related references

1.26 Using $Sub$$ to mix semihosted and nonsemihosted I/O functionality on page 1-47.

1.29 Direct semihosting C library function dependencies on page 1-51.

Related information

What is Semihosting?.

Tailoring the C library to your target hardware.

Tailoring the image memory map to your target hardware.

1.26 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.

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);
    }
}
```

Related concepts

1.25 Using the C and C++ libraries with an application in a semihosting environment on page 1-46.

1.27 Using the libraries in a nonsemihosting environment on page 1-48.

Related information

ELF for the ARM Architecture.

Use of $Super$$ and $Sub$$ to patch symbol definitions.

1.27 Using the libraries in a nonsemihosting environment

Some C library functions use semihosting. If you use the libraries in a nonsemihosting environment, you must ensure that semihosting function calls are dealt with appropriately.

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 **`armlink`** `--verbose --list err.txt`
2. Search `err.txt` for occurrences of `__I$use$semihosting`

   For example:

   ```
   ...
   Loading member sys_exit.o from c_4.l.
   reference : __I$use$semihosting
   definition: _sys_exit
   ...
   ```

   This shows that the semihosting-using function `_sys_exit` is linked-in from the C library. To prevent this, you must provide your own implementation of this function.

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

Related concepts

1.1 Mandatory linkage with the C library on page 1-18.

1.28 C++ exceptions in a non-semihosting environment on page 1-50.

Related references

1.26 Using $Sub$$ to mix semihosted and nonsemihosted I/O functionality on page 1-47.

1.30 Indirect semihosting C library function dependencies on page 1-52.

1.31 C library API definitions for targeting a different environment on page 1-53.

Related information

What is Semihosting?.

--list=filename linker option.

--verbose linker option.

1.28 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()`.

Related concepts

1.27 Using the libraries in a nonsemihosting environment on page 1-48.

1.29 Direct semihosting C library function dependencies

The following table shows the functions that depend directly on semihosting.

Table 1-2 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.

Related concepts

1.25 Using the C and C++ libraries with an application in a semihosting environment on page 1-46.

1.42 Initialization of the execution environment and execution of the application on page 1-66.

Related references

1.60 Modification of C library functions for error signaling, error handling, and program exit on page 1-88.

1.70 Tailoring input/output functions in the C and C++ libraries on page 1-99.

1.30 Indirect semihosting C library function dependencies on page 1-52.

1.79 Tailoring non-input/output C library functions on page 1-111.

4.54 __user_setup_stackheap() on page 4-247.

1.30 Indirect semihosting C library function dependencies

The following table shows functions that depend indirectly on one or more of the directly dependent functions.

You can use this table 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 1-3 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.).

Related concepts

1.27 Using the libraries in a nonsemihosting environment on page 1-48.

Related references

1.29 Direct semihosting C library function dependencies on page 1-51.

1.61 Avoiding the heap and heap-using library functions supplied by ARM on page 1-89.

1.60 Modification of C library functions for error signaling, error handling, and program exit on page 1-88.

1.70 Tailoring input/output functions in the C and C++ libraries on page 1-99.

1.79 Tailoring non-input/output C library functions on page 1-111.

1.31 C library API definitions for targeting a different environment

In addition to the direct and indirect semihosting dependent functions, there are a number of functions and files that might be useful when building for a different environment.

The following table shows these functions and files.

Table 1-4 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.

Related concepts

1.27 Using the libraries in a nonsemihosting environment on page 1-48.

1.49 Assembler macros that tailor locale functions in the C library on page 1-73.

Related information

--list=filename linker option.

--verbose linker option.

1.32 Building an application without the C library

If your application does not initialize the C library, a number of functions are not available in your application.

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()`.

The following table 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 1-5 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.

Table 1-5 Standalone C library functions (continued)

Function	Description
`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.
`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.

Table 1-5 Standalone C library functions (continued)

Function	Description
`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.
`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.

Related concepts

1.1 Mandatory linkage with the C library on page 1-18.

1.2 C and C++ runtime libraries on page 1-19.

1.33 Creating an application as bare machine C without the C library on page 1-57.

1.49 Assembler macros that tailor locale functions in the C library on page 1-73.

1.34 Integer and floating-point compiler functions and building an application without the C library on page 1-58.

1.39 Using high-level functions when exploiting the C library on page 1-63.

1.38 Using low-level functions when exploiting the C library on page 1-62.

Related references

1.70 Tailoring input/output functions in the C and C++ libraries on page 1-99.

1.60 Modification of C library functions for error signaling, error handling, and program exit on page 1-88.

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

Bare machine C 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.

Related concepts

1.34 Integer and floating-point compiler functions and building an application without the C library on page 1-58.

1.37 Customized C library startup code and access to C library functions on page 1-61.

1.38 Using low-level functions when exploiting the C library on page 1-62.

1.39 Using high-level functions when exploiting the C library on page 1-63.

1.35 Bare machine integer C on page 1-59.

1.36 Bare machine C with floating-point processing on page 1-60.

1.40 Using malloc() when exploiting the C library on page 1-64.

Related references

1.32 Building an application without the C library on page 1-54.

1.34 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. These functions require `__rt_raise()`.

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.

Related concepts

1.33 Creating an application as bare machine C without the C library on page 1-57.

1.91 Compiler generated and library-resident helper functions on page 1-126.

Related references

1.32 Building an application without the C library on page 1-54.

Related information

Application Binary Interface (ABI) for the ARM Architecture.

1.35 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, there are a number of requirements you must meet.

These requirements are:

- 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.

Related concepts

1.33 Creating an application as bare machine C without the C library on page 1-57.

1.36 Bare machine C with floating-point processing on page 1-60.

Related references

1.32 Building an application without the C library on page 1-54.

1.15 Use of the __user_libspace static data area by the C libraries on page 1-33.

Related information

--fpu=name compiler option.

1.36 Bare machine C with floating-point processing

If you want to use floating-point processing in an application without the C library, there are a number of requirements you must fulfil.

These requirements are:

- 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 static base register, 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()`).

Related concepts

1.33 Creating an application as bare machine C without the C library on page 1-57.

1.35 Bare machine integer C on page 1-59.

1.14 Use of static data in the C libraries on page 1-31.

Related references

1.32 Building an application without the C library on page 1-54.

1.15 Use of the __user_libspace static data area by the C libraries on page 1-33.

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

If you build an application with customized startup code, you must either avoid functions that require initialization or provide the initialization and low-level support 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.

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.

Related concepts

1.33 Creating an application as bare machine C without the C library on page 1-57.

1.24 Using the C library with an application on page 1-45.

1.35 Bare machine integer C on page 1-59.

1.38 Using low-level functions when exploiting the C library on page 1-62.

1.39 Using high-level functions when exploiting the C library on page 1-63.

1.40 Using malloc() when exploiting the C library on page 1-64.

Related references

1.32 Building an application without the C library on page 1-54.

1.15 Use of the __user_libspace static data area by the C libraries on page 1-33.

1.70 Tailoring input/output functions in the C and C++ libraries on page 1-99.

1.38 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.

Related concepts

1.39 Using high-level functions when exploiting the C library on page 1-63.

1.33 Creating an application as bare machine C without the C library on page 1-57.

1.37 Customized C library startup code and access to C library functions on page 1-61.

Related references

1.32 Building an application without the C library on page 1-54.

1.39 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`.

Related concepts

1.38 Using low-level functions when exploiting the C library on page 1-62.

1.33 Creating an application as bare machine C without the C library on page 1-57.

1.37 Customized C library startup code and access to C library functions on page 1-61.

Related references

1.32 Building an application without the C library on page 1-54.

1.40 Using malloc() when exploiting the C library

If heap support is required for bare machine C, you must implement `_init_alloc()` and `__rt_heap_extend()`.

`_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`.

Related concepts

1.33 Creating an application as bare machine C without the C library on page 1-57.

1.37 Customized C library startup code and access to C library functions on page 1-61.

1.41 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.

Functions whose names start with a single or double underscore 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.

Related concepts

1.43 C++ initialization, construction and destruction on page 1-67.

1.44 Legacy support for C$$pi_ctorvec instead of .init_array on page 1-68.

1.47 Library functions called from main() on page 1-71.

1.42 Initialization of the execution environment and execution of the application on page 1-66.

1.46 Emergency buffer memory for exceptions on page 1-70.

1.48 Program exit and the assert macro on page 1-72.

1.45 Exceptions system initialization on page 1-69.

1.47 Library functions called from main() on page 1-71.

Related references

4.23 __rt_entry on page 4-215.

4.25 __rt_exit() on page 4-217.

4.28 __rt_lib_init() on page 4-220.

4.29 __rt_lib_shutdown() on page 4-221.

1.42 Initialization of the execution environment and execution of the application

You can customize execution intialization by defining your own `__main` that branches to `__rt_entry`.

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()`.

Related concepts

1.47 Library functions called from main() on page 1-71.

1.41 Tailoring the C library to a new execution environment on page 1-65.

1.43 C++ initialization, construction and destruction on page 1-67.

Related references

1.29 Direct semihosting C library function dependencies on page 1-51.

4.23 __rt_entry on page 4-215.

4.28 __rt_lib_init() on page 4-220.

1.43 C++ initialization, construction and destruction

The C++ Standard places certain requirements on the construction and destruction of objects with static storage duration, and the ARM C++ compiler uses the `.init_array` area to achieve this.

The `.init_array` area 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.

Related concepts

1.44 Legacy support for C$$pi_ctorvec instead of .init_array on page 1-68.

1.41 Tailoring the C library to a new execution environment on page 1-65.

1.44 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)
    }
}
```

Related concepts

1.43 C++ initialization, construction and destruction on page 1-67.

1.41 Tailoring the C library to a new execution environment on page 1-65.

1.45 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`.

Related concepts

1.41 Tailoring the C library to a new execution environment on page 1-65.

1.46 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`.

Related concepts

1.41 Tailoring the C library to a new execution environment on page 1-65.

1.47 Library functions called from main()

The function `main()` can call a number of user-customizable functions in the C library.

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`.

Related concepts

1.42 Initialization of the execution environment and execution of the application on page 1-66.

1.41 Tailoring the C library to a new execution environment on page 1-65.

1.41 Tailoring the C library to a new execution environment on page 1-65.

1.49 Assembler macros that tailor locale functions in the C library on page 1-73.

Related references

1.70 Tailoring input/output functions in the C and C++ libraries on page 1-99.

1.79 Tailoring non-input/output C library functions on page 1-111.

1.60 Modification of C library functions for error signaling, error handling, and program exit on page 1-88.

1.48 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 a number of conditions.

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

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`.

Related concepts

1.41 Tailoring the C library to a new execution environment on page 1-65.

1.49 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.`

Related concepts

1.47 Library functions called from main() on page 1-71.

1.53 Runtime selection of the locale subsystem in the C library on page 1-77.

Related references

1.31 C library API definitions for targeting a different environment on page 1-53.

1.32 Building an application without the C library on page 1-54.

1.50 Link time selection of the locale subsystem in the C library on page 1-74.

1.54 Definition of locale data blocks in the C library on page 1-78.

1.55 LC_CTYPE data block on page 1-81.

1.56 LC_COLLATE data block on page 1-83.

1.57 LC_MONETARY data block on page 1-84.

1.58 LC_NUMERIC data block on page 1-85.

1.59 LC_TIME data block on page 1-86.

4.14 lconv structure on page 4-204.

4.9 _get_lconv() on page 4-199.

4.15 localeconv() on page 4-206.

4.32 setlocale() on page 4-224.

4.7 _findlocale() on page 4-197.

1.50 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.

Related concepts

1.49 Assembler macros that tailor locale functions in the C library on page 1-73.

1.52 Shift-JIS and UTF-8 implementation on page 1-76.

Related references

1.51 ISO8859-1 implementation on page 1-75.

1.51 ISO8859-1 implementation

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.

The following table shows the ISO8859-1 (Latin-1 alphabet) locale categories.

Table 1-6 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.

Related references

1.50 Link time selection of the locale subsystem in the C library on page 1-74.

1.52 Shift-JIS and UTF-8 implementation

The Shift-JIS and UTF-8 locales let you use Japanese and Unicode characters.

The following table shows the Shift-JIS (Japanese characters) or UTF-8 (Unicode characters) locale categories.

Table 1-7 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.

Related references

1.50 Link time selection of the locale subsystem in the C library on page 1-74.

1.53 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.

The `rt_locale.h` and `rt_locale.s` header files describe what must be implemented and provide some useful support macros.

Related concepts

1.49 Assembler macros that tailor locale functions in the C library on page 1-73.

Related references

4.32 setlocale() on page 4-224.

1.54 Definition of locale data blocks in the C library

Locale data blocks let you customize your own locales.

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.

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.

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.

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`.

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:

```
    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
```

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:

```
    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
```

Related concepts

1.49 Assembler macros that tailor locale functions in the C library on page 1-73.

Related references

1.55 LC_CTYPE data block on page 1-81.

1.56 LC_COLLATE data block on page 1-83.

1.57 LC_MONETARY data block on page 1-84.

1.58 LC_NUMERIC data block on page 1-85.

1.59 LC_TIME data block on page 1-86.

1.55 LC_CTYPE data block

The LC_CTYPE data block configures character classification and conversion.

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.

Example `LC_CTYPE` data block

```
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
```

Related concepts

1.49 Assembler macros that tailor locale functions in the C library on page 1-73.

Related references

1.54 Definition of locale data blocks in the C library on page 1-78.

1.56 LC_COLLATE data block

The LC_COLLATE data block configures collation of strings.

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.

Example `LC_COLLATE` data block

```
        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
```

Related concepts

1.49 Assembler macros that tailor locale functions in the C library on page 1-73.

Related references

1.54 Definition of locale data blocks in the C library on page 1-78.

1.51 ISO8859-1 implementation on page 1-75.

1.57 LC_MONETARY data block

The LC_MONETARY data block configures formatting of monetary values.

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.

Example `LC_MONETARY` data block

```
        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
```

Related concepts

1.49 Assembler macros that tailor locale functions in the C library on page 1-73.

Related references

1.54 Definition of locale data blocks in the C library on page 1-78.

1.58 LC_NUMERIC data block

The LC_NUMERIC data block configures formatting of numeric values that are not monetary.

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.

Example `LC_NUMERIC` data block

```
        LC_NUMERIC_begin c_numeric, "C"
        LC_NUMERIC_point "."
        LC_NUMERIC_thousands ""
        LC_NUMERIC_grouping ""
        LC_NUMERIC_end
```

Related concepts

1.49 Assembler macros that tailor locale functions in the C library on page 1-73.

Related references

1.54 Definition of locale data blocks in the C library on page 1-78.

1.59 LC_TIME data block

The LC_TIME data block configures formatting of date and time values.

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. The example below 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.

Example `LC_TIME` data block

```
        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
```

Related concepts

1.49 Assembler macros that tailor locale functions in the C library on page 1-73.

Related references

1.54 Definition of locale data blocks in the C library on page 1-78.

1.60 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 following table shows the trap and error-handling functions.

Table 1-8 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.

Related concepts

1.47 Library functions called from main() on page 1-71.

Related references

1.29 Direct semihosting C library function dependencies on page 1-51.

1.30 Indirect semihosting C library function dependencies on page 1-52.

1.32 Building an application without the C library on page 1-54.

4.41 _sys_exit() on page 4-234.

4.6 errno on page 4-196.

4.24 __rt_errno_addr() on page 4-216.

4.19 __raise() on page 4-210.

4.30 __rt_raise() on page 4-222.

4.5 __default_signal_handler() on page 4-195.

4.51 _ttywrch() on page 4-244.

4.26 __rt_fp_status_addr() on page 4-218.

1.61 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`.

Related concepts

1.63 Heap1, standard heap implementation on page 1-91.

1.64 Heap2, alternative heap implementation on page 1-92.

1.65 Using a heap implementation from bare machine C on page 1-93.

Related references

1.30 Indirect semihosting C library function dependencies on page 1-52.

1.62 C library support for memory allocation functions on page 1-90.

Related information

--list=filename linker option.

--verbose linker option.

1.62 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.

Related concepts

1.63 Heap1, standard heap implementation on page 1-91.

1.64 Heap2, alternative heap implementation on page 1-92.

1.65 Using a heap implementation from bare machine C on page 1-93.

Related references

1.61 Avoiding the heap and heap-using library functions supplied by ARM on page 1-89.

4.2 alloca() on page 4-192.

1.63 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.

Related concepts

1.64 Heap2, alternative heap implementation on page 1-92.

1.65 Using a heap implementation from bare machine C on page 1-93.

Related references

1.61 Avoiding the heap and heap-using library functions supplied by ARM on page 1-89.

1.62 C library support for memory allocation functions on page 1-90.

1.64 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.

