ARM® Compiler toolchain

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

ARM DUI 0492A (ID052210)

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

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Confidentiality	Change
28 May 2010	A	Non-Confidential	ARM Compiler v4.1 Release

Proprietary Notice

Confidentiality Status

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

Unrestricted Access is an ARM internal classification.

Product Status

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

Web Address

`http://www.arm.com`

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

Chapter 1
Conventions and feedback

The following topics describe typographical conventions and how to give feedback:

1.1 Typographical conventions

The following typographical conventions are used in this book:

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

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

`monospace italic`

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

`monospace bold`

Denotes language keywords when used outside example code.

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

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

1.2 Giving feedback

ARM welcomes feedback on this product and its documentation:

Feedback on this product

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

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

Feedback on the documentation

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

- the title
- the document number, ARM DUI 0492A
- if viewing online, the topic names your comments apply to
- if viewing a PDF version of a document, the page numbers your comments apply to
- a concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

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

1.2.1 See also

Other information

- ARM Information Center, `http://infocenter.arm.com/help/index.jsp`
- ARM Technical Support Knowledge Articles, http://infocenter.arm.com/help/topic/com.arm.doc.faqs/index.html
- Keil Distributors, `http://www.keil.com/distis`.

Chapter 2
The C and C++ libraries

The following topics document 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:

2.1 __aeabi_errno_addr()

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

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

2.1.1 Syntax

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

2.1.2 See also

Reference

- *errno* on page 2-10.

Other information

- *C Library ABI for the ARM Architecture*, http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0039-.

2.2 alloca()

Defined in `alloca.h`, this function allocates local storage in a function. It returns a pointer to *size* bytes of memory, or `NULL` if insufficient memory is available. The default implementation returns an eight-byte aligned block of memory.

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

Note

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

`alloca()` behaves identically to `malloc()` except that `alloca()` has automatic memory de-allocation.

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

2.2.1 Syntax

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

2.3 clock()

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

2.3.1 Syntax

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

2.3.2 Usage

The default implementation of this function uses semihosting.

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

Note

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

2.3.3 Returns

The returned value is an unsigned integer.

2.3.4 See also

Reference

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

- *Direct semihosting C library function dependencies* on page 2-44.

2.4 _clock_init()

Defined in rt_misc.h, this is an initialization function for clock(). It is not part of the C library standard, but is supported by the ARM C library as an extension.

2.4.1 Syntax

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

2.4.2 Usage

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

Note

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

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

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

2.4.3 See also

Reference

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

- *Direct semihosting C library function dependencies* on page 2-44.

2.5 __default_signal_handler()

Defined in rt_misc.h, this 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 is supported by the ARM C library as an extension.

2.5.1 Syntax

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

2.5.2 Usage

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

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

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

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

Note

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

2.5.3 See also

Concepts

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

- *ISO-compliant implementation of signals supported by the signal() function in the C library and additional type arguments* on page 2-132.

Reference

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

- *_ttywrch()* on page 2-65
- *_sys_exit()* on page 2-55.

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

- *Indirect semihosting C library function dependencies* on page 2-46.

2.6 errno

The C library `errno` variable is defined in the implicit static data area of the library. This area is identified by `__user_libspace()`. The function that returns the address of `errno` is:

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

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

Note

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

2.6.1 Returns

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

2.6.2 See also

Concepts

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

- *Use of static data in the C libraries* on page 2-21
- *Use of the __user_libspace static data area by the C libraries* on page 2-24.

Reference

- *__aeabi_errno_addr()* on page 2-5
- *__rt_errno_addr()* on page 2-34.

Other information

- *Application Binary Interface for the ARM Architecture*, http://infocenter/help/index.jsp?topic=/com.arm.doc.ihi0036-

2.7 _findlocale()

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

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

2.7.1 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.

2.7.2 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.

2.7.3 Returns

Returns a pointer to the requested data block.

2.7.4 See also

Concepts

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

- *Assembler macros that tailor locale functions in the C library* on page 2-76
- *Link time selection of the locale subsystem in the C library* on page 2-77
- *Runtime selection of the locale subsystem in the C library* on page 2-80
- *Definition of locale data blocks in the C library* on page 2-81.

Reference

- *lconv structure* on page 2-20.

2.8 _fisatty()

Defined in `stdio.h`, this 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 is supported by the ARM C library as an extension.

2.8.1 Syntax

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

The return value indicates the stream destination:

0 A file.

1 A terminal.

Negative An error.

2.8.2 See also

Tasks

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

- *Tailoring input/output functions in the C and C++ libraries* on page 2-108.

Reference

- *_sys_istty()* on page 2-57.

2.9 _get_lconv()

Defined in `locale.h`, `_get_lconv()` performs the same function as the standard C library function, `localeconv()`, except that it delivers the result in user-provided memory instead of an internal static variable. `_get_lconv()` sets the components of an `lconv` structure with values appropriate for the formatting of numeric quantities.

2.9.1 Syntax

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

2.9.2 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.

2.9.3 Returns

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

2.9.4 See also

Reference

- *localeconv()* on page 2-23.

2.10 getenv()

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

2.10.1 Syntax

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

2.10.2 Usage

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

If you re-implement `getenv()`, it is recommended 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.

2.10.3 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()`.

2.11 _getenv_init()

Defined in rt_misc.h, this function enables a user version of getenv() to initialize itself. It is not part of the C library standard, but is supported by the ARM C library as an extension.

2.11.1 Syntax

```
void _getenv_init(void);
```

2.11.2 Usage

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

2.12 __heapstats()

Defined in stdlib.h, this function displays statistics on the state of the storage allocation heap. The default implementation in the compiler gives information on how many free blocks exist, and estimates their size ranges.

Example 2-1 shows an example of the output from __heapstats().

Example 2-1 Output from __heapstats()

```
32272 bytes in 2 free blocks (avge size 16136)
1 blocks 2^12+1 to 2^13
1 blocks 2^13+1 to 2^14
```

Line 1 of the output displays the total number of bytes, the number of free blocks, and the average size. The following lines give an estimate of the size of each block in bytes, expressed as a range. __heapstats() does not give information on the number of used blocks.

The function outputs its results by calling the output function *dprint()*, that must work like fprintf(). The first parameter passed to *dprint()* is the supplied pointer *param*. You can pass fprintf() itself, provided you cast it to the right function pointer type. This type is defined as a **typedef** for convenience. It is called __heapprt. For example:

```
__heapstats((__heapprt)fprintf, stderr);
```

Note

If you call fprintf() on a stream that you have not already sent output to, the library calls malloc() internally to create a buffer for the stream. If this happens in the middle of a call to __heapstats(), the heap might be corrupted. Therefore, you must ensure you have already sent some output to stderr.

