



Arm[®] Compiler

Version 6.6

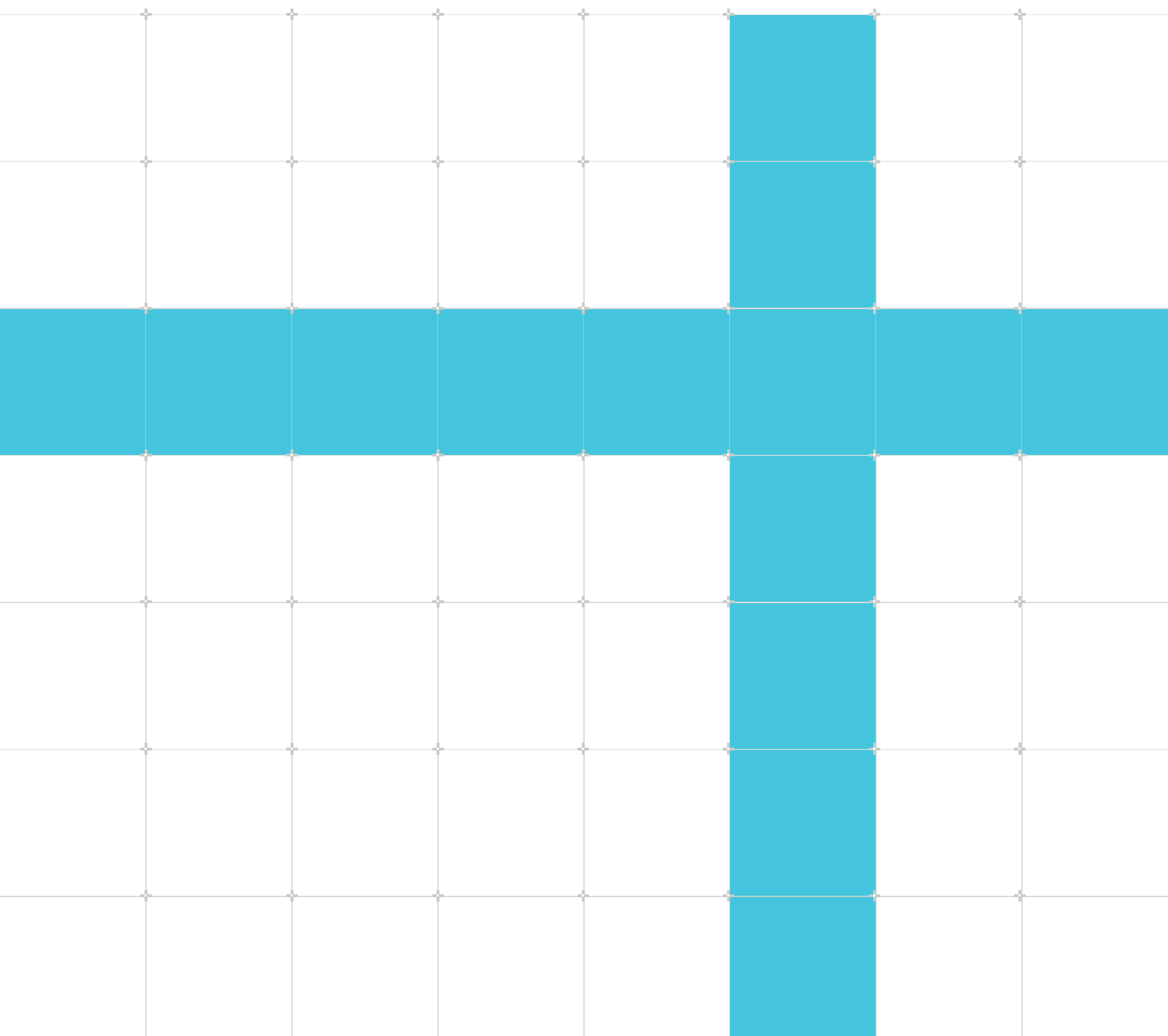
Arm[®] C and C++ Libraries and Floating-Point Support User Guide

Non-Confidential

Copyright © 2014–2017, 2019–2020, 2023 Arm Limited (or its affiliates).
All rights reserved.

Issue

DUI0808_I_en



Arm® Compiler

Arm® C and C++ Libraries and Floating-Point Support User Guide

Copyright © 2014–2017, 2019–2020, 2023 Arm Limited (or its affiliates). All rights reserved.

Release information

Document history

Issue	Date	Confidentiality	Change
A	14 March 2014	Non-Confidential	Arm Compiler v6.00 Release
B	15 December 2014	Non-Confidential	Arm Compiler v6.01 Release
C	30 June 2015	Non-Confidential	Arm Compiler v6.02 Release
D	18 November 2015	Non-Confidential	Arm Compiler v6.3 Release
E	24 February 2016	Non-Confidential	Arm Compiler v6.4 Release
F	29 June 2016	Non-Confidential	Arm Compiler v6.5 Release
G	4 November 2016	Non-Confidential	Arm Compiler v6.6 Release
H	8 May 2017	Non-Confidential	Arm Compiler v6.6.1 Release
I	29 November 2017	Non-Confidential	Arm Compiler v6.6.2 Release
J	28 August 2019	Non-Confidential	Arm Compiler v6.6.3 Release
K	26 August 2020	Non-Confidential	Arm Compiler v6.6.4 Release
L	31 January 2023	Non-Confidential	Arm Compiler v6.6.5 Release

Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED “AS IS”. ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm’s trademark usage guidelines at <https://www.arm.com/company/policies/trademarks>.

Copyright © 2014–2017, 2019–2020, 2023 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

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.

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on <https://support.developer.arm.com>

To provide feedback on the document, fill the following survey: <https://developer.arm.com/documentation-feedback-survey>.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

List of Figures.....	13
List of Tables.....	14

1. Introduction.....	15
1.1 Conventions.....	15
1.2 Other information.....	16
 2. The Arm C and C++ Libraries.....	 17
2.1 Support level definitions.....	17
2.2 Mandatory linkage with the C library.....	21
2.3 C and C++ runtime libraries.....	22
2.3.1 Summary of the C and C++ runtime libraries.....	22
2.3.2 Compliance with the Application Binary Interface (ABI) for the Arm architecture.....	23
2.3.3 Increase portability of object files to other CLIBABI implementations.....	24
2.3.4 Arm C and C++ library directory structure.....	24
2.3.5 Selection of Arm C and C++ library variants based on build options.....	25
2.3.6 T32 C libraries.....	27
2.4 C and C++ library features.....	27
2.5 C++ and C libraries and the std namespace.....	28
2.6 Multithreaded support in Arm C libraries.....	28
2.6.1 Arm C libraries and multithreading.....	28
2.6.2 Arm C libraries and reentrant functions.....	29
2.6.3 Arm C libraries and thread-safe functions.....	29
2.6.4 Use of static data in the C libraries.....	30
2.6.5 Use of the __user_libspace static data area by the C libraries.....	31
2.6.6 C library functions to access subsections of the __user_libspace static data area.....	32
2.6.7 Re-implementation of legacy function __user_libspace() in the C library.....	33
2.6.8 Management of locks in multithreaded applications.....	33
2.6.9 How to ensure re-implemented mutex functions are called.....	35
2.6.10 Using the Arm C library in a multithreaded environment.....	36
2.6.11 Thread safety in the Arm C library.....	37
2.6.12 The floating-point status word in a multithreaded environment.....	38
2.7 Multithreaded support in Arm C++ libraries [ALPHA].....	38
2.7.1 Arm C++ libraries and multithreading [ALPHA].....	38
2.7.2 Clocks [ALPHA].....	40
2.7.3 Mutexes [ALPHA].....	41
2.7.4 Condition variables [ALPHA].....	42
2.7.5 Threads [ALPHA].....	44
2.7.6 Miscellaneous functions [ALPHA].....	47

2.7.7 Thread safety in the Arm C++ library.....	47
2.7.8 Supported C++ Concurrency Features [ALPHA].....	48
2.7.9 Guard variables [ALPHA].....	48
2.7.10 Exceptions [ALPHA].....	49
2.7.11 Thread local storage [ALPHA].....	50
2.7.12 Standard library concurrency constructs [ALPHA].....	50
2.7.13 Thread-safe initialization of Mutexes and Condition variables [ALPHA].....	52
2.8 Support for building an application with the C library.....	53
2.8.1 Using the C library with an application.....	54
2.8.2 Using the C and C++ libraries with an application in a semihosting environment.....	54
2.8.3 Using \$Sub\$\$ to mix semihosted and nonsemihosted I/O functionality.....	55
2.8.4 Using the libraries in a nonsemihosting environment.....	56
2.8.5 Direct semihosting C library function dependencies.....	57
2.8.6 Indirect semihosting C library function dependencies.....	58
2.8.7 C library API definitions for targeting a different environment.....	59
2.9 Support for building an application without the C library.....	60
2.9.1 Standalone C library functions.....	60
2.9.2 Creating an application as bare machine C without the C library.....	63
2.9.3 Integer and floating-point compiler functions and building an application without the C library.....	63
2.9.4 Bare machine integer C.....	64
2.9.5 Bare machine C with floating-point processing.....	64
2.9.6 Customized C library startup code and access to C library functions.....	65
2.9.7 Using low-level functions when exploiting the C library.....	66
2.9.8 Using high-level functions when exploiting the C library.....	66
2.9.9 Using malloc() when exploiting the C library.....	67
2.10 Tailoring the C library to a new execution environment.....	67
2.10.1 Initialization of the execution environment and execution of the application.....	67
2.10.2 C++ initialization, construction and destruction.....	68
2.10.3 Exceptions system initialization.....	69
2.10.4 Library functions called from main().....	70
2.10.5 Program exit and the assert macro.....	70
2.11 Assembler macros that tailor locale functions in the C library.....	71
2.11.1 Link time selection of the locale subsystem in the C library.....	72
2.11.2 Runtime selection of the locale subsystem in the C library.....	73
2.11.3 Definition of locale data blocks in the C library.....	74

2.11.4 LC_CTYPE data block.....	75
2.11.5 LC_COLLATE data block.....	78
2.11.6 LC_MONETARY data block.....	79
2.11.7 LC_NUMERIC data block.....	80
2.11.8 LC_TIME data block.....	80
2.12 Modification of C library functions for error signaling, error handling, and program exit.....	82
2.13 Stack and heap memory allocation and the Arm C and C++ libraries.....	82
2.13.1 Library heap usage requirements of the Arm C and C++ libraries.....	83
2.13.2 Choosing a heap implementation for memory allocation functions.....	84
2.13.3 Stack pointer initialization and heap bounds.....	86
2.13.4 Legacy support for __user_initial_stackheap().....	89
2.13.5 Avoiding the heap and heap-using library functions supplied by Arm.....	89
2.14 Tailoring input/output functions in the C and C++ libraries.....	90
2.15 Target dependencies on low-level functions in the C and C++ libraries.....	91
2.16 The C library printf family of functions.....	93
2.17 The C library scanf family of functions.....	93
2.18 Redefining low-level library functions to enable direct use of high-level library functions in the C library.....	94
2.19 The C library functions fread(), fgets() and gets().....	96
2.20 Re-implementing __backspace() in the C library.....	97
2.21 Re-implementing __backspacewc() in the C library.....	98
2.22 Redefining target-dependent system I/O functions in the C library.....	99
2.23 Tailoring non-input/output C library functions.....	100
2.24 Real-time integer division in the Arm libraries.....	100
2.25 ISO C library implementation definition.....	101
2.25.1 How the Arm C library fulfills ISO C specification requirements.....	101
2.25.2 mathlib error handling.....	102
2.25.3 ISO-compliant implementation of signals supported by the signal() function in the C library and additional type arguments.....	103
2.25.4 ISO-compliant C library input/output characteristics.....	104
2.25.5 Standard C++ library implementation definition.....	106
2.26 C library functions and extensions.....	109
2.27 Avoid linking in the Arm C library.....	110
2.28 C and C++ library naming conventions.....	112
2.29 Using macro __ARM_WCHAR_NO_IO to disable FILE declaration and wide I/O function prototypes.....	116
2.30 Using library functions with execute-only memory.....	116

3. The Arm C Micro-library.....	117
3.1 About microlib.....	117
3.2 Differences between microlib and the default C library.....	117
3.3 Library heap usage requirements of microlib.....	119
3.4 ISO C features missing from microlib.....	120
3.5 Building an application with microlib.....	121
3.6 Configuring the stack and heap for use with microlib.....	122
3.7 Entering and exiting programs linked with microlib.....	124
3.8 Tailoring the microlib input/output functions.....	124
 4. Floating-point Support.....	 126
4.1 About floating-point support.....	126
4.2 Controlling the Arm floating-point environment.....	127
4.2.1 Floating-point functions for compatibility with Microsoft products.....	127
4.2.2 C99-compatible functions for controlling the Arm floating-point environment.....	128
4.2.3 C99 rounding mode and floating-point exception macros.....	129
4.2.4 Exception flag handling.....	129
4.2.5 Functions for handling rounding modes.....	131
4.2.6 Functions for saving and restoring the whole floating-point environment.....	131
4.2.7 Functions for temporarily disabling exceptions.....	132
4.2.8 Arm floating-point compiler extensions to the C99 interface.....	133
4.2.9 Example of a custom exception handler.....	134
4.2.10 Exception trap handling by signals.....	135
4.3 mathlib double and single-precision floating-point functions.....	136
4.4 IEEE 754 arithmetic.....	137
4.4.1 Basic data types for IEEE 754 arithmetic.....	137
4.4.2 Single precision data type for IEEE 754 arithmetic.....	137
4.4.3 Double precision data type for IEEE 754 arithmetic.....	139
4.4.4 Sample single precision floating-point values for IEEE 754 arithmetic.....	140
4.4.5 Sample double precision floating-point values for IEEE 754 arithmetic.....	141
4.4.6 IEEE 754 arithmetic and rounding.....	142
4.4.7 Exceptions arising from IEEE 754 floating-point arithmetic.....	143
4.4.8 Exception types recognized by the Arm floating-point environment.....	144
 5. The C and C++ Library Functions Reference.....	 146
5.1 __aeabi_errno_addr().....	146
5.2 alloca().....	146

5.3 clock()	147
5.4 _clock_init()	148
5.5 __default_signal_handler()	148
5.6 errno	149
5.7 _findlocale()	150
5.8 _fsatty()	150
5.9 _get_lconv()	151
5.10 getenv()	152
5.11 _getenv_init()	152
5.12 __heapstats()	153
5.13 __heapvalid()	154
5.14 lconv structure	155
5.15 localeconv()	156
5.16 _membitcpybl(), _membitcpybb(), _membitcpyhl(), _membitcpyhb(), _membitcpywl(), _membitcpywb(), _membitmovebl(), _membitmovebb(), _membitmovehl(), _membitmovehb(), _membitmovewl(), _membitmovewb()	157
5.17 _platform_pre_stackheap_init()	158
5.18 posix_memalign()	159
5.19 __raise()	160
5.20 _rand_r()	161
5.21 remove()	162
5.22 rename()	162
5.23 __rt_entry	163
5.24 __rt_exit()	163
5.25 __rt_fp_status_addr()	164
5.26 __rt_heap_extend()	164
5.27 __rt_lib_init()	165
5.28 __rt_lib_shutdown()	166
5.29 __rt_raise()	167
5.30 __rt_stackheap_init()	168
5.31 setlocale()	168
5.32 _srand_r()	169
5.33 strcasecmp()	170
5.34 strlcat()	170
5.35 strlcpy()	171
5.36 strncasecmp()	171

5.37	_sys_close()	172
5.38	_sys_command_string()	172
5.39	_sys_ensure()	173
5.40	_sys_exit()	173
5.41	_sys_flen()	174
5.42	_sys_istty()	174
5.43	_sys_open()	175
5.44	_sys_read()	176
5.45	_sys_seek()	176
5.46	_sys_tmpnam()	177
5.47	_sys_write()	177
5.48	system()	178
5.49	time()	178
5.50	_ttywrch()	179
5.51	__user_heap_extend()	179
5.52	__user_heap_extnt()	180
5.53	__user_setup_stackheap()	181
5.54	__vectab_stack_and_reset	182
5.55	wcscasecmp()	183
5.56	wcsncasecmp()	183
5.57	wcstombs()	184
5.58	Thread-safe C library functions	184
5.59	C library functions that are not thread-safe	186
5.60	Legacy function __user_initial_stackheap()	189
6.	Floating-point Support Functions Reference	191
6.1	_clearfp()	191
6.2	_controlfp()	191
6.3	__fp_status()	193
6.4	gamma(), gamma_r()	195
6.5	__ieee_status()	195
6.6	j0(), j1(), jn(), Bessel functions of the first kind	199
6.7	significand(), fractional part of a number	199
6.8	_statusfp()	199
6.9	y0(), y1(), yn(), Bessel functions of the second kind	200
7.	Arm C and C++ Libraries and Floating-Point Support User Guide Changes	201

7.1 Changes for the Arm C and C++ Libraries and Floating-Point Support User Guide.....	201
--	-----

List of Figures

Figure 2-1: Integration boundaries in Arm Compiler for Embedded 6.....	19
Figure 4-1: IEEE 754 single-precision floating-point format.....	138
Figure 4-2: IEEE 754 double-precision floating-point format.....	139
Figure 6-1: Floating-point status word layout.....	193
Figure 6-2: IEEE status word layout.....	196

List of Tables

Table 2-1: C library callouts.....	31
Table 2-4: Direct semihosting dependencies.....	57
Table 2-5: Indirect semihosting dependencies.....	58
Table 2-6: Published API definitions.....	59
Table 2-7: Standalone C library functions.....	60
Table 2-8: Default ISO8859-1 locales.....	72
Table 2-9: Default Shift-JIS and UTF-8 locales.....	73
Table 2-10: Trap and error handling.....	82
Table 2-12: Signals supported by the signal() function.....	103
Table 2-13: perror() messages.....	105
Table 2-14: C library extensions.....	109
Table 4-1: Sample single-precision floating-point values.....	140
Table 4-2: Sample double-precision floating-point values.....	141
Table 5-1: Functions that are thread-safe.....	184
Table 5-2: Functions that are not thread-safe.....	186
Table 6-1: _controlfp argument macros.....	192
Table 6-2: Status word bit modification.....	196
Table 6-3: Rounding mode control.....	197
Table 7-1: Changes between 6.6.5 (revision L) and 6.6.4 (revision K).....	201
Table 7-2: Changes between 6.6.4 (revision K) and 6.6.3 (revision J).....	201

1. Introduction

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. It is also available as a PDF.

1.1 Conventions

The following subsections describe conventions used in Arm documents.

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: developer.arm.com/glossary.

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
<i>italic</i>	Citations.
bold	Interface elements, such as menu names. Terms in descriptive lists, where appropriate.
monospace	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
monospace <u>underline</u>	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
<and>	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: <pre>MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2></pre>
SMALL CAPITALS	Terms that have specific technical meanings as defined in the <i>Arm® Glossary</i> . For example, IMPLEMENTATION DEFINED , IMPLEMENTATION SPECIFIC , UNKNOWN , and UNPREDICTABLE .
 Caution	Recommendations. Not following these recommendations might lead to system failure or damage.
 Warning	Requirements for the system. Not following these requirements might result in system failure or damage.
 Danger	Requirements for the system. Not following these requirements will result in system failure or damage.

Convention	Use
 Note	An important piece of information that needs your attention.
 Tip	A useful tip that might make it easier, better or faster to perform a task.
 Remember	A reminder of something important that relates to the information you are reading.

1.2 Other information

See the Arm website for other relevant information.

- [Arm® Developer](#).
- [Arm® Documentation](#).
- [Technical Support](#).
- [Arm® Glossary](#).

2. The Arm C and C++ Libraries

Describes the Arm® C and C++ libraries.

2.1 Support level definitions

This describes the levels of support for various Arm® Compiler 6 features.

Arm Compiler 6 is built on Clang and LLVM technology. Therefore, it has more functionality than the set of product features described in the documentation. The following definitions clarify the levels of support and guarantees on functionality that are expected from these features.

Arm welcomes feedback regarding the use of all Arm Compiler 6 features, and intends to support users to a level that is appropriate for that feature. You can contact support at <https://developer.arm.com/support>.

Identification in the documentation

All features that are documented in the Arm Compiler 6 documentation are product features, except where explicitly stated. The limitations of non-product features are explicitly stated.

Product features

Product features are suitable for use in a production environment. The functionality is well tested, and is expected to be stable across feature and update releases.

- Arm intends to give advance notice of significant functionality changes to product features.
- If you have a support and maintenance contract, Arm provides full support for use of all product features.
- Arm welcomes feedback on product features.
- Any issues with product features that Arm encounters or is made aware of are considered for fixing in future versions of Arm Compiler.

In addition to fully supported product features, some product features are only alpha or beta quality.

Beta product features

Beta product features are implementation complete, but have not been sufficiently tested to be regarded as suitable for use in production environments.

Beta product features are identified with [BETA].

- Arm endeavors to document known limitations on beta product features.
- Beta product features are expected to eventually become product features in a future release of Arm Compiler 6.
- Arm encourages the use of beta product features, and welcomes feedback on them.

- Any issues with beta product features that Arm encounters or is made aware of are considered for fixing in future versions of Arm Compiler.

Alpha product features

Alpha product features are not implementation complete, and are subject to change in future releases, therefore the stability level is lower than in beta product features.

Alpha product features are identified with [ALPHA].

- Arm endeavors to document known limitations of alpha product features.
- Arm encourages the use of alpha product features, and welcomes feedback on them.
- Any issues with alpha product features that Arm encounters or is made aware of are considered for fixing in future versions of Arm Compiler.

Community features

Arm Compiler 6 is built on LLVM technology and preserves the functionality of that technology where possible. This means that there are more features available in Arm Compiler that are not listed in the documentation. These extra features are known as community features. For information on these community features, see the [Clang Compiler User's Manual](#).

Where community features are referenced in the documentation, they are identified with [COMMUNITY].

- Arm makes no claims about the quality level or the degree of functionality of these features, except when explicitly stated in this documentation.
- Functionality might change significantly between feature releases.
- Arm makes no guarantees that community features remain functional across update releases, although changes are expected to be unlikely.