Related concepts

1.63 Heap1, standard heap implementation on page 1-91.

1.65 Using a heap implementation from bare machine C on page 1-93.

Related references

1.61 Avoiding the heap and heap-using library functions supplied by ARM on page 1-89.

1.62 C library support for memory allocation functions on page 1-90.

1.65 Using a heap implementation from bare machine C

When using a heap implementation from bare machine C, you must define the base and top of the heap as well as providing a heap extension function.

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.

Related concepts

1.63 Heap1, standard heap implementation on page 1-91.

1.64 Heap2, alternative heap implementation on page 1-92.

Related references

1.61 Avoiding the heap and heap-using library functions supplied by ARM on page 1-89.

1.62 C library support for memory allocation functions on page 1-90.

1.66 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 `__inital_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 for defining where the stack pointer starts and for 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()`

Related concepts

1.69 Legacy support for __user_initial_stackheap() on page 1-98.

1.67 Defining __initial_sp, __heap_base and __heap_limit on page 1-96.

Related tasks

1.68 Extending heap size at runtime on page 1-97.

Related references

4.52 __user_heap_extend() on page 4-245.

4.53 __user_heap_extent() on page 4-246.

4.61 Legacy function __user_initial_stackheap() on page 4-257.

4.27 __rt_heap_extend() on page 4-219.

4.31 __rt_stackheap_init() on page 4-223.

4.54 __user_setup_stackheap() on page 4-247.

4.55 __vectab_stack_and_reset on page 4-248.

4.54 __user_setup_stackheap() on page 4-247.

Related information

Placing the stack and heap.

Specifying stack and heap using the scatter file.

1.67 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 `__initial_sp`, `__heap_base`, and `__heap_limit` symbols.

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).

Related concepts

1.66 Stack pointer initialization and heap bounds on page 1-94.

Related tasks

1.68 Extending heap size at runtime on page 1-97.

1.68 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.

Related concepts

1.67 Defining __initial_sp, __heap_base and __heap_limit on page 1-96.

1.66 Stack pointer initialization and heap bounds on page 1-94.

Related references

4.27 __rt_heap_extend() on page 4-219.

1.69 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_stackheap()` 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.

Related concepts

1.66 Stack pointer initialization and heap bounds on page 1-94.

Related references

4.61 Legacy function __user_initial_stackheap() on page 4-257.

4.54 __user_setup_stackheap() on page 4-247.

4.54 __user_setup_stackheap() on page 4-247.

1.70 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:

- Avoid the high-level library functions.
- Redefine the low-level library functions.
- Redefine the system I/O 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.

Related concepts

1.47 Library functions called from main() on page 1-71.

1.72 The C library printf family of functions on page 1-102.

1.73 The C library scanf family of functions on page 1-103.

1.74 Redefining low-level library functions to enable direct use of high-level library functions in the C library on page 1-104.

1.75 The C library functions fread(), fgets() and gets() on page 1-106.

1.76 Re-implementing __backspace() in the C library on page 1-107.

1.77 Re-implementing __backspacewc() in the C library on page 1-109.

1.78 Redefining target-dependent system I/O functions in the C library on page 1-110.

Related references

1.29 Direct semihosting C library function dependencies on page 1-51.

1.30 Indirect semihosting C library function dependencies on page 1-52.

1.32 Building an application without the C library on page 1-54.

1.71 Target dependencies on low-level functions in the C and C++ libraries on page 1-100.

4.38 _sys_close() on page 4-231.

4.39 _sys_command_string() on page 4-232.

4.40 _sys_ensure() on page 4-233.

4.42 _sys_flen() on page 4-235.

4.43 _sys_istty() on page 4-236.

4.44 _sys_open() on page 4-237.

4.45 _sys_read() on page 4-238.

4.46 _sys_seek() on page 4-239.

4.47 _sys_tmpnam() on page 4-240.

4.48 _sys_write() on page 4-241.

4.18 #pragma import(_main_redirection) on page 4-209.

4.8 _fisatty() on page 4-198.

4.43 _sys_istty() on page 4-236.

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

Higher-level C and C++ library input/output functions are built upon 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.

The following table shows the dependencies of the higher-level functions on lower-level functions.

`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 1-9 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	-	-	-

Table 1-9 Input/output dependencies (continued)

High-level function	Low-level object									
perror	x	-	x	x	-	-	-	-	-	-
printf	x	-	x	x	x	-	-	-	-	-
putchar	x	-	x	x	-	-	-	-	-	-
puts	x	-	x	x	-	-	-	-	-	-
putwchar	x	-	x	-	-	-	-	x	-	-
scanf	x	x	-	-	-	x	-	-	x	-
vfprintf	x	-	-	x	x	-	-	-	-	-
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.

Related concepts

1.72 The C library printf family of functions on page 1-102.

1.73 The C library scanf family of functions on page 1-103.

1.74 Redefining low-level library functions to enable direct use of high-level library functions in the C library on page 1-104.

1.75 The C library functions fread(), fgets() and gets() on page 1-106.

1.76 Re-implementing __backspace() in the C library on page 1-107.

1.77 Re-implementing __backspacewc() in the C library on page 1-109.

1.78 Redefining target-dependent system I/O functions in the C library on page 1-110.

Related references

1.70 Tailoring input/output functions in the C and C++ libraries on page 1-99.

Related information

ISO C Reference.

1.72 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`.

Related concepts

1.73 The C library scanf family of functions on page 1-103.

1.74 Redefining low-level library functions to enable direct use of high-level library functions in the C library on page 1-104.

1.75 The C library functions fread(), fgets() and gets() on page 1-106.

1.76 Re-implementing __backspace() in the C library on page 1-107.

1.77 Re-implementing __backspacewc() in the C library on page 1-109.

1.78 Redefining target-dependent system I/O functions in the C library on page 1-110.

Related references

1.70 Tailoring input/output functions in the C and C++ libraries on page 1-99.

1.71 Target dependencies on low-level functions in the C and C++ libraries on page 1-100.

1.73 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`.

Related concepts

1.72 The C library printf family of functions on page 1-102.

1.74 Redefining low-level library functions to enable direct use of high-level library functions in the C library on page 1-104.

1.75 The C library functions fread(), fgets() and gets() on page 1-106.

1.76 Re-implementing __backspace() in the C library on page 1-107.

1.77 Re-implementing __backspacewc() in the C library on page 1-109.

1.78 Redefining target-dependent system I/O functions in the C library on page 1-110.

Related references

1.70 Tailoring input/output functions in the C and C++ libraries on page 1-99.

1.71 Target dependencies on low-level functions in the C and C++ libraries on page 1-100.

1.74 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.

These examples 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.

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;
}
```

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.

Related concepts

1.72 The C library printf family of functions on page 1-102.

1.73 The C library scanf family of functions on page 1-103.

1.75 The C library functions fread(), fgets() and gets() on page 1-106.

1.76 Re-implementing __backspace() in the C library on page 1-107.

1.77 Re-implementing __backspacewc() in the C library on page 1-109.

1.78 Redefining target-dependent system I/O functions in the C library on page 1-110.

Related references

1.70 Tailoring input/output functions in the C and C++ libraries on page 1-99.

1.71 Target dependencies on low-level functions in the C and C++ libraries on page 1-100.

1.75 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.

Related concepts

1.72 The C library printf family of functions on page 1-102.

1.73 The C library scanf family of functions on page 1-103.

1.74 Redefining low-level library functions to enable direct use of high-level library functions in the C library on page 1-104.

1.76 Re-implementing __backspace() in the C library on page 1-107.

1.77 Re-implementing __backspacewc() in the C library on page 1-109.

1.78 Redefining target-dependent system I/O functions in the C library on page 1-110.

Related references

1.70 Tailoring input/output functions in the C and C++ libraries on page 1-99.

1.71 Target dependencies on low-level functions in the C and C++ libraries on page 1-100.

1.76 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.

Related concepts

1.72 The C library printf family of functions on page 1-102.

1.73 The C library scanf family of functions on page 1-103.

1.74 Redefining low-level library functions to enable direct use of high-level library functions in the C library on page 1-104.

1.75 The C library functions fread(), fgets() and gets() on page 1-106.

1.77 Re-implementing __backspacewc() in the C library on page 1-109.

1.78 Redefining target-dependent system I/O functions in the C library on page 1-110.

Related references

1.70 Tailoring input/output functions in the C and C++ libraries on page 1-99.

1.71 Target dependencies on low-level functions in the C and C++ libraries on page 1-100.

1.77 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.

Related concepts

1.72 The C library printf family of functions on page 1-102.

1.73 The C library scanf family of functions on page 1-103.

1.74 Redefining low-level library functions to enable direct use of high-level library functions in the C library on page 1-104.

1.75 The C library functions fread(), fgets() and gets() on page 1-106.

1.76 Re-implementing __backspace() in the C library on page 1-107.

1.78 Redefining target-dependent system I/O functions in the C library on page 1-110.

Related references

1.70 Tailoring input/output functions in the C and C++ libraries on page 1-99.

1.71 Target dependencies on low-level functions in the C and C++ libraries on page 1-100.

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

The default target-dependent I/O functions use semihosting. If any of these functions are redefined, then they must all be redefined.

The following example shows you how to redefine the required functions this, for a device that supports writing but not reading.

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 */
}
```

`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.

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.

Related concepts

1.72 The C library printf family of functions on page 1-102.

1.73 The C library scanf family of functions on page 1-103.

1.74 Redefining low-level library functions to enable direct use of high-level library functions in the C library on page 1-104.

1.75 The C library functions fread(), fgets() and gets() on page 1-106.

1.76 Re-implementing __backspace() in the C library on page 1-107.

1.77 Re-implementing __backspacewc() in the C library on page 1-109.

Related references

1.70 Tailoring input/output functions in the C and C++ libraries on page 1-99.

1.71 Target dependencies on low-level functions in the C and C++ libraries on page 1-100.

1.79 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.

Related concepts

1.47 Library functions called from main() on page 1-71.

Related references

1.29 Direct semihosting C library function dependencies on page 1-51.

1.30 Indirect semihosting C library function dependencies on page 1-52.

4.3 clock() on page 4-193.

4.4 _clock_init() on page 4-194.

4.50 time() on page 4-243.

4.21 remove() on page 4-213.

4.22 rename() on page 4-214.

4.49 system() on page 4-242.

4.10 getenv() on page 4-200.

4.11 _getenv_init() on page 4-201.

1.80 Real-time integer division in the ARM libraries

The ARM library provides a real-time division routine and a standard division routine.

The standard 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.

Related concepts

1.81 Selecting real-time division in the ARM libraries on page 1-113.

1.81 Selecting real-time division in the ARM libraries

Real-time division provides a faster worst-case cycle count at the expense of lower average performance.

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.

Related references

1.80 Real-time integer division in the ARM libraries on page 1-112.

1.82 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.

Related concepts

1.85 ISO-compliant C library input/output characteristics on page 1-119.

1.48 Program exit and the assert macro on page 1-72.

Related references

1.83 mathlib error handling on page 1-115.

1.84 ISO-compliant implementation of signals supported by the signal() function in the C library and additional type arguments on page 1-117.

1.86 Standard C++ library implementation definition on page 1-121.

1.92 C and C++ library naming conventions on page 1-127.

1.83 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.

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

Table 1-10 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
`ceil(x)`	`SNaN`	`SNaN`	None
`floor(x)`	`SNaN`	`SNaN`	None

Table 1-10 Mathematical functions in RVCT 4.0 and earlier (continued)

Function	Condition	Returned value	Error number
`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 this table, 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.

Related concepts

1.85 ISO-compliant C library input/output characteristics on page 1-119.

1.82 How the ARM C library fulfills ISO C specification requirements on page 1-114.

Related references

1.84 ISO-compliant implementation of signals supported by the signal() function in the C library and additional type arguments on page 1-117.

1.86 Standard C++ library implementation definition on page 1-121.

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

The `signal()` function supports a number of signals.

The following table 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 1-11 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	Signals 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	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.	-

---

[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.

Table 1-11 Signals supported by the signal() function (continued)

Signal	Number	Description	Additional argument
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

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 the above table, 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.

Related concepts

1.85 ISO-compliant C library input/output characteristics on page 1-119.

1.82 How the ARM C library fulfills ISO C specification requirements on page 1-114.

3.30 Exception types recognized by the ARM floating-point environment on page 3-186.

Related references

1.83 mathlib error handling on page 1-115.

1.86 Standard C++ library implementation definition on page 1-121.

1.60 Modification of C library functions for error signaling, error handling, and program exit on page 1-88.

4.19 __raise() on page 4-210.

4.30 __rt_raise() on page 4-222.

Related information

IEEE Standard for Floating-Point Arithmetic (IEEE 754), 1985 version.

--fpmode=model compiler option.

--exceptions, --no_exceptions compiler option.

---

[c] Not used in RVCT 2.1, and later.

1.85 ISO-compliant C library input/output characteristics

The generic ARM C library has defined input/output characteristics.

These input/output characteristics are as follows:

- 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 the following table.

Table 1-12 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().`

Related concepts

1.82 How the ARM C library fulfills ISO C specification requirements on page 1-114.

Related references

1.83 mathlib error handling on page 1-115.

1.84 ISO-compliant implementation of signals supported by the signal() function in the C library and additional type arguments on page 1-117.

1.86 Standard C++ library implementation definition on page 1-121.

1.86 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.

The following table describes these limitations.

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.

The following table describes the most important features missing from the current release.

Table 1-13 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.

Related information

Rogue Wave documentation.

1.87 C library functions and extensions

The ARM C library is fully compliant with the ISO C99 library standard and provides a number of GNU, POSIX, BSD-derived, and ARM compiler-specific extensions.

The following table describes these extensions.

Table 1-14 C library extensions

Function	Header file definition	Extension
`wcscasecmp()`	`wchar.h`	GNU extension with ARM library support
`wcsncasecmp()`	`wchar.h`	GNU extension with ARM library support
`wcstombs()`	`stdlib.h`	POSIX extended functionality
`posix_memalign()`	`stdlib.h`	POSIX extended functionality
`alloca()`	`alloca.h`	Common nonstandard 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

Related references

4.56 wcscasecmp() on page 4-249.
4.57 wcsncasecmp() on page 4-250.
4.58 wcstombs() on page 4-251.
4.2 alloca() on page 4-192.
4.36 strlcat() on page 4-229.
4.37 strlcpy() on page 4-230.
4.34 strcasecmp() on page 4-227.
4.35 strncasecmp() on page 4-228.
4.8 _fisatty() on page 4-198.
4.12 __heapstats() on page 4-202.
4.13 __heapvalid() on page 4-203.

1.88 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.

Related concepts

1.89 Link time selection of C and C++ libraries on page 1-124.

1.91 Compiler generated and library-resident helper functions on page 1-126.

Related tasks

1.90 Managing projects that have explicit C or C++ library names in makefiles on page 1-125.

Related references

1.92 C and C++ library naming conventions on page 1-127.

1.89 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.

Related concepts

1.91 Compiler generated and library-resident helper functions on page 1-126.

1.88 Persistence of C and C++ library names across releases of the ARM compilation tools on page 1-123.

Related tasks

1.90 Managing projects that have explicit C or C++ library names in makefiles on page 1-125.

Related references

1.92 C and C++ library naming conventions on page 1-127.

1.90 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.

Procedure

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.

Related concepts

1.89 Link time selection of C and C++ libraries on page 1-124.

1.91 Compiler generated and library-resident helper functions on page 1-126.

1.88 Persistence of C and C++ library names across releases of the ARM compilation tools on page 1-123.

Related references

1.92 C and C++ library naming conventions on page 1-127.

Related information

--info=topic[,topic,...] linker option.

1.91 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.

Related concepts

1.89 Link time selection of C and C++ libraries on page 1-124.

1.88 Persistence of C and C++ library names across releases of the ARM compilation tools on page 1-123.

Related tasks

1.90 Managing projects that have explicit C or C++ library names in makefiles on page 1-125.

Related references

1.92 C and C++ library naming conventions on page 1-127.

1.92 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 processors. 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 none of `v`, `m`, or `s` are 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 either:

- Uses position-dependent access to static data. This is the case for the main C libraries with prefixes `c` or `mc`.
- Does not access static data, or does so only with the help of the main C library. This is the case for `fplib` and `mathlib` libraries with prefixes `fz`, `fj`, `f`, `g`, `mf`, or `m`.

`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.

Related concepts

1.89 Link time selection of C and C++ libraries on page 1-124.

1.91 Compiler generated and library-resident helper functions on page 1-126.

1.88 Persistence of C and C++ library names across releases of the ARM compilation tools on page 1-123.

Related tasks

1.90 Managing projects that have explicit C or C++ library names in makefiles on page 1-125.

Related information

--enum_is_int compiler option.

--wchar16 compiler option.

--wchar32 compiler option.

--info=topic[,topic,...] linker option.

--thumb2_library linker option.

1.93 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.

1.94 Using library functions with execute-only memory

The ARM® Compiler lets you build applications for execute-only memory. However, the ARM C and C++ libraries are not execute-only compliant.