If you are using the default one-region memory model, heap memory is allocated only as it is required. This means that the amount of free heap changes as you allocate and deallocate memory. For example, the sequence:

```
int *ip;
__heapstats((__heapprt)fprintf,stderr);   // print initial free heap size
ip = malloc(200000);
free(ip);
__heapstats((__heapprt)fprintf,stderr);    // print heap size after freeing
```

gives output such as:

```
4076 bytes in 1 free blocks (avge size 4076)
1 blocks 2^10+1 to 2^11
2008180 bytes in 1 free blocks (avge size 2008180)
1 blocks 2^19+1 to 2^20
```

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

2.12.1 Syntax

```
void __heapstats(int (*dprint)(void *param, char const *format,...), void
*param);
```

2.13 __heapvalid()

Defined in stdlib.h, this function performs a consistency check on the heap. It outputs full information about every free block if the *verbose* parameter is nonzero. Otherwise, it only outputs errors.

The function outputs its results by calling the output function *dprint()*, that must work like fprintf(). The first parameter passed to *dprint()* is the supplied pointer *param*. You can pass fprintf() itself, provided you cast it to the right function pointer type. This type is defined as a **typedef** for convenience. It is called __heapprt. For example:

Example 2-2 Calling __heapvalid() with fprintf()

```
__heapvalid((__heapprt) fprintf, stderr, 0);
```

Note

If you call fprintf() on a stream that you have not already sent output to, the library calls malloc() internally to create a buffer for the stream. If this happens in the middle of a call to __heapvalid(), the heap might be corrupted. You must therefore ensure you have already sent some output to stderr. The code in Example 2-2 fails if you have not already written to the stream.

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

2.13.1 Syntax

```
int __heapvalid(int (*dprint)(void *param, char const *format,...), void *param,
int verbose);
```

2.14 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 shown in Example 2-3.

Example 2-3 lconv structure

```
struct lconv {
  char *decimal_point;
       /* The decimal point character used to format non monetary quantities */
  char *thousands_sep;
       /* The character used to separate groups of digits to the left of the */
       /* decimal point character in formatted non monetary quantities.      */
  char *grouping;
       /* A string whose elements indicate the size of each group of digits  */
       /* in formatted non monetary quantities. See below for more details.  */
  char *int_curr_symbol;
       /* The international currency symbol applicable to the current locale.*/
       /* The first three characters contain the alphabetic international    */
       /* currency symbol in accordance with those specified in ISO 4217.    */
       /* Codes for the representation of Currency and Funds. The fourth     */
       /* character (immediately preceding the null character) is the        */
       /* character used to separate the international currency symbol from  */
       /* the monetary quantity.                                             */
  char *currency_symbol;
       /* The local currency symbol applicable to the current locale.        */
  char *mon_decimal_point;
       /* The decimal point used to format monetary quantities.              */
  char *mon_thousands_sep;
       /* The separator for groups of digits to the left of the decimal point*/
       /* in formatted monetary quantities.                                  */
  char *mon_grouping;
       /* A string whose elements indicate the size of each group of digits  */
       /* in formatted monetary quantities. See below for more details.      */
  char *positive_sign;
       /* The string used to indicate a non negative-valued formatted         */
       /* monetary quantity.                                                 */
  char *negative_sign;
       /* The string used to indicate a negative-valued formatted monetary   */
       /* quantity.                                                          */
  char int_frac_digits;
       /* The number of fractional digits (those to the right of the         */
       /* decimal point) to be displayed in an internationally formatted     */
       /* monetary quantities.                                               */
  char frac_digits;
       /* The number of fractional digits (those to the right of the         */
```

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

In this example:

- The elements of `grouping` and `mon_grouping` (shown in Example 2-3 on page 2-20) are interpreted as follows:

  `CHAR_MAX` No additional grouping is to be performed.

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

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

- The value of `p_sign_posn` and `n_sign_posn` (shown in Example 2-3 on page 2-20) are interpreted as follows:

  `0` Parentheses surround the quantity and currency symbol.

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

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

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

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

2.14.1 See also

Reference

- *_get_lconv()* on page 2-14
- *localeconv()* on page 2-23.

2.15 localeconv()

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

2.15.1 Syntax

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

2.15.2 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.

2.15.3 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.

2.15.4 See also

Reference

- *_get_lconv()* on page 2-14
- *lconv structure* on page 2-20
- *setlocale()* on page 2-45.

2.16 _membitcpybl(), _membitcpybb(), _membitcpyhl(), _membitcpyhb(), _membitcpywl(), _membitcpywb(), _membitmovebl(), _membitmovebb(), _membitmovehl(), _membitmovehb(), _membitmovewl(), _membitmovewb()

Similar to the standard C library `memcpy()` and `memmove()` functions, these nonstandard C library functions provide bit-aligned memory operations. They are defined in `string.h`.

2.16.1 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);
```

2.16.2 Usage

The number of contiguous bits specified by *nbits* is copied, or moved (depending on the function being used), from a memory location starting *src_offset* bits after (or before if a negative offset) the address pointed to by *src*, to a location starting *dest_offset* bits after (or before if a negative offset) the address pointed to by *dest*.

To define a contiguous sequence of bits, a form of ordering is required. The variants of each function define this order, as follows:

- Functions whose second-last character is `b`, for example `_membitcpybl()`, are byte-oriented. Byte-oriented functions consider all of the bits in one byte to come before the bits in the next byte.
- Functions whose second-last character is `h` are halfword-oriented.
- Functions whose second-last character is `w` are word-oriented.

Within each byte, halfword, or word, the bits can be considered to go in different order depending on the endianness. Functions ending in `b`, for example `_membitmovewb()`, are bitwise big-endian. This means that the *Most Significant Bit* (MSB) of each byte, halfword, or word (as appropriate) is considered to be the first bit in the word, and the *Least Significant Bit* (LSB) is considered to be the last. Functions ending in `l` are bitwise little-endian. They consider the LSB to come first and the MSB to come last.

As with `memcpy()` and `memmove()`, the bitwise memory copying functions copy as fast as they can in their assumption that source and destination memory regions do not overlap, whereas the bitwise memory move functions ensure that source data in overlapping regions is copied before being overwritten.

On a little-endian platform, the bitwise big-endian functions are distinct, but the bitwise little-endian functions use the same bit ordering, so they are synonymous symbols that refer to the same function. On a big-endian platform, the bitwise big-endian functions are all effectively the same, but the bitwise little-endian functions are distinct.

2.17 posix_memalign()

Defined in `stdlib.h`, this function provides aligned memory allocation. It is fully POSIX-compliant.

2.17.1 Syntax