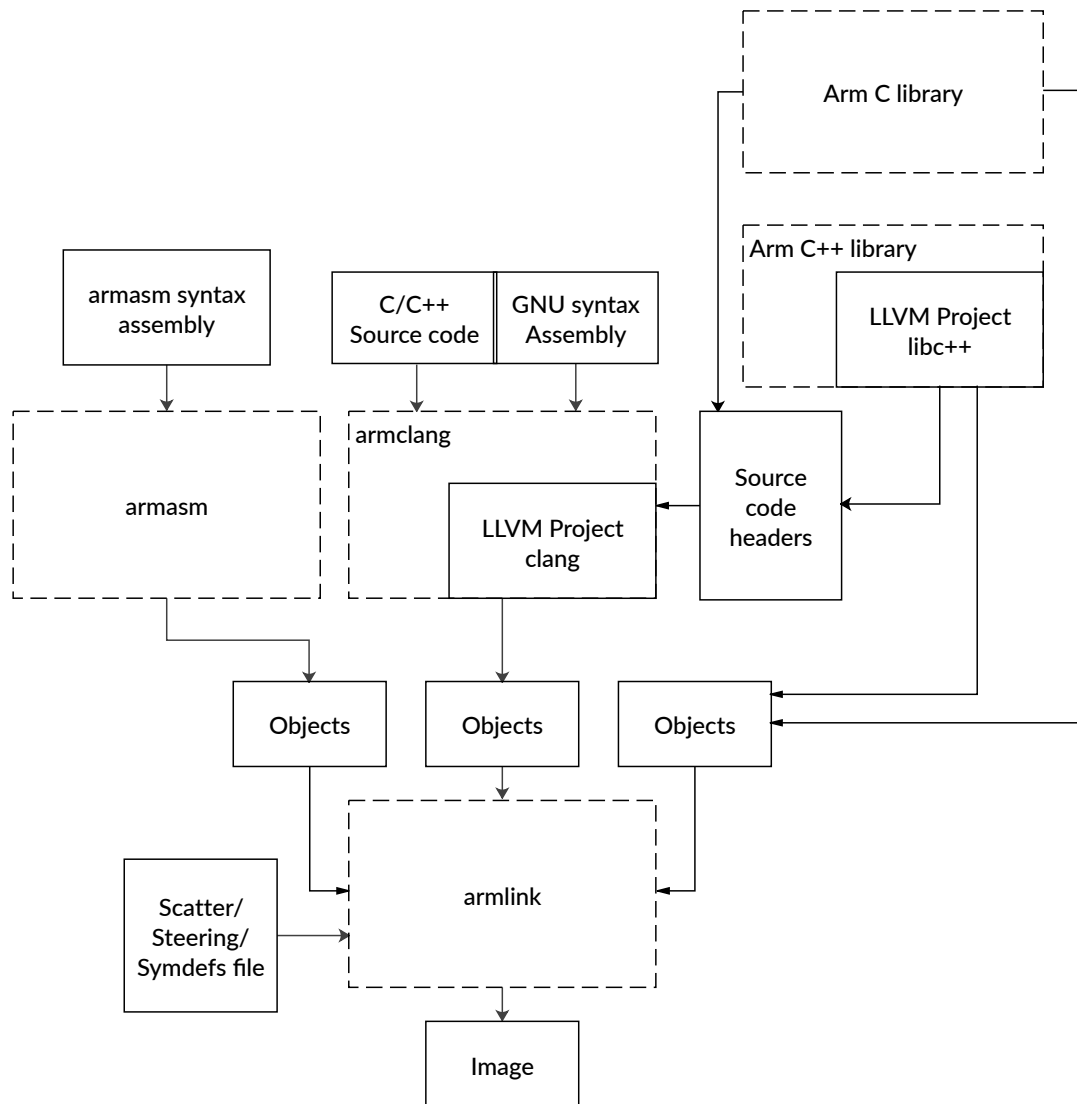
Some community features might become product features in the future, but Arm provides no roadmap for such features. Arm is interested in understanding your use of these features, and welcomes feedback on them. Arm supports customers using these features on a best-effort basis, unless the features are unsupported. Arm accepts defect reports on these features, but does not guarantee that these issues are to be fixed in future releases.

Guidance on use of community features

There are several factors to consider when assessing the likelihood of a community feature being functional:

- The following figure shows the structure of the Arm Compiler 6 toolchain:

Figure 2-1: Integration boundaries in Arm Compiler for Embedded 6.



The dashed boxes are toolchain components, and any interaction between these components is an integration boundary. Community features that span an integration boundary might have significant limitations in functionality. The exception to such features is if the interaction is codified in one of the standards supported by Arm Compiler 6. See [Application Binary Interface \(ABI\)](#). Community features that do not span integration boundaries are more likely to work as expected.

- Features primarily used when targeting hosted environments such as Linux or BSD might have significant limitations, or might not be applicable, when targeting bare-metal environments.

- The Clang implementations of compiler features, particularly those features that have been present for a long time in other toolchains, are likely to be mature. The functionality of new features, such as support for new language features, is likely to be less mature and therefore more likely to have limited functionality.

Deprecated features

A deprecated feature is one that Arm plans to remove from a future release of Arm Compiler. Arm does not make any guarantee regarding the testing or maintenance of deprecated features. Therefore, Arm does not recommend using a feature after it is deprecated.

For information on replacing deprecated features with supported features, see the Arm Compiler documentation and Release Notes. Where appropriate, each Arm Compiler document includes notes for features that are deprecated, and also provides entries in the changes appendix of that document.

Unsupported features

With both the product and community feature categories, specific features and use-cases are known not to function correctly, or are not intended for use with Arm Compiler 6.

Limitations of product features are stated in the documentation. Arm cannot provide an exhaustive list of unsupported features or use-cases for community features. The known limitations on community features are listed in [Community features](#).

List of known unsupported features

The following is an incomplete list of unsupported features, and might change over time:

- The Clang option `-stdlib=libstdc++` is not supported.
- C++ static initialization of local variables is not thread-safe when linked against the standard C++ libraries. For thread-safety, you must provide your own implementation of thread-safe functions as described in [Standard C++ library implementation definition](#).



This restriction does not apply to the [ALPHA]-supported multithreaded C++ libraries.

-
- Use of C11 library features is unsupported.
 - Any community feature that is exclusively related to non-Arm architectures is not supported.
 - Except for Armv6-M, compilation for targets that implement architectures lower than Armv7 is not supported.
 - The `long double` data type is not supported for AArch64 state because of limitations in the current Arm C library.
 - C complex arithmetic is not supported, because of limitations in the current Arm C library.
 - Complex numbers are defined in C++ as a template, `std::complex`. Arm Compiler supports `std::complex` with the `float` and `double` types, but not the `long double` type because of limitations in the current Arm C library.



For C code that uses complex numbers, it is not sufficient to recompile with the C++ compiler to make that code work. How you can use complex numbers depends on whether you are building for Armv8-M architecture-based processors.

- You must take care when mixing translation units that are compiled with and without the [COMMUNITY] `-fsigned-char` option, and that share interfaces or data structures.



The Arm ABI defines `char` as an unsigned byte, and this is the interpretation used by the C libraries supplied with the Arm compilation tools.

Alternatives to C complex numbers not being supported

If you are building for Armv8-M architecture-based processors, consider using the free and Open Source CMSIS-DSP library that includes a data type and library functions for complex number support in C. For more information about CMSIS-DSP and complex number support see the following sections of the CMSIS documentation:

- [Complex Math Functions](#)
- [Complex Matrix Multiplication](#)
- [Complex FFT Functions](#)

If you are not building for Armv8-M architecture-based processors, consider modifying the affected part of your project to use the C++ standard template library type `std::complex` instead.

2.2 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 information

[Summary of the C and C++ runtime libraries](#) on page 22

[Standalone C library functions](#) on page 60

2.3 C and C++ runtime libraries

Arm provides the C standardlib, C microlib, and C++ runtime libraries to support compiled C and C++.

2.3.1 Summary of the C and C++ runtime libraries

A summary of the C and C++ runtime libraries provided by Arm.

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 libc++ library.

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



Arm does not guarantee the compatibility of C++ compilation units compiled with different major or minor versions of Arm® Compiler and linked into a single image. Therefore, Arm recommends that you always build your C++ code from source with a single version of the toolchain.

You can mix C++ with C code or C libraries.

Related information

[Mandatory linkage with the C library](#) on page 21
[Standalone C library functions](#) on page 60
[The Arm C and C++ Libraries](#) on page 17
[The Arm C Micro-library](#) on page 117
[Standard C++ library implementation definition](#) on page 106
[ISO C library standard](#)
[IEEE Standard for Floating-Point Arithmetic \(IEEE 754\), 1985 version](#)

2.3.2 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 several supplemental specifications, for example the *Arm GNU/Linux ABI supplement*.

The *Base Standard ABI for the Arm Architecture* (BSABI) enables you to use A32 and T32 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 and the CLIBABI. The libc++ library conforms to the *C++ ABI for the Arm Architecture* (CPPABI), except for Array Construction and Delete helper functions.



Note

All C++ compilation units that are to be linked into a single image must be compiled with the same version of the C++ standard library ABI. If the ABI version changes between Arm Compiler releases, then you must recompile your object files.

If you are unable to recompile some of your object files, then contact Arm Support at <https://developer.arm.com/support>.

Related information

[Increase portability of object files to other CLIBABI implementations](#) on page 24
[Standard C++ library implementation definition](#) on page 106

Application Binary Interface (ABI) DWARF Debugging Standard

2.3.3 Increase 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.



This reduces the performance of some library operations.

You can use the following methods to request full CLIBABI portability

- 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 information

[Compliance with the Application Binary Interface \(ABI\) for the Arm architecture](#) on page 23
[Application Binary Interface \(ABI\)](#)

2.3.4 Arm C and C++ library directory structure

The libraries are installed in the `armlib` and `libcxx` subdirectories within `install_directorylib`.

armlib

Contains the variants of the Arm® C library, the floating-point arithmetic library (fplib), and the math library (mathlib).

libcxx

Contains all `libc++` and `libc++abi` libraries.

The accompanying header files for these libraries are installed in:

```
install_directory\include
```

To specify an alternative top-level `lib` directory, set either one of the environment variables `ARMCOMPILER6LIB` or `ARMLIB`, to point to the new directory, or use the `--libpath` option.

You must not identify the `armlib` and `libcxx` directories separately because the directory structure might change in future releases. The linker finds them from the location of `lib`.



- 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 libc++ and libc++abi libraries provided with Arm Compiler 6 are based on the open source libc++ and libc++abi libraries. The modifications made by Arm are covered by restrictions described in the end user license agreement.

2.3.5 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

A32 or T32 (AArch32 state instruction sets).



Microlib is not supported for AArch64 state.

Byte order

Big-endian or little-endian.

Floating-point support

- Software (SoftVFP).
- Hardware (VFP).
- Software or hardware with half-precision or double-precision extensions.
- No floating-point support.



In Armv8, VFP hardware is integral to the architecture. Software floating-point is supported for AArch32 state, but is not supported for AArch64 state.

Position independence

Position independent code uses PC-relative addressing modes where possible and otherwise accesses global data through the *Global Offset Table* (GOT).

Different ways to access your data are as follows:

- By absolute address.
- Relative to `sb` (read/write position-independent).
- Relative to `pc` (`-fbare-metal-pie`).

Different ways to access your code are as follows:

- By absolute address when appropriate.
- Relative to `pc` (read-only position independent).

A bare-metal *Position Independent Executable* (PIE) is an executable that does not need to be executed at a specific address but can be executed at any suitably aligned address.

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

You can only achieve position-independent C++ code with `-fbare-metal-pie`.



- Position independence is supported only in AArch32 state, and is not supported in microlib.
 - Bare-metal PIE support is deprecated in this release.
-

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

[-fropi, fnoropi \[BETA\] compiler option](#)

[-frwpi, fnorwpi \[BETA\] option](#)

[-marm compiler option](#)

[-mfpu compiler option](#)

[-mthumb compiler option](#)

[--fpu=name linker option](#)

[--ropi linker option](#)

[--rwpi linker option](#)

[--arm assembler option](#)

[--fpu assembler option](#)

[--thumb assembler option](#)

2.3.6 T32 C libraries

There are several variations of the T32 libraries. It depends on the architecture target or processor as to which one is used.

Arm®v7-A and Armv7-R use a T32 library. It contains a small number of A32 instructions that are used to significantly improve performance.

Armv7-R, Armv7E-M, and Armv8-R.mainline have their own T32 library.

Armv6-M and Armv8-R.baseline have their own T32 library.

2.4 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 DSTREAM debug and trace unit, the Arm RVI debug unit, and the *Fixed Virtual Platform* (FVP) models.

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

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

- The `malloc` family.
- The `ctype` family.
- All the locale-specific functions.

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

Functions in the C library are responsible for:

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

2.5 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");
```



Note

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

2.6 Multithreaded support in Arm C libraries

Describes the features that are supported by the Arm C libraries for creating multithreaded applications.

2.6.1 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 information

[Using the Arm C library in a multithreaded environment](#) on page 36

[Arm C libraries and reentrant functions](#) on page 29

[Arm C libraries and thread-safe functions](#) on page 29

2.6.2 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.



A reentrant function must not call non-reentrant functions.

Related information

[Arm C libraries and thread-safe functions](#) on page 29

[Arm C libraries and multithreading](#) on page 28

2.6.3 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 information

[Arm C libraries and reentrant functions](#) on page 29

[Arm C libraries and multithreading](#) on page 28

2.6.4 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 `-fnorwpi`. This is the default.
- At a fixed address relative to the static base, register `r9`, when compiled with `-frwpi`.
- At a fixed address relative to the program counter (`pc`), when compiled with `-fbare-metal-pie`.



Bare-metal PIE support is deprecated in this release.

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 `-fnorwpi`, 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 `-frwpi`, 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 `tz_*` and `ej_*` do not use static data and are always reentrant. For software floating-point, the `fx_*` and `gx_*` libraries use static data to store the Floating-Point (FP) status word. For hardware floating-point, the `fx_*` and `gx_*` libraries do not 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, `-fnorwpi`.
 - Reentrant build options, `-frwpi`.
- Some functions have static data implicit in their definitions. You must not use these in a reentrant application unless you build it with `-frwpi` and the callers use different values in `sb`.



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 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 2-1: C library callouts

Function	Description
<code>__rt_errno_addr()</code>	Called to get the address of the variable <code>errno</code>
<code>__rt_fp_status_addr()</code>	Called by the floating-point support code to get the address of the floating-point status word
locale functions	The function <code>__user_libspace()</code> 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.



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

Related information

[Re-implementation of legacy function `\_\_user\_libspace\(\)` in the C library](#) on page 33

[Assembler macros that tailor locale functions in the C library](#) on page 71

[Arm C libraries and multithreading](#) on page 28

[__rt_fp_status_addr\(\)](#) on page 164

[-fropi, -fnoropi option](#)

[-frwpi, -fnorwpi option](#)

2.6.5 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 several different types of data.

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.



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

Related information

[__aeabi_atexit\(\) in C++ ABI for the Arm Architecture](#)

2.6.6 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 memory for storing data that is global to an entire process. This data is shared between all threads.

In AArch32 state, returns a pointer to 96 bytes of 4-byte aligned memory.

In AArch64 state, returns a pointer to 192 bytes of 8-byte aligned memory.

`__user_perthread_libspace()`

Returns a pointer to memory 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.

In AArch32 state, returns a pointer to 96 bytes of 4-byte aligned memory.

In AArch64 state, returns a pointer to 192 bytes of 8-byte aligned memory.

Related information

[Use of the `\_\_user\_libspace` static data area by the C libraries](#) on page 31

[Re-implementation of legacy function `\_\_user\_libspace\(\)` in the C library](#) on page 33

2.6.7 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 information

[C library functions to access subsections of the `\_\_user\_libspace` static data area](#) on page 32

2.6.8 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 pointer-sized 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 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 AArch32 and 8 bytes of memory for AArch64. This memory holds the mutex data structure. `__Heap_Initialize()` is one such function.

For the C library, a mutex is specified as a single pointer-sized 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, ...);  
}
```



- `_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 information

[How to ensure re-implemented mutex functions are called](#) on page 35

[Using the Arm C library in a multithreaded environment](#) on page 36

[Thread safety in the Arm C library](#) on page 37

[Thread safety in the Arm C++ library](#) on page 47

2.6.9 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 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 information

[Using the Arm C library in a multithreaded environment](#) on page 36

[Thread safety in the Arm C library](#) on page 37

[Thread safety in the Arm C++ library](#) on page 47

[Management of locks in multithreaded applications](#) on page 33

2.6.10 Using the Arm C library in a multithreaded environment

There are several requirements you must fulfill 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.



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

- A thread-switching mechanism.



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 information

[Arm C libraries and multithreading](#) on page 28

2.6.11 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 information

[How to ensure re-implemented mutex functions are called](#) on page 35

[Using the Arm C library in a multithreaded environment](#) on page 36

[Thread safety in the Arm C++ library](#) on page 47

[Management of locks in multithreaded applications](#) on page 33

2.6.12 The floating-point status word in a multithreaded environment

Applicable to variants of the software floating-point libraries that require a status word, 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.



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.

In Arm® Compiler 6, floating-point library variants are selected by default. For more information see the `armclang` command-line option `-ffp-mode`.

Related information

[Thread safety in the Arm C library](#) on page 37

2.7 Multithreaded support in Arm C++ libraries [ALPHA]

Describes the features that the Arm C++ libraries support for creating multithreaded applications. These features are [ALPHA]-supported.



This topic describes an [ALPHA] feature. See [Support level definitions](#).

2.7.1 Arm C++ libraries and multithreading [ALPHA]

The C++ Thread Porting *Application Programming Interface* (API) is an [ALPHA]-supported API that enables the use of C++11 concurrency constructs with Arm® Compiler 6. Operating system or library vendors must provide an implementation of this API to enable the seamless use of C++11 concurrency constructs within user applications.



This topic describes an [ALPHA] feature. See [Support level definitions](#).

The C++11 standard offers several high-level concurrency constructs intended to simplify parallel programming and make multithreaded programs portable across platforms. Future versions of

the C++ standard are set to introduce additional high-level concurrency constructs. For more information, see <https://isocpp.org/std/status>.

Most standard library implementations expect an underlying operating system or library platform to provide a comprehensive set of primitive concurrency constructs on top of which these higher level constructs can be built.

The default standard C++ library supplied with Arm Compiler 6 has been built without concurrency support to avoid passing these dependencies to target bare-metal systems.

Arm Compiler 6 includes a special variant of the standard C++ library that enables support for C++11 concurrency constructs. This library variant requires platform vendors to provide an implementation of the threading API described in this document. On certain architectures, for example the Armv6-M architecture, this threaded library variant might contain library calls to various `__atomic_*` functions. On these architectures, platform vendors must also provide an implementation of an atomics library as discussed in [LLVM Atomic Instructions and Concurrency Guide](#).

To select the threaded standard C++ library variant instead of the default variant use the compiler options `-std=c++11 -D_ARM_LIBCPP_EXTERNAL_THREADS` together with linker option `--stdlib=threaded_libc++`.

Platform vendors can selectively implement subsets of the porting API based on the dependencies between the high-level C++11 concurrency constructs and the underlying platform concurrency primitives.

Arm Compiler 6 provides C++ libraries that are based on open source LLVM technology. The libraries for multithreaded applications are:

- The standard library, `libc++`.
- Low level support for the standard library, `libc++abi`.
- Exception unwinding support library, `libunwind`.

The C++ Thread Porting API enables these libraries to correctly operate in different multithreaded environments.



The C++ Thread Porting API is independent of the multithreaded support API provided in the Arm C library. For more information, see [Multithreaded support in Arm C libraries](#). Platform vendors must implement both of these APIs, to the respective specifications, to ensure correct operation of multithreaded programs. The C++ Thread Porting API is declared in the `<arm-tp1.h>` header file of the Arm Compiler 6 distribution.

The C++ Thread Porting API functional areas include:

- Clocks.
- Mutexes.

- Conditional variables.
- Threads.
- Miscellaneous functions.

2.7.2 Clocks [ALPHA]

The C++ Thread Porting API provides clock functions in the `<arm-tp1.h>` header file.



This topic describes an [ALPHA] feature. See [Support level definitions](#).

Types

```
struct timespec {  
    time_t tv_sec;  
    unsigned long tv_nsec;  
};
```

Functions

```
int __ARM_TPL_clock_realtime(timespec* ts);
```

```
int __ARM_TPL_clock_monotonic(timespec* ts);
```

Usage

The function `__ARM_TPL_clock_realtime()` must populate the argument with the current system-wide (wall-clock) time.

The function `__ARM_TPL_clock_monotonic()` must populate the argument with the elapsed time since some fixed point in time.



Time measurements produced by `__ARM_TPL_clock_monotonic()` must be steady, in that, those measurements must increase at a fixed rate relative to the real time.

Returns

These functions must return zero if successful, or return non-zero if not successful to indicate an error.

2.7.3 Mutexes [ALPHA]

The C++ Thread Porting API provides mutex functions in the `<arm-tpl.h>` header file.



This topic describes an [ALPHA] feature. See [Support level definitions](#).

Types

```
struct __ARM_TPL_mutex_t {
    _Atomic uintptr_t data;
};
```

Functions

```
int __ARM_TPL_recursive_mutex_init(__ARM_TPL_mutex_t* __m);
```

```
int __ARM_TPL_mutex_lock(__ARM_TPL_mutex_t* __m);
```

```
int __ARM_TPL_mutex_trylock(__ARM_TPL_mutex_t* __m);
```

```
int __ARM_TPL_mutex_unlock(__ARM_TPL_mutex_t* __m);
```

```
int __ARM_TPL_mutex_destroy(__ARM_TPL_mutex_t* __m);
```

Usage

The API uses the `__ARM_TPL_mutex_t` type to encapsulate a pointer to an underlying platform-specific mutex type. The semantics of these functions are:

- The functions `__ARM_TPL_mutex_lock()` and `__ARM_TPL_mutex_trylock()` must operate on an initialize-on-first-use basis with respect to `__m->data`. If the value `__m->data` is zero, an implementation must first initialize `__m->data` to point to a valid platform mutex before carrying out the requested locking operation. This initialization must be thread-safe. For more information, see [Thread-safe initialization of Mutexes and Condition variables \[ALPHA\]](#).
- The function `__ARM_TPL_mutex_lock()` must lock the mutex represented at `__m`, blocking the calling thread until the mutex becomes available. If the function is successful, it must return zero, with the calling thread as the owner of the underlying mutex. If the function is not successful, it must return non-zero.
- The function `__ARM_TPL_mutex_trylock()` is similar to `__ARM_TPL_mutex_lock()`, except if the mutex at `__m` is already locked, it must return immediately unsuccessfully. If the function successfully performs the lock, it must return zero. Otherwise, it must return non-zero.
- The function `__ARM_TPL_mutex_unlock()` and `__ARM_TPL_mutex_destroy()` must return zero if the value `__m->data` is zero. Otherwise, they must perform the requested operation on the

platform mutex pointed to by `__m->data` and then return zero if successful, or return non-zero if not successful.