If your application calls library functions, the library objects included in the image are not execute-only compliant. You must ensure these objects are not assigned to an execute-only memory region.

Chapter 2
The ARM C Micro-library

Describes the C micro-library (microlib).

It contains the following:

2.1 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()`.

Related concepts

2.2 Differences between microlib and the default C library on page 2-134.

2.3 Library heap usage requirements of the ARM C micro-library on page 2-135.

2.4 ISO C features missing from microlib on page 2-136.

2.5 Building an application with microlib on page 2-138.

2.8 Entering and exiting programs linked with microlib on page 2-141.

Related tasks

2.6 Creating an initial stack pointer for use with microlib on page 2-139.

2.7 Creating the heap for use with microlib on page 2-140.

Related references

2.9 Tailoring the microlib input/output functions on page 2-142.

2.2 Differences between microlib and the default C library

There are a number of differences between microlib and the default C library.

The main differences 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.

Related concepts

2.1 About microlib on page 2-133.

2.3 Library heap usage requirements of the ARM C micro-library on page 2-135.

2.4 ISO C features missing from microlib on page 2-136.

2.5 Building an application with microlib on page 2-138.

2.8 Entering and exiting programs linked with microlib on page 2-141.

Related tasks

2.6 Creating an initial stack pointer for use with microlib on page 2-139.

2.7 Creating the heap for use with microlib on page 2-140.

Related references

2.9 Tailoring the microlib input/output functions on page 2-142.

Related information

--fpmode=model compiler option.

#pragma import(__use_full_stdio).

#pragma import(__use_smaller_memcpy).

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

Library heap usage requirements for microlib differ to those of standardlib.

The differences are:

- 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.

Related concepts

1.4 Library heap usage requirements of the ARM C and C++ libraries on page 1-21.

2.1 About microlib on page 2-133.

2.2 Differences between microlib and the default C library on page 2-134.

2.4 ISO C features missing from microlib on page 2-136.

2.5 Building an application with microlib on page 2-138.

2.8 Entering and exiting programs linked with microlib on page 2-141.

Related tasks

2.6 Creating an initial stack pointer for use with microlib on page 2-139.

2.7 Creating the heap for use with microlib on page 2-140.

Related references

2.9 Tailoring the microlib input/output functions on page 2-142.

2.4 ISO C features missing from microlib

microlib does not support all ISO C90 features.

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.

Related concepts

2.1 About microlib on page 2-133.

2.2 Differences between microlib and the default C library on page 2-134.

2.3 Library heap usage requirements of the ARM C micro-library on page 2-135.

2.5 Building an application with microlib on page 2-138.

2.8 Entering and exiting programs linked with microlib on page 2-141.

Related tasks

2.6 Creating an initial stack pointer for use with microlib on page 2-139.

2.7 Creating the heap for use with microlib on page 2-140.

Related references

2.9 Tailoring the microlib input/output functions on page 2-142.

1.31 C library API definitions for targeting a different environment on page 1-53.

1.32 Building an application without the C library on page 1-54.

4.3 clock() on page 4-193.

4.4 _clock_init() on page 4-194.

Related information

#pragma import(__use_full_stdio).

2.5 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 `--library_type=microlib` with the linker to override all other options.

Compiler option

```
armcc --library_type=microlib -c main.c
armcc -c extra.c
armlink -o image.axf main.o extra.o
```

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`.

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
```

The request to the linker to use microlib is made as a result of assembling `more.s` with `--library_type=microlib`.

Linker option

```
armcc -c main.c
armcc -c extra.c
armlink --library_type=microlib -o image.axf main.o extra.o
```

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.

Related concepts

2.1 About microlib on page 2-133.
2.2 Differences between microlib and the default C library on page 2-134.
2.3 Library heap usage requirements of the ARM C micro-library on page 2-135.
2.4 ISO C features missing from microlib on page 2-136.
2.8 Entering and exiting programs linked with microlib on page 2-141.

Related tasks

2.6 Creating an initial stack pointer for use with microlib on page 2-139.
2.7 Creating the heap for use with microlib on page 2-140.

Related references

2.9 Tailoring the microlib input/output functions on page 2-142.

Related information

--library_type=lib compiler option.
input-file-list linker option.

2.6 Creating an initial stack pointer for use with microlib

To use microlib, you must specify an initial pointer for the stack. You can specify the initial pointer in a scatter file or using the `__initial_sp` symbol.

There are a number of methods you can use to specify an initial pointer for the stack.

Procedure

- Use a scatter file.

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

Examples

To set up the initial stack pointer using assembly language:

```
    EXPORT __initial_sp
__initial_sp EQU 0x100000        ; equal to the top of the stack
```

To set up the initial stack pointer using embedded assembler in C:

```
__asm void dummy_function(void)
{
    EXPORT __initial_sp
__initial_sp EQU 0x100000        ; equal to the top of the stack
}
```

Related concepts

2.1 About microlib on page 2-133.

2.2 Differences between microlib and the default C library on page 2-134.

2.3 Library heap usage requirements of the ARM C micro-library on page 2-135.

2.4 ISO C features missing from microlib on page 2-136.

2.5 Building an application with microlib on page 2-138.

2.8 Entering and exiting programs linked with microlib on page 2-141.

1.67 Defining __initial_sp, __heap_base and __heap_limit on page 1-96.

Related tasks

2.7 Creating the heap for use with microlib on page 2-140.

Related references

2.9 Tailoring the microlib input/output functions on page 2-142.

Related information

About scatter-loading.

2.7 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.

There are a number of methods you can use to specify the start and end of the heap.

Procedure

- Use a scatter file.

  The scatter file method uses `ARM_LIB_HEAP` and `ARM_LIB_STACKHEAP`.
- Define symbols `__heap_base` and `__heap_limit`.

  The `__heap_limit` symbol must point to the byte beyond the last byte in the heap region.

Examples

To set up the heap pointers using 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
```

To set up the heap pointer using 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
}
```

Related concepts

2.1 About microlib on page 2-133.

2.2 Differences between microlib and the default C library on page 2-134.

2.3 Library heap usage requirements of the ARM C micro-library on page 2-135.

2.4 ISO C features missing from microlib on page 2-136.

2.5 Building an application with microlib on page 2-138.

2.8 Entering and exiting programs linked with microlib on page 2-141.

1.67 Defining __initial_sp, __heap_base and __heap_limit on page 1-96.

Related tasks

2.6 Creating an initial stack pointer for use with microlib on page 2-139.

Related references

2.9 Tailoring the microlib input/output functions on page 2-142.

Related information

About scatter-loading.

2.8 Entering and exiting programs linked with microlib

Microlib requires a `main()` function that takes no arguments and never returns.

Use `main()` to begin your program. Do not declare `main()` to take arguments. Microlib does not support command-line arguments from an operating system.

Your program must not return from `main()`. This is because microlib does not contain any code to handle exit from `main()`. Microlib does not support programs that call `exit()`.

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()
}
```

Related concepts

2.1 About microlib on page 2-133.

2.2 Differences between microlib and the default C library on page 2-134.

2.3 Library heap usage requirements of the ARM C micro-library on page 2-135.

2.4 ISO C features missing from microlib on page 2-136.

2.5 Building an application with microlib on page 2-138.

Related tasks

2.6 Creating an initial stack pointer for use with microlib on page 2-139.

2.7 Creating the heap for use with microlib on page 2-140.

Related references

2.9 Tailoring the microlib input/output functions on page 2-142.

2.9 Tailoring the microlib input/output functions

Microlib provides a limited `stdio` subsystem. To use high-level I/O functions you must reimplement the base I/O 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`.

Related concepts

2.1 About microlib on page 2-133.
2.2 Differences between microlib and the default C library on page 2-134.
2.3 Library heap usage requirements of the ARM C micro-library on page 2-135.
2.4 ISO C features missing from microlib on page 2-136.
2.5 Building an application with microlib on page 2-138.
2.8 Entering and exiting programs linked with microlib on page 2-141.

Related tasks

2.6 Creating an initial stack pointer for use with microlib on page 2-139.
2.7 Creating the heap for use with microlib on page 2-140.

Related references

1.70 Tailoring input/output functions in the C and C++ libraries on page 1-99.

Chapter 3

Floating-point Support

Describes ARM support for floating-point computations.

It contains the following:

3.1 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.

Related concepts

3.2 The software floating-point library, fplib on page 3-146.

3.22 IEEE 754 arithmetic on page 3-175.

Related information

--fpu=name compiler option.

3.2 The software floating-point library, fplib

The software floating-point library, fplib, provides software implementations of floating-point operations.

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.

Related concepts

3.3 Calling fplib routines on page 3-147.

Related references

3.4 fplib arithmetic on numbers in a particular format on page 3-148.

3.5 fplib conversions between floats, long longs, doubles, and ints on page 3-150.

3.6 fplib comparisons between floats and doubles on page 3-152.

3.7 fplib C99 functions on page 3-154.

3.3 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.

Software floating-point library routines are declared in one of two header files:

- A small number of fplib routines that implement C99 functionality are declared in the standard header file `math.h`.
- All other 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.

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

```
IMPORT nextafter
BL nextafter
```

Related concepts

3.2 The software floating-point library, fplib on page 3-146.

Related references

3.4 fplib arithmetic on numbers in a particular format on page 3-148.

3.5 fplib conversions between floats, long longs, doubles, and ints on page 3-150.

3.6 fplib comparisons between floats and doubles on page 3-152.

3.7 fplib C99 functions on page 3-154.

Related information

Application Binary Interface (ABI) for the ARM Architecture.

Compiler support for floating-point computations and linkage.

3.4 fplib arithmetic on numbers in a particular format

fplib provides a number of routines to perform arithmetic on numbers in a particular format.

The following table describes these routines. Arguments and return types are always in the same format.

Table 3-1 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 note a)
`_frnd`	**`float`**	**`float`**	Return x rounded to an integer (see note b)
`_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 notes a and c)
`_drnd`	**`double`**	**`double`**	Return x rounded to an integer (see note b)
`_dsqrt`	**`double`**	**`double`**	Return square root of x

Notes on arithmetic routines

a

Functions that perform the IEEE 754 remainder operation. This is defined to take two numbers, x and y, and return a number z so that $z = x - n * y$, where n is an integer. To return an exactly correct result, n is chosen so that z is no bigger than half of x (so that z might be negative even if both x and y are positive). The IEEE 754 remainder function is not the same as the operation performed by the C library function `fmod`, where z always has the same sign as x. Where the IEEE 754 specification gives two acceptable choices of n, the even one is chosen. This behavior occurs independently of the current rounding mode.

b

Functions that perform the IEEE 754 round-to-integer operation. This takes a number and rounds it to an integer (in accordance with the current rounding mode), but returns that integer in the floating-point number format rather than as a C **`int`** variable. To convert a number to an **`int`** variable, you must use the `_ffix` routines.

c

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

Related concepts

3.2 The software floating-point library, fplib on page 3-146.

3.3 Calling fplib routines on page 3-147.

Related references

3.5 fplib conversions between floats, long longs, doubles, and ints on page 3-150.

3.6 fplib comparisons between floats and doubles on page 3-152.

3.7 fplib C99 functions on page 3-154.

Related information

Application Binary Interface (ABI) for the ARM Architecture.

3.5 fplib conversions between floats, long longs, doubles, and ints

fplib provides a number of routines to perform conversions between number formats.

The following table describes these routines.

Table 3-2 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 note a)
`_ffix_r`	**`float`**	**`int`**
`_ffixu`	**`float`**	**`unsigned int`** (see note a)
`_ffixu_r`	**`float`**	**`unsigned int`**
`_dfix`	**`double`**	**`int`** (see note a)
`_dfix_r`	**`double`**	**`int`**
`_dfixu`	**`double`**	**`unsigned int`** (see note a)
`_dfixu_r`	**`double`**	**`unsigned int`**
`_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 note a)
`_ll_sfrom_f_r`	**`float`**	**`long long`**
`_ll_ufrom_f`	**`float`**	**`unsigned long long`** (see note a)
`_ll_ufrom_f_r`	**`float`**	**`unsigned long long`**
`_ll_sfrom_d`	**`double`**	**`long long`** (see note a)
`_ll_sfrom_d_r`	**`double`**	**`long long`**
`_ll_ufrom_d`	**`double`**	**`unsigned long long`** (see note a)
`_ll_ufrom_d_r`	**`double`**	**`unsigned long long`**

Notes on rounding

a

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.

Related concepts

3.2 The software floating-point library, fplib on page 3-146.

3.3 Calling fplib routines on page 3-147.

Related references

3.4 fplib arithmetic on numbers in a particular format on page 3-148.

3.6 fplib comparisons between floats and doubles on page 3-152.

3.7 fplib C99 functions on page 3-154.

Related information

Application Binary Interface (ABI) for the ARM Architecture.

3.6 fplib comparisons between floats and doubles

fplib provides a number of routines to perform comparisons between floating-point numbers.

The following table describes these routines.

Table 3-3 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

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`.

Related concepts

3.2 The software floating-point library, fplib on page 3-146.

3.3 Calling fplib routines on page 3-147.

Related references

3.4 fplib arithmetic on numbers in a particular format on page 3-148.

3.5 fplib conversions between floats, long longs, doubles, and ints on page 3-150.

3.7 fplib C99 functions on page 3-154.

3.7 fplib C99 functions

fplib provides a number of routines that implement C99 functionality.

The following table describes these functions.

Table 3-4 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

Related concepts

3.2 The software floating-point library, fplib on page 3-146.

3.3 Calling fplib routines on page 3-147.

Related references

3.4 fplib arithmetic on numbers in a particular format on page 3-148.

3.5 fplib conversions between floats, long longs, doubles, and ints on page 3-150.

3.6 fplib comparisons between floats and doubles on page 3-152.

3.8 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.

Related concepts

3.9 Floating-point functions for compatibility with Microsoft products on page 3-156.

3.17 Writing a custom exception trap handler on page 3-164.

3.18 Example of a custom exception handler on page 3-170.

3.19 Exception trap handling by signals on page 3-172.

3.29 Exceptions arising from IEEE 754 floating-point arithmetic on page 3-185.

Related references

3.10 C99-compatible functions for controlling the ARM floating-point environment on page 3-157.

3.16 ARM floating-point compiler extensions to the C99 interface on page 3-163.

5.5 __ieee_status() on page 5-266.

5.3 __fp_status() on page 5-263.

3.9 Floating-point functions for compatibility with Microsoft products

Functions defined in `float.h` give compatibility with Microsoft products to ease porting of floating-point code to the ARM architecture.

These functions require you to select a floating-point model that supports exceptions. For example, `--fpmode=ieee_full` or `--fpmode=ieee_fixed`.

Related concepts

3.8 Controlling the ARM floating-point environment on page 3-155.

Related references

5.1 _clearfp() on page 5-260.

5.2 _controlfp() on page 5-261.

5.8 _statusfp() on page 5-271.

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

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`.

Note

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

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.

Related concepts

3.8 Controlling the ARM floating-point environment on page 3-155.

3.12 Exception flag handling on page 3-159.

3.13 Functions for handling rounding modes on page 3-160.

3.14 Functions for saving and restoring the whole floating-point environment on page 3-161.

3.15 Functions for temporarily disabling exceptions on page 3-162.

3.17 Writing a custom exception trap handler on page 3-164.

3.18 Example of a custom exception handler on page 3-170.

3.19 Exception trap handling by signals on page 3-172.

3.29 Exceptions arising from IEEE 754 floating-point arithmetic on page 3-185.

Related references

3.11 C99 rounding mode and floating-point exception macros on page 3-158.

3.16 ARM floating-point compiler extensions to the C99 interface on page 3-163.

Related information

--fpmode=model compiler option.

3.11 C99 rounding mode and floating-point exception macros

C99 defines a macro for each rounding mode and each exception

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 C99 rounding mode and exception 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.

Related concepts

3.13 Functions for handling rounding modes on page 3-160.

Related references

3.10 C99-compatible functions for controlling the ARM floating-point environment on page 3-157.

Related information

--fpmode=model compiler option.

3.12 Exception flag handling

The `feclearexcept()`, `fetestexcept()`, and `feraiseexcept()` functions let you clear, test and raise exceptions. The `fegetexceptflag()` and `fesetexceptflag()` functions let you save and restore information about a given exception.

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 these functions as follows:

```
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 all information 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

`You can use fesetexceptflag()` 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.

Related references

3.10 C99-compatible functions for controlling the ARM floating-point environment on page 3-157.

Related information

--fpmode=model compiler option.

3.13 Functions for handling rounding modes

The `fegetround()` and `fesetround` functions let you get and set the current rounding mode.

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 these functions as follows:

```
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.

Related references

3.10 C99-compatible functions for controlling the ARM floating-point environment on page 3-157.

3.11 C99 rounding mode and floating-point exception macros on page 3-158.

Related information

--fpmode=model compiler option.