```
int posix_memalign(void **memptr, size_t alignment, size_t size);
```

2.17.2 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.

2.17.3 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.

2.17.4 See also

Other information

- The Open Group Base Specifications, *IEEE Std 1003.1*, http://www.opengroup.org

2.18 #pragma import(_main_redirection)

This pragma must be defined when redirecting standard input, output and error streams at runtime.

2.18.1 Syntax

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

2.18.2 See also

Reference

Compiler Reference:

- *Environment* on page C-3.

2.19 __raise()

Defined in rt_misc.h, this function raises a signal to indicate a runtime anomaly. It is not part of the C library standard, but is supported by the ARM C library as an extension.

2.19.1 Syntax

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

where:

signal is an integer that holds the signal number.

type is an integer, string constant or variable that provides additional information about the circumstances that the signal was raised in, for some kinds of signal.

2.19.2 Usage

This function calls the normal C signal mechanism or the default signal handler.

You can replace the __raise() function by defining:

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

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

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

The default signal handler of the library uses the *type* parameter of __raise() to vary the messages it outputs.

2.19.3 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*.

2.19.4 See also

Concepts

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

- *Thread safety in the ARM C library* on page 2-34.

Reference

- *__default_signal_handler()* on page 2-9
- *_sys_exit()* on page 2-55
- *_ttywrch()* on page 2-65.

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

- *Indirect semihosting C library function dependencies* on page 2-46
- *ISO-compliant implementation of signals supported by the signal() function in the C library and additional type arguments* on page 2-132.

2.20 _rand_r()

Defined in stdlib.h, this is a reentrant version of the rand() function.

2.20.1 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.

2.20.2 Usage

This function enables you to explicitly supply your own buffer in thread-local storage.

2.20.3 See also

Reference

- *C library functions that are not thread-safe* on page 2-82
- *_srand_r()* on page 2-47.

2.21 remove()

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

2.21.1 Syntax

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

2.21.2 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.

2.21.3 Returns

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

2.21.4 See also

Reference

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

- *Direct semihosting C library function dependencies* on page 2-44.

2.22 rename()

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

2.22.1 Syntax

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

2.22.2 Usage

The default implementation of this function uses semihosting.

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

2.22.3 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.

2.22.4 See also

Reference

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

- *Direct semihosting C library function dependencies* on page 2-44.

2.23 __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-load regions have been relocated to their execution addresses.

2.23.1 Usage

The default implementation of __rt_entry:

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

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

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

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

_sys_exit() Exits directly to the execution environment. It does not shut down the library and does not call atexit() functions.

2.24 __rt_errno_addr()

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

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

Note

This function is associated with pre-ABI versions of the compilation tools. However, it remains supported to ensure that object files compiled with those tools link successfully. Unless you are working with object files compiled with pre-ABI versions of the tools, use `__aeabi_errno_addr()` instead of `__rt_errno_addr()`.

2.24.1 Syntax

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

2.24.2 See also

Reference

- *__aeabi_errno_addr()* on page 2-5
- *errno* on page 2-10.

Other information

- *Application Binary Interface for the ARM Architecture*, http://infocenter/help/index.jsp?topic=/com.arm.doc.ihi0036-

2.25 __rt_exit()

Defined in rt_misc.h, this function shuts down the library but does not call functions registered with atexit(). atexit()-registered functions are called by exit().

__rt_exit() is not part of the C library standard, but is supported by the ARM C library as an extension.

2.25.1 Syntax

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

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

2.25.2 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().

2.25.3 Returns

This function does not return.

2.26 __rt_fp_status_addr()

Defined in rt_fp.h, this function returns the address of the floating-point status word, which resides by default in __user_libspace. It is not part of the C library standard, but is supported by the ARM C library as an extension.

2.26.1 Syntax

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

2.26.2 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.

2.26.3 Returns

The address of the floating-point status word.

2.26.4 See also

Concepts

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

- *Thread safety in the ARM C library* on page 2-34.

2.27 __rt_heap_extend()

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

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

2.27.1 Syntax

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

2.27.2 Usage

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

The default implementation has the following characteristics:

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

2.27.3 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.

2.27.4 See also

Reference

- *__rt_stackheap_init()* on page 2-42
- *__user_heap_extend()* on page 2-66.

Other information

- *Procedure Call Standard for the ARM Architecture*, http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0042-

2.28 __rt_lib_init()

Defined in rt_misc.h, this is the library initialization function and is the companion to __rt_lib_shutdown().

2.28.1 Syntax

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

where:

heapbase is the start of the heap memory block.

heaptop is the end of the heap memory block.

2.28.2 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.

2.28.3 Returns

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

```
struct __argc_argv
{   int argc;
    char **argv;
    int r2, r3;   // optional extra arguments that on entry to main() are
};                // found in registers R2 and R3.
```

2.29 __rt_lib_shutdown()

Defined in rt_misc.h, this is the library shutdown function and is the companion to __rt_lib_init().

2.29.1 Syntax

```
void __rt_lib_shutdown(void);
```

2.29.2 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.

2.30 __rt_raise()

Defined in rt_misc.h, this function raises a signal to indicate a runtime anomaly. It is not part of the C library standard, but is supported by the ARM C library as an extension.

2.30.1 Syntax

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

where:

signal is an integer that holds the signal number.

type is an integer, string constant or variable that provides additional information about the circumstances that the signal was raised in, for some kinds of signal.

2.30.2 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*.

2.30.3 See also

Reference

- *__raise()* on page 2-28
- *_sys_exit()* on page 2-55.

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

- *ISO-compliant implementation of signals supported by the signal() function in the C library and additional type arguments* on page 2-132.

2.31 __rt_stackheap_init()

Defined in rt_misc.h, this function sets up the stack pointer and returns a region of memory for use as the initial heap. It is called from the library initialization code.

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

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

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

2.31.1 See also

Reference

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

2.32 Selecting the one-region memory model automatically

Define one special execution region, `ARM_LIB_STACKHEAP`, in your scatter-loading description file. You must mark this region with the `EMPTY` attribute.

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

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

2.32.1 See also

Tasks

- *__user_initial_stackheap() and scatter-loading description files* on page 2-72.

Concepts

Using the Linker:

- *About scatter-loading* on page 8-3.

2.33 Selecting the two-region memory model automatically

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

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

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

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

The following segment is part of a scatter-loading description file that defines both ARM_LIB_HEAP and ARM_LIB_STACK, as an example:

```
;; Heap starts at 1MB and grows upwards
ARM_LIB_HEAP 0x20100000 EMPTY 0x100000-0x8000
{
}

;; Stack starts at the end of the 2MB of RAM and grows downwards for 32KB
ARM_LIB_STACK 0x20200000 EMPTY -0x8000
{
}
```

2.33.1 See also

Tasks

- *__user_initial_stackheap() and scatter-loading description files* on page 2-72.

Concepts

Using the Linker:

- *About scatter-loading* on page 8-3.

Reference

- *__user_initial_stackheap()* on page 2-68.

2.34 setlocale()

Defined in locale.h, this function selects the appropriate locale as specified by the *category* and *locale* arguments.

2.34.1 Syntax

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

2.34.2 Usage

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

Valid *locale* values depend on which __use_*X*_ctype symbols are imported (__use_iso8859_ctype, __use_sjis_ctype, __use_utf8_ctypte), and on user-defined locales.

Locale categories

The values of *category* are:

LC_COLLATE — Affects the behavior of strcoll().

LC_CTYPE — Affects the behavior of the character handling functions.

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

LC_NUMERIC — Affects the decimal-point character for the formatted input/output functions and the string conversion functions and the numeric formatting information returned by localeconv().

LC_TIME — Can affect the behavior of strftime(). For currently supported locales, the option has no effect.

LC_ALL — Affects all locale categories. This is the bitwise OR of all the locale categories.

2.34.3 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().

2.34.4 See also

Concept

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

- *ISO8859-1 implementation* on page 2-78
- *Shift-JIS and UTF-8 implementation* on page 2-79
- *Definition of locale data blocks in the C library* on page 2-81.

Reference

- *lconv structure* on page 2-20.

2.35 _srand_r()

Defined in stdlib.h, this is a reentrant version of the srand() function.

2.35.1 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().

2.35.2 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.

2.35.3 See also

Reference

- *C library functions that are not thread-safe* on page 2-82
- *_rand_r()* on page 2-30.

2.36 strcasecmp()

Defined in string.h, this function performs a case-insensitive string comparison test.

2.36.1 Syntax

```
extern _ARMABI int strcasecmp(const char *s1, const char *s2);
```

2.36.2 See also

Other information

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

2.37 strncasecmp()

Defined in string.h, this function performs a case-insensitive string comparison test of not more than a specified number of characters.

2.37.1 Syntax

```
extern _ARMABI int strncasecmp(const char *s1, const char *s2, size_t n);
```

2.37.2 See also

Other information

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

2.38 strlcat()

Defined in string.h, this function concatenates two strings. It appends up to *size*-strlen(*dst*)-1 bytes from the NUL-terminated string *src* to the end of *dst*. It takes the full size of the buffer, not only the length, and terminates the result with NUL as long as *size* is greater than 0. Include a byte for the NUL in your *size* value.

The strlcat() function returns the total length of the string that *would* have been created if there was unlimited space. This might or might not be equal to the length of the string *actually* created, depending on whether there was enough space. This means that you can call strlcat() once to find out how much space is required, then allocate it if you do not have enough, and finally call strlcat() a second time to create the required string.

This function is a common BSD-derived extension to many C libraries.

2.38.1 Syntax

```
extern size_t strlcat(char *dst, *src, size_t size);
```

2.39 strlcpy()

Defined in string.h, this function copies up to *size*-1 characters from the NUL-terminated string *src* to *dst*. It takes the full size of the buffer, not only the length, and terminates the result with NUL as long as *size* is greater than 0. Include a byte for the NUL in your *size* value.

The strlcpy() function returns the total length of the string that *would* have been copied if there was unlimited space. This might or might not be equal to the length of the string *actually* copied, depending on whether there was enough space. This means that you can call strlcpy() once to find out how much space is required, then allocate it if you do not have enough, and finally call strlcpy() a second time to do the required copy.

This function is a common BSD-derived extension to many C libraries.

2.39.1 Syntax