- The function `__ARM_TPL_mutex_unlock()` must unlock the mutex represented at `__m`. If the mutex at `__m` was initialized as a recursive mutex, it is unlocked only when the lock count reaches zero. This function must return zero if successful, or return non-zero if not successful.
- The function `__ARM_TPL_mutex_destroy()` must destroy the mutex represented at `__m`. It is guaranteed that an already destroyed `__ARM_TPL_mutex_t` object is not re-referenced through any API functions afterward. `__ARM_TPL_mutex_destroy()` must return zero if successful, or return non-zero if not successful.
- The function `__ARM_TPL_recursive_mutex_init()` must initialize the platform mutex pointed to by `__m->data` as a recursive mutex. There is no requirement for this initialization to be thread-safe. This function must return zero if successful, or non-zero if not successful.

Returns

These functions must return zero if successful, or return non-zero if not successful to indicate an error.

2.7.4 Condition variables [ALPHA]

The C++ Thread Porting API provides functions for condition variables in the `<arm-tp1.h>` header file.



This topic describes an [ALPHA] feature. See [Support level definitions](#).

Types

```
struct __ARM_TPL_condvar_t {
    _Atomic uintptr_t data;
};
```

Functions

```
int __ARM_TPL_condvar_signal(__ARM_TPL_condvar_t* __cv);
```

```
int __ARM_TPL_condvar_broadcast(__ARM_TPL_condvar_t* __cv);
```

```
int __ARM_TPL_condvar_wait(__ARM_TPL_condvar_t* __cv, __ARM_TPL_mutex_t* __m);
```

```
int __ARM_TPL_condvar_timedwait(__ARM_TPL_condvar_t* __cv, __ARM_TPL_mutex_t* __m,
    timespec* __ts);
```

```
int __ARM_TPL_condvar_destroy(__ARM_TPL_condvar_t* __cv);
```

Usage

The C++ Thread Porting API uses the `__ARM_TPL_condvar_t` type to encapsulate a pointer to an underlying platform-specific condition variable type. The semantics of the functions are:

- The functions `__ARM_TPL_condvar_wait()` and `__ARM_TPL_condvar_timedwait()` must operate on an initialize-on-first-use basis with respect to `__cv->data`. If the value `__cv->data` is zero, an implementation must first initialize `__cv->data` to point to a valid platform condition variable before carrying out the requested operation. This initialization must be thread-safe. For more information, see [Thread-safe initialization of Mutexes and Condition variables \[ALPHA\]](#).
- The function `__ARM_TPL_condvar_wait()` must cause the calling thread (which is guaranteed to be the owner of the mutex `__m`) to block until the condition variable, `__cv`, is signaled by a different thread or the calling thread is interrupted. For the duration where the calling thread is blocked, the mutex `__m` must be unlocked. When this function returns, the unblocked thread must be the owner of the mutex, `__m`, regardless of the reason for unblocking. The reason for unblocking might be:
 - Condition variable is signaled.
 - The current thread is interrupted.

This function must return zero to indicate success when a thread is unblocked as a result of the condition variable being signaled. This function must return a non-zero value to indicate any error conditions.

- The function `__ARM_TPL_condvar_timedwait()` behaves similar to `__ARM_TPL_condvar_wait()`, except that it allows an explicit time limit to be specified for the blocking operation. If this function returns due to the timeout expiring, its return value shall be non-zero.
- The functions `__ARM_TPL_condvar_signal()`, `__ARM_TPL_condvar_broadcast()`, and `__ARM_TPL_condvar_destroy()` must return zero if the value `__cv->data` is zero. Otherwise, they must perform the requested operation on the platform condition variable pointed to by `__cv->data` and return zero if successful, or return non-zero if not successful.
- The function `__ARM_TPL_condvar_signal()` must unblock at least one of the threads blocked on the condition variable `__cv`. The function `__ARM_TPL_condvar_broadcast()` must unblock all the threads blocked on the condition variable `__cv`. If more than one thread is unblocked as a result of a call to one of these functions, they must all contend for the respective mutexes with which they originally invoked `__ARM_TPL_condvar_wait()` or `__ARM_TPL_condvar_timedwait()` functions.
- The function `__ARM_TPL_condvar_destroy()` must destroy the condition variable represented in `__cv`. It is guaranteed that an already destroyed `__ARM_TPL_condvar_t` object is not referenced through any API functions afterward. When successful, `__ARM_TPL_condvar_destroy()` must return zero, or a non-zero value if not successful.

Returns

These functions must return zero if successful, or return non-zero if not successful to indicate an error.

2.7.5 Threads [ALPHA]

The C++ Thread Porting API provides thread function prototypes in the `<arm-tp1.h>` header file.



This topic describes an [ALPHA] feature. See [Support level definitions](#).

Types

```
typedef uint32_t __ARM_TPL_thread_id;

struct __ARM_TPL_thread_t {
    uintptr_t data;
};

typedef uint32_t __ARM_TPL_tls_key;
```

Functions

```
int __ARM_TPL_thread_create(__ARM_TPL_thread_t* __t, void* (*__f)(void*), void*
__arg);
```

```
__ARM_TPL_thread_id __ARM_TPL_thread_get_current_id();
```

```
__ARM_TPL_thread_id __ARM_TPL_thread_get_id(const __ARM_TPL_thread_t* __t);
```

```
int __ARM_TPL_thread_id_compare(__ARM_TPL_thread_id t1, __ARM_TPL_thread_id t2);
```

```
int __ARM_TPL_thread_join(__ARM_TPL_thread_t* __t);
```

```
int __ARM_TPL_thread_detach(__ARM_TPL_thread_t* __t);
```

```
void __ARM_TPL_thread_yield();
```

```
void __ARM_TPL_thread_nanosleep(const timespec *req, timespec *rem);
```

```
unsigned __ARM_TPL_thread_hw_concurrency();
```

```
int __ARM_TPL_tls_create(__ARM_TPL_tls_key* __key, void (*__at_exit)(void*));
```

```
void __ARM_TPL_tls_set(__ARM_TPL_tls_key __key, void* __p);
```

```
void* __ARM_TPL_tls_get(__ARM_TPL_tls_key __key);
```

Usage

The C++ Thread Porting API uses the `__ARM_TPL_thread_t` type to encapsulate a pointer to an underlying platform-specific thread type. The types `__ARM_TPL_thread_id` and `__ARM_TPL_tls_key` are identifiers of threads and instances of thread local storage created within the system. The semantics of the functions are:

- The `__ARM_TPL_thread_create()` function must initialize `__t->data` to point to a newly allocated system thread structure. There is no requirement for this initialization to be thread safe. The newly allocated thread must be scheduled to execute the `__f` routine with `__arg` as its sole argument.
- The function `__ARM_TPL_thread_get_current_id()` must return the thread identifier for the calling thread.
- The function `__ARM_TPL_thread_get_id()` must return the corresponding thread identifier for the argument `__t`.

- The function `__ARM_TPL_thread_id_compare()` must return positive if `t1 > t2`, zero if `t1 == t2`, and negative if `t1 < t2`.
- The function `__ARM_TPL_thread_join()` must cause the calling thread to block until the thread represented in `__t` terminates. This function must return zero (success) immediately if `__t` has already terminated. If the argument `__t` does not represent a joinable thread or refers to the calling thread itself, this function must return a non-zero value to indicate error. When a thread `__t` has been joined to, it is guaranteed not to be accessed again. Therefore, any system resources accessible through `__t` must be reclaimed before returning from this function. All threads are created in a joinable state, calling either `__ARM_TPL_thread_join()` or `__ARM_TPL_thread_detach()` on an argument `__t` makes the thread represented in `__t` non-joinable.
- The function `__ARM_TPL_thread_detach()` must cleanup any system resources accessible through `__t` while allowing the underlying thread to continue execution. As with `__ARM_TPL_thread_join()`, invoking this function on an argument `__t` causes the underlying thread to become non-joinable. This function must return zero on success or non-zero if not successful.
- The function `__ARM_TPL_thread_yield()` must force the calling thread to relinquish the processor until it becomes eligible for execution again.
- The function `__ARM_TPL_thread_nanosleep()` must cause the calling thread to be blocked for a minimum interval specified by `req`. The thread might be interrupted due to a signal being delivered to it, in which case either the corresponding signal handler must be invoked or the process must be terminated. When the argument `rem` is provided (non-null), and the function returns before having the requested time interval elapsed, `rem` must be populated to indicate the remaining time interval of the original request (time requested - actual time elapsed).
- The function `__ARM_TPL_thread_hw_concurrency()` must return the number of concurrent threads supported by the underlying platform. The sole use of this function is for the implementation of the `std::hardware_concurrency()` function.
- The `__ARM_TPL_tls_create()` function must initialize `*__key` to identify a unique process-wide thread local storage. Upon creation, each `__key` must be bound to a NULL value, as if `__ARM_TPL_tls_set(*__key, NULL)` was invoked. Individual threads within the current process might later bind thread specific values to this key using the `__ARM_TPL_tls_set()` function. There is no requirement for the initialization of `*__key` to be thread safe. If the `__at_exit` argument is provided (non-null), and a thread has a non-null binding for that `*__key` at the point of termination, the system must ensure that `__at_exit(void*)` is invoked with the current binding for `*__key` as the sole argument.
- The function `__ARM_TPL_tls_set()` must associate the value `__p` with `*__key` for the calling thread. It is guaranteed that `__key` has been obtained using `__ARM_TPL_tls_create()`. It is also guaranteed that `__ARM_TPL_tls_set()` is not invoked from thread-local destructor functions registered in `__ARM_TPL_tls_create()`. This function must return zero if successful or a non-zero value if not successful.
- The function `__ARM_TPL_tls_get()` must return the value bound to the `__key` argument for the calling thread. It is guaranteed that `__ARM_TPL_tls_get()` is not invoked from thread-local destructor functions registered in `__ARM_TPL_tls_create()`.

Returns

These functions must return zero if successful, or return non-zero to indicate an error if not successful.

2.7.6 Miscellaneous functions [ALPHA]

The C++ Thread Porting API provides functions in the `<arm-tp1.h>` header file.



This topic describes an [ALPHA] feature. See [Support level definitions](#).

Types

```
typedef volatile unsigned long __ARM_TPL_exec_once_flag;
```

Function

```
void __ARM_TPL_execute_once(__ARM_TPL_exec_once_flag *__flag, void(*__func)(void));
```

Usage

The first invocation of the `__ARM_TPL_execute_once()` function by any thread within the current process for a given `__func` argument must result in a call to the `__func()` routine. Subsequent calls to `__ARM_TPL_execute_once()` for the same `__func` argument must not have any effect. The argument `__flag` can be used to determine whether the routine `__func` has already been invoked or not.

Returns

This function must return `void`.

2.7.7 Thread safety in the Arm C++ library

The functions contained within the `libc++` library are fully-supported for use in single-threaded environments, and [ALPHA]-supported for use in multithreaded environments.



This topic includes descriptions of [ALPHA] features. See [Support level definitions](#).

The C++ Thread Porting API enables support for the C++11 concurrency constructs in the Arm® Compiler 6 C++ library. This implies that the C++ library as a whole is exposed to multithreaded environments, and users must consider the overall thread safety aspects of the library. The thread

safety provided by the C++ Thread Porting API applies to the use of shared global data within Arm Compiler 6 C++ libraries. The default C++ libraries of Arm Compiler 6 are not thread safe. They are only intended to be used in single-threaded environments. The [ALPHA]-supported multithreaded C++ libraries are guaranteed to be thread safe if the C++ Thread Porting API is implemented and the necessary steps are taken to ensure that the Arm Compiler 6 C libraries are also thread safe. For more information, see [Multithreaded support in Arm C libraries](#).

Related information

[How to ensure re-implemented mutex functions are called](#) on page 35

[Using the Arm C library in a multithreaded environment](#) on page 36

[Thread safety in the Arm C library](#) on page 37

[Management of locks in multithreaded applications](#) on page 33

2.7.8 Supported C++ Concurrency Features [ALPHA]

The following sections identify the high-level C++ concurrency constructs supported by the multithreaded Arm® C++ libraries. For each of the features, the underlying section of the thread porting API required for the correct functionality of that feature is also identified.



This topic describes an [ALPHA] feature. See [Support level definitions](#).

Some language features require extra runtime support when operating in a multithreaded environment. The C++ language features of concern here are initialization of guard variables and exceptions.

2.7.9 Guard variables [ALPHA]

To ensure thread-safety of certain initializations, the compiler calls out to helper functions in the libcplusplus library.



This topic describes an [ALPHA] feature. See [Support level definitions](#).

For example, in the following code snippet, the compiler must ensure that the Counter `c` is constructed exactly once, even when multiple threads call `getCount()`.

```
class Counter {
public:
    Counter(int x) : _m(x) {}
    int inc() {return _m++;}
private:
```

```
int _m;
}

int getCount() {
    static Counter c(42);
    return c.inc();
}
```

To support such thread-safe initializations, a platform vendor must provide the implementations for the following subset of constructs from the [Mutexes \[ALPHA\]](#) and [Condition variables \[ALPHA\]](#) sections of the API:

Types

```
struct __ARM_TPL_mutex_t{
    __Atomic uintptr_t data;
};
struct __ARM_TPL_condvar_t{
    __Atomic uintptr_t data;
};
```

Functions

```
int __ARM_TPL_mutex_lock(__ARM_TPL_mutex_t* __m);
```

```
int __ARM_TPL_mutex_unlock(__ARM_TPL_mutex_t* __m);
```

```
int __ARM_TPL_condvar_broadcast(__ARM_TPL_condvar_t* __cv);
```

```
int __ARM_TPL_condvar_wait(__ARM_TPL_condvar_t* __cv, __ARM_TPL_mutex_t* __m);
```

The ABI functions `__cxa_guard_acquire()`, `__cxa_guard_release()`, and `__cxa_guard_abort()` need not be re-implemented under this scheme.

2.7.10 Exceptions [ALPHA]

The C++ runtime (libc++abi and libunwind) must take special measures when allocating and handling exceptions in a threaded environment.



This topic describes an [ALPHA] feature. See [Support level definitions](#).

An implementation of the following subset of the thread porting API is required for the correct operation of exceptions in such an environment:

Types

```
struct __ARM_TPL_mutex_t{
```

```
_Atomic uintptr_t data;
};
typedef uint32_t __ARM_TPL_tls_key;
```

Functions

```
int __ARM_TPL_mutex_lock(__ARM_TPL_mutex_t* __m);
```

```
int __ARM_TPL_mutex_unlock(__ARM_TPL_mutex_t* __m);
```

```
int __ARM_TPL_tls_create(__ARM_TPL_tls_key* __key, void (*__at_exit) (void*));
```

```
void* __ARM_TPL_tls_get(__ARM_TPL_tls_key __key);
```

```
void __ARM_TPL_tls_set(__ARM_TPL_tls_key __key, void* __p);
```

```
void __ARM_TPL_call_once(volatile unsigned long& flag, void* arg, void(*func)
(void*));
```

2.7.11 Thread local storage [ALPHA]

The thread storage duration specifier of C++, `thread_local`, is not supported.



This topic describes an [ALPHA] feature. See [Support level definitions](#).

2.7.12 Standard library concurrency constructs [ALPHA]

The C++ standard library, beginning with the C++11 standard, provides various high-level concurrency constructs. Presently, these constructs are spread across the headers `<atomic>`, `<chrono>`, `<mutex>`, `<shared_mutex>`, `<condition_variable>`, `<thread>`, and `<future>`.



This topic describes an [ALPHA] feature. See [Support level definitions](#).

The following sections identify how the functionality of each of these headers maps to the Arm® Compiler thread porting API introduced in [Arm C++ libraries and multithreading \[ALPHA\]](#). They also identify any additional dependencies or expected limitations.

<atomic>

The functionality of this header does not depend on the thread porting API.

The following table summarizes the level of support for the <atomic> header on various Arm architectures.

Architecture	T (Template parameter)	atomic<T>	atomic<T*>
Armv7-A, Armv7-R, Armv8-A, Armv8-R	Any types	Supported	Supported
Armv7-M, Armv8-M	Integral types (including <stdint.h> defined types)	Supported	Supported
Armv7-M, Armv8-M	Complex types	Unsupported	Supported
Armv6-M	Any types	Unsupported	Unsupported

None of the targets support the functions `atomic_thread_fence()` and `atomic_signal_fence()`.

<chrono>

This header requires a full implementation of the [Clocks \[ALPHA\]](#) section of the thread porting API.

<mutex>, <shared_mutex>

These headers require a full implementation of the [Mutexes \[ALPHA\]](#) section of the thread porting API. In addition, the time-related subset of constructs (for example, `std::timed_mutex` and `std::recursive_timed_mutex`) defined in these headers requires an implementation of the [Clocks \[ALPHA\]](#) section of the porting API.

<condition_variable>

This header requires a full implementation of the [Condition variables \[ALPHA\]](#) section of the thread porting API. In addition, the time-related subset of constructs (for example, `std::condition_variable::wait_for()`) defined in this header requires an implementation of the [Clocks \[ALPHA\]](#) section of the porting API.

<thread>, <future>

These headers require a full implementation of the [Threads \[ALPHA\]](#) section of the thread porting API. In addition, the time-related subset of constructs (for example, `std::thread::sleep_until()` and `std::future::wait_for()`) defined in these headers requires an implementation of the [Clocks \[ALPHA\]](#) section of the porting API.

2.7.13 Thread-safe initialization of Mutexes and Condition variables [ALPHA]

The Mutexes and Condition Variable parts of the porting API must adopt an initialize-on-first-use strategy. Implementations must ensure that such initializations are thread-safe.



This topic describes an [ALPHA] feature. See [Support level definitions](#).

Consider the following sample implementation of the `__ARM_TPL_mutex_lock()` function:

```
// [1] Include the header for your operating system, which defines a
// platform-specific API for mutexes. The names below were created for this
// example only.
#include <platform.h>

// Assume that platform.h declares the following types and functions:
// struct platform_mutex_t;
// platform_mutex_t *alloc_platform_mutex();
// void lock_platform_mutex(platform_mutex_t *p);
// void unlock_platform_mutex(platform_mutex_t *p);
// void destroy_platform_mutex(platform_mutex_t *p);

// [2] Include the Arm TPL header, which defines the functions that you must
// implement.
#include <arm-tpl.h>

// [3] Implement the Arm TPL functions according to the API that your operating
// system provides.

void __ARM_TPL_mutex_lock(__ARM_TPL_mutex_t* __m) {
    if (__m->data == 0) {
        __m->data = static_cast<uintptr_t>(alloc_platform_mutex());
    }
    lock_platform_mutex(reinterpret_cast<platform_mutex_t*>(__m->data));
}
```

The anatomy of this snippet can be understood as follows:

1. Assume the underlying system (included through `platform.h`) provides the type `__platform_mutex_t` and the functions `alloc_platform_mutex()`, `lock_platform_mutex()`, `unlock_platform_mutex()`, and `destroy_platform_mutex()`.
2. The porting API header (`arm-tpl.h`) is then included, which defines the type `__ARM_TPL_mutex_t` and the prototypes for the various porting API functions.
3. The implementations of the various porting API functions follow.

This implementation of `__ARM_TPL_mutex_lock()` method leads to a race condition if multiple threads attempt to lock the same `std::mutex` object. Therefore, an implementation must ensure that `__m->data` initializes atomically. The following sections illustrate possible solutions to this problem.

Global locking

An implementation may employ a platform provided mutex to guard the initialization of `*__m` as follows:

```
static platform_mutex_t guard_mut;

void __ARM_TPL_mutex_lock(__ARM_TPL_mutex_t* __m) {
    volatile __ARM_TPL_mutex_t* __vm = __m;
    if (__vm->data == 0) {
        lock_platform_mutex(&guard_mut);
        if (__vm->data == 0)
            __vm->data = static_cast<uintptr_t>(alloc_platform_mutex());
        unlock_platform_mutex(&guard_mut);
    }
    lock_platform_mutex(static_cast<platform_mutex_t*>(__vm));
}
```



This solution could result in reduced performance because threads must contend for the shared mutex `guard_mut` for each initial `std::mutex` lock operation.

Lock free

An implementation avoiding global locking is achievable using the lock-free concurrency constructs available through the `<stdatomic.h>` header. The following snippet atomically attempts to initialize `__m->data`, and undo its attempt if another thread has already done the initialization:

```
#include <stdint>
#include <stdatomic.h>

int __ARM_TPL_mutex_lock(__ARM_TPL_mutex_t* __m) {
    if (__m->data == 0) {
        uintptr_t mut_new =
            reinterpret_cast<uintptr_t>(alloc_platform_mutex());
        uintptr_t mut_null = 0;
        if (!atomic_compare_exchange_strong(&__m->data, &mut_null, mut_new))
            destroy_platform_mutex(reinterpret_cast<platform_mutex_t*>(mut_new));
        return lock_platform_mutex(reinterpret_cast<platform_mutex_t*>(__m->data));
    }
}
```

The Arm®v6-M architecture does not support this method.

2.8 Support for building an application with the C library

Describes the Arm® Compiler features that are supported when building an application with the C library.

2.8.1 Using the C library with an application

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

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

- 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 information

[Using the C and C++ libraries with an application in a semihosting environment](#) on page 54

[Using the libraries in a nonsemihosting environment](#) on page 56

[Standalone C library functions](#) on page 60

2.8.2 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 A32 or T32 semihosting trap instructions for AArch32 state or A64 semihosting trap instruction for AArch64 state.

The execution environment can be provided by either:

- Using the standard semihosting functionality that is present by default in, for example, the Arm DSTREAM debug and trace unit.
- 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.

Arm Compiler supports semihosting by generating trap instructions such as `HLT`, `SVC`, or `BKPT` depending on the architecture or profile. Debug agents can trap these instructions to perform semihosting operations on the host.

Architecture	Instruction Set	Trap Instruction
Armv8-A and Armv8-R	A64	HLT 0xF000
Armv8-A and Armv8-R	A32	HLT 0xF000
		SVC 0x123456
Armv8-A and Armv8-R	T32	HLT 0x3C
		SVC 0xAB
Armv7-A and Armv7-R	A32	HLT 0xF000

Architecture	Instruction Set	Trap Instruction
		SVC 0x123456
Armv7-A and Armv7-R	T32	HLT 0x3C SVC 0xAB
Any architecture with M-profile	T32	BKPT 0xAB

For AArch32 in architectures with A-profile or R-profile, Arm Compiler supports two different semihosting implementations:

- Semihosting using the `svc` instruction. This is the default and legacy implementation.
- Semihosting using the `HLT` instruction. This implementation is required for semihosting in hardware debug environments with mixed AArch32 and AArch64 states.

There are separate libraries for SVC-based and HLT-based semihosting. Arm Compiler uses the HLT-based semihosting library if your code references the symbol `__use_hlt_semihosting`. To do this, either:

- `IMPORT __use_hlt_semihosting` from assembly language.
- `__asm(".global __use_hlt_semihosting\n\t")` from C.