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

The `fegetenv` and `fesetenv` functions let you save and restore the entire 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 defines these functions as follows:

```
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.

Related references

3.10 C99-compatible functions for controlling the ARM floating-point environment on page 3-157.

Related information

--fpmode=model compiler option.

3.15 Functions for temporarily disabling exceptions

The `feholdexcept` and `feupdateenv` functions let you temporarily disable exception trapping.

Note

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

These functions let you 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.

Related references

3.10 C99-compatible functions for controlling the ARM floating-point environment on page 3-157.

5.5 __ieee_status() on page 5-266.

Related information

--fpmode=model compiler option.

3.16 ARM floating-point compiler extensions to the C99 interface

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.

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 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.

Related concepts

3.8 Controlling the ARM floating-point environment on page 3-155.

3.17 Writing a custom exception trap handler on page 3-164.

3.18 Example of a custom exception handler on page 3-170.

3.17 Writing a custom exception trap handler on page 3-164.

3.18 Example of a custom exception handler on page 3-170.

3.19 Exception trap handling by signals on page 3-172.

3.29 Exceptions arising from IEEE 754 floating-point arithmetic on page 3-185.

Related references

3.10 C99-compatible functions for controlling the ARM floating-point environment on page 3-157.

5.5 __ieee_status() on page 5-266.

Related information

--fpmode=model compiler option.

3.17 Writing a custom exception trap handler

Custom exception trap handlers let you override the default exception handling behavior. For example, when converting Fortran code you might want to override the division by zero exception to return 1 rather than an invalid operation exception.

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 value returned from this function is of type `__ieee_value_t` and is used as the result of the operation that caused the exception.

The function must be declared `__softfp` in order to be usable as a handler.

The parameters to this function are:

`op1, op2`

These specify 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`**.)

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 The following table.

Table 3-5 FE_EX_INTYPE_MASK operand type flags

Flag	Operand type
`FE_EX_INTYPE_FLOAT`	**`float`**
`FE_EX_INTYPE_DOUBLE`	**`double`**
`FE_EX_INTYPE_FD`	**`float double`**
`FE_EX_INTYPE_DF`	**`double float`**
`FE_EX_INTYPE_HALF`	**`short`**
`FE_EX_INTYPE_INT`	**`int`**
`FE_EX_INTYPE_UINT`	**`unsigned int`**
`FE_EX_INTYPE_LONGLONG`	**`long long`**
`FE_EX_INTYPE_ULONGLONG`	**`unsigned long long`**

`edata & FE_EX_OUTTYPE_`*`MASK`*

This gives the type of the operands to the function, as one of the type values shown in The following table.

Table 3-6 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 the following table.

Table 3-7 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.
`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 the following table.

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

Table 3-8 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

Related concepts

3.18 Example of a custom exception handler on page 3-170.

3.19 Exception trap handling by signals on page 3-172.

3.29 Exceptions arising from IEEE 754 floating-point arithmetic on page 3-185.

3.8 Controlling the ARM floating-point environment on page 3-155.

Related references

3.16 ARM floating-point compiler extensions to the C99 interface on page 3-163.

3.16 ARM floating-point compiler extensions to the C99 interface on page 3-163.

3.10 C99-compatible functions for controlling the ARM floating-point environment on page 3-157.

5.5 __ieee_status() on page 5-266.

Related information

--fpmode=model compiler option.

--strict, --no_strict compiler option.

3.18 Example of a custom exception handler

This example exception trap handler overrides the division by zero exception to return 1 rather than an invalid operation exception.

Note

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

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.

After the handler is installed, dividing 0.0 by 0.0 returns 1.0.

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);
}
```

Related concepts

3.17 Writing a custom exception trap handler on page 3-164.

3.19 Exception trap handling by signals on page 3-172.

3.29 Exceptions arising from IEEE 754 floating-point arithmetic on page 3-185.

3.8 Controlling the ARM floating-point environment on page 3-155.

Related references

3.16 ARM floating-point compiler extensions to the C99 interface on page 3-163.

3.16 ARM floating-point compiler extensions to the C99 interface on page 3-163.

3.10 C99-compatible functions for controlling the ARM floating-point environment on page 3-157.

Related information

--fpmode=model compiler option.

3.19 Exception trap handling by signals

You can use the `SIGFPE` signal to handle 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`.

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`.

Related concepts

3.17 Writing a custom exception trap handler on page 3-164.

3.18 Example of a custom exception handler on page 3-170.

3.29 Exceptions arising from IEEE 754 floating-point arithmetic on page 3-185.

3.8 Controlling the ARM floating-point environment on page 3-155.

Related references

3.16 ARM floating-point compiler extensions to the C99 interface on page 3-163.

3.10 C99-compatible functions for controlling the ARM floating-point environment on page 3-157.

4.30 __rt_raise() on page 4-222.

Related information

--fpmode=model compiler option.

3.20 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__`.

Related information

WG14 - C N965, Optional support for Signaling NaNs.

Predefined macros.

3.21 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++.

3.22 IEEE 754 arithmetic

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

Related concepts

3.24 Single precision data type for IEEE 754 arithmetic on page 3-177.

3.25 Double precision data type for IEEE 754 arithmetic on page 3-179.

3.28 IEEE 754 arithmetic and rounding on page 3-184.

3.29 Exceptions arising from IEEE 754 floating-point arithmetic on page 3-185.

Related references

3.23 Basic data types for IEEE 754 arithmetic on page 3-176.

3.26 Sample single precision floating-point values for IEEE 754 arithmetic on page 3-180.

3.27 Sample double precision floating-point values for IEEE 754 arithmetic on page 3-182.

3.23 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.

Related concepts

3.22 IEEE 754 arithmetic on page 3-175.

3.24 Single precision data type for IEEE 754 arithmetic on page 3-177.

3.25 Double precision data type for IEEE 754 arithmetic on page 3-179.

3.28 IEEE 754 arithmetic and rounding on page 3-184.

3.29 Exceptions arising from IEEE 754 floating-point arithmetic on page 3-185.

Related references

3.26 Sample single precision floating-point values for IEEE 754 arithmetic on page 3-180.

3.27 Sample double precision floating-point values for IEEE 754 arithmetic on page 3-182.

Related information

IEEE Standard for Floating-Point Arithmetic (IEEE 754), 1985 version.

3.24 Single precision data type for IEEE 754 arithmetic

A **`float`** value is 32 bits wide.

The structure is:

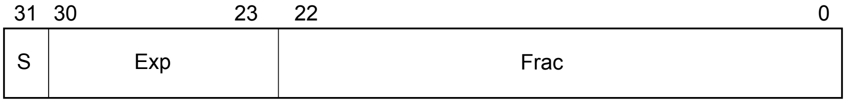

Figure 3-1 IEEE 754 single-precision floating-point format

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 can 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 can 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*.

Related concepts

3.22 IEEE 754 arithmetic on page 3-175.

3.25 Double precision data type for IEEE 754 arithmetic on page 3-179.

3.28 IEEE 754 arithmetic and rounding on page 3-184.

3.29 Exceptions arising from IEEE 754 floating-point arithmetic on page 3-185.

Related references

3.23 Basic data types for IEEE 754 arithmetic on page 3-176.

3.26 Sample single precision floating-point values for IEEE 754 arithmetic on page 3-180.

3.27 Sample double precision floating-point values for IEEE 754 arithmetic on page 3-182.

Related information

IEEE Standard for Floating-Point Arithmetic (IEEE 754), 1985 version.

3.25 Double precision data type for IEEE 754 arithmetic

A **`double`** value is 64 bits wide.

The structure is:

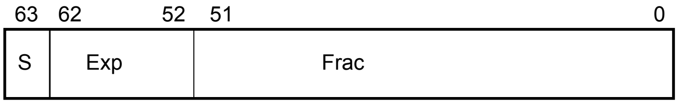

Figure 3-2 IEEE 754 double-precision floating-point format

As with single-precision `float` data types, `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 representing infinity and NaNs is `0x7FF` (2047) instead of `0xFF`.

Related concepts

3.22 IEEE 754 arithmetic on page 3-175.

3.24 Single precision data type for IEEE 754 arithmetic on page 3-177.

3.28 IEEE 754 arithmetic and rounding on page 3-184.

3.29 Exceptions arising from IEEE 754 floating-point arithmetic on page 3-185.

Related references

3.23 Basic data types for IEEE 754 arithmetic on page 3-176.

3.26 Sample single precision floating-point values for IEEE 754 arithmetic on page 3-180.

3.27 Sample double precision floating-point values for IEEE 754 arithmetic on page 3-182.

Related information

IEEE Standard for Floating-Point Arithmetic (IEEE 754), 1985 version.

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

Sample `float` bit patterns, together with their mathematical values.

Table 3-9 Sample single-precision floating-point values

Float value	S	Exp	Frac	Mathematical value	Notes
`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

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.

Related concepts

3.22 IEEE 754 arithmetic on page 3-175.

3.24 Single precision data type for IEEE 754 arithmetic on page 3-177.

3.25 Double precision data type for IEEE 754 arithmetic on page 3-179.

3.28 IEEE 754 arithmetic and rounding on page 3-184.

3.29 Exceptions arising from IEEE 754 floating-point arithmetic on page 3-185.

Related references

3.23 Basic data types for IEEE 754 arithmetic on page 3-176.

3.27 Sample double precision floating-point values for IEEE 754 arithmetic on page 3-182.

Related information

IEEE Standard for Floating-Point Arithmetic (IEEE 754), 1985 version.

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

Sample **`double`** bit patterns, together with their mathematical values.

Table 3-10 Sample double-precision floating-point values

Double value	S	Exp	Frac	Mathematical value	Notes
`0x3FF0000000000000`	`0`	`0x3FF`	`000...000`	`1.0`	-
`0xBFF0000000000000`	`1`	`0x3FF`	`000...000`	`-1.0`	-
`0x3FF0000000000001`	`0`	`0x3FF`	`000...001`	`1.000 000 000 000 000 222`	a
`0x3FE8000000000000`	`0`	`0x3FE`	`100...000`	`0.75`	-
`0x0010000000000000`	`0`	`0x001`	`000...000`	$2.23*10^{-308}$	b
`0x0000000000000001`	`0`	`0x000`	`000...001`	$4.94*10^{-324}$	c
`0x7FEFFFFFFFFFFFFF`	`0`	`0x7FE`	`111...111`	$1.80*10^{308}$	d
`0x7FF0000000000000`	`0`	`0x7FF`	`000...000`	Plus infinity	-
`0xFFF0000000000000`	`1`	`0x7FF`	`000...000`	Minus infinity	-
`0x0000000000000000`	`0`	`0x000`	`000...000`	`0.0`	e
`0x7FF0000000000001`	`0`	`0x7FF`	`000...001`	Signaling NaN	f
`0x7FF8000000000000`	`0`	`0x7FF`	`100...000`	Quiet NaN	f

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.

Related concepts

3.22 IEEE 754 arithmetic on page 3-175.

3.24 Single precision data type for IEEE 754 arithmetic on page 3-177.

3.25 Double precision data type for IEEE 754 arithmetic on page 3-179.

3.28 IEEE 754 arithmetic and rounding on page 3-184.

3.29 Exceptions arising from IEEE 754 floating-point arithmetic on page 3-185.

Related references

3.23 Basic data types for IEEE 754 arithmetic on page 3-176.

3.26 Sample single precision floating-point values for IEEE 754 arithmetic on page 3-180.

Related information

IEEE Standard for Floating-Point Arithmetic (IEEE 754), 1985 version.

3.28 IEEE 754 arithmetic and rounding

IEEE 754 defines different rounding rules to use when calculating arithmetic results.

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.

Related concepts

3.22 IEEE 754 arithmetic on page 3-175.

3.24 Single precision data type for IEEE 754 arithmetic on page 3-177.

3.25 Double precision data type for IEEE 754 arithmetic on page 3-179.

3.29 Exceptions arising from IEEE 754 floating-point arithmetic on page 3-185.

Related references

3.23 Basic data types for IEEE 754 arithmetic on page 3-176.

3.26 Sample single precision floating-point values for IEEE 754 arithmetic on page 3-180.

3.27 Sample double precision floating-point values for IEEE 754 arithmetic on page 3-182.

Related information

IEEE Standard for Floating-Point Arithmetic (IEEE 754), 1985 version.

3.29 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.

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.

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.

Related concepts

3.17 Writing a custom exception trap handler on page 3-164.

3.18 Example of a custom exception handler on page 3-170.

3.19 Exception trap handling by signals on page 3-172.

3.8 Controlling the ARM floating-point environment on page 3-155.

3.22 IEEE 754 arithmetic on page 3-175.

3.24 Single precision data type for IEEE 754 arithmetic on page 3-177.

3.25 Double precision data type for IEEE 754 arithmetic on page 3-179.

3.28 IEEE 754 arithmetic and rounding on page 3-184.

Related references

3.16 ARM floating-point compiler extensions to the C99 interface on page 3-163.

3.10 C99-compatible functions for controlling the ARM floating-point environment on page 3-157.

3.23 Basic data types for IEEE 754 arithmetic on page 3-176.

3.26 Sample single precision floating-point values for IEEE 754 arithmetic on page 3-180.

3.27 Sample double precision floating-point values for IEEE 754 arithmetic on page 3-182.

Related information

IEEE Standard for Floating-Point Arithmetic (IEEE 754), 1985 version.

3.30 Exception types recognized by the ARM floating-point environment

The ARM floating-point environment recognizes a number of different 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.

Related concepts

3.17 Writing a custom exception trap handler on page 3-164.

3.12 Exception flag handling on page 3-159.

3.18 Example of a custom exception handler on page 3-170.

3.19 Exception trap handling by signals on page 3-172.

3.22 IEEE 754 arithmetic on page 3-175.

3.29 Exceptions arising from IEEE 754 floating-point arithmetic on page 3-185.

Related references

1.84 ISO-compliant implementation of signals supported by the signal() function in the C library and additional type arguments on page 1-117.

3.26 Sample single precision floating-point values for IEEE 754 arithmetic on page 3-180.

Related information

IEEE Standard for Floating-Point Arithmetic (IEEE 754), 1985 version.

3.31 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.

Related information

Using VFP with RVDS, Application Note 133.

Limitations on hardware handling of floating-point arithmetic.

Implementation of Vector Floating-Point (VFP) support code.

Chapter 4
The C and C++ Library Functions

Documents standard C and C++ library functions that are extensions to the C Standard or that differ in some way to the standard.

Some of the standard functions interact with the ARM retargetable semihosting environment. Such functions are also documented.

It contains the following:

4.1 __aeabi_errno_addr()

The `__aeabi_errno_addr()` returns the address of the C library `errno` variable when the C library attempts to read or write `errno`.

The library provides a default implementation. It is unlikely that you have to re-implement this function.

This function is not part of the C library standard, but the ARM C library supports it as an extension.

Syntax

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

Related references

4.6 errno on page 4-196.

4.24 __rt_errno_addr() on page 4-216.

Related information

C Library ABI for the ARM Architecture.

4.2 alloca()

Defined in `alloca.h`, the `alloca()` function allocates local storage in a function. It returns a pointer to *`size`* bytes of memory.

The default implementation returns an eight-byte aligned block of memory on the stack.

Memory returned from `alloca()` must never be passed to `free()`. Instead, the memory is de-allocated automatically when the function that called `alloca()` returns.

Note

`alloca()` must not be called through a function pointer. You must take care when using `alloca()` and `setjmp()` in the same function, because memory allocated by `alloca()` between calling `setjmp()` and `longjmp()` is de-allocated by the call to `longjmp()`.

This function is a common nonstandard extension to many C libraries.

Syntax

```
void *alloca(size_t size);
```

Related concepts

1.13 ARM C libraries and thread-safe functions on page 1-30.

Related references

4.59 Thread-safe C library functions on page 4-252.

1.32 Building an application without the C library on page 1-54.

4.3 clock()

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

Syntax

```
clock_t clock(void);
```

Usage

The default implementation of this function uses semihosting.

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

Note

If you re-implement `clock()` you must also re-implement `_clock_init()`.

Returns

The returned value is an unsigned integer.

Related references

1.29 Direct semihosting C library function dependencies on page 1-51.

4.4 _clock_init()

Defined in `rt_misc.h`, the `_clock_init()` function is an initialization function for `clock()`.

It is not part of the C library standard, but the ARM C library supports it as an extension.

Syntax

```
void _clock_init(void);
```

Usage

This is a function that you can re-implement in an implementation-specific way. It is called from the library initialization code, so you do not have to call it from your application code.

Note

You must re-implement this function if you re-implement `clock()`.

The initialization that `_clock_init()` applies enables `clock()` to return the time that has elapsed since the program was started.