```
extern size_t strlcpy(char *dst, const char *src, size_t size);
```

2.40 _sys_close()

Defined in `rt_sys.h`, this function closes a file previously opened with `_sys_open()`.

2.40.1 Syntax

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

2.40.2 Usage

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

2.40.3 Returns

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

2.40.4 See also

Reference

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

- *Direct semihosting C library function dependencies* on page 2-44.

2.41 _sys_command_string()

Defined in rt_sys.h, this function retrieves the command line used to invoke the current application from the environment that called the application.

2.41.1 Syntax

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

where:

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

len is the length of the buffer.

2.41.2 Usage

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

Note

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

2.41.3 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.

2.41.4 See also

Reference

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

- *Direct semihosting C library function dependencies* on page 2-44.

2.42 _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`).

2.43 _sys_exit()

Defined in rt_sys.h, this is the library exit function. All exits from the library eventually call _sys_exit().

2.43.1 Syntax

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

2.43.2 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.

2.43.3 Returns

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

2.43.4 See also

Reference

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

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

- *Direct semihosting C library function dependencies* on page 2-44.

2.44 _sys_flen()

Defined in rt_sys.h, this function returns the current length of a file.

2.44.1 Syntax

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

2.44.2 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.

2.44.3 Returns

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

2.44.4 See also

Reference

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

- *Direct semihosting C library function dependencies* on page 2-44.

2.45 _sys_istty()

Defined in rt_sys.h, this function determines if a file handle identifies a terminal.

2.45.1 Syntax

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

2.45.2 Usage

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

2.45.3 Returns

The return value is one of the following values:

0 There is no interactive device.

1 There is an interactive device.

other An error occurred.

2.45.4 See also

Reference

- *_fisatty()* on page 2-13.

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

- *Direct semihosting C library function dependencies* on page 2-44.

2.46 _sys_open()

Defined in rt_sys.h, this function opens a file.

2.46.1 Syntax

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

2.46.2 Usage

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

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

2.46.3 Returns

The return value is -1 if an error occurs.

2.46.4 See also

Other information

- *install_directory*\include\...\rt_sys.h

2.46.5 See also

Reference

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

- *Direct semihosting C library function dependencies* on page 2-44.

2.47 _sys_read()

Defined in rt_sys.h, this function reads the contents of a file into a buffer.

2.47.1 Syntax

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

Note

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

2.47.2 Returns

The return value is one of the following:

- The number of bytes *not* read (that is, *len* - *result* number of bytes were read).
- An error indication.
- An EOF indicator. The EOF indication involves the setting of 0x80000000 in the normal result.

Reading up to and including the last byte of data does not turn on the EOF indicator. The EOF indicator is only reached when an attempt is made to read beyond the last byte of data. The target-independent code is capable of handling:

- the EOF indicator being returned in the same read as the remaining bytes of data that precede the EOF
- the EOF indicator being returned on its own after the remaining bytes of data have been returned in a previous read.

2.47.3 See also

Reference

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

- *Direct semihosting C library function dependencies* on page 2-44.

2.48 _sys_seek()

Defined in rt_sys.h, this function puts the file pointer at offset *pos* from the beginning of the file.

2.48.1 Syntax

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

2.48.2 Usage

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

2.48.3 Returns

The result is:

- non-negative if no error occurs
- negative if an error occurs.

2.48.4 See also

Reference

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

- *Direct semihosting C library function dependencies* on page 2-44.

2.49 _sys_tmpnam()

Defined in rt_sys.h, this function converts the file number *fileno* for a temporary file to a unique filename, for example, tmp0001.

2.49.1 Syntax

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

2.49.2 Usage

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

2.49.3 Returns

Returns the filename in *name*.

2.49.4 See also

Reference

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

- *Direct semihosting C library function dependencies* on page 2-44.

2.50 _sys_write()

Defined in rt_sys.h, this function writes the contents of a buffer to a file previously opened with _sys_open().

2.50.1 Syntax

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

Note

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

2.50.2 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.

2.50.3 See also

Reference

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

- *Direct semihosting C library function dependencies* on page 2-44.

2.51 system()

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

2.51.1 Syntax

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

2.51.2 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.

2.51.3 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.

2.51.4 See also

Reference

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

- *Direct semihosting C library function dependencies* on page 2-44.

2.52 time()

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

The default implementation of this function uses semihosting.

2.52.1 Syntax

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

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

2.52.2 Returns

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

2.52.3 See also

Reference

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

- *Direct semihosting C library function dependencies* on page 2-44.

2.53 _ttywrch()

Defined in rt_sys.h, this function writes a character to the console. The console might have been redirected. You can use this function as a last resort error handling routine.

2.53.1 Syntax

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

2.53.2 Usage

The default implementation of this function uses semihosting.

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

2.53.3 See also

Concepts

Developing Software for ARM® Processors:

- Chapter 8 *Semihosting*.

Reference

- *__raise()* on page 2-28

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

- *Direct semihosting C library function dependencies* on page 2-44.

2.54 __user_heap_extend()

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

2.54.1 Syntax

```
extern unsigned __user_heap_extend(int var0, void **base, unsigned
requested_size);
```

2.54.2 Usage

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

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

2.54.3 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.

2.54.4 See also

Other information

- *Procedure Call Standard for the ARM Architecture*, http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0042-

2.55 __user_heap_extent()

If defined, this function returns the base address and maximum range of the heap. See `rt_misc.h`.

2.55.1 Syntax

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

2.55.2 Usage

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

2.55.3 See also

Concepts

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

- *C library support for memory allocation functions* on page 2-98.

2.56 __user_initial_stackheap()

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

2.56.1 Syntax

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

2.56.2 Usage

__user_initial_stackheap() returns the:

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

If this function is reimplemented, it must:

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

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

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

To create a version of __user_initial_stackheap() that inherits sp from the execution environment and does not have a heap, set r0 and r2 to the value of r1 and return.

The definition of __initial_stackheap in rt_misc.h is:

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

Note

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

The examples directory provided with the ARM compilation tools contains example reimplementations of this function.

2.56.3 Returns

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

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

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

2.56.4 See also

Concepts

- *__user_initial_stackheap() and migration to ARM Compiler 4.1 from RVCT v3.x* on page 2-71
- *__user_initial_stackheap() and migration to ARM Compiler 4.1 from RVCT v2.x and earlier* on page 2-70.

Reference

- *__user_setup_stackheap()* on page 2-73.

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

- *Direct semihosting C library function dependencies* on page 2-44.

2.57 `__user_initial_stackheap()` and migration to ARM Compiler 4.1 from RVCT v2.x and earlier

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

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

2.57.1 See also

Tasks

- *__user_initial_stackheap() and scatter-loading description files* on page 2-72.

Reference

- *__user_initial_stackheap()* on page 2-68
- *__user_setup_stackheap()* on page 2-73.

2.58 `__user_initial_stackheap()` and migration to ARM Compiler 4.1 from RVCT v3.x

In RVCT v3.x and later versions of RVCT, and in ARM Compiler 4.1, the library includes more implementations of `__user_initial_stackheap()` (see `rt_misc.h`), and can select the correct implementation for you automatically from information given in a scatter-loading description file. This means that it is not necessary to reimplement this function if you are using scatter-loading files.

2.58.1 See also

Tasks

- *__user_initial_stackheap() and scatter-loading description files* on page 2-72.

Reference

- *__user_initial_stackheap()* on page 2-68.

2.59 `__user_initial_stackheap()` and scatter-loading description files

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

Tasks

- *Selecting the one-region memory model automatically* on page 2-43
- *Selecting the two-region memory model automatically* on page 2-44.

2.60 __user_setup_stackheap()

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

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 or returning.

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

__user_setup_stackheap() returns the:

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

Note

__user_setup_stackheap() must be reimplemented in assembler.

2.60.1 See also

Reference

- *__user_initial_stackheap()* on page 2-68.

2.61 __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-loading description file, an error is generated to the following effect:

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

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

The following segment is part of a scatter-loading description file. It includes a minimal vector table illustrating the use of __vectab_stack_and_reset to place the initial sp and pc values at addresses 0x0 and 0x4 in the vector table:

```
;; Maximum of 256 exceptions (256*4 bytes == 0x400)
VECTORS 0x0 0x400
{
    ; First two entries provided by library
    ; Remaining entries provided by the user in exceptions.c