If you do not use the symbol `__use_hlt_semihosting`, then by default, Arm Compiler emits `svc` instructions for semihosting calls. This symbol does not have an effect on M-profile architectures, or in AArch64 state.

Arm strongly discourages mixing `HLT` and `svc` semihosting mechanisms within the same executable. The library only uses either `svc` or `HLT` instructions, rather than a mixture. However, you must ensure that you do not mix `svc` and `HLT` instructions when using:

- inline assembly.
- `<arm_compat.h>` header file.

Related information

[Using `\$Sub\$\$` to mix semihosted and nonsemihosted I/O functionality](#) on page 55

[Direct semihosting C library function dependencies](#) on page 57

2.8.3 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:

```
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 information

[Using the C and C++ libraries with an application in a semihosting environment](#) on page 54

2.8.4 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 `armasm` assembly language.
- `__asm(".global __use_no_semihosting\n\t")` for C or C++ code.



`IMPORT __use_no_semihosting` is only required to be added to a single assembly source file. Similarly, `__asm(".global __use_no_semihosting\n\t")` 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. For example, to determine which objects are using semihosting when using an Arm®v8-R processor:

1. Link with `armlink --cpu=8-M --verbose --list err.txt`
2. Search `err.txt` for occurrences of `__I$use$semihosting`

For example:

```
...
Loading member sys_exit.o from c_2.1.
reference : __I$use$semihosting
definition: _sys_exit
...
```

This example shows that the semihosting-using function `_sys_exit` is linked-in from the C library. To prevent the C library being linked-in, 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 information

[Using \\$Sub\\$\\$ to mix semihosted and nonsemihosted I/O functionality](#) on page 55

[Mandatory linkage with the C library](#) on page 21

2.8.5 Direct semihosting C library function dependencies

A table showing the functions that depend directly on semihosting.

Table 2-4: Direct semihosting dependencies

Function	Description
<code>__user_initial_stackheap()</code>	Sets up and returns the locations of the stack and the heap. If you are using a scatter file at the link stage, you might have to re-implement this function. The linker issues an error when no semihosting is requested and <code>__user_initial_stackheap()</code> is not re-implemented.
<code>_sys_exit()</code>	Error signaling, error handling, and program exit.
<code>_ttywrch()</code>	

Function	Description
<code>_sys_command_string()</code> <code>_sys_close()</code> <code>_sys_iserror()</code> <code>_sys_istty()</code> <code>_sys_flen()</code> <code>_sys_open()</code> <code>_sys_read()</code> <code>_sys_seek()</code> <code>_sys_write()</code> <code>_sys_tmpnam()</code>	Tailoring input/output functions in the C and C++ libraries.
<code>clock()</code> <code>_clock_init()</code> <code>remove()</code> <code>rename()</code> <code>system()</code> <code>time()</code>	Tailoring other C library functions.

Related information

[Using the C and C++ libraries with an application in a semihosting environment](#) on page 54

2.8.6 Indirect semihosting C library function dependencies

A table showing 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:

- `__asm(".global __use_no_semihosting\n\t")` in C source code.
- `IMPORT __use_no_semihosting` in `armasm` assembly language source code.

Table 2-5: Indirect semihosting dependencies

Function	Usage
<code>__user_setup_stackheap()</code>	Sets up and returns the locations of the stack and the heap.

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

Related information

[Direct semihosting C library function dependencies](#) on page 57

2.8.7 C library API definitions for targeting a different environment

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

The following table shows these functions and files.

Table 2-6: Published API definitions

File or function	Description
<code>__main()</code> , <code>__rt_entry()</code>	Initializes the runtime environment and executes the user application
<code>__rt_lib_init()</code> , <code>__rt_exit()</code> , <code>__rt_lib_shutdown()</code>	Initializes or finalizes the runtime library
<code>LC_CTYPE</code> <code>locale</code>	Defines the character properties for the local alphabet
<code>rt_sys.h</code>	A C header file describing all the functions whose default (semihosted) implementations use semihosting calls
<code>rt_heap.h</code>	A C header file describing the storage management abstract data type
<code>rt_locale.h</code>	A C header file describing the five locale category <i>fling systems</i> , and defining some macros that are useful for describing the contents of locale categories
<code>rt_misc.h</code>	A C header file describing miscellaneous unrelated public interfaces to the C library
<code>rt_memory.s</code>	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.



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 information

[--list=filename linker option](#)

[--verbose linker option](#)

2.9 Support for building an application without the C library

Describes the Arm® Compiler features that are supported and not supported when building an application without the C library.

2.9.1 Standalone C library functions

If your application does not initialize the C library, several 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.

Table 2-7: Standalone C library functions

Function	Description
<code>alloca.h</code>	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.
<code>assert.h</code>	Functions listed in this file require high-level <code>stdio</code> , <code>__rt_raise()</code> , and <code>_sys_exit()</code> . You must be familiar with tailoring error signaling, error handling, and program exit to use this header file.
<code>ctype.h</code>	Functions listed in this file require the locale functions.

Function	Description
<code>errno.h</code>	Functions in this file work without the requirement for any library initialization or function re-implementation.
<code>fenv.h</code>	Functions in this file work without the requirement for any library initialization and only require the re-implementation of <code>__rt_raise()</code> .
<code>float.h</code>	This file does not contain any code. The definitions in the file do not require library initialization or function re-implementation.
<code>inttypes.h</code>	Functions listed in this file require the locale functions.
<code>limits.h</code>	Functions in this file work without the requirement for any library initialization or function re-implementation.
<code>locale.h</code>	Call <code>setlocale()</code> before calling any function that uses locale functions. For example: <pre>setlocale(LC_ALL, "C");</pre> See the contents of <code>locale.h</code> for more information on the following functions and data structures: <code>setlocale()</code> selects the appropriate locale as specified by the category and locale arguments. <code>lconv</code> is the structure used by locale functions for formatting numeric quantities according to the rules of the current locale. <code>localeconv()</code> creates an <code>lconv</code> structure and returns a pointer to it. <code>_get_lconv()</code> fills the <code>lconv</code> structure pointed to by the parameter. This ISO extension removes the requirement for static data within the library. <code>locale.h</code> also contains constant declarations used with locale functions.
<code>math.h</code>	For functions in this file to work, you must first call <code>_fp_init()</code> and re-implement <code>__rt_raise()</code> .
<code>setjmp.h</code>	Functions in this file work without any library initialization or function re-implementation.
<code>signal.h</code>	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. <code>__rt_raise()</code> can be re-implemented for error and exit handling. You must be familiar with tailoring error signaling, error handling, and program exit.
<code>stdarg.h</code>	Functions listed in this file work without any library initialization or function re-implementation.
<code>stddef.h</code>	This file does not contain any code. The definitions in the file do not require library initialization or function re-implementation.
<code>stdint.h</code>	This file does not contain any code. The definitions in the file do not require library initialization or function re-implementation.

Function	Description
stdio.h	The following dependencies or limitations apply to these functions: - The high-level functions such as <code>printf()</code>, <code>scanf()</code>, <code>puts()</code>, <code>fgets()</code>, <code>fread()</code>, <code>fwrite()</code>, and <code>perror()</code> depend on lower-level stdio functions <code>fgetc()</code>, <code>fputc()</code>, and <code>__backspace()</code>. You must re-implement these lower-level functions when using the standalone C library. However, you cannot re-implement the <code>_sys_</code> prefixed functions (for example, <code>_sys_read()</code>) when using the standalone C library because the layer of <code>stdio</code> that calls the <code>_sys_</code> functions requires library initialization. You must be familiar with tailoring the input/output functions in the C and C++ libraries. - The <code>printf()</code> and <code>scanf()</code> family of functions require locale. - The <code>remove()</code> and <code>rename()</code> functions are system-specific and probably not usable in your application.
stdlib.h	Most functions in this file work without any library initialization or function re-implementation. The following functions depend on other functions being instantiated correctly: <code>ato*()</code> requires locale. <code>strto*()</code> requires locale. <code>malloc()</code>, <code>calloc()</code>, <code>realloc()</code>, and <code>free()</code> require heap functions. <code>atexit()</code> is not available when building an application without the C library.
string.h	Functions in this file work without any library initialization, except for <code>strcoll()</code> and <code>strxfrm()</code>, that require locale.
time.h	<code>mktime()</code> and <code>localtime()</code> can be used immediately <code>time()</code> and <code>clock()</code> are system-specific and are probably not usable unless re-implemented <code>asctime()</code>, <code>ctime()</code>, and <code>strftime()</code> require locale.
wchar.h	Wide character library functions added to ISO C by <i>Normative Addendum 1</i> in 1994. The following dependencies or limitations apply to these functions: - The high-level functions such as <code>swprintf()</code>, <code>vswprintf()</code>, <code>swscanf()</code>, and <code>vswscanf()</code> depend on lower-level stdio functions such as <code>fgetwc()</code> and <code>fputwc()</code>. You must re-implement these lower-level functions when using the standalone C library. See Target dependencies on low-level functions in the C and C++ libraries for more information. - The high-level functions such as <code>swprintf()</code>, <code>vswprintf()</code>, <code>swscanf()</code>, and <code>vswscanf()</code> require locale. - All the conversion functions (for example, <code>btowc</code>, <code>wctob</code>, <code>mbtowc</code>, and <code>wcrtomb</code>) require locale. <code>wscoll()</code> and <code>wcsxfrm()</code> require locale.
wctype.h	Wide character library functions added to ISO C by <i>Normative Addendum 1</i> in 1994. This requires locale.

Related information

[Creating an application as bare machine C without the C library](#) on page 63

[Assembler macros that tailor locale functions in the C library](#) on page 71

[Tailoring input/output functions in the C and C++ libraries](#) on page 90

[Modification of C library functions for error signaling, error handling, and program exit](#) on page 81

[Integer and floating-point compiler functions and building an application without the C library](#) on page 63

[Using high-level functions when exploiting the C library](#) on page 66

[Using low-level functions when exploiting the C library](#) on page 66

2.9.2 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 information

[Standalone C library functions](#) on page 60

2.9.3 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. (Arm®v7-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 information

[Creating an application as bare machine C without the C library](#) on page 63

2.9.4 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 several 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.
- For AArch32 targets, build your application using `-mfpu=none`. For AArch64 targets, use `-mcpu` or `-march` to disable floating-point instructions and registers.

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 information

[Creating an application as bare machine C without the C library](#) on page 63

2.9.5 Bare machine C with floating-point processing

If you want to use floating-point processing in an application without the C library, there are several requirements you must fulfill.

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.

The register state 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 `-mfpu=none` option.

Certain floating-point modes when used with software floating-point support require a floating-point status word. This is enabled by default in Arm® Compiler 6, but you can disable it with the `armclang` command-line option `-ffp-mode=fast`. In such cases, 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 information

[Creating an application as bare machine C without the C library](#) on page 63

2.9.6 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`.
- `armclang` uses full IEEE math by default, therefore `__rt_fp_status_addr()` is always required.
- 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 information

[Creating an application as bare machine C without the C library](#) on page 63

2.9.7 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.



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



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 information

[Using high-level functions when exploiting the C library](#) on page 66

[Standalone C library functions](#) on page 60

2.9.8 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 information

[Using low-level functions when exploiting the C library](#) on page 66

[Standalone C library functions](#) on page 60

2.9.9 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 information

[Creating an application as bare machine C without the C library](#) on page 63

2.10 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 information

[Initialization of the execution environment and execution of the application](#) on page 67

[C++ initialization, construction and destruction](#) on page 68

[Exceptions system initialization](#) on page 69

[Library functions called from main\(\)](#) on page 70

[Program exit and the assert macro](#) on page 70

2.10.1 Initialization of the execution environment and execution of the application

You can customize execution initialization 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`. Use the `armclang` option `-e` or `armlink` option `--entry` to specify `__main` as the entry point. For example:

```
.global __rt_entry
.global __main
__main:
    B __rt_entry
```

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

1. Sets up the stack and the heap by one of several 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 information

[Direct semihosting C library function dependencies](#) on page 57

2.10.2 C++ initialization, construction and destruction

The C++ standard places certain requirements on the construction and destruction of objects with static storage duration. The static constructors are executed before `main()`, the destructors are called after the program exits.

The library must ensure that all the static constructors within a translation unit are called in the order of declaration, and the static destructors are called in reverse order of declaration. There is no way to determine the initialization order between translation units.

Each translation unit containing static constructors has an initialization function. This function calls the static constructors for the translation unit, and registers the static destructors with a call to `__aeabi_atexit()`. Function-local static objects with destructors also register their destructors using `__aeabi_atexit()`.

The location of the per translation unit initialization function is stored in an `.init_array` section. At link time the `.init_array` sections must be collated together into a single contiguous `.init_array` section. The linker generates an error if this is not possible.

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()` which calls the static destructors registered with `__aeabi_atexit()`.



The `__aeabi_atexit()` function calls `malloc()`.

Related information

[Tailoring the C library to a new execution environment](#) on page 67

2.10.3 Exceptions system initialization

The exceptions system can be initialized either on demand (that is, when first used), or before entering `main()`.

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 entering `main()`, 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()`.



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 information

[Tailoring the C library to a new execution environment](#) on page 67

2.10.4 Library functions called from main()

The function `main()` can call several 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 `error`.

Related information

[Initialization of the execution environment and execution of the application](#) on page 67

[Tailoring the C library to a new execution environment](#) on page 67

[Assembler macros that tailor locale functions in the C library](#) on page 71

[Tailoring input/output functions in the C and C++ libraries](#) on page 90

[Tailoring non-input/output C library functions](#) on page 100

[Modification of C library functions for error signaling, error handling, and program exit](#) on page 81

2.10.5 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 several conditions:

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:
 - `__OPT_SMALL_ASSERT` is defined.
 - `__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()`.

Related information

[Tailoring the C library to a new execution environment](#) on page 67

2.11 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 information

[Link time selection of the locale subsystem in the C library](#) on page 71

[Runtime selection of the locale subsystem in the C library](#) on page 73

[Definition of locale data blocks in the C library](#) on page 73

[LC_CTYPE data block](#) on page 75

[LC_COLLATE data block](#) on page 78

[LC_MONETARY data block](#) on page 79

[LC_NUMERIC data block](#) on page 80

[LC_TIME data block](#) on page 80

2.11.1 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 of your own locales in each category as you choose, and you can name your own 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 information

[ISO8859-1 implementation](#) on page 72

[Shift-JIS and UTF-8 implementation](#) on page 73

2.11.1.1 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 2-8: Default ISO8859-1 locales

Symbol	Description
<code>__use_iso8859_ctype</code>	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.
<code>__use_iso8859_collate</code>	Selects the <code>strcoll</code> / <code>strxfrm</code> collation table appropriate to the Latin-1 alphabet. The default C locale does not require a collation table.
<code>__use_iso8859_monetary</code>	Selects the Sterling monetary category using Latin-1 coding.
<code>__use_iso8859_numeric</code>	Selects separation of thousands with commas in the printing of numeric values.
<code>__use_iso8859_locale</code>	Selects all the ISO8859-1 selections described in this table.

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

2.11.1.2 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 2-9: Default Shift-JIS and UTF-8 locales

Function	Description
<code>__use_sjis_ctype</code>	Sets the character set to the Shift-JIS multibyte encoding of Japanese characters
<code>__use_utf8_ctype</code>	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 A6 and DF 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.

2.11.2 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 information

[setlocale\(\)](#) on page 168

2.11.3 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. You define each locale block in the same way with a `_begin` macro, some data macros, and an `_end` macro.

`LC_TYPE_begin prefix, name`

Begins the definition of a locale block.

`LC_TYPE_function`

Specifies the data for a locale block.



- When specifying locale data, you must call the macro repeatedly for each respective function.
 - To specify the data for your locale block, call the macros for that locale type in the order specified for that particular locale type.
-

`LC_TYPE_end`

Ends the definition of a locale block.

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.

function

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

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 information

[LC_CTYPE data block](#) on page 75
[LC_COLLATE data block](#) on page 78
[LC_MONETARY data block](#) on page 79
[LC_NUMERIC data block](#) on page 80
[LC_TIME data block](#) on page 80

2.11.4 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:

<code>__S</code>	whitespace characters
<code>__P</code>	punctuation characters
<code>__B</code>	printable space characters
<code>__L</code>	lowercase letters
<code>__U</code>	uppercase letters
<code>__N</code>	decimal digits
<code>__C</code>	control characters
<code>__X</code>	hexadecimal digit letters A-F and a-f
<code>__A</code>	alphabetic but neither uppercase nor lowercase, such as Japanese katakana.



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

Note

`__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(char32_t *pwc, char c, mbstate_t *pstate, int
    wchar32);
size_t internal_wcrtomb(char *s, char32_t w, mbstate_t *pstate, int wchar32);
```

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.

The *wchar32* parameter specifies whether the wide character is 32-bit (1) or 16-bit (0). If your code does not use the C11/C++11 headers `<uchar.h>` or `<cuchar>`, the *wchar32* parameter can be ignored because it defaults to the current definition of *wchar_t*.

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" ; 0x00-0x08
LC_CTYPE_table "_C|_S, _C|_S, _C|_S, _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" ; 0x0E-0x0F
LC_CTYPE_table "_C, _C, _C, _C, _C, _C, _C, _C" ; 0x10-0x1F
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, _P" ; ) * + , - . /
LC_CTYPE_table "_N, _N, _N, _N, _N, _N, _N, _N" ; 0-9
LC_CTYPE_table "_P, _P, _P, _P, _P, _P, _P, _P" ; : ; < = > ? @
LC_CTYPE_table "_U|_X, _U|_X, _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" ; G-P
LC_CTYPE_table "_U, _U, _U, _U, _U, _U, _U, _U" ; Q-Z
LC_CTYPE_table "_P, _P, _P, _P, _P, _P, _P, _P" ; [ \ ] ^ _ `
LC_CTYPE_table "_L|_X, _L|_X, _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" ; g-p
LC_CTYPE_table "_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"
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_wrtomb
LC_CTYPE_multibyte utf8_mbrtowc, utf8_wrtomb, 6
LC_CTYPE_end
```

Related information

[Definition of locale data blocks in the C library](#) on page 73

2.11.5 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, 0x77, 0x79, 0x7a, 0x7b, 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 information

[Definition of locale data blocks in the C library](#) on page 73

2.11.6 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_cursymbol 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_cursymbol "", ""
LC_MONETARY_point ""
LC_MONETARY_thousands ""
LC_MONETARY_grouping ""
LC_MONETARY_end
```

Related information

[Definition of locale data blocks in the C library](#) on page 73

2.11.7 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 information

[Definition of locale data blocks in the C library](#) on page 73

2.11.8 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 "%a %b %e %T %Y", "%m/%d/%y", "%H:%M:%S"
LC_TIME_c99format "%I:%M:%S %p"
LC_TIME_end
```

Related information

[Definition of locale data blocks in the C library](#) on page 73

2.12 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.



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 2-10: Trap and error handling

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

Related information

[Direct semihosting C library function dependencies](#) on page 57

2.13 Stack and heap memory allocation and the Arm C and C++ libraries

The Arm® C and C++ libraries require you to specify where the stack pointer begins, but specifying the heap is optional. However, some library functions use the heap, either explicitly (for example `malloc`) or implicitly (for example `fopen`).

If you are providing a heap, you must:

- Understand the heap usage requirements of the Arm C and C++ libraries.
- Configure the size and placement of the heap.
- Consider which heap implementation you want to use.

If you are not providing a heap, you must:

- Understand the heap usage requirements of the Arm C and C++ libraries.
- Understand how to avoid or reimplement the heap-using functions.

2.13.1 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.



The memory sizes quoted might change in future releases.

Related information

[Library heap usage requirements of microlib](#) on page 119

2.13.2 Choosing a heap implementation 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 available heap implementations are:

- Heap1, the default implementation, implements the smallest and simplest heap manager.
- Heap2 provides an implementation with the performance cost of `malloc()` or `free()` growing logarithmically with the number of free blocks.



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

Heap1

Heap1, the default implementation, implements the smallest and simplest heap manager. The heap is managed as a single-linked list of free blocks that are held in increasing address order. This implementation has low overheads. However, the performance cost of `malloc()` or `free()` grows linearly with the number of free blocks and might be too slow for some use cases.

If you expect more than 100 unallocated blocks, Arm recommends that you use Heap2 when you require near constant-time performance.

The allocation policy is first-fit by address. For AArch32, the smallest block that can be allocated is 4 bytes and there is an extra overhead of 4 bytes. For AArch64, the smallest allocation is 8 bytes, and there is an extra overhead of 8 bytes per allocation. For AArch64 with `__use_memtag_heap`, the smallest allocation is 0 bytes, and there is an extra overhead of 16 bytes per allocation.

Heap2

Heap2 provides an 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. For AArch32, the smallest block that can be allocated is 12 bytes and there is an extra overhead of 4 bytes. For AArch64, the smallest allocation is 24 bytes and there is an extra overhead of 8 bytes per allocation. For AArch64 with `__use_memtag_heap`, the smallest allocation is 16 bytes and there is an extra overhead of 16 bytes per allocation.

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 one of the following:

- `IMPORT __use_realtime_heap` when using the legacy `armasm`-syntax assembly language.
- `__asm(".global __use_realtime_heap\n\t")` from C.

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

The heap must fit within 16MB of address space.

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()`).

For AArch32 targets, the heap bounds are given by:

```
struct __heap_extent {
    unsigned base;
    size_t range;
};
__attribute__((value_in_regs)) struct __heap_extent __user_heap_extent(
    unsigned ignore1, size_t ignore2);
```

For AArch64 targets, the heap bounds are given by:

```
struct __heap_extent {
    unsigned long base;
    size_t range;
};
__attribute__((value_in_regs)) struct __heap_extent __user_heap_extent(
    unsigned long ignore1, size_t ignore2);
```

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 implement `__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 implement `__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.



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 information

[Avoiding the heap and heap-using library functions supplied by Arm](#) on page 89

2.13.3 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.

You must always specify where the stack pointer begins. The initial stack pointer must be aligned to a multiple of 8 bytes for AArch32 and a multiple of 16 bytes for AArch64.

You might have to configure the heap if, for example:

- You intend to use Arm library functions that use the heap, for example, `malloc()`, `calloc()`.
- You define `argc` and `argv` command-line arguments for `main()`.

If you are using the C library's initialization code, use any of the following methods to configure the stack and heap:

- Use the symbols `__initial_sp`, `__heap_base`, and `__heap_limit`.
- Use a scatter file to define `ARM_LIB_STACKHEAP`, `ARM_LIB_STACK`, or `ARM_LIB_HEAP` regions.
- Implement `__user_setup_stackheap()` or `__user_initial_stackheap()`.



The first two methods are the only methods that microlib supports for defining where the stack pointer starts and for defining the heap bounds.

If you are not using the C library's initialization code (see [Standalone C library functions](#)), use the following method to configure the stack and heap:

- Set up the stack pointer manually at your application's entry point.
- Call `_init_alloc()` to set up an initial heap region, and implement `__rt_heap_extend()` if you need to add memory to it later.

Configuring the stack and heap with symbols

Define the symbol `__initial_sp` to point to the top of the stack.

If using the heap, also define symbols `__heap_base` and `__heap_limit`.

You can define these symbols in an assembly language file.

For example:

```
__attribute__((naked)) void dummy_function(void)
{
    __asm(".global __initial_sp\n\t"
          ".global __heap_base\n\t"
          ".global __heap_limit\n\t"
          ".equ __initial_sp, STACK_BASE\n\t"
          ".equ __heap_base, HEAP_BASE\n\t")
}
```

```
    }.equ __heap_limit, (HEAP_BASE+HEAP_SIZE)\n\t"
};
```

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)
```



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).