An example of how you might re-implement `_clock_init()` might be to set the timer to zero. However, if your implementation of `clock()` relies on a system timer that cannot be reset, then `_clock_init()` could instead read the time at startup (when called from the library initialization code), with `clock()` subsequently subtracting the time that was read at initialization, from the current value of the timer. In both cases, some form of initialization is required of `_clock_init()`.

Related references

1.29 Direct semihosting C library function dependencies on page 1-51.

4.5 __default_signal_handler()

Defined in `rt_misc.h`, the `__default_signal_handler()` function handles a raised signal. The default action is to print an error message and exit.

This function is not part of the C library standard, but the ARM C library supports it as an extension.

Syntax

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

Usage

The default signal handler returns a nonzero value to indicate that the caller has to arrange for the program to exit. You can replace the default signal handler by defining:

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

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

A complete list of the defined signals is in `signal.h`.

Note

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

Related references

1.30 Indirect semihosting C library function dependencies on page 1-52.

4.19 __raise() on page 4-210.

4.51 _ttywrch() on page 4-244.

4.41 _sys_exit() on page 4-234.

4.30 __rt_raise() on page 4-222.

1.84 ISO-compliant implementation of signals supported by the signal() function in the C library and additional type arguments on page 1-117.

4.6 errno

The C library `errno` variable is defined in the implicit static data area of the library.

This area is identified by `__user_libspace()`. The function that returns the address of `errno` is:

```
(*(volatile int *) __aeabi_errno_addr())
```

You can define `__aeabi_errno_addr()` if you want to place `errno` at a user-defined location instead of the default location identified by `__user_libspace()`.

Note

Legacy versions of `errno.h` might define `errno` in terms of `__rt_errno_addr()` rather than `__aeabi_errno_addr()`. The function name `__rt_errno_addr()` is a legacy from pre-ABI versions of the tools, and is still supported to ensure that object files generated with those tools link successfully.

Returns

The return value is a pointer to a variable of type **`int`**, containing the currently applicable instance of `errno`.

Related concepts

1.14 Use of static data in the C libraries on page 1-31.

Related references

4.1 __aeabi_errno_addr() on page 4-191.

4.24 __rt_errno_addr() on page 4-216.

1.15 Use of the __user_libspace static data area by the C libraries on page 1-33.

Related information

Application Binary Interface for the ARM Architecture.

4.7 _findlocale()

Defined in `rt_locale.h`, `_findlocale()` searches a set of contiguous locale data blocks for the requested locale, and returns a pointer to that locale.

This function is not part of the C library standard, but the ARM C library supports it as an extension.

Syntax

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

Where:

`index`

is a pointer to a set of locale data blocks that are contiguous in memory and that end with a terminating value (set by the `LC_index_end` macro).

`name`

is the name of the locale to find.

Usage

You can use `_findlocale()` as an optional helper function when defining your own locale setup.

The `_get_lc_*()` functions, for example, `_get_lc_ctype()`, are expected to return a pointer to a locale definition created using the assembler macros. If you only want to write one locale definition, you can write an implementation of `_get_lc_ctype()` that always returns the same pointer. However, if you want to use different locale definitions at runtime, then the `_get_lc_*()` functions have to be able to return a different data block depending on the name passed to them as an argument. `_findlocale()` provides an easy way to do this.

Returns

Returns a pointer to the requested data block.

Related concepts

1.49 Assembler macros that tailor locale functions in the C library on page 1-73.

1.49 Assembler macros that tailor locale functions in the C library on page 1-73.

1.53 Runtime selection of the locale subsystem in the C library on page 1-77.

Related references

1.50 Link time selection of the locale subsystem in the C library on page 1-74.

1.54 Definition of locale data blocks in the C library on page 1-78.

4.14 lconv structure on page 4-204.

4.9 _get_lconv() on page 4-199.

4.15 localeconv() on page 4-206.

4.32 setlocale() on page 4-224.

4.8 _fisatty()

Defined in `stdio.h`, the `_fisatty()` function determines whether the given `stdio` stream is attached to a terminal device or a normal file.

It calls the `_sys_istty()` low-level function on the underlying file handle.

This function is not part of the C library standard, but the ARM C library supports it as an extension.

Syntax

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

The return value indicates the stream destination:

0
A file.

1
A terminal.

Negative
An error.

Related references

4.43 _sys_istty() on page 4-236.

4.9 _get_lconv()

Defined in `locale.h`, `_get_lconv()` performs the same function as the standard C library function, `localeconv()`, except that it delivers the result in user-provided memory instead of an internal static variable.

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

Syntax

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

Usage

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

Returns

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

Related concepts

1.49 Assembler macros that tailor locale functions in the C library on page 1-73.

Related references

4.7 _findlocale() on page 4-197.

4.14 lconv structure on page 4-204.

4.15 localeconv() on page 4-206.

4.32 setlocale() on page 4-224.

4.10 getenv()

This is the standard C library `getenv()` function from `stdlib.h`. It gets the value of a specified environment variable.

Syntax

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

Usage

The default implementation returns `NULL`, indicating that no environment information is available.

If you re-implement `getenv()`, ARM recommends that you re-implement it in such a way that it searches some form of environment list for the input string, *`name`*. The set of environment names and the method for altering the environment list are implementation-defined. `getenv()` does not depend on any other function, and no other function depends on `getenv()`.

A function closely associated with `getenv()` is `_getenv_init()`. `_getenv_init()` is called during startup if it is defined, to enable a user re-implementation of `getenv()` to initialize itself.

Returns

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

4.11 _getenv_init()

Defined in `rt_misc.h`, the `_getenv_init()` function enables a user version of `getenv()` to initialize itself.

It is not part of the C library standard, but the ARM C library supports it as an extension.

Syntax

```
void _getenv_init(void);
```

Usage

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

4.12 __heapstats()

Defined in `stdlib.h`, the `__heapstats()` function displays statistics on the state of the storage allocation heap.

The default implementation in the compiler gives information on how many free blocks exist, and estimates their size ranges.

The `__heapstats()` function generates output as follows:

```
32272 bytes in 2 free blocks (avge size 16136)
1 blocks 2^12+1 to 2^13
1 blocks 2^13+1 to 2^14
```

Line 1 of the output displays the total number of bytes, the number of free blocks, and the average size. The following lines give an estimate of the size of each block in bytes, expressed as a range. `__heapstats()` does not give information on the number of used blocks.

The function outputs its results by calling the output function *`dprint()`*, that must work like `fprintf()`. The first parameter passed to *`dprint()`* is the supplied pointer *`param`*. You can pass `fprintf()` itself, provided you cast it to the right function pointer type. This type is defined as a **`typedef`** for convenience. It is called `__heapprt`. For example:

```
__heapstats((__heapprt)fprintf, stderr);
```

Note

If you call `fprintf()` on a stream that you have not already sent output to, the library calls `malloc()` internally to create a buffer for the stream. If this happens in the middle of a call to `__heapstats()`, the heap might be corrupted. Therefore, you must ensure you have already sent some output to `stderr`.

If you are using the default one-region memory model, heap memory is allocated only as it is required. This means that the amount of free heap changes as you allocate and deallocate memory. For example, the sequence:

```
int *ip;
__heapstats((__heapprt)fprintf,stderr);   // print initial free heap size
ip = malloc(200000);
free(ip);
__heapstats((__heapprt)fprintf,stderr);    // print heap size after freeing
```

gives output such as:

```
4076 bytes in 1 free blocks (avge size 4076)
1 blocks 2^10+1 to 2^11
2008180 bytes in 1 free blocks (avge size 2008180)
1 blocks 2^19+1 to 2^20
```

This function is not part of the C library standard, but the ARM C library supports it as an extension.

Syntax

```
void __heapstats(int (*dprint)(void *param, char const *format,...), void
*param);
```

4.13 __heapvalid()

Defined in `stdlib.h`, the `__heapvalid()` function performs a consistency check on the heap.

It outputs full information about every free block if the *`verbose`* parameter is nonzero. Otherwise, it only outputs errors.

The function outputs its results by calling the output function *`dprint()`*, that must work like `fprintf()`. The first parameter passed to *`dprint()`* is the supplied pointer *`param`*. You can pass `fprintf()` itself, provided you cast it to the right function pointer type. This type is defined as a **`typedef`** for convenience. It is called `__heapprt`. For example:

```
__heapvalid((__heapprt) fprintf, stderr, 0);
```

Note

If you call `fprintf()` on a stream that you have not already sent output to, the library calls `malloc()` internally to create a buffer for the stream. If this happens in the middle of a call to `__heapvalid()`, the heap might be corrupted. You must therefore ensure you have already sent some output to `stderr`. The example code fails if you have not already written to the stream.

This function is not part of the C library standard, but the ARM C library supports it as an extension.

Syntax

```
int __heapvalid(int (*dprint)(void *param, char const *format,...), void
*param, int verbose);
```

4.14 lconv structure

Defined in `locale.h`, the `lconv` structure contains numeric formatting information

The structure is filled by the functions `_get_lconv()` and `localeconv()`.

The definition of `lconv` from `locale.h` is as follows.

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

The elements of `grouping` and `mon_grouping` are interpreted as follows:

`CHAR_MAX`

No additional grouping is to be performed.

`0`

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

`other`

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

The value of `p_sign_posn` and `n_sign_posn` are interpreted as follows:

`0`

Parentheses surround the quantity and currency symbol.

`1`

The sign string precedes the quantity and currency symbol.

`2`

The sign string is after the quantity and currency symbol.

`3`

The sign string immediately precedes the currency symbol.

`4`

The sign string immediately succeeds the currency symbol.

Related concepts

1.49 Assembler macros that tailor locale functions in the C library on page 1-73.

Related references

4.7 _findlocale() on page 4-197.
4.9 _get_lconv() on page 4-199.
4.15 localeconv() on page 4-206.
4.32 setlocale() on page 4-224.

4.15 localeconv()

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

Syntax

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

Usage

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

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

This function is not thread-safe, because it uses an internal static buffer. `_get_lconv()` provides a thread-safe alternative.

Returns

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

Related concepts

1.49 Assembler macros that tailor locale functions in the C library on page 1-73.

Related references

4.7 _findlocale() on page 4-197.

4.14 lconv structure on page 4-204.

4.9 _get_lconv() on page 4-199.

4.32 setlocale() on page 4-224.

4.16 _membitcpybl(), _membitcpybb(), _membitcpyhl(), _membitcpyhb(), _membitcpywl(), _membitcpywb(), _membitmovebl(), _membitmovebb(), _membitmovehl(), _membitmovehb(), _membitmovewl(), _membitmovewb()

Similar to the standard C library `memcpy()` and `memmove()` functions, these nonstandard C library functions provide bit-aligned memory operations.

They are defined in `string.h`.

Syntax

```
void _membitcpy[b|h|w][b|l](void *dest, const void *src, int dest_offset, int
src_offset, size_t nbits);
```

```
void _membitmove[b|h|w][b|l](void *dest, const void *src, int dest_offset, int
src_offset, size_t nbits);
```

Usage

The number of contiguous bits specified by ***`nbits`*** is copied, or moved (depending on the function being used), from a memory location starting *`src_offset`* bits after (or before if a negative offset) the address pointed to by *`src`*, to a location starting *`dest_offset`* bits after (or before if a negative offset) the address pointed to by *`dest`*.

To define a contiguous sequence of bits, a form of ordering is required. The variants of each function define this order, as follows:

- Functions whose second-last character is `b`, for example `_membitcpybl()`, are byte-oriented. Byte-oriented functions consider all of the bits in one byte to come before the bits in the next byte.
- Functions whose second-last character is `h` are halfword-oriented.
- Functions whose second-last character is `w` are word-oriented.

Within each byte, halfword, or word, the bits can be considered to go in different order depending on the endianness. Functions ending in `b`, for example `_membitmovewb()`, are bitwise big-endian. This means that the *Most Significant Bit* (MSB) of each byte, halfword, or word (as appropriate) is considered to be the first bit in the word, and the *Least Significant Bit* (LSB) is considered to be the last. Functions ending in `l` are bitwise little-endian. They consider the LSB to come first and the MSB to come last.

As with `memcpy()` and `memmove()`, the bitwise memory copying functions copy as fast as they can in their assumption that source and destination memory regions do not overlap, whereas the bitwise memory move functions ensure that source data in overlapping regions is copied before being overwritten.

On a little-endian platform, the bitwise big-endian functions are distinct, but the bitwise little-endian functions use the same bit ordering, so they are synonymous symbols that refer to the same function. On a big-endian platform, the bitwise big-endian functions are all effectively the same, but the bitwise little-endian functions are distinct.

4.17 posix_memalign()

Defined in `stdlib.h`, the `posix_memalign()` function provides aligned memory allocation.

It is fully POSIX-compliant.

Syntax

```
int posix_memalign(void **memptr, size_t alignment, size_t size);
```

Usage

This function allocates *`size`* bytes of memory at an address that is a multiple of *`alignment`*.

The value of *`alignment`* must be a power of two and a multiple of `sizeof(void *)`.

You can free memory allocated by `posix_memalign()` using the standard C library `free()` function.

Returns

The returned address is written to the `void *` variable pointed to by *`memptr`*.

The integer return value from the function is zero on success, or an error code on failure.

If no block of memory can be found with the requested size and alignment, the function returns `ENOMEM` and the value of `*`*`memptr`* is undefined.

Related information

The Open Group Base Specifications, IEEE Std 1003.1.

4.18 #pragma import(_main_redirection)

This pragma must be defined when redirecting standard input, output and error streams at runtime.

Syntax

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

Related information

Environment.

4.19 __raise()

Defined in `rt_misc.h`, the `__raise()` function raises a signal to indicate a runtime anomaly.

It is not part of the C library standard, but the ARM C library supports it as an extension.

Syntax

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

where:

`signal`
is an integer that holds the signal number.

`type`
is an integer, string constant or variable that provides additional information about the circumstances that the signal was raised in, for some kinds of signal.

Usage

If the user has configured the handling of the signal by calling `signal()` then `__raise()` takes the action specified by the user. That is, either to ignore the signal or to call the user-provided handler function. Otherwise, `__raise()` calls `__default_signal_handler()`, which provides the default signal handling behavior.

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

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

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

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

The default signal handler of the library uses the *`type`* parameter of `__raise()` to vary the messages it outputs.

Returns

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

no return
The handler performs a long jump or restart.

0
The signal was handled.

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

Related concepts

1.21 Thread safety in the ARM C library on page 1-41.

Related references

1.30 Indirect semihosting C library function dependencies on page 1-52.

4.5 __default_signal_handler() on page 4-195.

4.51 _ttywrch() on page 4-244.

4.41 _sys_exit() on page 4-234.

4.30 __rt_raise() on page 4-222.

1.84 ISO-compliant implementation of signals supported by the signal() function in the C library and additional type arguments on page 1-117.

4.20 _rand_r()

Defined in `stdlib.h`, the `_rand_r()` function is a reentrant version of the `rand()` function.

Syntax

```
int _rand_r(struct _rand_state * buffer);
```

where:

`buffer`
is a pointer to a user-supplied buffer storing the state of the random number generator.

Usage

This function enables you to explicitly supply your own buffer in thread-local storage.

Related references

4.60 C library functions that are not thread-safe on page 4-255.

4.33 _srand_r() on page 4-226.

4.21 remove()

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

Syntax

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

Usage

The default implementation of this function uses semihosting.

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

Returns

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

Related references

1.29 Direct semihosting C library function dependencies on page 1-51.

4.22 rename()

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

Syntax

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

Usage

The default implementation of this function uses semihosting.

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

Returns

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

Related references

1.29 Direct semihosting C library function dependencies on page 1-51.

4.23 __rt_entry

The symbol `__rt_entry` is the starting point for a program using the ARM C library.

Control passes to `__rt_entry` after all scatter-loaded regions have been relocated to their execution addresses.

Usage

The default implementation of `__rt_entry`:

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

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

`exit()`

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

`__rt_exit()`

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

`_sys_exit()`

Exits directly to the execution environment. It does not shut down the library and does not call `atexit()` functions.

4.24 __rt_errno_addr()

The `__rt_errno_addr()` function is called to get the address of the C library `errno` variable when the C library attempts to read or write `errno`.

The library provides a default implementation. It is unlikely that you have to reimplement this function.

This function is not part of the C library standard, but the ARM C library supports it as an extension.

Note

This function is associated with pre-ABI versions of the compilation tools. However, it remains supported to ensure that object files compiled with those tools link successfully. Unless you are working with object files compiled with pre-ABI versions of the tools, use `__aeabi_errno_addr()` instead of `__rt_errno_addr()`.

Syntax

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

Related references

4.1 __aeabi_errno_addr() on page 4-191.

4.6 errno on page 4-196.

Related information

Application Binary Interface for the ARM Architecture.

4.25 __rt_exit()

Defined in `rt_misc.h`, the `__rt_exit()` function shuts down the library but does not call functions registered with `atexit()`.

`atexit()`-registered functions are called by `exit()`.