    * (:gdef:__vectab_stack_and_reset, +FIRST)
    * (exceptions_area)
}

CODE 0x400 FIXED
{
    * (+RO)
}
```

2.61.1 See also

Concepts

Using the Linker:

- *About scatter-loading* on page 8-3.

2.62 wcscasecmp()

Defined in wchar.h, this function performs a case-insensitive string comparison test on wide characters. It is a GNU extension to the libraries. It is not POSIX-standardized.

2.62.1 Syntax

```
int wcscasecmp(const wchar_t * __restrict s1, const wchar_t * __restrict s2);
```

2.63 wcsncasecmp()

Defined in wchar.h, this function performs a case-insensitive string comparison test of not more than a specified number of wide characters. It is a GNU extension to the libraries. It is not POSIX-standardized.

2.63.1 Syntax

```
int wcsncasecmp(const wchar_t * __restrict s1, const wchar_t * __restrict s2,
size_t n);
```

2.64 wcstombs()

Defined in wchar.h, this function works as described in the ISO C standard, with extended functionality as specified by POSIX, that is, if *s* is a NULL pointer, wcstombs() returns the length required to convert the entire array regardless of the value of *n*, but no values are stored.

2.64.1 Syntax

```
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

2.65 Thread-safe C library functions

The following table shows the C library functions that are thread-safe.

Table 2-1 Functions that are thread-safe

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

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

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

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

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

2.65.1 See also

Concepts

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

- *Thread safety in the ARM C library* on page 2-34.

2.66 C library functions that are not thread-safe

The following table shows the C library functions that are not thread-safe.

Table 2-2 Functions that are not thread-safe

Functions	Description
`asctime()`, `localtime()`, `strtok()`	These functions are all thread-unsafe. Each contains a static buffer that might be overwritten by another thread between a call to the function and the subsequent use of its return value. ARM supplies reentrant versions, `_asctime_r()`, `_localtime_r()`, and `_strtok_r()`. ARM recommends that you use these functions instead to ensure safety. **Note** These reentrant versions take additional parameters. `_asctime_r()` takes an additional parameter that is a pointer to a buffer that the output string is written into. `_localtime_r()` takes an additional parameter that is a pointer to a `struct tm`, that the result is written into. `_strtok_r()` takes an additional parameter that is a pointer to a **`char`** pointer to the next token.
`exit()`	Do not call `exit()` in a multithreaded program even if you have provided an implementation of the underlying `_sys_exit()` that actually terminates all threads. In this case, `exit()` cleans up *before* calling `_sys_exit()` so disrupts other threads.
`gamma()`[a], `lgamma()`	These extended mathlib functions use a global variable, `_signgam`, so are not thread-safe.

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

Functions	Description
`mbrlen()`, `mbsrtowcs()`, `mbrtowc()`, `wcrtomb()`, `wcsrtombs()`	The C89 multibyte conversion functions (defined in `stdlib.h`) are not thread-safe, for example `mblen()` and `mbtowc()`, because they contain internal static state that is shared between all threads without locking. However, the extended restartable versions (defined in `wchar.h`) are thread-safe, for example `mbrtowc()` and `wcrtomb()`, provided you pass in a pointer to your own `mbstate_t` object. You must exclusively use these functions with non-NULL `mbstate_t *` parameters if you want to ensure thread-safety when handling multibyte strings.
`rand()`, `srand()`	These functions keep internal state that is both global and unprotected. This means that calls to `rand()` are never thread-safe. ARM recommends that you do one of the following: • Use ARM-supplied reentrant versions `_rand_r()` and `_srand_r()`. These use user-provided buffers instead of static data within the C library. • Use your own locking to ensure that only one thread ever calls `rand()` at a time, for example, by defining `$Sub$$rand()` if you want to avoid changing your code. • Arrange that only one thread ever needs to generate random numbers. • Supply your own random number generator that can have multiple independent instances. **Note** `_rand_r()` and `_srand_r()` both take an additional parameter that is a pointer to a buffer storing the state of the random number generator.

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

Functions	Description
`setlocale()`, `localeconv()`	`setlocale()` is used for setting and reading locale settings. The locale settings are global across all threads, and are not protected by a lock. If two threads call `setlocale()` to simultaneously modify the locale settings, or if one thread reads the settings while another thread is modifying them, data corruption might occur. Also, many other functions, for example `strtod()` and `sprintf()`, read the current locale settings. Therefore, if one thread calls `setlocale()` concurrently with another thread calling such a function, there might be unexpected results. Multiple threads *reading* the settings simultaneously is thread-safe in simple cases and if no other thread is simultaneously modifying those settings, but where internally an intermediate buffer is required for more complicated returned results, unexpected results can occur unless you use a reentrant version of `setlocale()`. ARM recommends that you either: • Choose the locale you want and call `setlocale()` once to initialize it. Do this before creating any additional threads in your program so that any number of threads can read the locale settings concurrently without interfering with one another. • Use the ARM-supplied reentrant version, `_setlocale_r()`. This returns a string that is either a pointer to a constant string, or a pointer to a string stored in a user-supplied buffer that can be used for thread-local storage, rather than using memory within the C library. Be aware that `_setlocale_r()` is not fully thread-safe when accessed concurrently to *change* locale settings. This access is not lock-protected. Also, be aware that `localeconv()` is not thread-safe. Call the ARM function `_get_lconv()` with a pointer to a user-supplied buffer instead.

a. If migrating from RVCT, be aware that `gamma()` is deprecated in ARM Compiler 4.1.

2.66.1 See also

Concepts

- *Thread safety in the ARM C library* on page 2-34.

Reference

- *_rand_r()* on page 2-30
- *_srand_r()* on page 2-47.

Chapter 3
Floating-point support

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

3.1 _clearfp()

Defined in `float.h`, this function is provided for compatibility with Microsoft products.

`_clearfp()` clears all five exception sticky flags and returns their previous values. The `_controlfp()` argument macros, for example `_EM_INVALID` and `_EM_ZERODIVIDE`, can be used to test bits of the returned result.

The function prototype for `_clearfp()` is:

```
unsigned _clearfp(void);
```

Note

This function requires you to select a floating-point model that supports exceptions. For example, `--fpmode=ieee_full` or `--fpmode=ieee_fixed`.

3.1.1 See also

Tasks

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

- *Floating-point functions for compatibility with Microsoft products* on page 4-17.

Reference

- *_controlfp()* on page 3-3
- *_statusfp()* on page 3-15.