Configuring the stack and heap with a scatter file

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.

Configuring the stack and heap with `__user_setup_stackheap()`

Implement `__user_setup_stackheap()` to set up the stack pointer and return the bounds of the initial heap region.

Configuring the heap from bare machine C using `_init_alloc` and `__rt_heap_extend`

If you are using a heap implementation from bare machine C (that is an application that does not define `main()` and does not initialize the C library) you must define the base and top of the heap as well as providing a heap extension function.

1. Call `_init_alloc(base, top)` to define the base and top of the memory you want to manage as a heap.



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.

Stack and heap collision detection

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 `__asm(".global __use_two_region_memory\n\t").`



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()`

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 information

[Legacy support for `\_\_user\_initial\_stackheap\(\)` on page 88](#)

[__user_heap_extend\(\) on page 179](#)

[__user_heap_extent\(\) on page 180](#)

[Legacy function `\_\_user\_initial\_stackheap\(\)` on page 188](#)

[__rt_heap_extend\(\) on page 164](#)

[__rt_stackheap_init\(\) on page 167](#)

[__user_setup_stackheap\(\) on page 181](#)

[__vectab_stack_and_reset on page 182](#)

2.13.4 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. However Arm recommends not using this option if possible.



Arm recommends that you use `__user_setup_stackheap()` instead of `__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.

Exceptions

When you must create the heap and stack in C code rather than in assembly code, you cannot use the `__user_setup_stackheap()` function. Therefore, you must use the `__user_initial_stackheap()` function instead.

If your implementation is sufficiently complex that it warrants the use of a temporary stack when setting up the initial heap and stack, use either:

- `__user_setup_stackheap()` and manually set up the temporary stack yourself.
- `__user_initial_stackheap()`, which sets up the temporary stack for you.

Related information

[Stack pointer initialization and heap bounds](#) on page 85

2.13.5 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.
- `__asm(".global __use_no_heap\n\t")` 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>

__asm(".global __use_no_heap\n\t");
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 information

[Indirect semihosting C library function dependencies](#) on page 58

2.14 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. These low-level functions call system I/O functions that 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 information

[Direct semihosting C library function dependencies](#) on page 57

2.15 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`.
- When targeting the strict ANSI C standard, you must provide your own implementation of the `__FILE` structure. For example:

```
struct __FILE { int handle; /* Add whatever you need here */ };
```
- 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`.

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()`.

High-level function	Low-level object									
	1	2	3	4	5	6	7	8	9	10
<code>fgets</code>	X	-	-	-	X	X	-	-	-	-
<code>fgetws</code>	X	-	-	-	-	-	X	-	-	-
<code>fprintf</code>	X	-	-	X	X	-	-	-	-	-
<code>fputs</code>	X	-	-	X	-	-	-	-	-	-
<code>fputws</code>	X	-	-	-	-	-	-	X	-	-
<code>fread</code>	X	-	-	-	-	X	-	-	-	-
<code>fscanf</code>	X	-	-	-	-	X	-	-	X	-
<code>fwprintf</code>	X	-	-	-	X	-	-	X	-	-
<code>fwrite</code>	X	-	-	X	-	-	-	-	-	-
<code>fwscanf</code>	X	-	-	-	-	-	X	-	-	X
<code>getchar</code>	X	X	-	-	-	X	-	-	-	-
<code>gets</code>	X	X	-	-	X	X	-	-	-	-
<code>getwchar</code>	X	X	-	-	-	-	X	-	-	-
<code>perror</code>	X	-	X	X	-	-	-	-	-	-
<code>printf</code>	X	-	X	X	X	-	-	-	-	-
<code>putchar</code>	X	-	X	X	-	-	-	-	-	-
<code>puts</code>	X	-	X	X	-	-	-	-	-	-
<code>putwchar</code>	X	-	X	-	-	-	-	X	-	-
<code>scanf</code>	X	X	-	-	-	X	-	-	X	-
<code>vfprintf</code>	X	-	-	X	X	-	-	-	-	-
<code>vfscanf</code>	X	-	-	-	-	X	-	-	X	-
<code>vfwprintf</code>	X	-	-	-	X	-	-	X	-	-
<code>vfwscanf</code>	X	-	-	-	-	-	X	-	-	X
<code>vprintf</code>	X	-	X	X	X	-	-	-	-	-
<code>vscanf</code>	X	X	-	-	-	X	-	-	X	-
<code>vwprintf</code>	X	-	X	-	X	-	-	X	-	-
<code>vwscanf</code>	X	X	-	-	-	-	X	-	-	X
<code>wprintf</code>	X	-	X	-	X	-	-	X	-	-
<code>wscanf</code>	X	X	-	-	-	-	X	-	-	X

Related information

[Tailoring input/output functions in the C and C++ libraries](#) on page 90

[The C library printf family of functions on page 93](#)

[The C library scanf family of functions on page 93](#)

[Redefining low-level library functions to enable direct use of high-level library functions in the C library on page 94](#)

[The C library functions fread\(\), fgets\(\) and gets\(\) on page 96](#)

[Re-implementing __backspace\(\) in the C library on page 97](#)

[Re-implementing __backspacewc\(\) in the C library on page 98](#)

[Redefining target-dependent system I/O functions in the C library on page 98](#)

2.16 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 information

[The C library scanf family of functions on page 93](#)

[Redefining low-level library functions to enable direct use of high-level library functions in the C library on page 94](#)

[The C library functions fread\(\), fgets\(\) and gets\(\) on page 96](#)

[Re-implementing __backspace\(\) in the C library on page 97](#)

[Re-implementing __backspacewc\(\) in the C library on page 98](#)

[Redefining target-dependent system I/O functions in the C library on page 98](#)

[Tailoring input/output functions in the C and C++ libraries on page 90](#)

[Target dependencies on low-level functions in the C and C++ libraries on page 91](#)

2.17 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 information

[The C library printf family of functions](#) on page 93

[Redefining low-level library functions to enable direct use of high-level library functions in the C library](#) on page 94

[The C library functions fread\(\), fgetc\(\) and gets\(\)](#) on page 96

[Re-implementing __backspace\(\) in the C library](#) on page 97

[Re-implementing __backspacewc\(\) in the C library](#) on page 98

[Redefining target-dependent system I/O functions in the C library](#) on page 98

[Tailoring input/output functions in the C and C++ libraries](#) on page 90

[Target dependencies on low-level functions in the C and C++ libraries](#) on page 91

2.18 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");
}

```

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:



Note

```

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); // sendchar(&ch) would not work everywhere
    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 information

[The C library printf family of functions](#) on page 93

[The C library scanf family of functions](#) on page 93

[The C library functions fread\(\), fgetc\(\) and gets\(\)](#) on page 96

[Re-implementing __backspace\(\) in the C library](#) on page 97

[Re-implementing __backspacewc\(\) in the C library](#) on page 98

[Redefining target-dependent system I/O functions in the C library](#) on page 98

[Tailoring input/output functions in the C and C++ libraries](#) on page 90

[Target dependencies on low-level functions in the C and C++ libraries](#) on page 91

2.19 The C library functions fread(), fgetc() and gets()

The functions `fread()`, `fgetc()`, 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 information

[The C library printf family of functions](#) on page 93

[The C library scanf family of functions](#) on page 93

[Redefining low-level library functions to enable direct use of high-level library functions in the C library on page 94](#)

[Re-implementing `\_\_backspace\(\)` in the C library on page 97](#)

[Re-implementing `\_\_backspacewc\(\)` in the C library on page 98](#)

[Redefining target-dependent system I/O functions in the C library on page 98](#)

[Tailoring input/output functions in the C and C++ libraries on page 90](#)

[Target dependencies on low-level functions in the C and C++ libraries on page 91](#)

2.20 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.



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 `ungetc()` of a character into a stream followed immediately by a `__backspace()` with no intervening read, never occurs.

Related information

[The C library printf family of functions](#) on page 93

[The C library scanf family of functions](#) on page 93

[Redefining low-level library functions to enable direct use of high-level library functions in the C library](#) on page 94

[The C library functions fread\(\), fgetc\(\) and gets\(\)](#) on page 96

[Re-implementing __backspacewc\(\) in the C library](#) on page 98

[Redefining target-dependent system I/O functions in the C library](#) on page 98

[Tailoring input/output functions in the C and C++ libraries](#) on page 90

[Target dependencies on low-level functions in the C and C++ libraries](#) on page 91

2.21 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 information

[The C library printf family of functions](#) on page 93

[The C library scanf family of functions](#) on page 93

[Redefining low-level library functions to enable direct use of high-level library functions in the C library](#) on page 94

[The C library functions fread\(\), fgetc\(\) and gets\(\)](#) on page 96

[Re-implementing __backspace\(\) in the C library](#) on page 97

[Redefining target-dependent system I/O functions in the C library](#) on page 98

[Tailoring input/output functions in the C and C++ libraries](#) on page 90

[Target dependencies on low-level functions in the C and C++ libraries](#) on page 91

2.22 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 function prototypes are contained in `rt_sys.h`. These functions are called by the C standard I/O library functions. For example, `_sys_open()` is called by `fopen()` and `freopen()`. `_sys_open()` uses the strings `__stdin_name`, `__stdout_name`, and `__stderr_name` during C library initialization to identify which standard I/O device handle to return. You can leave their values as the default (`:tt`) if `_sys_open()` does not use them.



`stdin`, `stdout`, and `stderr` are assumed to be interactive devices. They are line-buffered at program startup, regardless of what `_sys_istty` reports for them. An exception is if they have been redirected on the command line.

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

Example of retargeting the system I/O functions

```
/*
 * These names are used during library initialization as the
 * file names opened for stdin, stdout, and stderr.
 * As we define _sys_open() to always return the same file handle,
 * these can be left as their default values.
 */
const char __stdin_name[] = ":tt";
const char __stdout_name[] = ":tt";
const char __stderr_name[] = ":tt";

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 information

[The C library `printf` family of functions](#) on page 93

[The C library `scanf` family of functions](#) on page 93

[Redefining low-level library functions to enable direct use of high-level library functions in the C library](#) on page 94

[The C library functions `fread\(\)`, `fgets\(\)` and `gets\(\)`](#) on page 96

[Re-implementing `\_\_backspace\(\)` in the C library](#) on page 97

[Re-implementing `\_\_backspacewc\(\)` in the C library](#) on page 98

[Tailoring input/output functions in the C and C++ libraries](#) on page 90

[Target dependencies on low-level functions in the C and C++ libraries](#) on page 91

2.23 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 information

[Direct semihosting C library function dependencies](#) on page 57

2.24 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.



Real-time division is not available in the libraries for the Armv6-M and Armv8-M.baseline architecture.



The Armv7-M, Armv7-R, and Armv8-M.mainline architectures support hardware floating-point divide. Code running on these architectures do not require the library divide routines.

Select the real-time divide routine using either of the following methods:

- `IMPORT __use_realtime_division` from assembly language.
- `__asm(".global __use_realtime_division\n\t")` from C.

2.25 ISO C library implementation definition

Describes how the libraries fulfill the requirements of the ISO specification.

2.25.1 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
```



The behavior of the `assert` macro depends on the conditions in operation at the most recent occurrence of `#include <assert.h>`. See [Program exit and the assert macro](#) for more information about the behavior of the `assert` macro.

- The following functions test for character values in the range `EOF` (-1) to 255 inclusive:
 - `isalnum()`
 - `isalpha()`
 - `isctrl()`
 - `islower()`
 - `isprint()`
 - `isupper()`
 - `ispunct()`
- The fully POSIX-compliant functions `remquo()`, `remquof()` and `remquo1()` 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.

Related information

[mathlib error handling](#) on page 102

[ISO-compliant implementation of signals supported by the `signal\(\)` function in the C library and additional type arguments](#) on page 103

[ISO-compliant C library input/output characteristics](#) on page 104

[Program exit and the `assert` macro](#) on page 70

[C and C++ library naming conventions](#) on page 112

2.25.2 mathlib error handling

The error handling of mathematical functions is consistent with Annex F of the ISO/IEC C99 standard.

Related information

[ISO-compliant implementation of signals supported by the `signal\(\)` function in the C library and additional type arguments](#) on page 103

[ISO-compliant C library input/output characteristics](#) on page 104

[How the Arm C library fulfills ISO C specification requirements](#) on page 101

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

The `signal()` function supports several 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 2-12: Signals supported by the `signal()` function

Signal	Number	Description	Additional argument
<code>SIGABRT</code>	1	Returned when the <code>abort()</code> function is called. The <code>abort()</code> function is triggered when there is an untrapped C++ exception, or when an assertion fails.	None
<code>SIGFPE</code>	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 <code>FE_EX_INEXACT</code> , <code>FE_EX_UNDERFLOW</code> , <code>FE_EX_OVERFLOW</code> , <code>FE_EX_DIVBYZERO</code> , <code>FE_EX_INVALID</code> , <code>DIVBYZERO</code> ¹
<code>SIGILL</code>	3	Illegal instruction.	None
<code>SIGINT</code> ²	4	Attention request from user.	None
<code>SIGSEGV</code> ²	5	Bad memory access.	None
<code>SIGTERM</code> ²	6	Termination request.	None
<code>SIGSTAK</code>	7	Obsolete.	None
<code>SIGRTRED</code>	8	Redirection failed on a runtime library input/output stream.	Name of file or device being re-opened to redirect a standard stream
<code>SIGRTMEM</code>	9	Out of heap space during initialization or after corruption.	Size of failed request
<code>SIGUSR1</code>	10	User-defined.	User-defined
<code>SIGUSR2</code>	11	User-defined.	User-defined
<code>SIGPVFN</code>	12	A pure virtual function was called from C++.	-
<code>SIGCPPL</code>	13	Not supported.	-
<code>SIGOUTOFHEAP</code>	14	Not supported.	Size of the failed request in bytes

¹ These constants are defined in `fenv.h`. `FE_EX_DIVBYZERO` is for floating-point division while `DIVBYZERO` is for integer division.

² The library never generates this signal. It is available for you to raise manually, if required.

Signal	Number	Description	Additional argument
reserved	>=15	Reserved.	Reserved

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 using a fully-conforming floating-point model, such as the `armclang` default.

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 information

[mathlib error handling](#) on page 102

[ISO-compliant C library input/output characteristics](#) on page 104

[How the Arm C library fulfills ISO C specification requirements](#) on page 101

2.25.4 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.
- `stdin`, `stdout`, and `stderr` are assumed to be interactive devices. They are line-buffered at program startup, regardless of what `_sys_istty` reports for them. An exception is if they have been redirected on the command line.
- `localtime()` is implemented and returns the local time. `gmtime()` is not implemented and returns `NULL`. Therefore converting between time-zones is not supported.
- 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 `0`.
- `fscanf()` treats `%p` arguments 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 2-13: perror() messages

Error	Message
0	No error (<code>errno = 0</code>)
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 information

[mathlib error handling](#) on page 102

[ISO-compliant implementation of signals supported by the `signal\(\)` function in the C library and additional type arguments](#) on page 103

[How the Arm C library fulfills ISO C specification requirements](#) on page 101

2.25.5 Standard C++ library implementation definition

The Standard C++ library in Arm® Compiler 6 is based on the LLVM libc++ project.



This topic includes descriptions of [ALPHA] and [COMMUNITY] features. See [Support level definitions](#).

The following sections describe the limitations of the Standard C++ library in Arm Compiler 6.

All supported standard versions

For all supported standards, the libc++ library deviates from the standard library as follows:

- For `std::vector<bool>::const_reference`, the standards require the `const_reference` type to be `bool`. However, in libc++ the `const_reference` type is an **IMPLEMENTATION DEFINED**, read-only bit reference class.
- For `std::bitset<N>`, the standards require `bool operator[](size_t pos) const;` to return `bool`. However, in libc++ `bool operator[](size_t pos) const;` returns an **IMPLEMENTATION DEFINED**, read-only bit reference object.

Support for C++98

Arm Compiler 6 C++ libraries support the C++98 standard, except:

- Where the C++11 standard deviates from the C++98 standard. For example, where `std::deque<T>::insert()` returns an iterator, as required by the C++11 standard, but the C++98 standard requires it to return `void`. Information about how the C++11 standard deviates from the C++98 standard is available in Annex "C Compatibility" of the C++11 standard definition.
- Where the libc++ library deviates from the C++98 standard library:

For `std::raw_storage_iterator`, the C++98 standard requires the `raw_storage_iterator` class template to be inherited from `std::iterator<std::output_iterator_tag, void, void, void, void>`. However, in libc++ the

`raw_storage_iterator` class template is inherited from an instantiation of `std::iterator` with a different list of template arguments.

Support for C++03

Arm Compiler 6 C++ libraries support the C++03 standard, except:

- Where the C++11 standard deviates from the C++03 standard. For example, where `std::deque<T>::insert()` returns an iterator, as required by the C++11 standard, but the C++03 standard requires it to return `void`. Information about how the C++11 standard deviates from the C++03 standard is available in Annex "C Compatibility" of the C++11 standard definition.
- Where the `libc++` library deviates from the C++98 standard library:

For `std::raw_storage_iterator`, the C++98 standard requires the `raw_storage_iterator` class template to be inherited from `std::iterator<std::output_iterator_tag, void, void, void, void>`. However, in `libc++` the `raw_storage_iterator` class template is inherited from an instantiation of `std::iterator` with a different list of template arguments.

- When compiling with any optimization other than `-O0`, Arm Compiler might omit a call to a replaceable global allocation function and the corresponding deallocation function. When it does so, the storage is instead provided by the implementation or provided by extending the allocation of another `new` expression. This rule is part of the C++14 standard, but Arm Compiler also applies the rule for C++03 and C++11.

Support for C++11

Arm Compiler 6 C++ libraries support the majority of C++11.

- Thread support, `<thread>`, and other concurrency features, are [ALPHA] supported.
- When compiling with any optimization other than `-O0`, Arm Compiler might omit a call to a replaceable global allocation function and the corresponding deallocation function. When it does so, the storage is instead provided by the implementation or provided by extending the allocation of another `new` expression. This rule is part of the C++14 standard, but Arm Compiler also applies the rule for C++03 and C++11.

Support for exceptions

Arm Compiler 6 supports:

- C++ libraries with exceptions.
- C++ libraries without exceptions. These libraries are compiled with the `-fno-exceptions` option.

Linking objects that have been compiled with `-fno-exceptions` automatically selects the libraries without exceptions. You can use the linker option `--no_exceptions` to diagnose whether the objects being linked contain exceptions.



- By default, C++ sources are compiled with exceptions support. You can use the `-fno-exceptions` option to disable exceptions support.
- By default, C sources are compiled without exceptions support. You can use the `-fexceptions` option to enable exceptions support.

- If an exception propagates into a function that has been compiled without exceptions support, then the program terminates.
- If the C++ libraries built without exception support is put in an error state, then an exception is not thrown, but the program behavior is undefined.

Support for Array Construction and Delete helper functions

Arm Compiler 6 is not compatible with C++ objects from other compilers that use Array Construction and Delete helper functions.

Support for Atomic [ALPHA]

Arm Compiler 6 provides access to the Atomic operations library `<atomic>` as an [ALPHA] feature.



Arm Compiler 6 does not provide an implementation of `libatomic`. You must either provide an implementation of `libatomic` or only use the atomic operation library for types for which the hardware can provide synchronization primitives.

Support for multithreading [ALPHA]

The default standard C++ library shipped in Arm Compiler 6 does not support multithreading. This variant of the standard C++ library does not support the concurrency constructs available through the headers, that includes `<thread>` and `<mutex>`. A multi-threaded [ALPHA]-supported variant of the C++ library is also included in Arm Compiler 6.

Support for thread-safe static initialization of local variables in C++

The default C++ library in Arm Compiler 6 contains trivial implementations of the following one-time construction API functions, which are not thread-safe:

```
extern "C" int __cxa_guard_acquire(int*guard_object);  
extern "C" void __cxa_guard_release(int *guard_object);  
extern "C" void __cxa_guard_abort(int *guard_object);
```



- This does not apply to the [ALPHA]-supported multi-threaded C++ libraries.
- For thread-safe static initialization of local variables in C++, you must provide your own thread-safe implementation of these functions.

These API functions are described in more detail in the [Run-time ABI for the Arm Architecture](#). On Armv6-M, you must also provide thread-safe atomic helperfunctions.

Support for Armv6-M atomic helper functions

The Arm Compiler 6 Armv6-M libraries contain trivial implementations of the following atomic helper functions, which are not thread-safe:

```
uint32_t __user_cmpxchg_4(uint32_t *ptr, uint32_t old, uint32_t new);
uint16_t __user_cmpxchg_2(uint16_t *ptr, uint16_t old, uint16_t new);
uint8_t __user_cmpxchg_1(uint8_t *ptr, uint8_t old, uint8_t new);
uint64_t __atomic_exchange_8(uint64_t *dest, uint64_t val, int model);
uint32_t __atomic_exchange_4(uint32_t *dest, uint32_t val, int model);
uint16_t __atomic_exchange_2(uint16_t *dest, uint16_t val, int model);
uint8_t __atomic_exchange_1(uint8_t *dest, uint8_t val, int model);
```



For atomic access on Armv6-M, you must provide your own thread-safe implementation of the atomic helper functions.

Library extensions

All libc++ experimental features that are not part of the C++ standard are [COMMUNITY]-supported. In particular, the headers `<ext/*>` and `<experimental/*>` are [COMMUNITY]-supported.

Related information

[The LLVM Compiler Infrastructure](#)

[The LLVM libc++ library](#)

2.26 C library functions and extensions

The Arm C library is fully compliant with the ISO C99 library standard and provides several GNU, POSIX, BSD-derived, and Arm® Compiler-specific extensions.

The following table describes these extensions.