`__rt_exit()` is not part of the C library standard, but the ARM C library supports it as an extension.

Syntax

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

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

Usage

Shuts down the C library by calling `__rt_lib_shutdown()`, and then calls `_sys_exit()` to terminate the application. Reimplement `_sys_exit()` rather than `__rt_exit()`.

Returns

This function does not return.

Related references

4.41 _sys_exit() on page 4-234.

4.26 __rt_fp_status_addr()

Defined in `rt_fp.h`, the `__rt_fp_status_addr()` function returns the address of the floating-point status word.

By default, the floating-point status word resides in `__user_libspace`.

It is not part of the C library standard, but the ARM C library supports it as an extension.

Syntax

Syntax

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

Usage

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

Returns

The address of the floating-point status word.

Related concepts

1.21 Thread safety in the ARM C library on page 1-41.

4.27 __rt_heap_extend()

Defined in `rt_heap.h`, the `__rt_heap_extend()` function returns a new eight-byte aligned block of memory to add to the heap, if possible.

If you reimplement `__rt_stackheap_init()`, you must reimplement this function. An incomplete prototype implementation is in `rt_memory.s`.

This function is not part of the C library standard, but the ARM C library supports it as an extension.

Syntax

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

Usage

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

The default implementation has the following characteristics:

- The returned size is either:
  - — A multiple of eight bytes of at least the requested size.
  - — 0, denoting that the request cannot be honored.
- The returned base address is aligned on an eight-byte boundary.
- Size is measured in bytes.
- The function is subject only to *ARM Architecture Procedure Call Standard* (AAPCS) constraints.

Returns

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

Related concepts

1.66 Stack pointer initialization and heap bounds on page 1-94.

Related references

4.31 __rt_stackheap_init() on page 4-223.
4.52 __user_heap_extend() on page 4-245.
4.53 __user_heap_extent() on page 4-246.
4.54 __user_setup_stackheap() on page 4-247.

Related information

Procedure Call Standard for the ARM Architecture.

4.28 __rt_lib_init()

Defined in `rt_misc.h`, this is the library initialization function and is the companion to `__rt_lib_shutdown()`.

Syntax

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

where:

`heapbase`
is the start of the heap memory block.

`heaptop`
is the end of the heap memory block.

Usage

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

Returns

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

```
struct __argc_argv
{   int argc;
    char **argv;
    int r2, r3;   // optional extra arguments that on entry to main() are
};                // found in registers R2 and R3.
```

4.29 __rt_lib_shutdown()

Defined in `rt_misc.h`, `__rt_lib_shutdown()` is the library shutdown function and is the companion to `__rt_lib_init()`.

Syntax

```
void __rt_lib_shutdown(void);
```

Usage

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

4.30 __rt_raise()

Defined in `rt_misc.h`, the `__rt_raise()` function raises a signal to indicate a runtime anomaly.

It is not part of the C library standard, but the ARM C library supports it as an extension.

Syntax

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

where:

`signal`
is an integer that holds the signal number.

`type`
is an integer, string constant or variable that provides additional information about the circumstances that the signal was raised in, for some kinds of signal.

Usage

Redefine this function to replace the entire signal handling mechanism for the library. The default implementation calls `__raise()`.

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

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

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

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

Related references

4.5 __default_signal_handler() on page 4-195.

4.19 __raise() on page 4-210.

4.51 _ttywrch() on page 4-244.

4.41 _sys_exit() on page 4-234.

1.84 ISO-compliant implementation of signals supported by the signal() function in the C library and additional type arguments on page 1-117.

4.31 __rt_stackheap_init()

Defined in `rt_misc.h`, the `__rt_stackheap_init()` function sets up the stack pointer and returns a region of memory for use as the initial heap.

It is called from the library initialization code.

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

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

This function is not part of the C library standard, but the ARM C library supports it as an extension.

Related concepts

1.66 Stack pointer initialization and heap bounds on page 1-94.

Related references

4.27 __rt_heap_extend() on page 4-219.
4.52 __user_heap_extend() on page 4-245.
4.53 __user_heap_extent() on page 4-246.
4.54 __user_setup_stackheap() on page 4-247.

4.32 setlocale()

Defined in `locale.h`, the `setlocale()` function selects the appropriate locale as specified by the *`category`* and *`locale`* arguments.

Syntax

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

Usage

Use the `setlocale()` function to change or query part or all of the current locale. The effect of the *`category`* argument for each value is:

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

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

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

`LC_NUMERIC`
Affects the decimal-point character for the formatted input/output functions and the string conversion functions and the numeric formatting information returned by `localeconv()`.

`LC_TIME`
Can affect the behavior of `strftime()`. For currently supported locales, the option has no effect.

`LC_ALL`
Affects all locale categories. This is the bitwise OR of all the locale categories.

A value of `"C"` for *`locale`* specifies the minimal environment for C translation. An empty string, `""`, for *`locale`* specifies the implementation-defined native environment. At program startup, the equivalent of `setlocale(LC_ALL, "C")` is executed.

Valid *`locale`* values depend on which `__use_X_ctype` symbols are imported (`__use_iso8859_ctype`, `__use_sjis_ctype`, `__use_utf8_ctypte`), and on user-defined locales.

Returns

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

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

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

Related concepts

1.49 Assembler macros that tailor locale functions in the C library on page 1-73.

1.52 Shift-JIS and UTF-8 implementation on page 1-76.

Related references

4.7 _findlocale() on page 4-197.

4.14 lconv structure on page 4-204.

4.9 _get_lconv() on page 4-199.

4.15 localeconv() on page 4-206.

1.51 ISO8859-1 implementation on page 1-75.

1.54 Definition of locale data blocks in the C library on page 1-78.

4.33 _srand_r()

Defined in `stdlib.h`, this is a reentrant version of the `srand()` function.

Syntax

```
int _srand_r(struct _rand_state * buffer, unsigned int seed);
```

where:

`buffer`

is a pointer to a user-supplied buffer storing the state of the random number generator.

`seed`

is a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to `_rand_r()`.

Usage

This function enables you to explicitly supply your own buffer that can be used for thread-local storage.

If `_srand_r()` is repeatedly called with the same seed value, the same sequence of pseudo-random numbers is repeated. If `_rand_r()` is called before any calls to `_srand_r()` have been made with the same buffer, undefined behavior occurs because the buffer is not initialized.

Related references

4.60 C library functions that are not thread-safe on page 4-255.

4.20 _rand_r() on page 4-212.

4.34 strcasecmp()

Defined in `string.h`, the `strcasecmp()` function performs a case-insensitive string comparison test.

Syntax

```
extern _ARMABI int strcasecmp(const char *s1, const char *s2);
```

Related information

Application Binary Interface for the ARM Architecture.

4.35 strncasecmp()

Defined in `string.h`, the `strncasecmp()` function performs a case-insensitive string comparison test of not more than a specified number of characters.

Syntax

```
extern _ARMABI int strncasecmp(const char *s1, const char *s2, size_t n);
```

Related information

Application Binary Interface for the ARM Architecture.

4.36 strlcat()

Defined in `string.h`, the `strlcat()` function concatenates two strings.

It appends up to `size-strlen(dst)-1` bytes from the `NUL`-terminated string `src` to the end of `dst`. It takes the full size of the buffer, not only the length, and terminates the result with `NUL` as long as `size` is greater than 0. Include a byte for the `NUL` in your `size` value.

The `strlcat()` function returns the total length of the string that *would* have been created if there was unlimited space. This might or might not be equal to the length of the string *actually* created, depending on whether there was enough space. This means that you can call `strlcat()` once to find out how much space is required, then allocate it if you do not have enough, and finally call `strlcat()` a second time to create the required string.

This function is a common BSD-derived extension to many C libraries.

Syntax

```
extern size_t strlcat(char *dst, *src, size_t size);
```

4.37 strlcpy()

Defined in `string.h`, the `strlcpy()` function copies up to *`size`*`-1` characters from the `NUL`-terminated string *`src`* to *`dst`*.

It takes the full size of the buffer, not only the length, and terminates the result with `NUL` as long as *`size`* is greater than 0. Include a byte for the `NUL` in your *`size`* value.

The `strlcpy()` function returns the total length of the string that *would* have been copied if there was unlimited space. This might or might not be equal to the length of the string *actually* copied, depending on whether there was enough space. This means that you can call `strlcpy()` once to find out how much space is required, then allocate it if you do not have enough, and finally call `strlcpy()` a second time to do the required copy.

This function is a common BSD-derived extension to many C libraries.

Syntax

```
extern size_t strlcpy(char *dst, const char *src, size_t size);
```

4.38 _sys_close()

Defined in `rt_sys.h`, the `_sys_close()` function closes a file previously opened with `_sys_open()`.

Syntax

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

Usage

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

Returns

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

Related references

1.29 Direct semihosting C library function dependencies on page 1-51.

4.39 _sys_command_string()

Defined in `rt_sys.h`, the `_sys_command_string()` function retrieves the command line that invoked the current application from the environment that called the application.

Syntax

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

where:

`cmd`

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

`len`

is the length of the buffer.

Usage

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

Note

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

Returns

The function must return either:

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

Related references

1.29 Direct semihosting C library function dependencies on page 1-51.

4.40 _sys_ensure()

This function is deprecated. It is never called by any other library function, and you are not required to re-implement it if you are retargeting standard I/O (`stdio`).

4.41 _sys_exit()

Defined in `rt_sys.h`, this is the library exit function. All exits from the library eventually call `_sys_exit()`.

Syntax

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

Usage

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

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

Returns

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

Related references

1.29 Direct semihosting C library function dependencies on page 1-51.

4.5 __default_signal_handler() on page 4-195.

4.19 __raise() on page 4-210.

4.51 _ttywrch() on page 4-244.

4.30 __rt_raise() on page 4-222.

4.25 __rt_exit() on page 4-217.

4.42 _sys_flen()

Defined in `rt_sys.h`, the `_sys_flen()` function returns the current length of a file.

Syntax

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

Usage

This function is used by `_sys_seek()` to convert an offset relative to the end of a file into an offset relative to the beginning of the file.

You do not have to define `_sys_flen()` if you do not intend to use `fseek()`.

If you retarget at system `_sys_*()` level, you must supply `_sys_flen()`, even if the underlying system directly supports seeking relative to the end of a file.

Returns

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

Related references

1.29 Direct semihosting C library function dependencies on page 1-51.

4.43 _sys_istty()

Defined in `rt_sys.h`, the `_sys_istty()` function determines if a file handle identifies a terminal.

Syntax

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

Usage

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

Returns

The return value is one of the following values:

0

There is no interactive device.

1

There is an interactive device.

other

An error occurred.

Related references

1.29 Direct semihosting C library function dependencies on page 1-51.

4.8 _fisatty() on page 4-198.

4.44 _sys_open()

Defined in `rt_sys.h`, the `_sys_open()` function opens a file.

Syntax

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

Usage

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

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

Returns

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

Related references

1.29 Direct semihosting C library function dependencies on page 1-51.

4.45 _sys_read()

Defined in `rt_sys.h`, the `_sys_read()` function reads the contents of a file into a buffer.

Syntax

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

Note

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

Returns

The return value is one of the following:

- The number of bytes *not* read (that is, `len - result` number of bytes were read).
- An error indication.
- An `EOF` indicator. The `EOF` indication involves the setting of `0x80000000` in the normal result.

Reading up to and including the last byte of data does not turn on the `EOF` indicator. The `EOF` indicator is only reached when an attempt is made to read beyond the last byte of data. The target-independent code is capable of handling:

- The `EOF` indicator being returned in the same read as the remaining bytes of data that precede the `EOF`.
- The `EOF` indicator being returned on its own after the remaining bytes of data have been returned in a previous read.

Related references

1.29 Direct semihosting C library function dependencies on page 1-51.

4.46 _sys_seek()

Defined in `rt_sys.h`, the `_sys_seek()` function puts the file pointer at offset *`pos`* from the beginning of the file.

Syntax

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

Usage

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

Returns

The result is:

- Negative if an error occurs.
- Non-negative if no error occurs.

Related references

1.29 Direct semihosting C library function dependencies on page 1-51.

4.47 _sys_tmpnam()

Defined in `rt_sys.h`, the `_sys_tmpnam()` function converts the file number *`fileno`* for a temporary file to a unique filename, for example, `tmp0001`.

Syntax

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

Usage

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

Returns

Returns the filename in *`name`*.

Related references

1.29 Direct semihosting C library function dependencies on page 1-51.

4.48 _sys_write()

Defined in `rt_sys.h`, the `_sys_write()` function writes the contents of a buffer to a file previously opened with `_sys_open()`.

Syntax

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

Note

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

Returns

The return value is either:

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

Related references

1.29 Direct semihosting C library function dependencies on page 1-51.

4.49 system()

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

Syntax

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

Usage

The default implementation of this function uses semihosting.

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

Returns

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

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

Related references

1.29 Direct semihosting C library function dependencies on page 1-51.

4.50 time()

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

The default implementation of this function uses semihosting.

Syntax

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

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

Returns

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

Related references

1.29 Direct semihosting C library function dependencies on page 1-51.

4.51 _ttywrch()

Defined in `rt_sys.h`, the `_ttywrch()` function writes a character to the console.

The console might have been redirected. You can use this function as a last resort error handling routine.

Syntax

```
void _ttywrch(int ch);
```

Usage

The default implementation of this function uses semihosting.

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

Related references

1.29 Direct semihosting C library function dependencies on page 1-51.

4.5 __default_signal_handler() on page 4-195.

4.19 __raise() on page 4-210.

4.41 _sys_exit() on page 4-234.

4.30 __rt_raise() on page 4-222.

4.19 __raise() on page 4-210.

Related information

What is Semihosting?.

4.52 __user_heap_extend()

Defined in `rt_misc.h`, the `__user_heap_extend()` function can be defined to return extra blocks of memory, separate from the initial one, to be used by the heap.

If defined, this function must return the size and base address of an eight-byte aligned heap extension block.

Syntax

```
extern unsigned __user_heap_extend(int var0, void **base, unsigned
requested_size);
```

Usage

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

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

Returns

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

Related concepts

1.66 Stack pointer initialization and heap bounds on page 1-94.

Related references

4.27 __rt_heap_extend() on page 4-219.
4.31 __rt_stackheap_init() on page 4-223.
4.53 __user_heap_extent() on page 4-246.
4.54 __user_setup_stackheap() on page 4-247.

Related information

Procedure Call Standard for the ARM Architecture.

4.53 __user_heap_extent()

If defined, the `__user_heap_extent()` function returns the base address and maximum range of the heap.

See `rt_misc.h`.

Syntax

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

Usage

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

Related concepts

1.66 Stack pointer initialization and heap bounds on page 1-94.

Related references

4.27 __rt_heap_extend() on page 4-219.

4.31 __rt_stackheap_init() on page 4-223.

4.52 __user_heap_extend() on page 4-245.

4.54 __user_setup_stackheap() on page 4-247.

1.62 C library support for memory allocation functions on page 1-90.

4.54 __user_setup_stackheap()

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

If you define this function, it is called by the C library during program start-up.

When `__user_setup_stackheap()` is called, `sp` has the same value it had on entry to the application. If this was set to a valid value before calling the C library initialization code, it can be left at this value. If `sp` is not valid, `__user_setup_stackheap()` must change this value before using any stack and before returning.

`__user_setup_stackheap()` returns the:

- Heap base in `r0` (if the program uses the heap).
- Stack base in `sp`.
- Heap limit in `r2` (if the program uses the heap and uses two-region memory).

If this function is re-implemented, it must:

- Not corrupt registers other than `r0` to `r3`, `ip` and `sp`.
- Maintain eight-byte alignment of the heap by ensuring that the heap base is a multiple of eight.

To create a version of `__user_setup_stackheap()` that inherits `sp` from the execution environment and does not have a heap, set `r0` and `r2` to `zero` and return.

There is no limit to the size of the stack. However, if the heap region grows into the stack, `malloc()` attempts to detect the overlapping memory and fails the new memory allocation request.

Note

Any re-implementation of `__user_setup_stackheap()` must be in assembler.

Related concepts

1.66 Stack pointer initialization and heap bounds on page 1-94.

1.66 Stack pointer initialization and heap bounds on page 1-94.

1.69 Legacy support for __user_initial_stackheap() on page 1-98.

Related references

1.29 Direct semihosting C library function dependencies on page 1-51.

4.27 __rt_heap_extend() on page 4-219.

4.31 __rt_stackheap_init() on page 4-223.

4.52 __user_heap_extend() on page 4-245.

4.53 __user_heap_extent() on page 4-246.

4.61 Legacy function __user_initial_stackheap() on page 4-257.

4.55 __vectab_stack_and_reset

`__vectab_stack_and_reset` is a library section that provides a way for the initial values of `sp` and `pc` to be placed in the vector table, starting at address `0` for M-profile processors, such as Cortex-M1 and Cortex-M3 embedded applications.