3.2 _controlfp()

Defined in float.h, this function is provided for compatibility with Microsoft products. It enables you to control exception traps and rounding modes.

The function prototype for _controlfp() is:

```
unsigned int _controlfp(unsigned int new, unsigned int mask);
```

Note

This function requires you to select a floating-point model that supports exceptions. For example, --fpmode=ieee_full or --fpmode=ieee_fixed.

_controlfp() also modifies a control word using a mask to isolate the bits to modify. For every bit of mask that is zero, the corresponding control word bit is unchanged. For every bit of mask that is nonzero, the corresponding control word bit is set to the value of the corresponding bit of new. The return value is the previous state of the control word.

Note

This is different behavior to that of __ieee_status() or __fp_status(), where you can toggle a bit by setting a zero in the mask word and a one in the flags word.

Table 3-1 describes the macros you can use to form the arguments to _controlfp().

Table 3-1 _controlfp argument macros

Macro	Description
_MCW_EM	Mask containing all exception bits
_EM_INVALID	Bit describing the Invalid Operation exception
_EM_ZERODIVIDE	Bit describing the Divide by Zero exception
_EM_OVERFLOW	Bit describing the Overflow exception
_EM_UNDERFLOW	Bit describing the Underflow exception
_EM_INEXACT	Bit describing the Inexact Result exception
_MCW_RC	Mask for the rounding mode field
_RC_CHOP	Rounding mode value describing Round Toward Zero

Table 3-1 _controlfp argument macros (continued)

Macro	Description
_RC_UP	Rounding mode value describing Round Up
_RC_DOWN	Rounding mode value describing Round Down
_RC_NEAR	Rounding mode value describing Round To Nearest

Note

The values of these macros are not guaranteed to remain the same in future versions of ARM products. To ensure that your code continues to work if the value changes in future releases, use the macro rather than its value.

For example, to set the rounding mode to round down, call:

```
_controlfp(_RC_DOWN, _MCW_RC);
```

To trap the Invalid Operation exception and untrap all other exceptions:

```
_controlfp(_EM_INVALID, _MCW_EM);
```

To untrap the Inexact Result exception:

```
_controlfp(0, _EM_INEXACT);
```

3.2.1 See also

Tasks

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

- *Floating-point functions for compatibility with Microsoft products* on page 4-17.

Reference

- *__ieee_status()* on page 3-9
- *__fp_status()* on page 3-5
- *_clearfp()* on page 3-2
- *_statusfp()* on page 3-15.

3.3 __fp_status()

Some older versions of the ARM libraries implemented a function called `__fp_status()` that manipulated a status word in the floating-point environment. This is the same as `__ieee_status()` but it uses an older style of status word layout. The compiler still supports the `__fp_status()` function for backwards compatibility. `__fp_status()` is defined in `stdlib.h`.

The function prototype for `__fp_status()` is:

```
unsigned int __fp_status(unsigned int mask, unsigned int flags);
```

Note

This function requires you to select a floating-point model that supports exceptions. For example, `--fpmode=ieee_full` or `--fpmode=ieee_fixed`.

The layout of the status word as seen by `__fp_status()` is shown in Figure 3-1.

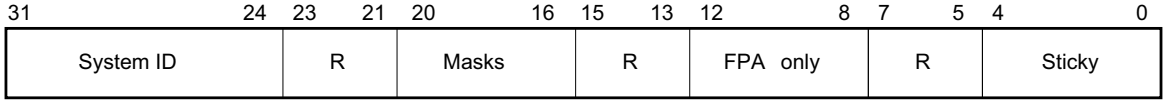

Figure 3-1 Floating-point status word layout

The fields in Figure 3-1 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 is not supported by the ARM compilation tools. Any attempt to write to these bits is ignored.

- Bits 16 to 20 (values `0x10000` to `0x100000`) are the exception masks. These control whether each exception is trapped or not. If a bit is set to 1, the corresponding exception is trapped. If a bit is set to 0, the corresponding exception sets its sticky flag and returns a plausible result.
- Bits 24 to 31 contain the system ID that cannot be changed. It is set to `0x40` for software floating-point, to `0x80` or above for hardware floating-point, and to `0` or `1` if a hardware floating-point environment is being faked by an emulator.
- Bits marked `R` are reserved. They cannot be written to by the `__fp_status()` call, and you must ignore anything you find in them.

The rounding mode cannot be changed with the `__fp_status()` call.

In addition to defining the `__fp_status()` call itself, `stdlib.h` also defines the following constants to be used for the arguments:

```
#define __fpsr_IXE  0x100000
#define __fpsr_UFE  0x80000
#define __fpsr_OFE  0x40000
#define __fpsr_DZE  0x20000
#define __fpsr_IOE  0x10000
#define __fpsr_IXC  0x10
#define __fpsr_UFC  0x8
#define __fpsr_OFC  0x4
#define __fpsr_DZC  0x2
#define __fpsr_IOC  0x1
```

For example, to trap the Invalid Operation exception and untrap all other exceptions, you would call `__fp_status()` with the following input parameters:

```
__fp_status(_fpsr_IXE | _fpsr_UFE | _fpsr_OFE |
            _fpsr_DZE | _fpsr_IOE, _fpsr_IOE);
```

To untrap the Inexact Result exception:

```
__fp_status(_fpsr_IXE, 0);
```

To clear the Underflow sticky flag:

```
__fp_status(_fpsr_UFC, 0);
```

3.3.1 See also

Tasks

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

- *Controlling the ARM floating-point environment* on page 4-16.

Reference

- *__ieee_status()* on page 3-9.

3.4 gamma(), gamma_r()

These functions both compute the logarithm of the gamma function.They are synonyms for `lgamma` and `lgamma_r`.

```
double gamma(double x);
double gamma_r(double x, int *);
```

Note

Despite their names, these functions compute the logarithm of the gamma function, not the gamma function itself.

Note

If you are migrating from RVCT, be aware that these functions are deprecated in ARM Compiler 4.1.

3.4.1 See also

Reference

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

- *Nonstandard functions in mathlib* on page 4-38.

3.5 __ieee_status()

The ARM compiler toolchain supports an interface to the status word in the floating-point environment. This interface is provided as function `__ieee_status()` and it is generally the most efficient function to use for modifying the status word for VFP. `__ieee_status()` is defined in `fenv.h`.

The function prototype for `__ieee_status()` is:

```
unsigned int __ieee_status(unsigned int mask, unsigned int flags);
```

Note

This function requires you to select a floating-point model that supports exceptions. For example, `--fpmode=ieee_full` or `--fpmode=ieee_fixed`.