Table 2-14: C library extensions

Function	Header file definition	Extension
<code>wscasecmp()</code>	<code>wchar.h</code>	GNU extension with Arm library support
<code>wcsncasecmp()</code>	<code>wchar.h</code>	GNU extension with Arm library support
<code>wcstombs()</code>	<code>stdlib.h</code>	POSIX extended functionality
<code>posix_memalign()</code>	<code>stdlib.h</code>	POSIX extended functionality
<code>alloca()</code>	<code>alloca.h</code>	Common nonstandard extension to many C libraries

Function	Header file definition	Extension
<code>strncpy()</code>	<code>string.h</code>	Common BSD-derived extension to many C libraries
<code>strlcat()</code>	<code>string.h</code>	Common BSD-derived extension to many C libraries
<code>strcasecmp()</code>	<code>string.h</code>	Standardized by POSIX
<code>strncasecmp()</code>	<code>string.h</code>	Standardized by POSIX
<code>_fisatty()</code>	<code>stdio.h</code>	Specific to Arm Compiler
<code>__heapstats()</code>	<code>stdlib.h</code>	Specific to Arm Compiler
<code>__heapvalid()</code>	<code>stdlib.h</code>	Specific to Arm Compiler

Related information

[wcscasecmp\(\)](#) on page 183
[wcsncasecmp\(\)](#) on page 183
[wcstombs\(\)](#) on page 183
[alloca\(\)](#) on page 146
[strlcat\(\)](#) on page 170
[strncpy\(\)](#) on page 171
[strcasecmp\(\)](#) on page 170
[strncasecmp\(\)](#) on page 171
[_fisatty\(\)](#) on page 150
[__heapstats\(\)](#) on page 152
[__heapvalid\(\)](#) on page 154

2.27 Avoid linking in the Arm C library

The C runtime libraries provided with Arm® Compiler 6 are suitable for various Arm-based projects. However, some projects might have certain requirements that mean it is necessary to avoid using all or part of the standard C library.



This topic includes descriptions of [COMMUNITY] features. See [Support level definitions](#).

For example:

- The project must use a certified functional safety (FuSa) C library to make it easier to fulfill the safety requirements for the project.
- The project uses alternative libraries provided by the operating system (OS) vendor.
- The project has some custom requirements to re-implement certain C library functionality.

The following sections expand on the information provided in [Standalone C library functions](#).

The following sections do not:

- Describe how to fully avoid the C++ library.
- Explain how to develop your startup and initialization code that must run before the `main` function is reached.

The C libraries provided in Arm Compiler 6

Arm Compiler 6 provides the following C runtime libraries:

- C standardlib.
- C microlib.

The standard C library (standardlib) is the default C library that projects are likely to use. The micro library (microlib) is an alternative to the standard C library. Microlib focuses in particular on smaller code size, but with some documented limitations and restrictions.

Build options required to avoid the C library

The following compiler and linker options are required to avoid the C library being used explicitly by the user and implicitly by the compiler:

Compiler options

- `-fno-builtin` prevents the compiler from transforming calls to standard library functions based on built-in knowledge about how those functions behave.

This behavior applies to functions such as `printf` rather than `__builtin_name` functions, despite the name. The compiler knows something about functions such as `printf`, and sometimes transforms the source code based on that understanding. However, the compiler still expects the library to provide an implementation of those functions.

For example, if your code calls `printf("hello, world\n")`, the compiler might convert it into `puts("hello, world")` because it knows from the descriptions of those two functions in the C standard that they perform the same operations. But the `puts()` function cannot perform all the operations of `printf` by itself. If you write a more complicated call involving formatting such as `%d`, then the compiler has to emit a call to the `printf` library function.

- `-nostdlibinc` prevents the compiler from using the Arm standard C and C++ library header files.
- `-nostdlib` prevents the compiler from using the Arm standard C and C++ libraries.

Also, if you are working in a freestanding, non-hosted, environment you can specify the [COMMUNITY] option `-ffreestanding`. This option:

- Asserts that compilation targets a freestanding environment.
- Implies `-fno-builtin`.
- Sets the macro `STD_C_HOSTED` to 0.

Linker options

- `--no_scanlib` prevents the linker from scanning the Arm libraries to resolve references. As a consequence of using this option, the Arm supplied libraries are not used by the linker and you must include your own libraries.

Source code changes to avoid the C library

The function label `main()` has a special significance. The presence of a `main()` function forces the linker to link in the initialization code in `__main`. The `__main` function calls the following initialization functions:

- `__scatterload` (scatter-loading memory initialization code).
- `__rt_entry` (runtime library initialization code).

Without a function labeled `main()`, the initialization sequence is not linked in, and as a result, some standard C library functionality is not supported.

To prevent a reference to `__main`, either:

- Specify a different `main` function, for example, `my_main()`.
- Link with the `--no_startup` option.

Related information

[__rt_entry](#) on page 162

[-fno-builtin](#)

[-nostdlib](#)

[-nostdlibinc](#)

[--scanlib](#), [--no_scanlib](#)

[--startup=symbol](#), [--no_startup](#)

2.28 C and C++ library naming conventions

The library filename identifies how the variant was built.



This topic includes descriptions of [ALPHA] features. See [Support level definitions](#).



The library naming convention described in this documentation applies to the current release of the Arm compilation tools. Do not rely on C and C++ library names. They might change in future releases.

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.

The values for the fields of the filename, and the relevant build options are:

```
root /prefix _arch[fpu][position-independence][enum][wchar][exception]
[threading].endian
```

root

armlib

Arm® C library.

libcxx

libc++ library.

prefix

c

ISO C and C++ basic runtime support.

libcpp

libc++ library.

libcppabi

libc++abi runtime library.

f

IEEE-compliant library with a fixed rounding mode (round to nearest) and no inexact exceptions.

fj

IEEE-compliant library with a fixed rounding mode (round to nearest) and no exceptions.

fz

Behaves like the ϵ_j library, but additionally flushes denormals and infinities to zero.

This library behaves like the Arm VFP in Fast mode. This is the default.

g

IEEE-compliant library with configurable rounding mode and all IEEE exceptions.

h

Compiler support (helper function) 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 A32 only library for use with the Armv4 architecture. This is not available for C++.

t

An A32/T32 interworking library for use with the Armv4T architecture. This is not available for C++.

5

An A32/T32 interworking library for use with the Armv5T architecture and later. This is not available for C++.

p

A T32 only library for use with the Armv6-M architecture.

w

A T32 only library for use with the Armv7-M architecture.

2

A combined A32 and T32 library for use with Cortex®-A and Cortex-R series processors.

8

An A32/T32 interworking library for use with the Armv8 architecture, in AArch32 state.

o

An A64 library for use with the Armv8 architecture, in AArch64 state.

fpu

m

A variant of the library for processors that have single-precision hardware floating-point only, such as the Cortex-M4 processor.

v

Uses VFP instruction set.

s

Soft VFP.



If none of **v**, **m**, or **s** are present in a library name, the library provides no floating-point support.

position-independence

e

Position-independent access to static data.

f

FPIC addressing is enabled (used by `armclang` option `-bare-metal-pie`).



- Position independence is only supported in AArch32 state. If both `e` and `f` are not present in a library name, the library uses position-dependent access to static data.
- Bare-metal PIE support is deprecated in this release.

enum

n

Compatible with the default compiler option, `-fno-short-enums`.

wchar

u

Indicates the size of `wchar_t`. When present, the library is compatible with the compiler option, `-fno-short-wchar`. Otherwise, it is compatible with `-fshort-wchar`.

exception

x

This only applies to the C++ library. `x` indicates that the library is built with exception handling. Libraries without `x` are built without exception handling.

threading

z

This only applies to the C++ library. `z` indicates that the library is a multithreaded [ALPHA]-supported variant of the C++ library. Libraries without `z` are built without multithreading support.

endian

l

Little-endian.

b

Big-endian.

For example:

```
armlib/c_2.b  
libcxx/libc++_8f.l
```



Not all variant/name combinations are valid. See the `armlib` and `libcxx` directories for the libraries that are supplied with the Arm Compiler.

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

Related information

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

2.29 Using macro `__ARM_WCHAR_NO_IO` to disable `FILE` declaration and wide I/O function prototypes

You can define the macro `__ARM_WCHAR_NO_IO` to cause the `wchar.h` and `wchar` 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.

2.30 Using library functions with execute-only memory

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.



Arm does not provide libraries that are built without literal pools. The libraries still use literal pools, even when you use the `-mexecute-only` option.

3. The Arm C Micro-library

Describes microlib, the C micro-library.

3.1 About microlib

Microlib is an alternative library to the default C library. It is intended for use with deeply embedded applications that must fit into extremely small memory footprints.

These applications do not run under an operating system.



- Microlib does not attempt to be an ISO C-compliant library.
 - Microlib has no support for AArch64 execution state.
-

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 information

[Differences between microlib and the default C library](#) on page 117

[Library heap usage requirements of microlib](#) on page 119

[ISO C features missing from microlib](#) on page 120

[Building an application with microlib](#) on page 121

[Entering and exiting programs linked with microlib](#) on page 123

[Configuring the stack and heap for use with microlib](#) on page 122

[Tailoring the microlib input/output functions](#) on page 124

3.2 Differences between microlib and the default C library

There are several 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. In `main`, `argc` and `argv` parameters are undefined and cannot be used to access command-line arguments.
- Microlib provides limited support for C99 functions. Specifically, microlib does not support the following C99 functions:

- `<fenv.h>` functions:

<code>feclearexcept</code>	<code>fegetenv</code>	<code>fegetexceptflag</code>
<code>fegetround</code>	<code>feholdexcept</code>	<code>feraiseexcept</code>
<code>fesetenv</code>	<code>fesetexceptflag</code>	<code>fesetround</code>
<code>fetestexcept</code>	<code>feupdateenv</code>	

- Wide characters in general:

<code>btowc</code>	<code>fgetwc</code>	<code>fgetws</code>	<code>fputwc</code>
<code>fputws</code>	<code>fwide</code>	<code>fwprintf</code>	<code>fwscanf</code>
<code>getwc</code>	<code>getwchar</code>	<code>iswalnum</code>	<code>iswalpha</code>
<code>iswblank</code>	<code>iswcntrl</code>	<code>iswctype</code>	<code>iswdigit</code>
<code>iswgraph</code>	<code>iswlower</code>	<code>iswprint</code>	<code>iswpunct</code>
<code>iswspace</code>	<code>iswupper</code>	<code>iswxdigit</code>	<code>mblen</code>
<code>mbrlen</code>	<code>mbsinit</code>	<code>mbsrtowcs</code>	<code>mbstowcs</code>
<code>mbtowc</code>	<code>putwc</code>	<code>putwchar</code>	<code>swprintf</code>
<code>swscanf</code>	<code>towctrans</code>	<code>towlower</code>	<code>towupper</code>
<code>ungetwc</code>	<code>vfwprintf</code>	<code>vfwscanf</code>	<code>vswprintf</code>
<code>vswscanf</code>	<code>vwprintf</code>	<code>vwscanf</code>	<code>wcscat</code>
<code>wcschr</code>	<code>wcscmp</code>	<code>wscoll</code>	<code>wcscspn</code>
<code>wcsftime</code>	<code>wcslen</code>	<code>wcsncat</code>	<code>wcsncmp</code>
<code>wcsncpy</code>	<code>wcspbrk</code>	<code>wcsrchr</code>	<code>wcsrtombs</code>
<code>wcsspn</code>	<code>wcsstr</code>	<code>wcstod</code>	<code>wcstof</code>
<code>wcstoimax</code>	<code>wcstok</code>	<code>wcstol</code>	<code>wcstold</code>
<code>wcstoll</code>	<code>wcstombs</code>	<code>wcstoul</code>	<code>wcstoull</code>
<code>wcstoumax</code>	<code>wcsxfrm</code>	<code>wctob</code>	<code>wctomb</code>
<code>wctrans</code>	<code>wctype</code>	<code>wmemchr</code>	<code>wmemcmp</code>
<code>wmemcpy</code>	<code>wmemmove</code>	<code>wmemset</code>	<code>wprintf</code>
<code>wscanf</code>			

- Auxiliary `<math.h>` functions:

<code>ilogb</code>	<code>ilogbf</code>	<code>ilogbl</code>
<code>lgamma</code>	<code>lgammaf</code>	<code>lgammal</code>
<code>logb</code>	<code>logbf</code>	<code>logbl</code>
<code>nextafter</code>	<code>nextafterf</code>	<code>nextafterl</code>
<code>nexttoward</code>	<code>nexttowardf</code>	<code>nexttowardl</code>

- Functions relating to program startup and shutdown and other OS interaction:

<code>_Exit</code>	<code>atexit</code>	<code>exit</code>
<code>_system</code>	<code>time</code>	

- 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 only with the `armclang` command-line option `-ffp-mode=fast`.
- The level of ANSI C `stdio` support that is provided can be controlled with `__asm(".global __use_full_stdio\n\t").`
- `__asm(".global __use_smaller_memcpy\n\t")` 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.
- Microlib has no support in AArch64 state.
- When compiling a program that uses the microlib character classification functions in `ctype.h`, if the variable to be classified is not an ASCII character, the behavior of these functions is undefined.

Related information

[About microlib](#) on page 117

[Library heap usage requirements of microlib](#) on page 119

[ISO C features missing from microlib](#) on page 120

[Building an application with microlib](#) on page 121

[Entering and exiting programs linked with microlib](#) on page 123

[Configuring the stack and heap for use with microlib](#) on page 122

[Tailoring the microlib input/output functions](#) on page 124

3.3 Library heap usage requirements of microlib

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.



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

Related information

[Library heap usage requirements of the Arm C and C++ libraries](#) on page 83

3.4 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 `__asm(".global __use_full_stdio\n\t").` Use of this assembler directive 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 position independent compilation modes.

Although ROPI code can be linked with microlib, the resulting binary is not ROPI-compliant overall.

Command-line arguments

In `main`, `argc` and `argv` parameters are undefined and cannot be used to access command-line arguments.

Related information

[About microlib](#) on page 117

[Differences between microlib and the default C library](#) on page 117

[Library heap usage requirements of microlib](#) on page 119

[Building an application with microlib](#) on page 121

[Entering and exiting programs linked with microlib](#) on page 123

[Configuring the stack and heap for use with microlib](#) on page 122

[Tailoring the microlib input/output functions](#) on page 124

3.5 Building an application with microlib

To build a program using microlib, you must use the command-line option `--library_type=microlib`. This option can be used by the compiler, assembler or linker.

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

Compiler option



When compiling, `--library_type` must be used with `-Wl`.

```
armclang --target arm-arm-none-eabi -march=armv8-a -Wl,--library_type=microlib  
main.c
```

```
armclang --target arm-arm-none-eabi -march=armv8-a -c extra.c
armlink --cpu=8-A.32 -o image.axf main.o extra.o
```

Specifying `-wl,--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 `-wl,--library_type=microlib` is unnecessary, because the request to link against microlib exists in the object file generated by compiling `main.c`.

Assembler option

```
armclang --target arm-arm-none-eabi -march=armv8-a -c main.c
armclang --target arm-arm-none-eabi -march=armv8-a -c extra.c
armasm --cpu=8-A.32 --library_type=microlib more.s
armlink --cpu=8-A.32 -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

```
armclang --target arm-arm-none-eabi -march=armv8-a -c main.c
armclang --target arm-arm-none-eabi -march=armv8-a -c extra.c
armlink --cpu=8-A.32 --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 information

[About microlib](#) on page 117

[Differences between microlib and the default C library](#) on page 117

[Library heap usage requirements of microlib](#) on page 119

[ISO C features missing from microlib](#) on page 120

[Entering and exiting programs linked with microlib](#) on page 123

[Configuring the stack and heap for use with microlib](#) on page 122

[Tailoring the microlib input/output functions](#) on page 124

3.6 Configuring the stack and heap 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.

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

To configure the stack and heap for use with microlib, use either of the following methods:

- Define the symbol `__initial_sp` to point to the top of the stack. If using the heap, also define symbols `__heap_base` and `__heap_limit`.

`__initial_sp` must be aligned to a multiple of eight bytes.

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

- 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.

Examples

To set up the initial stack and heap pointers using `armasm` assembly language:

```
.global __initial_sp
.equ __initial_sp, 0x10000          ; top of the stack
.global __heap_base
.equ __heap_base, 0x400000         ; start of the heap
.global __heap_limit
.equ __heap_limit, 0x800000        ; end of the heap
```

To set up the initial stack pointer using inline assembler in C.

```
__asm(".global __initial_sp\n\t"
      ".equ __initial_sp, 0x10000\n\t" /* equal to the top of the stack */
);
```

To set up the heap pointer using inline assembler in C.

```
__asm(".global __heap_base\n\t"
      ".equ __heap_base, 0x400000\n\t" /* equal to the start of the heap */
      ".global __heap_limit\n\t"
      ".equ __heap_limit, 0x800000\n\t" /* equal to the end of the heap */
);
```

Related information

[About microlib](#) on page 117

[Differences between microlib and the default C library](#) on page 117

[Library heap usage requirements of microlib](#) on page 119

[ISO C features missing from microlib](#) on page 120

[Building an application with microlib](#) on page 121

[Entering and exiting programs linked with microlib](#) on page 123

[Tailoring the microlib input/output functions](#) on page 124

3.7 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 information

[About microlib](#) on page 117

[Differences between microlib and the default C library](#) on page 117

[Library heap usage requirements of microlib](#) on page 119

[ISO C features missing from microlib](#) on page 120

[Building an application with microlib](#) on page 121

[Configuring the stack and heap for use with microlib](#) on page 122

[Tailoring the microlib input/output functions](#) on page 124

3.8 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()`.



Conversions that are not supported in microlib are `%lc`, `%ls`, and `%a`.

Related information

[About microlib](#) on page 117

[Differences between microlib and the default C library](#) on page 117

[Library heap usage requirements of microlib](#) on page 119

[ISO C features missing from microlib](#) on page 120

[Building an application with microlib](#) on page 121

[Entering and exiting programs linked with microlib](#) on page 123

[Configuring the stack and heap for use with microlib](#) on page 122

4. Floating-point Support

Describes Arm support for floating-point computations.

4.1 About floating-point support

The Arm floating-point environment is an implementation of the IEEE 754-1985 standard for binary floating-point arithmetic. However, there is no guarantee that **armclang** generates floating-point exceptions in compliance with the IEEE 754-1985 standard when compiling C/C++ code.

The underlying library system is IEEE 754-1985 compliant, at least if you use the `-ffp-mode=full` model that enables all the optional features. Therefore, if you write floating-point code in assembly language, it behaves as you expect. But if you write your floating-point code in C, compiler optimizations might affect which exceptions you get.



In Arm®v8, floating-point hardware is integral to the architecture. Software floating-point is supported for AArch32 state, but is not supported for AArch64 state.

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 `-mfloat-abi=hard` selects a hardware VFP coprocessor and the option `-mfloat-abi=soft` specifies that arithmetic operations are to be performed in software, without the use of any coprocessor instructions.



In Arm Compiler 6, there is no command-line option to exclude both hardware and software floating-point computations. If the compiler encounters floating-point types in the source code, it uses software-based floating-point library functions.

Related information

[IEEE 754 arithmetic](#) on page 136

[-ffp-mode](#)

[-mfloat-abi](#)

4.2 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.



The Arm® Compiler toolchain does not support floating-point exception trapping for AArch64 targets.

Related information

[Floating-point functions for compatibility with Microsoft products](#) on page 127

[C99-compatible functions for controlling the Arm floating-point environment](#) on page 127

[C99 rounding mode and floating-point exception macros](#) on page 128

[Exception flag handling](#) on page 129

[Functions for handling rounding modes](#) on page 130

[Functions for saving and restoring the whole floating-point environment](#) on page 131

[Functions for temporarily disabling exceptions](#) on page 132

[Arm floating-point compiler extensions to the C99 interface](#) on page 133

[Example of a custom exception handler](#) on page 134

[Exception trap handling by signals](#) on page 135

4.2.1 Floating-point functions for compatibility with Microsoft products

Functions declared 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. In Arm Compiler 6 this is disabled by default, and can be enabled by the `armclang` command-line option `-fno-fast-math`.

Related information

[Controlling the Arm floating-point environment](#) on page 126

[_clearfp\(\)](#) on page 191

[_controlfp\(\)](#) on page 191

[_statusfp\(\)](#) on page 199

4.2.2 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 Arm® Compiler toolchain does not support floating-point exception trapping for AArch64 targets.



The following functionality requires a floating-point model that supports exceptions. In Arm Compiler 6 this is disabled by default, and can be enabled by the `armclang` command-line option `-ffp-mode=full`.

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 declared in `fenv.h`.

C99 defines two data types, `fenv_t` and `except_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 `except_t` is defined to hold all the information relevant to a given set of exceptions.

Related information

[Controlling the Arm floating-point environment](#) on page 126

[C99 rounding mode and floating-point exception macros](#) on page 128

[Exception flag handling](#) on page 129

[Functions for handling rounding modes](#) on page 130

[Functions for saving and restoring the whole floating-point environment](#) on page 131

[Functions for temporarily disabling exceptions](#) on page 132

[Arm floating-point compiler extensions to the C99 interface](#) on page 133

4.2.3 C99 rounding mode and floating-point exception macros

C99 defines a macro for each rounding mode and each exception



The following functionality requires a floating-point model that supports exceptions. In Arm® Compiler 6 this is disabled by default, and can be enabled by the `armclang` command-line option `-ffp-mode=full`.

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 information

[Functions for handling rounding modes](#) on page 130

[C99-compatible functions for controlling the Arm floating-point environment](#) on page 127

4.2.4 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.



The Arm® Compiler toolchain does not support floating-point exception trapping for AArch64 targets.



The following functionality requires a floating-point model that supports exceptions. In Arm Compiler 6 this is disabled by default, and can be enabled by the `armclang` command-line option `-ffp-mode=full`.

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.



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 information

[C99-compatible functions for controlling the Arm floating-point environment](#) on page 127

4.2.5 Functions for handling rounding modes

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



The following functionality requires a floating-point model that supports exceptions. In Arm® Compiler 6 this is disabled by default, and can be enabled by the `armclang` command-line option `-ffp-mode=full`.

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 information

[C99 rounding mode and floating-point exception macros](#) on page 128

[C99-compatible functions for controlling the Arm floating-point environment](#) on page 127

4.2.6 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.



The following functionality requires a floating-point model that supports exceptions. In Arm® Compiler 6 this is disabled by default, and can be enabled by the `armclang` command-line option `-ffp-mode=full`.

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 information

[C99-compatible functions for controlling the Arm floating-point environment](#) on page 127

4.2.7 Functions for temporarily disabling exceptions

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



The Arm® Compiler toolchain does not support floating-point exception trapping for AArch64 targets.