`__vectab_stack_and_reset` requires the existence of a `main()` function in your source code. Without a `main()` function, if you place the `__vectab_stack_and_reset` section in a scatter file, an error is generated to the following effect:

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

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

The following segment is part of a scatter file. It includes a minimal vector table illustrating the use of `__vectab_stack_and_reset` to place the initial `sp` and `pc` values at addresses `0x0` and `0x4` in the vector table:

```
;; Maximum of 256 exceptions (256*4 bytes == 0x400)
VECTORS 0x0 0x400
{
    ; First two entries provided by library
    ; Remaining entries provided by the user in exceptions.c
    * (:gdef:__vectab_stack_and_reset, +FIRST)
    * (exceptions_area)
}
CODE 0x400 FIXED
{
    * (+RO)
}
```

Related concepts

1.66 Stack pointer initialization and heap bounds on page 1-94.

Related information

About scatter-loading.

4.56 wcscasecmp()

Defined in `wchar.h`, the `wcscasecmp()` function performs a case-insensitive string comparison test on wide characters.

It is a GNU extension to the libraries. It is not POSIX-standardized.

Syntax

```
int wcscasecmp(const wchar_t * __restrict s1, const wchar_t * __restrict s2);
```

4.57 wcsncasecmp()

Defined in `wchar.h`, the `wcsncasecmp()` function performs a case-insensitive string comparison test of not more than a specified number of wide characters.

It is a GNU extension to the libraries. It is not POSIX-standardized.

Syntax

```
int wcsncasecmp(const wchar_t * __restrict s1, const wchar_t * __restrict s2,
size_t n);
```

4.58 wcstombs()

Defined in `wchar.h`, the `wcstombs()` function works as described in the ISO C standard, with extended functionality as specified by POSIX.

That is, if *`s`* is a `NULL` pointer, `wcstombs()` returns the length required to convert the entire array regardless of the value of *`n`*, but no values are stored.

Syntax

```
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

4.59 Thread-safe C library functions

The following table shows the C library functions that are thread-safe.

Table 4-1 Functions that are thread-safe

Functions	Description
`calloc(), free(), malloc(), realloc()`	The heap functions are thread-safe if the `_mutex_*` functions are implemented. All threads share a single heap and use mutexes to avoid data corruption when there is concurrent access. Each heap implementation is responsible for doing its own locking. If you supply your own allocator, it must also do its own locking. This enables it to do fine-grained locking if required, rather than protecting the entire heap with a single mutex (coarse-grained locking).
`alloca()`	`alloca()` is thread-safe because it allocates memory on the stack.
`abort(), raise(), signal(), fenv.h`	The ARM signal handling functions and floating-point exception traps are thread-safe. The settings for signal handlers and floating-point traps are global across the entire process and are protected by locks. Data corruption does not occur if multiple threads call `signal()` or an `fenv.h` function at the same time. However, be aware that the effects of the call act on all threads and not only on the calling thread.
`clearerr(), fclose(), feof(),ferror(), fflush(), fgetc(),fgetpos(), fgets(), fopen(),fputc(), fputs(), fread(),freopen(), fseek(), fsetpos(),ftell(), fwrite(), getc(),getchar(), gets(), perror(),putc(), putchar(), puts(),rewind(), setbuf(), setvbuf(),tmpfile(), tmpnam(), ungetc()`	The `stdio` library is thread-safe if the `_mutex_*` functions are implemented. Each individual stream is protected by a lock, so two threads can each open their own `stdio` stream and use it, without interfering with one another. If two threads both want to read or write the same stream, locking at the `fgetc()` and `fputc()` level prevents data corruption, but it is possible that the individual characters output by each thread might be interleaved in a confusing way. **Note** `tmpnam()` also contains a static buffer but this is only used if the argument is NULL. To ensure that your use of `tmpnam()` is thread-safe, supply your own buffer space.
`fprintf(), printf(), vfprintf(), vprintf(), fscanf(), scanf()`	When using these functions: • The standard C `printf()` and `scanf()` functions use `stdio` so they are thread-safe. • The standard C `printf()` function is susceptible to changes in the locale settings if called in a multithreaded program.
`clock()`	`clock()` contains static data that is written once at program startup and then only ever read. Therefore, `clock()` is thread-safe provided no extra threads are already running at the time that the library is initialized.

Table 4-1 Functions that are thread-safe (continued)

Functions	Description
`errno`	`errno` is thread-safe. Each thread has its own `errno` stored in a `__user_perthread_libspace` block. This means that each thread can call `errno`-setting functions independently and then check `errno` afterwards without interference from other threads.
`atexit()`	The list of exit functions maintained by `atexit()` is process-global and protected by a lock. In the worst case, if more than one thread calls `atexit()`, the order that exit functions are called cannot be guaranteed.
`abs(), acos(), asin(),atan(), atan2(), atof(),atol(), atoi(), bsearch(),ceil(), cos(), cosh(),difftime(), div(), exp(),fabs(), floor(), fmod(),frexp(), labs(), ldexp(),ldiv(), log(), log10(),memchr(), memcmp(), memcpy(),memmove(), memset(), mktime(),modf(), pow(), qsort(),sin(), sinh(), sqrt(),strcat(), strchr(), strcmp(),strcpy(), strcspn(), strlcat(),strlcpy(), strlen(), strncat(),strncmp(), strncpy(), strpbrk(),strrchr(), strspn(), strstr(),strxfrm(), tan(), tanh()`	These functions are inherently thread-safe.
`longjmp(), setjmp()`	Although `setjmp()` and `longjmp()` keep data in `__user_libspace`, they call the `__alloca_*` functions, that are thread-safe.
`remove(), rename(), time()`	These functions use interrupts that communicate with the ARM debugging environments. Typically, you have to reimplement these for a real-world application.

Table 4-1 Functions that are thread-safe (continued)

Functions	Description
`snprintf(), sprintf(), vsnprintf(),vsprintf(), sscanf(), isalnum(),isalpha(), iscntrl(), isdigit(),isgraph(), islower(), isprint(),ispunct(), isspace(), isupper(),isxdigit(), tolower(), toupper(),strcoll(), strtod(), strtol(),strtoul(), strftime()`	When using these functions, the string-based functions read the locale settings. Typically, they are thread-safe. However, if you change locale in mid-session, you must ensure that these functions are not affected. The string-based functions, such as `sprintf()` and `sscanf()`, do not depend on the `stdio` library.
`stdin, stdout, stderr`	These functions are thread-safe.

Related concepts

1.21 Thread safety in the ARM C library on page 1-41.

Related references

4.2 alloca() on page 4-192.

4.60 C library functions that are not thread-safe

The following table shows the C library functions that are not thread-safe.

Table 4-2 Functions that are not thread-safe

Functions	Description
`asctime()`, `localtime()`, `strtok()`	These functions are all thread-unsafe. Each contains a static buffer that might be overwritten by another thread between a call to the function and the subsequent use of its return value. ARM supplies reentrant versions, `_asctime_r()`, `_localtime_r()`, and `_strtok_r()`. ARM recommends that you use these functions instead to ensure safety. **Note** These reentrant versions take additional parameters. `_asctime_r()` takes an additional parameter that is a pointer to a buffer that the output string is written into. `_localtime_r()` takes an additional parameter that is a pointer to a `struct tm`, that the result is written into. `_strtok_r()` takes an additional parameter that is a pointer to a **`char`** pointer to the next token.
`exit()`	Do not call `exit()` in a multithreaded program even if you have provided an implementation of the underlying `_sys_exit()` that actually terminates all threads. In this case, `exit()` cleans up *before* calling `_sys_exit()` so disrupts other threads.
`gamma()`, `lgamma()`, `lgammaf()`, `lgammal()` [d]	These extended mathlib functions use a global variable, `_signgam`, so are not thread-safe.
`mbrlen()`, `mbsrtowcs()`, `mbrtowc()`, `wcrtomb()`, `wcsrtombs()`	The C89 multibyte conversion functions (defined in `stdlib.h`) are not thread-safe, for example `mblen()` and `mbtowc()`, because they contain internal static state that is shared between all threads without locking. However, the extended restartable versions (defined in `wchar.h`) are thread-safe, for example `mbrtowc()` and `wcrtomb()`, provided you pass in a pointer to your own `mbstate_t` object. You must exclusively use these functions with non-NULL `mbstate_t *` parameters if you want to ensure thread-safety when handling multibyte strings.

---

[d] If migrating from RVCT, be aware that `gamma()` is deprecated in ARM Compiler 4.1 and later.

Table 4-2 Functions that are not thread-safe (continued)

Functions	Description
`rand()`, `srand()`	These functions keep internal state that is both global and unprotected. This means that calls to `rand()` are never thread-safe. ARM recommends that you do one of the following: • Use the reentrant versions `_rand_r()` and `_srand_r()` supplied by ARM. These use user-provided buffers instead of static data within the C library. • Use your own locking to ensure that only one thread ever calls `rand()` at a time, for example, by defining `$Sub$$rand()` if you want to avoid changing your code. • Arrange that only one thread ever needs to generate random numbers. • Supply your own random number generator that can have multiple independent instances. **Note** `_rand_r()` and `_srand_r()` both take an additional parameter that is a pointer to a buffer storing the state of the random number generator.
`setlocale()`, `localeconv()`	`setlocale()` is used for setting and reading locale settings. The locale settings are global across all threads, and are not protected by a lock. If two threads call `setlocale()` to simultaneously modify the locale settings, or if one thread reads the settings while another thread is modifying them, data corruption might occur. Also, many other functions, for example `strtod()` and `sprintf()`, read the current locale settings. Therefore, if one thread calls `setlocale()` concurrently with another thread calling such a function, there might be unexpected results. Multiple threads *reading* the settings simultaneously is thread-safe in simple cases and if no other thread is simultaneously modifying those settings, but where internally an intermediate buffer is required for more complicated returned results, unexpected results can occur unless you use a reentrant version of `setlocale()`. ARM recommends that you either: • Choose the locale you want and call `setlocale()` once to initialize it. Do this before creating any additional threads in your program so that any number of threads can read the locale settings concurrently without interfering with one another. • Use the reentrant version `_setlocale_r()` supplied by ARM. This returns a string that is either a pointer to a constant string, or a pointer to a string stored in a user-supplied buffer that can be used for thread-local storage, rather than using memory within the C library. Be aware that `_setlocale_r()` is not fully thread-safe when accessed concurrently to *change* locale settings. This access is not lock-protected. Also, be aware that `localeconv()` is not thread-safe. Call the ARM function `_get_lconv()` with a pointer to a user-supplied buffer instead.

Related concepts

1.21 Thread safety in the ARM C library on page 1-41.

Related references

4.20 _rand_r() on page 4-212.

4.33 _srand_r() on page 4-226.

4.61 Legacy function __user_initial_stackheap()

If you have legacy source code you might see `__user_initial_stackheap()`, from `rt_misc.h`. This is an old function that is only supported for backwards compatibility with legacy source code.

The modern equivalent is `__user_setup_stackheap()`.

Syntax

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

Usage

`__user_initial_stackheap()` returns the:

- Heap base in `r0`.
- Stack base in `r1`, that is, the highest address in the stack region.
- Heap limit in `r2`.

If this function is reimplemented, it must:

- Use no more than 88 bytes of stack.
- Not corrupt registers other than `r12` (`ip`).
- Maintain eight-byte alignment of the heap.

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

To create a version of `__user_initial_stackheap()` that inherits `sp` from the execution environment and does not have a heap, set `r0` and `r2` to the value of `r1` and return.

There is no limit to the size of the stack. However, if the heap region grows into the stack, `malloc()` attempts to detect the overlapping memory and fails the new memory allocation request.

The definition of `__initial_stackheap` in `rt_misc.h` is:

```
struct __initial_stackheap {
    unsigned heap_base; /* low-address end of initial heap */
    unsigned stack_base; /* high-address end of initial stack */
    unsigned heap_limit; /* high-address end of initial heap */
    unsigned stack_limit; /* unused */
};
```

Note

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

Related concepts

1.66 Stack pointer initialization and heap bounds on page 1-94.

1.69 Legacy support for __user_initial_stackheap() on page 1-98.

Related references

1.29 Direct semihosting C library function dependencies on page 1-51.

4.54 __user_setup_stackheap() on page 4-247.

1.29 Direct semihosting C library function dependencies on page 1-51.

Chapter 5
Floating-point Support Functions

Describes ARM support for floating-point computations.

It contains the following:

5.1 _clearfp()

Defined in `float.h`, the `_clearfp()` function is provided for compatibility with Microsoft products.

`_clearfp()` clears all five exception sticky flags and returns their previous values. You can use the `_controlfp()` argument macros, for example `_EM_INVALID` and `_EM_ZERODIVIDE`, to test bits of the returned result.

The function prototype for `_clearfp()` is:

```
unsigned _clearfp(void);
```

Note

This function requires you to select a floating-point model that supports exceptions. For example, `--fpmode=ieee_full` or `--fpmode=ieee_fixed`.

Related concepts

3.9 Floating-point functions for compatibility with Microsoft products on page 3-156.

Related references

5.2 _controlfp() on page 5-261.

5.8 _statusfp() on page 5-271.

5.2 _controlfp()

Defined in `float.h`, the `_controlfp()` function is provided for compatibility with Microsoft products. It enables you to control exception traps and rounding modes.

The function prototype for `_controlfp()` is:

```
unsigned int _controlfp(unsigned int new, unsigned int mask);
```

Note

This function requires you to select a floating-point model that supports exceptions. For example, `--fpmode=ieee_full` or `--fpmode=ieee_fixed`.

`_controlfp()` also modifies a control word using a mask to isolate the bits to modify. For every bit of `mask` that is zero, the corresponding control word bit is unchanged. For every bit of `mask` that is nonzero, the corresponding control word bit is set to the value of the corresponding bit of `new`. The return value is the previous state of the control word.

Note

This is different behavior to that of `__ieee_status()` or `__fp_status()`, where you can toggle a bit by setting a zero in the mask word and a one in the flags word.

The following table describes the macros you can use to form the arguments to `_controlfp()`.

Table 5-1 _controlfp argument macros

Macro	Description
`_MCW_EM`	Mask containing all exception bits
`_EM_INVALID`	Bit describing the Invalid Operation exception
`_EM_ZERODIVIDE`	Bit describing the Divide by Zero exception
`_EM_OVERFLOW`	Bit describing the Overflow exception
`_EM_UNDERFLOW`	Bit describing the Underflow exception
`_EM_INEXACT`	Bit describing the Inexact Result exception
`_MCW_RC`	Mask for the rounding mode field
`_RC_CHOP`	Rounding mode value describing Round Toward Zero
`_RC_UP`	Rounding mode value describing Round Up
`_RC_DOWN`	Rounding mode value describing Round Down
`_RC_NEAR`	Rounding mode value describing Round To Nearest

Note

The values of these macros are not guaranteed to remain the same in future versions of ARM products. To ensure that your code continues to work if the value changes in future releases, use the macro rather than its value.

For example, to set the rounding mode to round down, call:

```
_controlfp(_RC_DOWN, _MCW_RC);
```

To trap the Invalid Operation exception and untrap all other exceptions:

```
_controlfp(_EM_INVALID, _MCW_EM);
```

To untrap the Inexact Result exception:

```
_controlfp(0, _EM_INEXACT);
```

Related concepts

3.9 Floating-point functions for compatibility with Microsoft products on page 3-156.

Related references

5.1 _clearfp() on page 5-260.

5.8 _statusfp() on page 5-271.

5.5 __ieee_status() on page 5-266.

5.3 __fp_status() on page 5-263.

5.3 __fp_status()

Some older versions of the ARM libraries implemented a function called `__fp_status()` that manipulated a status word in the floating-point environment.

This is the same as `__ieee_status()` but it uses an older style of status word layout. The compiler still supports the `__fp_status()` function for backwards compatibility. `__fp_status()` is defined in `stdlib.h`.

The function prototype for `__fp_status()` is:

```
unsigned int __fp_status(unsigned int mask, unsigned int flags);
```

Note

This function requires you to select a floating-point model that supports exceptions. For example, `--fpmode=ieee_full` or `--fpmode=ieee_fixed`.

The layout of the status word as seen by `__fp_status()` is as follows:

31 24	23 21	20 16	15 13	12 8	7 5	4 0
System ID	R	Masks	R	FPA only	R	Sticky

Figure 5-1 Floating-point status word layout

The fields in the status word are as follows:

- Bits 0 to 4 (values `0x1` to `0x10`, respectively) are the sticky flags, or cumulative flags, for each exception. The sticky flag for an exception is set to 1 whenever that exception happens and is not trapped. Sticky flags are never cleared by the system, only by the user. The mapping of exceptions to bits is:
  - bit 0 (`0x01`) is for the Invalid Operation exception
  - Bit 1 (`0x02`) is for the Divide by Zero exception.
  - Bit 2 (`0x04`) is for the Overflow exception.
  - Bit 3 (`0x08`) is for the Underflow exception.
  - Bit 4 (`0x10`) is for the Inexact Result exception.
- Bits 8 to 12 (values `0x100` to `0x1000`) control various aspects of the *Floating-Point Architecture* (FPA). The FPA is obsolete and the ARM compilation tools do not support it. Any attempt to write to these bits is ignored.
- Bits 16 to 20 (values `0x10000` to `0x100000`) are the exception masks. These control whether each exception is trapped or not. If a bit is set to 1, the corresponding exception is trapped. If a bit is set to 0, the corresponding exception sets its sticky flag and returns a plausible result.
- Bits 24 to 31 contain the system ID that cannot be changed. It is set to `0x40` for software floating-point, to `0x80` or above for hardware floating-point, and to `0` or `1` if a hardware floating-point environment is being faked by an emulator.
- Bits marked `R` are reserved. They cannot be written to by the `__fp_status()` call, and you must ignore anything you find in them.

The rounding mode cannot be changed with the `__fp_status()` call.

In addition to defining the `__fp_status()` call itself, `stdlib.h` also defines the following constants to be used for the arguments:

```
#define __fpsr_IXE  0x100000
#define __fpsr_UFE  0x80000
#define __fpsr_OFE  0x40000
```

```
#define __fpsr_DZE  0x20000
#define __fpsr_IOE  0x10000
#define __fpsr_IXC  0x10
#define __fpsr_UFC  0x8
#define __fpsr_OFC  0x4
#define __fpsr_DZC  0x2
#define __fpsr_IOC  0x1
```

For example, to trap the Invalid Operation exception and untrap all other exceptions, you would call `__fp_status()` with the following input parameters:

```
__fp_status(_fpsr_IXE | _fpsr_UFE | _fpsr_OFE |
            _fpsr_DZE | _fpsr_IOE, _fpsr_IOE);
```

To untrap the Inexact Result exception:

```
__fp_status(_fpsr_IXE, 0);
```

To clear the Underflow sticky flag:

```
__fp_status(_fpsr_UFC, 0);
```

Related concepts

3.8 Controlling the ARM floating-point environment on page 3-155.

3.8 Controlling the ARM floating-point environment on page 3-155.

Related references

5.5 __ieee_status() on page 5-266.

5.4 gamma(), gamma_r()

The `gamma()` amd `gamma_r()` functions both compute the logarithm of the gamma function. They are synonyms for `lgamma` and `lgamma_r`.