`__ieee_status()` modifies the writable parts of the status word according to the parameters, and returns the previous value of the whole word.

The writable bits are modified by setting them to:

```
new = (old & ~mask) ^ flags;
```

Four different operations can be performed on each bit of the status word, depending on the corresponding bits in mask and flags. See Table 3-2.

Table 3-2 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 shown in Figure 3-2.

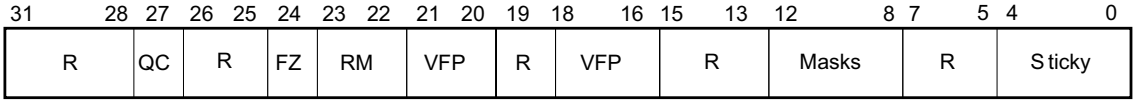

Figure 3-2 IEEE status word layout

The fields in Figure 3-2 on page 3-9 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 Table 3-3.

Table 3-3 Rounding mode control

Bits	Rounding mode
`00`	Round to nearest
`01`	Round up
`10`	Round down
`11`	Round toward zero

Note

The `fz*`, `fj*` and `f*` library variants support only the round-to-nearest rounding mode. If you require support for the other rounding modes, you must use the full IEEE `g*` libraries. (The relevant compiler options are `--fpmode=std`, `--fpmode=ieee_no_fenv` and `--fpmode=ieee_fixed`.)

- Bit 24 enables FZ (Flush to Zero) mode if it is set. In FZ mode, denormals are forced to zero to speed up processing because denormals can be difficult to work with and slow down floating-point systems. Setting this bit reduces accuracy but might increase speed.

  Note

  The FZ bit in the IEEE status word is not supported by any of the `fplib` variants. This means that switching between flushing to zero and not flushing to zero is not possible with any variant of `fplib` at *runtime*. However, flushing to zero or not flushing to zero can be set at compile time as a result of the library you choose to build with.

  Some functions are not provided in hardware. They exist only in the software floating-point libraries. So these functions cannot support the FZ mode, even when you are compiling for a hardware VFP architecture. As a result, behavior of the floating-point libraries is not consistent across all functions when you change the FZ mode dynamically.

- Bit 27 indicates that saturation has occurred in an advanced SIMD saturating integer operation. This is accessible through the `__ieee_status()` call.
- Bits marked R are reserved. They cannot be written to by the `__ieee_status()` call, and you must ignore anything you find in them.

In addition to defining the `__ieee_status()` call itself, `fenv.h` also defines the following constants to be used for the arguments:

```
#define FE_IEEE_FLUSHZERO          (0x01000000)
#define FE_IEEE_ROUND_TONEAREST    (0x00000000)
#define FE_IEEE_ROUND_UPWARD       (0x00400000)
#define FE_IEEE_ROUND_DOWNWARD     (0x00800000)
#define FE_IEEE_ROUND_TOWARDZERO   (0x00C00000)
#define FE_IEEE_ROUND_MASK         (0x00C00000)
#define FE_IEEE_MASK_INVALID       (0x00000100)
#define FE_IEEE_MASK_DIVBYZERO     (0x00000200)
#define FE_IEEE_MASK_OVERFLOW      (0x00000400)
#define FE_IEEE_MASK_UNDERFLOW     (0x00000800)
#define FE_IEEE_MASK_INEXACT       (0x00001000)
#define FE_IEEE_MASK_ALL_EXCEPT    (0x00001F00)
#define FE_IEEE_INVALID            (0x00000001)
#define FE_IEEE_DIVBYZERO          (0x00000002)
#define FE_IEEE_OVERFLOW           (0x00000004)
#define FE_IEEE_UNDERFLOW          (0x00000008)
#define FE_IEEE_INEXACT            (0x00000010)
#define FE_IEEE_ALL_EXCEPT         (0x0000001F)
```

For example, to set the rounding mode to round down, you would call:

```
__ieee_status(FE_IEEE_ROUND_MASK, FE_IEEE_ROUND_DOWNWARD);
```

To trap the Invalid Operation exception and untrap all other exceptions:

```
__ieee_status(FE_IEEE_MASK_ALL_EXCEPT, FE_IEEE_MASK_INVALID);
```

To untrap the Inexact Result exception:

```
__ieee_status(FE_IEEE_MASK_INEXACT, 0);
```

To clear the Underflow sticky flag:

```
__ieee_status(FE_IEEE_UNDERFLOW, 0);
```

3.5.1 See also

Tasks

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

- *Exceptions arising from IEEE 754 floating-point arithmetic* on page 4-49
- *Controlling the ARM floating-point environment* on page 4-16.

Concepts

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

- *C and C++ library naming conventions* on page 2-144.

Reference

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

3.6 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. `j`*n* computes the function of order *n*.

```
double j0(double x);
double j1(double x);
double jn(int n, double x);
```

If the absolute value of *x* exceeds π times 2^{52}, these functions return an `ERANGE` error, denoting total loss of significance in the result.

Note

If you are migrating from RVCT, be aware that these functions are deprecated in ARM Compiler 4.1.

3.6.1 See also

Reference

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

- *Nonstandard functions in mathlib* on page 4-38.

3.7 significand(), fractional part of a number

This function returns the fraction part of *x*, as a number between 1.0 and 2.0 (not including 2.0).

```
double significand(double x);
```

Note

If you are migrating from RVCT, be aware that this function is deprecated in ARM Compiler 4.1.

3.7.1 See also

Reference

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

- *Nonstandard functions in mathlib* on page 4-38.

3.8 _statusfp()

Defined in `float.h`, this function is provided for compatibility with Microsoft products. It returns the current value of the exception sticky flags. The `_controlfp()` argument macros, for example `_EM_INVALID` and `_EM_ZERODIVIDE`, can be used to test bits of the returned result.

The function prototype for `_statusfp() is`:

```
unsigned _statusfp(void);
```

Note

This function requires you to select a floating-point model that supports exceptions. For example, `--fpmode=ieee_full` or `--fpmode=ieee_fixed`.

3.8.1 See also

Tasks

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

- *Floating-point functions for compatibility with Microsoft products* on page 4-17.

Reference

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

3.9 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.

Note

If you are migrating from RVCT, be aware that these functions are deprecated in ARM Compiler 4.1.

3.9.1 See also

Reference

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

- *Nonstandard functions in mathlib* on page 4-38.