The following functionality requires a floating-point model that supports exceptions. In Arm Compiler 6 this is disabled by default, and can be enabled by the `armclang` command-line option `-ffp-mode=full`.

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 information

[C99-compatible functions for controlling the Arm floating-point environment](#) on page 127

4.2.8 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.



The Arm® Compiler toolchain does not support floating-point exception trapping for AArch64 targets.



The following functionality requires a floating-point model that supports exceptions. In Arm Compiler 6 this is disabled by default, and can be enabled by the `armclang` command-line option `-ffp-mode=full`.

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.

In AArch32 state, `fenv_t` and `fexcept_t` have the following structure:

```
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.

In AArch64 state, `fenv_t` and `fexcept_t` have the following structure:

```
typedef struct{
    unsigned __statusword;
} fenv_t, fexcept_t;
```

Related information

[Controlling the Arm floating-point environment](#) on page 126

[Example of a custom exception handler](#) on page 134

[__ieee_status\(\)](#) on page 195

[C99-compatible functions for controlling the Arm floating-point environment](#) on page 127

4.2.9 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.



The Arm® Compiler toolchain does not support floating-point exception trapping for AArch64 targets.



The following functionality requires a floating-point model that supports exceptions. In Arm Compiler 6 this is disabled by default, and can be enabled by the `armclang` command-line option `-ffp-mode=full`.

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 example

```
#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_INTTYPE_MASK) == FE_EX_INTTYPE_FLOAT)
        {
            if (op1.f == 0.0 && op2.f == 0.0)
            {
                ret.f = 1.0;
                return ret;
            }
        }
    }
}
```

```

    }
    if ((edata & FE_EX_INTTYPE_MASK) == FE_EX_INTTYPE_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 information

[Arm floating-point compiler extensions to the C99 interface](#) on page 133

4.2.10 Exception trap handling by signals

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



The Arm® Compiler toolchain does not support floating-point exception trapping for AArch64 targets.



The following functionality requires a floating-point model that supports exceptions. In Arm Compiler 6 this is disabled by default, and can be enabled by the `armclang` command-line option `-ffp-mode=full`.

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 information

[Example of a custom exception handler](#) on page 134

[Exceptions arising from IEEE 754 floating-point arithmetic](#) on page 143

[Controlling the Arm floating-point environment](#) on page 126

[Arm floating-point compiler extensions to the C99 interface](#) on page 133

[C99-compatible functions for controlling the Arm floating-point environment](#) on page 127

[__rt_raise\(\)](#) on page 167

4.3 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 C90, but that do exist in C99.

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

4.4 IEEE 754 arithmetic

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

Related information

[Basic data types for IEEE 754 arithmetic](#) on page 137

[Single precision data type for IEEE 754 arithmetic](#) on page 137

[Double precision data type for IEEE 754 arithmetic](#) on page 139

[Sample single precision floating-point values for IEEE 754 arithmetic](#) on page 140

[Sample double precision floating-point values for IEEE 754 arithmetic](#) on page 140

[IEEE 754 arithmetic and rounding](#) on page 142

[Exceptions arising from IEEE 754 floating-point arithmetic](#) on page 143

[Exception types recognized by the Arm floating-point environment](#) on page 143

4.4.1 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 information

[Sample single precision floating-point values for IEEE 754 arithmetic](#) on page 140

[Sample double precision floating-point values for IEEE 754 arithmetic](#) on page 140

[IEEE 754 arithmetic](#) on page 136

[Single precision data type for IEEE 754 arithmetic](#) on page 137

[Double precision data type for IEEE 754 arithmetic](#) on page 139

[IEEE 754 arithmetic and rounding](#) on page 142

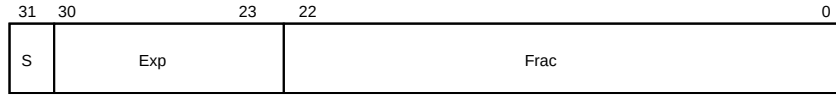
[Exceptions arising from IEEE 754 floating-point arithmetic](#) on page 143

4.4.2 Single precision data type for IEEE 754 arithmetic

A float value is 32 bits wide.

The structure is:

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



The `s` field gives the sign of the number. It is 0 for positive, or 1 for negative.

The `Exp` field gives the exponent of the number, as a power of two. It is biased by `0x7F` (127), so that very small numbers have exponents near zero and very large numbers have exponents near `0xFF` (255).

For example:

- If `Exp` = 7D (125), the number is between 0.25 and 0.5 (not including 0.5).
- If `Exp` = 7E (126), the number is between 0.5 and 1.0 (not including 1.0).
- If `Exp` = 7F (127), the number is between 1.0 and 2.0 (not including 2.0).
- If `Exp` = 80 (128), the number is between 2.0 and 4.0 (not including 4.0).
- If `Exp` = 81 (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.

For example, if `Exp` is `0x7F`:

- If `Frac` = 000000000000000000000000 (binary), the number is 1.0.
- If `Frac` = 100000000000000000000000 (binary), the number is 1.5.
- If `Frac` = 010000000000000000000000 (binary), the number is 1.25.
- If `Frac` = 110000000000000000000000 (binary), the number is 1.75.

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 information

[IEEE 754 arithmetic](#) on page 136

[Double precision data type for IEEE 754 arithmetic](#) on page 139

[IEEE 754 arithmetic and rounding](#) on page 142

[Exceptions arising from IEEE 754 floating-point arithmetic](#) on page 143

[Basic data types for IEEE 754 arithmetic](#) on page 137

[Sample single precision floating-point values for IEEE 754 arithmetic](#) on page 140

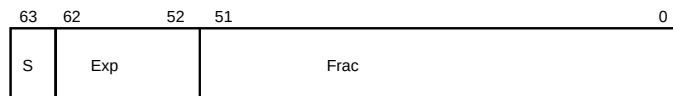
[Sample double precision floating-point values for IEEE 754 arithmetic](#) on page 140

4.4.3 Double precision data type for IEEE 754 arithmetic

A double value is 64 bits wide.

The structure is:

Figure 4-2: 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 3FF (1023) instead of 7F, so numbers between 1.0 and 2.0 have an `Exp` field of 3FF.
- The `Exp` value representing infinity and NaNs is 7FF (2047) instead of FF.

Related information

[IEEE 754 arithmetic](#) on page 136

[Single precision data type for IEEE 754 arithmetic](#) on page 137

[IEEE 754 arithmetic and rounding](#) on page 142

[Exceptions arising from IEEE 754 floating-point arithmetic](#) on page 143

[Basic data types for IEEE 754 arithmetic](#) on page 137

[Sample single precision floating-point values for IEEE 754 arithmetic](#) on page 140

[Sample double precision floating-point values for IEEE 754 arithmetic](#) on page 140

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

Sample float bit patterns, together with their mathematical values.

Table 4-1: Sample single-precision floating-point values

Float value	S	Exp	Frac	Mathematical value
0x3F800000	0	0x7F	000...000	1.0
0xBF800000	1	0x7F	000...000	-1.0
0x3F800001 ³	0	0x7F	000...001	1.000 000 119
0x3F400000	0	0x7E	100...000	0.75
0x00800000 ⁴	0	0x01	000...000	1.18×10^{-38}
0x00000001 ⁵	0	0x00	000...001	1.40×10^{-45}
0x7F7FFFFF ⁶	0	0xFE	111...111	3.40×10^{38}
0x7F800000	0	0xFF	000...000	Plus infinity
0xFF800000	1	0xFF	000...000	Minus infinity
0x00000000 ⁷	0	0x00	000...000	0.0
0x7FC00000 ⁸	0	0xFF	100...000	Quiet NaN

Related information

[Basic data types for IEEE 754 arithmetic](#) on page 137

[Sample double precision floating-point values for IEEE 754 arithmetic](#) on page 140

[IEEE 754 arithmetic](#) on page 136

[Single precision data type for IEEE 754 arithmetic](#) on page 137

[Double precision data type for IEEE 754 arithmetic](#) on page 139

[IEEE 754 arithmetic and rounding](#) on page 142

[Exceptions arising from IEEE 754 floating-point arithmetic](#) on page 143

-
- ³ 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.
- ⁴ 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.
- ⁵ The smallest positive number that can be distinguished from zero. This is the absolute lower limit of the format.
- ⁶ The largest finite number that can be stored. Attempting to increase this number by addition or multiplication causes overflow and generates infinity (in general).
- ⁷ 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.
- ⁸ There are two types of NaNs, signaling NaNs and quiet NaNs. Quiet NaNs have a 1 in the first bit of Frac, and signaling NaNs have a zero there. The difference is that signaling NaNs cause an exception when used, whereas quiet NaNs do not.

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

Sample double bit patterns, together with their mathematical values.

Table 4-2: Sample double-precision floating-point values

Double value	S	Exp	Frac	Mathematical value
0x3FF00000 00000000	0	0x3FF	000...000	1.0
0xBFF00000 00000000	1	0x3FF	000...000	-1.0
0x3FF00000 00000001 ⁹	0	0x3FF	000...001	1.000 000 000 000 000 222
0x3FE80000 00000000	0	0x3FE	100...000	0.75
0x00100000 00000000 ¹⁰	0	0x001	000...000	2.23×10^{-308}
0x00000000 00000001 ¹¹	0	0x000	000...001	4.94×10^{-324}
0x7FEFFFFFFF FFFFFFFF ¹²	0	0x7FE	111...111	1.80×10^{308}
0x7FF00000 00000000	0	0x7FF	000...000	Plus infinity
0xFFF00000 00000000	1	0x7FF	000...000	Minus infinity
0x00000000 00000000 ¹³	0	0x000	000...000	0.0
0x7FF00000 00000001	0	0x7FF	000...001	Signaling NaN
0x7FF80000 00000000 ¹⁴	0	0x7FF	100...000	Quiet NaN

Related information

[Basic data types for IEEE 754 arithmetic](#) on page 137

- ⁹ 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.
- ¹⁰ 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.
- ¹¹ The smallest positive number that can be distinguished from zero. This is the absolute lower limit of the format.
- ¹² The largest finite number that can be stored. Attempting to increase this number by addition or multiplication causes overflow and generates infinity (in general).
- ¹³ 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.
- ¹⁴ 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.

[Sample single precision floating-point values for IEEE 754 arithmetic](#) on page 140

[IEEE 754 arithmetic](#) on page 136

[Single precision data type for IEEE 754 arithmetic](#) on page 137

[Double precision data type for IEEE 754 arithmetic](#) on page 139

[IEEE 754 arithmetic and rounding](#) on page 142

[Exceptions arising from IEEE 754 floating-point arithmetic](#) on page 143

4.4.6 IEEE 754 arithmetic and rounding

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



The Arm® Compiler toolchain does not support floating-point exception trapping for AArch64 targets.

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 information

[IEEE 754 arithmetic](#) on page 136

[Single precision data type for IEEE 754 arithmetic](#) on page 137

[Double precision data type for IEEE 754 arithmetic](#) on page 139

[Exceptions arising from IEEE 754 floating-point arithmetic](#) on page 143

[Basic data types for IEEE 754 arithmetic](#) on page 137

[Sample single precision floating-point values for IEEE 754 arithmetic](#) on page 140

[Sample double precision floating-point values for IEEE 754 arithmetic](#) on page 140

4.4.7 Exceptions arising from IEEE 754 floating-point arithmetic

Floating-point arithmetic operations can run into various problems. These are known as exceptions, because they indicate unusual or exceptional situations.

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).



The Arm® Compiler toolchain does not support floating-point exception trapping for AArch64 targets.

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 information

[Example of a custom exception handler](#) on page 134

[Exception trap handling by signals](#) on page 135

[Controlling the Arm floating-point environment](#) on page 126

[Arm floating-point compiler extensions to the C99 interface](#) on page 133

[C99-compatible functions for controlling the Arm floating-point environment](#) on page 127

[__rt_raise\(\)](#) on page 167

4.4.8 Exception types recognized by the Arm floating-point environment

The Arm floating-point environment recognizes several different types of exception.



The Arm® Compiler toolchain does not support floating-point exception trapping for AArch64 targets.

The following types of exception are recognized:

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.



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 information

[Exception flag handling](#) on page 129

[Example of a custom exception handler](#) on page 134

[Exception trap handling by signals](#) on page 135

[IEEE 754 arithmetic](#) on page 136

[Exceptions arising from IEEE 754 floating-point arithmetic](#) on page 143

[ISO-compliant implementation of signals supported by the `signal\(\)` function in the C library and additional type arguments](#) on page 103

[Sample single precision floating-point values for IEEE 754 arithmetic](#) on page 140

[IEEE Standard for Floating-Point Arithmetic \(IEEE 754\), 1985 version](#)

5. The C and C++ Library Functions Reference

Describes the 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.

5.1 `__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`.

Syntax

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

Usage

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.

Related information

[C Library ABI for the Arm Architecture](#)

5.2 `alloca()`

Declared in `alloca.h`, the `alloca()` function allocates local storage in a function. It returns a pointer to the number of bytes of memory allocated.

Syntax

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

Usage

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.



`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.

Returns

Returns in `size` a pointer to the number of bytes of memory allocated.

Related information

[Arm C libraries and thread-safe functions](#) on page 29

[Standalone C library functions](#) on page 60

5.3 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.



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

Returns

The returned value is an unsigned integer.

Related information

[Direct semihosting C library function dependencies](#) on page 57

5.4 `_clock_init()`

Declared 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.



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()` later 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 information

[Direct semihosting C library function dependencies](#) on page 57

5.5 `__default_signal_handler()`

Declared 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, intptr_t 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, intptr_t 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`.



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

Related information

[ISO-compliant implementation of signals supported by the `signal\(\)` function in the C library and additional type arguments](#) on page 103

5.6 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()`.



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 information

[__aeabi_errno_addr\(\)](#) on page 146

5.7 `_findlocale()`

Declared 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 information

[Assembler macros that tailor locale functions in the C library](#) on page 71

[Link time selection of the locale subsystem in the C library](#) on page 71

[Runtime selection of the locale subsystem in the C library](#) on page 73

[Definition of locale data blocks in the C library](#) on page 73

5.8 `_fisatty()`

Declared 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 information

[_sys_istty\(\)](#) on page 174

5.9 _get_lconv()

Declared 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 information

[_findlocale\(\)](#) on page 149

[lconv structure](#) on page 154

[localeconv\(\)](#) on page 156

[setlocale\(\)](#) on page 168

5.10 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()`.

5.11 _getenv_init()

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

This function 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.

5.12 __heapstats()

Declared in `stdlib.h`, the `__heapstats()` function displays statistics on the state of the storage allocation heap.

Syntax

```
void __heapstats(int (*dprint)(void *param, char const *format,...), void *param);
```

Usage

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);
```



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.

5.13 __heapvalid()

Declared in `stdlib.h`, the `__heapvalid()` function performs a consistency check on the heap. This function assumes a single contiguous block of memory for the heap. If you use the `__rt_heap_extend()` function to add a non-contiguous block of memory to the heap, then you must not use `__heapvalid()`.

Syntax

```
int __heapvalid(int (*dprint)(void *param, char const *format,...), void *param, int verbose);
```

Usage

`__heapvalid()` outputs full information about every free block if the 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.

Related information

[__rt_heap_extend\(\)](#) on page 164

5.14 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 information

[_findlocale\(\)](#) on page 149

[_get_lconv\(\)](#) on page 151

[localeconv\(\)](#) on page 156

[setlocale\(\)](#) on page 168

5.15 localeconv()

Declared 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 information

[_findlocale\(\)](#) on page 149

[lconv structure](#) on page 154

[_get_lconv\(\)](#) on page 151

[setlocale\(\)](#) on page 168

5.16 [_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 Declared 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 `_membitcpyb1()`, 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.

5.17 `_platform_pre_stackheap_init()`

If defined, `_platform_pre_stackheap_init` is called by `__rt_entry` before stack and heap initialization. Define this function to perform hardware initialization after scatter-loading but before stack and heap initialization.

Because `_platform_pre_stackheap_init` is called before the stack initialization, either it must not use the stack or the SP must already be valid.

Invalidating the Armv8 instruction cache

To invalidate the Arm®v8 instruction cache after scatter-loading and before initialization of the stack and heap, you must:

- Implement instruction cache invalidation code in `_platform_pre_stackheap_init`.

- Ensure that all code that is executed from the program entry, up to and including `_platform_pre_stackheap_init`, is located in a root region.

Where a processor starts in AArch64 state, then switches to AArch32 state, it is possible that addresses are speculatively prefetched, and therefore cached, while in AArch64 state. If the MMU has remained off while in AArch64 state, a processor is allowed to speculatively prefetch from any address either within:

- The same page as an architecturally executed instruction.
- The following page, where `page` is the smallest supported granule `sizeof` for the processor.

If you have AArch64 startup code that switches to AArch32 state to run `__main` and then run C/C++ applications, then the cache invalidation must be done in AArch32 state.

Example

Invalidate caches in AArch64 as follows:

```
_platform_pre_stackheap_init:
    dsb ish          // ensure all previous stores have completed
                      // before invalidating
    ic ialluis       // I cache invalidate all inner shareable to PoU
                      // (which includes secondary cores)
    dsb ish          // ensure completion on inner shareable domain
                      // (which includes secondary cores)
    isb
    b InvalidateUDCaches // only needed if the MMU is on at this point
```



`b` is a tail-call to avoid saving the return address.

Related information

[__rt_entry](#) on page 162

[Placing code in a root region](#)

5.18 posix_memalign()

Declared in `stdlib.h`, the `posix_memalign()` function provides aligned memory allocation.

This function 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](#)

5.19 __raise()

Declared 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, intptr_t 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, intptr_t 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, intptr_t 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 information

[Thread safety in the Arm C library](#) on page 37

[ISO-compliant implementation of signals supported by the `signal\(\)` function in the C library and additional type arguments](#) on page 103

5.20 `_rand_r()`

Declared 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 information

[_srand_r\(\)](#) on page 169

5.21 remove()

This function 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 information

[Direct semihosting C library function dependencies](#) on page 57

5.22 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 with a name that is the string pointed to by `old` to be later 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 information

[Direct semihosting C library function dependencies](#) on page 57

5.23 __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. Performs hardware initialization by calling `_platform_pre_stackheap_init()`, if this function is defined.
2. Sets up the heap and stack.
3. Initializes the C library by calling `__rt_lib_init`.
4. Calls `main()`.
5. Shuts down the C library, by calling `__rt_lib_shutdown`.
6. 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.

Related information

[_platform_pre_stackheap_init\(\)](#) on page 158

5.24 __rt_exit()

Declared 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()`.

The `__rt_exit()` function 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 information

[_sys_exit\(\)](#) on page 173

5.25 __rt_fp_status_addr()

Declared 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`.

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

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 information

[Thread safety in the Arm C library](#) on page 37

5.26 __rt_heap_extend()

Declared in `rt_heap.h`, the `__rt_heap_extend()` function returns a new 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 size_t __rt_heap_extend(size_t 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 one of the following:
 - In AArch32 state, a multiple of 8 bytes of at least the requested size.
 - In AArch64 state, a multiple of 16 bytes of at least the requested size.
 - 0, denoting that the request cannot be honored.
- The returned base address is aligned on:
 - In AArch32 state, an 8-byte boundary.
 - In AArch64 state, a 16-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 information

[Stack pointer initialization and heap bounds](#) on page 85

5.27 __rt_lib_init()

Declared in `rt_misc.h`, this is the library initialization function and is the companion to `__rt_lib_shutdown()`.

Syntax

For AArch32 targets:

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

For AArch64 targets:

```
extern __attribute__((value_in_regs)) struct __argc_argv __rt_lib_init(unsigned long
heapbase, unsigned long 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.

For AArch32 targets:

```
struct __argc_argv
{
    int argc;
    char **argv;
    void *r2; // optional extra arguments that on entry to main() are
    void *r3; // found in registers R2 and R3.
};
```

For AArch64 targets:

```
struct __argc_argv
{
    long argc;
    char **argv;
    void *r2; // optional extra arguments that on entry to main() are
    void *r3; // found in registers X2 (alias for R2) and X3 (alias for R3).
};
```

5.28 __rt_lib_shutdown()

Declared 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.

5.29 __rt_raise()

Declared in `rt_misc.h`, the `__rt_raise()` function raises a signal to indicate a runtime anomaly.