```
double gamma(double x);
double gamma_r(double x, int *);
```

Note

Despite their names, these functions compute the logarithm of the gamma function, not the gamma function itself. To compute the gamma function itself, use `tgamma()`.

Note

If you are migrating from RVCT, be aware that these functions are deprecated in ARM Compiler 4.1 and later.

5.5 __ieee_status()

The ARM compiler toolchain supports an interface to the status word in the floating-point environment. This interface is provided as function `__ieee_status()` and it is generally the most efficient function to use for modifying the status word for VFP.

`__ieee_status()` is defined in `fenv.h`.

The function prototype for `__ieee_status()` is:

```
unsigned int __ieee_status(unsigned int mask, unsigned int flags);
```

Note

This function requires you to select a floating-point model that supports exceptions. For example, `--fpmode=ieee_full` or `--fpmode=ieee_fixed`.

`__ieee_status()` modifies the writable parts of the status word according to the parameters, and returns the previous value of the whole word.

The writable bits are modified by setting them to:

```
new = (old & ~mask) ^ flags;
```

Four different operations can be performed on each bit of the status word, depending on the corresponding bits in mask and flags.

Table 5-2 Status word bit modification

Bit of mask	Bit of flags	Effect
0	0	Leave alone
0	1	Toggle
1	0	Set to 0
1	1	Set to 1

The layout of the status word as seen by `__ieee_status()` is as follows:

31 28	27	26 25	24	23 22	21 20	19	18 16	15 13	12 8	7 5	4 0
R	QC	R	FZ	RM	VFP	R	VFP	R	Masks	R	Sticky

Figure 5-2 IEEE status word layout

The fields in the status word are as follows:

- Bits 0 to 4 (values `0x1` to `0x10`, respectively) are the sticky flags, or cumulative flags, for each exception. The sticky flag for an exception is set to 1 whenever that exception happens and is not trapped. Sticky flags are never cleared by the system, only by the user. The mapping of exceptions to bits is:
  - — bit 0 (`0x01`) is for the Invalid Operation exception
  - — Bit 1 (`0x02`) is for the Divide by Zero exception.
  - — Bit 2 (`0x04`) is for the Overflow exception.
  - — Bit 3 (`0x08`) is for the Underflow exception.
  - — Bit 4 (`0x10`) is for the Inexact Result exception.

- Bits 8 to 12 (values `0x100` to `0x1000`) are the exception masks. These control whether each exception is trapped or not. If a bit is set to 1, the corresponding exception is trapped. If a bit is set to 0, the corresponding exception sets its sticky flag and returns a plausible result.
- Bits 16 to 18, and bits 20 and 21, are used by VFP hardware to control the VFP vector capability. The `__ieee_status()` call does not let you modify these bits.
- Bits 22 and 23 control the rounding mode. See the following table.

Table 5-3 Rounding mode control

Bits	Rounding mode
`00`	Round to nearest
`01`	Round up
`10`	Round down
`11`	Round toward zero

Note

The `fz*`, `fj*` and `f*` library variants support only the round-to-nearest rounding mode. If you require support for the other rounding modes, you must use the full IEEE `g*` libraries. (The relevant compiler options are `--fpmode=std`, `--fpmode=ieee_no_fenv` and `--fpmode=ieee_fixed`.)

- Bit 24 enables FZ (Flush to Zero) mode if it is set. In FZ mode, denormals are forced to zero to speed up processing because denormals can be difficult to work with and slow down floating-point systems. Setting this bit reduces accuracy but might increase speed.

Note

— The FZ bit in the IEEE status word is not supported by any of the `fplib` variants. This means that switching between flushing to zero and not flushing to zero is not possible with any variant of `fplib` at *runtime*. However, flushing to zero or not flushing to zero can be set at compile time as a result of the library you choose to build with.

— Some functions are not provided in hardware. They exist only in the software floating-point libraries. So these functions cannot support the FZ mode, even when you are compiling for a hardware VFP architecture. As a result, behavior of the floating-point libraries is not consistent across all functions when you change the FZ mode dynamically.

- Bit 27 indicates that saturation has occurred in an advanced SIMD saturating integer operation. This is accessible through the `__ieee_status()` call.
- Bits marked R are reserved. They cannot be written to by the `__ieee_status()` call, and you must ignore anything you find in them.

In addition to defining the `__ieee_status()` call itself, `fenv.h` also defines the following constants to be used for the arguments:

```
#define FE_IEEE_FLUSHZERO          (0x01000000)
#define FE_IEEE_ROUND_TONEAREST    (0x00000000)
#define FE_IEEE_ROUND_UPWARD       (0x00400000)
#define FE_IEEE_ROUND_DOWNWARD     (0x00800000)
#define FE_IEEE_ROUND_TOWARDZERO   (0x00C00000)
#define FE_IEEE_ROUND_MASK         (0x00C00000)
#define FE_IEEE_MASK_INVALID       (0x00000100)
#define FE_IEEE_MASK_DIVBYZERO     (0x00000200)
#define FE_IEEE_MASK_OVERFLOW      (0x00000400)
#define FE_IEEE_MASK_UNDERFLOW     (0x00000800)
#define FE_IEEE_MASK_INEXACT       (0x00001000)
```

```
#define FE_IEEE_MASK_ALL_EXCEPT    (0x00001F00)
#define FE_IEEE_INVALID            (0x00000001)
#define FE_IEEE_DIVBYZERO          (0x00000002)
#define FE_IEEE_OVERFLOW           (0x00000004)
#define FE_IEEE_UNDERFLOW          (0x00000008)
#define FE_IEEE_INEXACT            (0x00000010)
#define FE_IEEE_ALL_EXCEPT         (0x0000001F)
```

For example, to set the rounding mode to round down, you would call:

```
__ieee_status(FE_IEEE_ROUND_MASK, FE_IEEE_ROUND_DOWNWARD);
```

To trap the Invalid Operation exception and untrap all other exceptions:

```
__ieee_status(FE_IEEE_MASK_ALL_EXCEPT, FE_IEEE_MASK_INVALID);
```

To untrap the Inexact Result exception:

```
__ieee_status(FE_IEEE_MASK_INEXACT, 0);
```

To clear the Underflow sticky flag:

```
__ieee_status(FE_IEEE_UNDERFLOW, 0);
```

Related concepts

3.8 Controlling the ARM floating-point environment on page 3-155.
3.8 Controlling the ARM floating-point environment on page 3-155.
3.29 Exceptions arising from IEEE 754 floating-point arithmetic on page 3-185.

Related references

3.16 ARM floating-point compiler extensions to the C99 interface on page 3-163.
5.3 __fp_status() on page 5-263.
1.92 C and C++ library naming conventions on page 1-127.

5.6 j0(), j1(), jn(), Bessel functions of the first kind

These functions compute Bessel functions of the first kind.

`j0` and `j1` compute the functions of order 0 and 1 respectively. `jn` computes the function of order *n*.

```
double j0(double x);
double j1(double x);
double jn(int n, double x);
```

If the absolute value of *x* exceeds pi times 2^52, these functions return an `ERANGE` error, denoting total loss of significance in the result.

Note

If you are migrating from RVCT, be aware that these functions are deprecated in ARM Compiler 4.1 and later.

5.7 significand(), fractional part of a number

The `significand()` function returns the fraction part of *x*, as a number between 1.0 and 2.0 (not including 2.0).

```
double significand(double x);
```

Note

If you are migrating from RVCT, be aware that this function is deprecated in ARM Compiler 4.1 and later.

5.8 _statusfp()

Defined in `float.h`, the `_statusfp()` function is provided for compatibility with Microsoft products. It returns the current value of the exception sticky flags.

You can use the `_controlfp()` argument macros, for example `_EM_INVALID` and `_EM_ZERODIVIDE`, to test bits of the returned result.

The function prototype for `_statusfp() is`:

```
unsigned _statusfp(void);
```

Note

This function requires you to select a floating-point model that supports exceptions. For example, `--fpmode=ieee_full` or `--fpmode=ieee_fixed`.

Related concepts

3.9 Floating-point functions for compatibility with Microsoft products on page 3-156.

Related references

5.1 _clearfp() on page 5-260.

5.2 _controlfp() on page 5-261.

5.9 y0(), y1(), yn(), Bessel functions of the second kind

These functions compute Bessel functions of the second kind.

`y0` and `y1` compute the functions of order 0 and 1 respectively. `yn` computes the function of order *n*.

```
double y0(double x);
double y1(double x);
double yn(int, double);
```

If *x* is positive and exceeds pi times 2^52, these functions return an `ERANGE` error, denoting total loss of significance in the result.

Note

If you are migrating from RVCT, be aware that these functions are deprecated in ARM Compiler 4.1 and later.

Appendix A

Libraries Document Revisions

Describes the technical changes that have been made to the ARM C and C++ Libraries and Floating-Point Support User Guide.

It contains the following:

A.1 Revisions for ARM C and C++ Libraries and Floating-Point Support User Guide

The following technical changes have been made to the ARM C and C++ Libraries and Floating-Point Support User Guide.

Table A-1 Differences between issue I and issue J

Change	Topics affected
Added the chapters from the ARM C and C++ Libraries and Floating-Point Support Reference into the ARM C and C++ Libraries and Floating-Point Support User Guide. The ARM C and C++ Libraries and Floating-Point Support Reference is no longer being provided as a separate document.	• *4 The C and C++ Library Functions on page 4-189*. • *5 Floating-point Support Functions on page 5-259*.
Added a topic about using library functions with literal pools.	*1.94 Using library functions with execute-only memory on page 1-131*.

Table A-2 Differences between issue H and issue I

Change	Topics affected
Where appropriate, changed the terminology that implied that 16-bit Thumb and 32-bit Thumb are separate instruction sets.	Various topics.
Clarified the descriptions of signal handling behavior.	• *4.5 __default_signal_handler() on page 4-195*. • *4.19 __raise() on page 4-210*.

Table A-3 Differences between issue G and issue H

Change	Topics affected
Noted the correct use of `_init_alloc(`*`base, top`*`)`.	*1.65 Using a heap implementation from bare machine C on page 1-93*.
Added thread safety information about the Rogue Wave Standard C++ library.	*1.22 Thread safety in the ARM C++ library on page 1-42*.
Corrected the default behavior of `__user_setup_stackheap()`, and reworded the note.	*4.54 __user_setup_stackheap() on page 4-247*.

Table A-4 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.
Explicitly stated that there are two ways to tailor I/O functions, with guidance about use.	*1.70 Tailoring input/output functions in the C and C++ libraries on page 1-99*.

Table A-4 Differences between issue F and issue G (continued)

Change	Topics affected
Made the terms for system I/O functions more consistent, and gave an example of how to redefine the functions.	• *1.70 Tailoring input/output functions in the C and C++ libraries on page 1-99*. • *1.78 Redefining target-dependent system I/O functions in the C library on page 1-110*.
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.	*1.84 ISO-compliant implementation of signals supported by the signal() function in the C library and additional type arguments on page 1-117*.

Table A-5 Differences between issue E and issue F

Change	Topics affected
Changed the ARMCC*nn*LIB environment variable to ARMCC*n*LIB.	*1.7 ARM C and C++ library directory structure on page 1-24*.
Added the values for the arguments to the `SIGFPE` signal.	*1.84 ISO-compliant implementation of signals supported by the signal() function in the C library and additional type arguments on page 1-117*.
Removed `clock()` from the list of unsupported ISO C features.	*2.4 ISO C features missing from microlib on page 2-136*.
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-6 Differences between issue D and issue E

Change	Topics affected
Corrected the library naming conventions.	*1.92 C and C++ library naming conventions on page 1-127*.
Added `#pragma import(__use_smaller_memcpy)`.	*2.2 Differences between microlib and the default C library on page 2-134*.

Table A-7 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.	*1.84 ISO-compliant implementation of signals supported by the signal() function in the C library and additional type arguments on page 1-117*.
Added subtitles to the topic.	*1.54 Definition of locale data blocks in the C library on page 1-78*.

Table A-7 Differences between issue C and issue D (continued)

Change	Topics affected
Restructured the Usage section of the `setlocale()` description.	*4.32 setlocale() on page 4-224*.
Added `lgammaf()`, `lgammal()` to the table of functions that are not thread-safe.	*4.60 C library functions that are not thread-safe on page 4-255*.

Table A-8 Differences between issue B and issue C

Change	Topics affected
Removed `alloca()` from the first sentence.	*1.4 Library heap usage requirements of the ARM C and C++ libraries on page 1-21*.
Removed the item describing `alloca` state from the list.	*1.15 Use of the __user_libspace static data area by the C libraries on page 1-33*.
Reworded the `alloca.h` table entry.	*1.32 Building an application without the C library on page 1-54*.
Reworded the overview of the `alloca()` function.	*4.2 alloca() on page 4-192*.
Reworded the `alloca()` table entry.	*4.59 Thread-safe C library functions on page 4-252*.

Table A-9 Differences between issue A and issue B

Change	Topics affected
New topic.	*1.66 Stack pointer initialization and heap bounds on page 1-94*.
New topic.	*1.67 Defining __initial_sp, __heap_base and __heap_limit on page 1-96*.
New topic.	*1.68 Extending heap size at runtime on page 1-97*.
New topic.	*3.21 mathlib double and single-precision floating-point functions on page 3-174*.
Textual clarification.	*2.8 Entering and exiting programs linked with microlib on page 2-141*.
`__user_setup_stackheap()` clarification.	*1.29 Direct semihosting C library function dependencies on page 1-51*.
`__user_libspace()` is a legacy function.	*1.17 Re-implementation of legacy function __user_libspace() in the C library on page 1-35*.
Reliance of `remquo()`, `remquof()`, and `remquol()`, on implementation-defined integer value.	*1.82 How the ARM C library fulfills ISO C specification requirements on page 1-114*.
`__user_initial_stackheap()` downgraded to legacy support only.	*1.69 Legacy support for __user_initial_stackheap() on page 1-98*.
Topic obsolete.	Memory models and the C library.
Topic obsolete.	Methods of modifying the runtime memory model with the C library.

Table A-9 Differences between issue A and issue B (continued)

Change	Topics affected
Topic obsolete.	User-defined C library memory models.
Topic obsolete.	Selecting the one-region memory model automatically.
Topic obsolete.	Selecting the two-region memory model automatically.
Textual clarifications.	*4.54 __user_setup_stackheap() on page 4-247*.
Textual clarifications.	*4.61 Legacy function __user_initial_stackheap() on page 4-257*.