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

Syntax

```
void __rt_raise(int signal, intptr_t 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 information

[ISO-compliant implementation of signals supported by the `signal\(\)` function in the C library and additional type arguments](#) on page 103

5.30 `__rt_stackheap_init()`

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 information

[Stack pointer initialization and heap bounds](#) on page 85

5.31 `setlocale()`

Declared 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` symbol is imported (`__use_iso8859_ctype`, `__use_sjis_ctype`, or `__use_utf8_ctype`), and on user-defined locales.



Only one `__use_x_ctype` symbol can be imported.

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 information

[ISO8859-1 implementation](#) on page 72

[Shift-JIS and UTF-8 implementation](#) on page 73

[Definition of locale data blocks in the C library](#) on page 73

5.32 _srand_r()

Declared 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 information

[_rand_r\(\)](#) on page 161

5.33 strcasecmp()

Declared 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](#)

5.34 strlcat()

Declared in `string.h`, the `strlcat()` function concatenates two strings.

Syntax

```
extern size_t strlcat(char *dst, const char *src, size_t size);
```

Usage

`strlcat()` 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.

5.35 `strlcpy()`

Declared in `string.h`, the `strlcpy()` function copies up to `size-1` characters from the `NUL`-terminated string `src` to `dst`.

Syntax

```
extern size_t strlcpy(char *dst, const char *src, size_t size);
```

Usage

`strlcpy()` 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.

5.36 `strncasecmp()`

Declared 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](#)

5.37 `_sys_close()`

Declared 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 information

[Direct semihosting C library function dependencies](#) on page 57

5.38 `_sys_command_string()`

Declared 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()`.



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 information

[Direct semihosting C library function dependencies](#) on page 57

5.39 `_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).

5.40 `_sys_exit()`

Declared 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 information

[Direct semihosting C library function dependencies](#) on page 57

5.41 `_sys_flen()`

Declared 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 information

[Direct semihosting C library function dependencies](#) on page 57

5.42 `_sys_istty()`

Declared in `rt_sys.h`, the `_sys_istty()` function determines whether a file handle is attached to an interactive device.

Syntax

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

Usage

The Arm® libraries call `__sys_istty()` to determine whether a file handle (that is being used for an output file stream) is attached to an interactive device.

For file streams that are attached to interactive devices, the Arm library:

- Provides unbuffered behavior by default, in the absence of a call to `setbuf()` or `setvbuf()`.
- Prohibits seeking.

Restriction

`stdin`, `stdout`, and `stderr` are assumed to be interactive devices. They are line-buffered at program startup, regardless of what `_sys_istty` reports for them. An exception is if they have been redirected on the command line.



This restriction does not apply when using `microlib`. In `microlib`, `stdin`, `stdout`, and `stderr` are always unbuffered.

Returns

The return value is one of the following values:

0

`fh` is not attached to an interactive device.

1

`fh` is attached to an interactive device.

other

An error occurred.

Related information

[Direct semihosting C library function dependencies](#) on page 57

5.43 `_sys_open()`

Declared 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 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 information

[Direct semihosting C library function dependencies](#) on page 57

5.44 `_sys_read()`

Declared 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);
```



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, minus the number of bytes that 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 information

[Direct semihosting C library function dependencies](#) on page 57

5.45 `_sys_seek()`

Declared 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 information

[Direct semihosting C library function dependencies](#) on page 57

5.46 _sys_tmpnam()

Declared 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 information

[Direct semihosting C library function dependencies](#) on page 57

5.47 _sys_write()

Declared 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);
```



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 information

[Direct semihosting C library function dependencies](#) on page 57

5.48 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 information

[Direct semihosting C library function dependencies](#) on page 57

5.49 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 `((time_t)-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 information

[Direct semihosting C library function dependencies](#) on page 57

5.50 _ttywrch()

Declared 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 information

[Direct semihosting C library function dependencies](#) on page 57

5.51 __user_heap_extend()

Declared 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 size_t __user_heap_extend(int var0, void **base, size_t 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 is one of the following:
 - In AArch32 state, a multiple of 8 bytes of at least the requested size.

- In AArch64 state, a multiple of 16 bytes of at least the requested size.
- 0, denoting that the request cannot be honored.
- The returned base address is aligned on:
 - In AArch32 state, an 8-byte boundary.
 - In AArch64 state, a 16-byte boundary.
- Size is measured in bytes.
- The function is subject only to *Procedure Call Standard for the Arm Architecture* (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 function `__user_heap_extend()` is only weakly referenced by the C library. This means that unused section elimination might remove the `__user_heap_extend()` function at link time, and in this case, the heap cannot be extended. To prevent this situation, you can use `armlink --keep` to prevent the function from being eliminated. Alternatively, include an explicit reference to `__user_heap_extend()` from a part of the application code that you are sure is not removed at link time.

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 information

[Stack pointer initialization and heap bounds](#) on page 85

5.52 __user_heap_extent()

If defined, the `__user_heap_extent()` function returns the bounds of the memory available to the Heap2 allocator.

See `rt_misc.h`.



If you provide an implementation of this function, then you must link with either the `--keep` or `--no_remove` `armlink` options. Otherwise, the unused section elimination feature of the linker might remove your implementation.

Syntax

For AArch32 targets:

```
extern __attribute__((value_in_regs)) struct __heap_extent  
__user_heap_extent(unsigned ignore1, size_t ignore2);
```

For AArch64 targets:

```
extern __attribute__((value_in_regs)) struct __heap_extent  
__user_heap_extent(unsigned long ignore1, size_t ignore2);
```

Usage

The parameters *ignore1* and *ignore2* are the default values for the base address and size of the heap. They are for information only and can be ignored.

You only need to implement this function if you are using the Heap2 allocator, which is also part of the C library. This function has no default implementation. The Heap2 allocator calls it during heap initialization to determine the maximum address range that the heap can occupy. The function returns the base address of the heap and the total number of bytes available to the heap, rounded up to the next power of two.

For example, if you want to specify that all your heap allocations comes from address 0x80000000 and above, and that the heap has a total maximum size of 3MiB, `__user_heap_extent()` must return `base=0x80000000` and `range=0x400000`, which is 3MiB rounded up to the next power of two.

Related information

[Stack pointer initialization and heap bounds](#) on page 85

5.53 __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, if the program uses the heap.
 - In AArch32 state, register R0 contains the heap base.
 - In AArch64 state, register X0 contains the heap base.
- Stack base in `sp`.

- Heap limit, if the program uses the heap and uses two-region memory.
 - In AArch32 state, register R2 contains the heap limit.
 - In AArch64 state, register X2 contains the heap limit.

If this function is re-implemented, it must:

- Preserve the registers required by the PCS, except for SP.
- Ensure alignment of the stack and heap:
 - In AArch32 state, ensure that the stack base and heap base are a multiple of 8 to maintain 8-byte alignment of the stack and heap.
 - In AArch64 state, ensure that the stack base and heap base are a multiple of 16 to maintain 16-byte alignment of the stack and heap.

To create a version of `__user_setup_stackheap()` that inherits `sp` from the execution environment and does not have a heap:

- In AArch32 state, set `r0` and `r2` to zero and return.
- In AArch64 state, set `x0` and `x2` 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.



Any re-implementation of `__user_setup_stackheap()` must be in assembler.

Related information

[Direct semihosting C library function dependencies](#) on page 57

5.54 `__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 0 and 4 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 information

[Stack pointer initialization and heap bounds](#) on page 85

5.55 wcscasecmp()

Declared in `wchar.h`, the `wcscasecmp()` function performs a case-insensitive string comparison test on wide characters.

This function 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);
```

5.56 wcsncasecmp()

Declared in `wchar.h`, the `wcsncasecmp()` function performs a case-insensitive string comparison test of not more than a specified number of wide characters.

This function 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);
```

5.57 wcstombs()

Declared 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);
```

5.58 Thread-safe C library functions

The following table shows the C library functions that are thread-safe.

Table 5-1: Functions that are thread-safe

Functions	Description
<code>calloc()</code> , <code>free()</code> , <code>malloc()</code> , <code>realloc()</code>	The heap functions are thread-safe if the <code>_mutex_*</code> 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).
<code>alloca()</code>	<code>alloca()</code> is thread-safe because it allocates memory on the stack.
<code>abort()</code> , <code>raise()</code> , <code>signal()</code> , <code>fenv.h</code>	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 <code>signal()</code> or an <code>fenv.h</code> 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.

Functions	Description
<code>clearerr()</code> , <code>fclose()</code> , <code>feof()</code> , <code>ferror()</code> , <code>fflush()</code> , <code>fgetc()</code> , <code>fputc()</code> , <code>fseek()</code> , <code>fwrite()</code> , <code>gets()</code> , <code>putchar()</code> , <code>putc()</code> , <code>tmpnam()</code>	The <code>stdio</code> library is thread-safe if the <code>_mutex_*</code> functions are implemented. Each individual stream is protected by a lock, so two threads can each open their own <code>stdio</code> stream and use it, without interfering with one another. If two threads both want to read or write the same stream, locking at the <code>fgetc()</code> and <code>fputc()</code> level prevents data corruption, but it is possible that the individual characters output by each thread might be interleaved in a confusing way. Note: <code>tmpnam()</code> also contains a static buffer but this is only used if the argument is <code>NULL</code>. To ensure that your use of <code>tmpnam()</code> is thread-safe, supply your own buffer space.
<code>fprintf()</code> , <code>printf()</code> , <code>vfprintf()</code> , <code>vprintf()</code> , <code>fscanf()</code> , <code>scanf()</code>	When using these functions: - The standard C <code>printf()</code> and <code>scanf()</code> functions use <code>stdio</code> so they are thread-safe. - The standard C <code>printf()</code> function is susceptible to changes in the locale settings if called in a multithreaded program.
<code>clock()</code>	<code>clock()</code> contains static data that is written once at program startup and then only ever read. Therefore, <code>clock()</code> is thread-safe provided no extra threads are already running at the time that the library is initialized.
<code>errno</code>	<code>errno</code> is thread-safe. Each thread has its own <code>errno</code> stored in a <code>__user_perthread_libspace</code> block. This means that each thread can call <code>errno</code>-setting functions independently and then check <code>errno</code> afterwards without interference from other threads.
<code>atexit()</code>	The list of exit functions maintained by <code>atexit()</code> is process-global and protected by a lock. In the worst case, if more than one thread calls <code>atexit()</code>, the order that exit functions are called cannot be guaranteed.
<code>abs()</code> , <code>acos()</code> , <code>asin()</code> , <code>atan()</code> , <code>atan2()</code> , <code>atof()</code> , <code>atol()</code> , <code>atoi()</code> , <code>bsearch()</code> , <code>ceil()</code> , <code>cos()</code> , <code>cosh()</code> , <code>difftime()</code> , <code>div()</code> , <code>exp()</code> , <code>fabs()</code> , <code>floor()</code> , <code>fmod()</code> , <code>frexp()</code> , <code>labs()</code> , <code>ldexp()</code> , <code>ldiv()</code> , <code>log()</code> , <code>log10()</code> , <code>memchr()</code> , <code>memcmp()</code> , <code>memcpy()</code> , <code>memmove()</code> , <code>memset()</code> , <code>mktime()</code> , <code>modf()</code> , <code>pow()</code> , <code>qsort()</code> , <code>sin()</code> , <code>sinh()</code> , <code>sqrt()</code> , <code>strcat()</code> , <code>strchr()</code> , <code>strcmp()</code> , <code>strcpy()</code> , <code>strcspn()</code> , <code>strlcat()</code> , <code>strncpy()</code> , <code>strlen()</code> , <code>strncat()</code> , <code>strncmp()</code> , <code>strncpy()</code> , <code>strpbrk()</code> , <code>strrchr()</code> , <code>strspn()</code> , <code>strstr()</code> , <code>strxfrm()</code> , <code>tan()</code> , <code>tanh()</code>	These functions are inherently thread-safe.
<code>longjmp()</code> , <code>setjmp()</code>	Although <code>setjmp()</code> and <code>longjmp()</code> keep data in <code>__user_libspace</code> , they call the <code>__alloca_*</code> functions, that are thread-safe.

Functions	Description
<code>remove()</code> , <code>rename()</code> , <code>time()</code>	These functions use interrupts that communicate with the Arm debugging environments. Typically, you have to reimplement these for a real-world application.
<code>snprintf()</code> , <code>sprintf()</code> , <code>vsnprintf()</code> , <code>vsprintf()</code> , <code>sscanf()</code> , <code>isalnum()</code> , <code>isalpha()</code> , <code>iscntrl()</code> , <code>isdigit()</code> , <code>isgraph()</code> , <code>islower()</code> , <code>isprint()</code> , <code>ispunct()</code> , <code>isspace()</code> , <code>isupper()</code> , <code>isxdigit()</code> , <code>tolower()</code> , <code>toupper()</code> , <code>strcoll()</code> , <code>strtod()</code> , <code>strtol()</code> , <code>strtoul()</code> , <code>strftime()</code>	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 <code>sprintf()</code> and <code>sscanf()</code> , do not depend on the <code>stdio</code> library.
<code>stdin</code> , <code>stdout</code> , <code>stderr</code>	These functions are thread-safe.

Related information

[alloca\(\)](#) on page 146

5.59 C library functions that are not thread-safe

The following table shows the C library functions that are not thread-safe.

Table 5-2: Functions that are not thread-safe

Functions	Description
<code>asctime()</code> , <code>localtime()</code> , <code>strtok()</code>	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, <code>_asctime_r()</code> , <code>_localtime_r()</code> , and <code>_strtok_r()</code> . Arm recommends that you use these functions instead to ensure safety. These reentrant versions take additional parameters. <code>_asctime_r()</code> takes an additional parameter that is a pointer to a buffer that the output string is written into. <code>_localtime_r()</code> takes an additional parameter that is a pointer to a <code>struct tm</code> , that the result is written into. <code>_strtok_r()</code> takes an additional parameter that is a pointer to a char pointer to the next token.
<code>exit()</code>	Do not call <code>exit()</code> in a multithreaded program even if you have provided an implementation of the underlying <code>_sys_exit()</code> that actually terminates all threads. In this case, <code>exit()</code> cleans up <i>before</i> calling <code>_sys_exit()</code> so disrupts other threads.
<code>gamma()</code> , <code>lgamma()</code> , <code>lgammaf()</code> , <code>lgammal()</code> ¹⁵	These extended mathlib functions use a global variable, <code>_signgam</code> , so are not thread-safe.

¹⁵ If migrating from RVCT, be aware that `gamma()` is deprecated in Arm Compiler 4.1 and later.

Functions	Description
<code>mbrlen()</code> , <code>mbsrtowcs()</code> , <code>mbrtowc()</code> , <code>wcrtomb()</code> , <code>wcsrtombs()</code>	The C90 multibyte conversion functions (declared in <code>stdlib.h</code>) are not thread-safe, for example <code>mblen()</code> and <code>mbtowc()</code>, because they contain internal static state that is shared between all threads without locking. However, the extended restartable versions (declared in <code>wchar.h</code>) are thread-safe, for example <code>mbrtowc()</code> and <code>wcrtomb()</code>, provided you pass in a pointer to your own <code>mbstate_t</code> object. You must exclusively use these functions with non-NULL <code>mbstate_t</code> * parameters if you want to ensure thread-safety when handling multibyte strings.
<code>rand()</code> , <code>srand()</code>	These functions keep internal state that is both global and unprotected. This means that calls to <code>rand()</code> are never thread-safe. Arm recommends that you do one of the following: Use the reentrant versions <code>_rand_r()</code> and <code>_srand_r()</code> 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 <code>rand()</code> at a time, for example, by defining <code>\$Sub \$ \$rand()</code> 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. <code>_rand_r()</code> and <code>_srand_r()</code> both take an additional parameter that is a pointer to a buffer storing the state of the random number generator.

Functions	Description
<code>setlocale()</code> , <code>localeconv()</code>	<code>setlocale()</code> 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 <code>setlocale()</code> 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 <code>strtod()</code> and <code>sprintf()</code>, read the current locale settings. Therefore, if one thread calls <code>setlocale()</code> 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 <code>setlocale()</code>. Arm recommends that you either: - Choose the locale you want and call <code>setlocale()</code> 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 <code>_setlocale_r()</code> 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. The buffer must be at least <code>_SETLOCALE_R_BUFSIZE</code> bytes long, including space for a trailing NUL. Be aware that <code>_setlocale_r()</code> is not fully thread-safe when accessed concurrently to <i>change</i> locale settings. This access is not lock-protected. Also, be aware that <code>localeconv()</code> is not thread-safe. Call the Arm function <code>_get_lconv()</code> with a pointer to a user-supplied buffer instead.

Related information

[_rand_r\(\)](#) on page 161

[_srand_r\(\)](#) on page 169

5.60 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.



Arm recommends not using `__user_initial_stackheap()` in new code. Instead, use its modern equivalent, `__user_setup_stackheap()`.

Syntax

For targets in AArch32 state:

```
extern __attribute__((value_in_regs)) struct __initial_stackheap
__user_initial_stackheap(unsigned R0, unsigned SP, unsigned R2, unsigned SL);
```

For targets in AArch64 state:

```
extern __attribute__((value_in_regs)) struct __initial_stackheap
__user_initial_stackheap(unsigned long R0, unsigned long SP, unsigned long R2,
unsigned long 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)` when targeting AArch32 state.
- Not corrupt registers other than `r16 (ip0)` and `r17 (ip1)` when targeting AArch64 state.
- Maintain 8-byte alignment of the heap when targeting AArch32 state.
- Maintain 16-byte alignment of the heap when targeting AArch64 state.

When `__user_initial_stackheap()` is called, the argument in `r1` is the value that `sp` had when `__main()` was called. 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.

For targets in AArch32 state, 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 */
};
```

For targets in AArch64 state, the definition of `__initial_stackheap` in `rt_misc.h` is:

```
struct __initial_stackheap {
    unsigned long heap_base; /* low-address end of initial heap */
    unsigned long stack_base; /* high-address end of initial stack */
    unsigned long heap_limit; /* high-address end of initial heap */
    unsigned long stack_limit; /* unused */
};
```



The value of `stack_base` is 1 greater than the highest address used by the stack because a full-descending stack is used.

Related information

[Stack pointer initialization and heap bounds](#) on page 85

6. Floating-point Support Functions Reference

Describes Arm support for floating-point functions.

6.1 `_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);
```



This function requires a floating-point model that supports exceptions. In Arm® Compiler 6 this is disabled by default, and can be enabled by the `armclang` command-line option `-ffp-mode=full`.

Related information

[_controlfp\(\)](#) on page 191

[_statusfp\(\)](#) on page 199

6.2 `_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 Arm® Compiler toolchain does not support floating-point exception trapping for AArch64 targets.

The function prototype for `_controlfp()` is:

```
unsigned int _controlfp(unsigned int new, unsigned int mask);
```



This function requires a floating-point model that supports exceptions. In Arm Compiler 6 this is disabled by default, and can be enabled by the `armclang` command-line option `-ffp-mode=full`.

`_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.



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 6-1: `_controlfp` argument macros

Macro	Description
<code>_MCW_EM</code>	Mask containing all exception bits
<code>_EM_INVALID</code>	Bit describing the Invalid Operation exception
<code>_EM_ZERODIVIDE</code>	Bit describing the Divide by Zero exception
<code>_EM_OVERFLOW</code>	Bit describing the Overflow exception
<code>_EM_UNDERFLOW</code>	Bit describing the Underflow exception
<code>_EM_INEXACT</code>	Bit describing the Inexact Result exception
<code>_MCW_RC</code>	Mask for the rounding mode field
<code>_RC_CHOP</code>	Rounding mode value describing Round Toward Zero
<code>_RC_UP</code>	Rounding mode value describing Round Up
<code>_RC_DOWN</code>	Rounding mode value describing Round Down
<code>_RC_NEAR</code>	Rounding mode value describing Round To Nearest



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 information

[_clearfp\(\)](#) on page 191

[_statusfp\(\)](#) on page 199

6.3 __fp_status()

The Arm® Compiler toolchain supports an interface to the status word in the floating-point environment. Some older versions of the Arm libraries implemented a function called `__fp_status()` to provide this interface.



The Arm Compiler toolchain does not support floating-point exception trapping for AArch64 targets.

`__fp_status()` 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 declared in `stdlib.h`.

The function prototype for `__fp_status()` is:

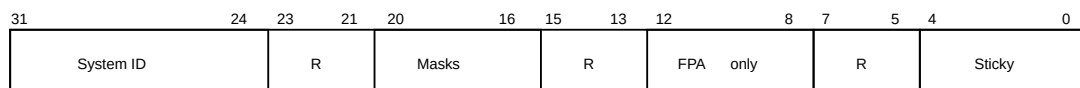
```
unsigned int __fp_status(unsigned int mask, unsigned int flags);
```



This function requires a floating-point model that supports exceptions. In Arm Compiler 6 this is disabled by default, and can be enabled by the `armclang` command-line option `-ffp-mode=full`.

The layout of the status word as seen by `__fp_status()` is as follows:

Figure 6-1: 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 40 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 information

[Controlling the Arm floating-point environment](#) on page 126

[__ieee_status\(\)](#) on page 195

6.4 gamma(), gamma_r()

The `gamma()` and `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 *);
```



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()`.



These functions are deprecated.

6.5 __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.



The Arm Compiler toolchain does not support floating-point exception trapping for AArch64 targets.

`__ieee_status()` is declared in `fenv.h`.

The function prototype for `__ieee_status()` is:

```
unsigned int __ieee_status(unsigned int mask, unsigned int flags);
```



This function requires a floating-point model that supports exceptions. In Arm Compiler 6 this is disabled by default, and can be enabled by the `armclang` command-line option `-ffp-mode=full`.

`__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;
```

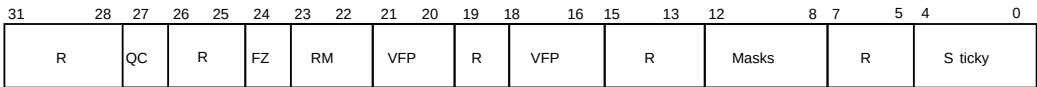
Four different operations can be performed on each bit of the status word, depending on the corresponding bits in mask and flags.

Table 6-2: 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:

Figure 6-2: 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 6-3: Rounding mode control

Bits	Rounding mode
00	Round to nearest
01	Round up
10	Round down
11	Round toward zero



The relevant libraries are selected by default in Arm Compiler 6. For more information, see the `armclang` command-line option `-ffp-mode`.

- 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.



- The FZ bit in the IEEE status word is not supported by any of the `fp11b` variants. This means that switching between flushing to zero and not flushing to zero is not possible with any variant of `fp11b` 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 information

[Controlling the Arm floating-point environment](#) on page 126

[Arm floating-point compiler extensions to the C99 interface](#) on page 133

[C and C++ library naming conventions](#) on page 112

[Exceptions arising from IEEE 754 floating-point arithmetic](#) on page 143

6.6 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 2⁵², these functions return an `ERANGE` error, denoting total loss of significance in the result.



These functions are deprecated.

6.7 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);
```



This function is deprecated.

6.8 _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);
```



This function requires a floating-point model that supports exceptions. In Arm® Compiler 6 this is disabled by default, and can be enabled by the `armclang` command-line option `-ffp-mode=full`.

Related information

[_clearfp\(\)](#) on page 191

[_controlfp\(\)](#) on page 191

6.9 `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 π times 2^{52} , these functions return an `ERANGE` error, denoting total loss of significance in the result.



These functions are deprecated.

7. Arm C and C++ Libraries and Floating-Point Support User Guide Changes

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

7.1 Changes for the Arm C and C++ Libraries and Floating-Point Support User Guide

Changes that have been made to the *Arm C and C++ Libraries and Floating-Point Support User Guide* are listed with the latest version first.

Table 7-1: Changes between 6.6.5 (revision L) and 6.6.4 (revision K)

Change	Topics affected
[SDCOMP-59739] Added note about new and delete potentially being omitted by optimizer in C++03 and C++11, as well as C++14 which is standard.	Standard C++ library implementation definition.
[SDCOMP-57264] Added note on mixing objects compiled with different C/C++ standards.	- Summary of the C and C++ runtime libraries. - Compliance with the Application Binary Interface (ABI) for the Arm architecture.

Table 7-2: Changes between 6.6.4 (revision K) and 6.6.3 (revision J)

Change	Topics affected
[SDCOMP-52286] Modified the statement that a mutex is always 4 bytes. A mutex is 4 bytes for AArch32 and 8 bytes for AArch64.	- Management of locks in multithreaded applications. - Choosing a heap implementation for memory allocation functions.
[SDCOMP-54116] Added descriptions for SIGCPPL and SIGOUTOFHEAP.	ISO-compliant implementation of signals supported by the <code>signal()</code> function in the C library and additional type arguments.
[SDCOMP-53622] Added a statement about stack and heap alignment for AArch32 and AArch64.	- Stack pointer initialization and heap bounds. <code>__user_setup_stackheap()</code>.
[SDCOMP-54283] Added a note about using the <code>armlink</code> options <code>--keep</code> or <code>--no_remove</code> when implementing your own version of <code>__user_heap_extent()</code> .	<code>__user_heap_extent()</code>.
[SDCOMP-55662] Added sentence for assumption about contiguous heap.	<code>__heapvalid()</code>.