Application Note 241

ARM Compiler C Library Startup and Initialization

Document number: ARM DAI 0241B

Non-Confidential

Application Note 241
ARM Compiler C Library Startup and Initialization

Release information

The following table lists the changes made to this document.

Change history

Date	Issue	Change
2 December 2010	A	ARM Compiler toolchain v4.1 build 561 Release
24 January 2014	B	Generic document for all ARM Compiler releases

Proprietary notice

Confidentiality status

Feedback on this application note

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

- the document title
- the document number
- the page numbers to which your comments apply
- a concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

ARM web address

http://www.arm.com

Table of Contents

1 Introduction

This document describes the C library startup code and the initialization functions that might be called during the startup of an application that has been compiled using the ARM Compiler. The document gives an overview of what the functions in the startup code do, and why they are present. You can use this document to verify the startup code of your application.

1.1 Version

This document describes the startup code for ARM Compiler. Functions in the startup code might change between different releases and patches of the toolchain. This document makes no guarantee about the continued operation of the library startup code in later releases or patches of the toolchain.

1.2 Additional reading

This section lists publications by ARM and by third parties.

See Infocenter, `http://infocenter.arm.com`, for access to ARM documentation.

1.2.1 ARM publications

The following documents contain information relevant to this document:

- *ARM Compiler toolchain Developing Software for ARM Processors* (ARM DUI 0471)
- *ARM Compiler toolchain ARM C and C++ Libraries and Floating-Point Support Reference* (ARM DUI 0492)
- *ARM Compiler toolchain Using ARM C and C++ Libraries and Floating-Point Support* (ARM DUI 0475)
- *ARM Compiler toolchain Linker Reference* (ARM DUI 0493).

2 Startup code

Embedded applications require an initialization sequence before the user-defined `main()` function starts. This is called the startup code or boot code. The ARM C library contains pre-compiled and pre-assembled code sections that are necessary to start an application. When linking your application, the linker includes the necessary code, based on the application, from the C library to create a custom startup code for the application.

Note. Your embedded application running on a target can perform other target hardware initializations before calling the C library startup code. See *Reset and Initialization* in *Developing Software for ARM Processors* for more information.

The startup code for one application might be different to the startup code for another application. The document does not describe the precise startup code for any particular user application. Also, the document does not describe how to customize the startup code yourself. For information on how to customize the startup code, see *Developing Software for ARM Processors*.

The startup code described in this document applies to the standard ARM C library. It does not apply to the ARM C micro-library. It is also common to architectures ARMv4T and later.

3 Entry point to the C library

The function `__main` is the entry point to the C library. Unless you change it, `__main` is the default entry point to the ELF image that the ARM linker (armlink) uses when creating the image. Figure 1 shows the functions called by `__main` during the C library startup.

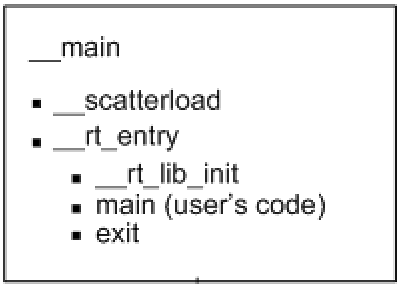

Figure 1 Overview of the functions called during the C library startup

`__rt_entry` and the functions called by `__rt_entry` are described in *Functions called by __rt_entry*.

3.1 __scatterload

Application code and data can be in a root region or a non-root region. Root regions have the same load-time and execution-time addresses. Non-root regions have different load-time and execution-time addresses. The root region contains a *region table* output by the ARM linker.

The region table contains the addresses of the non-root code and data regions that require initialization. The region table also contains a function pointer that indicates what initialization is needed for the region, for example a copying, zeroing, or decompressing function.

`__scatterload` goes through the region table and initializes the various execution-time regions. The function:

- Initializes the Zero Initialized (ZI) regions to zero
- Copies or decompresses the non-root code and data region from their load-time locations to the execute-time regions.

`__main` always calls this function during startup before calling `__rt_entry`.

3.2 See also

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

- *Initialization of the execution environment and execution of the application.*

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

- *Thread-safe C library functions.*

Developing Software for ARM Processors:

- *Tailoring the image memory map to your target hardware*
- *Local memory setup considerations*
- *Application startup*

- *Reset and initialization*
- *Scatter loading description file*
- *Root regions.*

4 Functions called by __rt_entry

__main calls __rt_entry to initialize the stack, heap and other C library sub systems. __rt_entry calls various initialization functions and then calls the user-level main().

This lists the functions that _rt_entry can call. The functions are listed in the order they get called:

1. _platform_pre_stackheap_init
2. __user_setup_stackheap or setup the Stack Pointer (SP) by another method
3. _platform_post_stackheap_init
4. __rt_lib_init
5. _platform_post_lib_init
6. main()
7. exit()

The _platform_* functions are not part of the standard C library. If you define them, then the linker places calls to them in __rt_entry.

main() is the entry point to the application at the user-level. Registers r0 and r1 contain the arguments to main(). If main() returns, its return value is passed to exit() and the application exits.

__rt_entry is also responsible for setting up the stack and heap. However, setting up the stack and heap depends on the method specified by the user. The stack and heap can be setup by any of the following methods:

- Calling __user_setup_stackheap. This also obtains the bounds of the memory used by the heap (heap top and heap base).
- Loading the SP with the value of the symbol __initial_sp.
- Using the top of the ARM_LIB_STACK or ARM_LIB_STACKHEAP region specified in the linker scatter file.

__rt_entry and __rt_lib_init do not exist as complete functions in the C library. Small sections of these functions are present in several internal objects that are part of the C library. Not all of these code sections are useful for a given user application. The linker decides which subset of those code sections are needed for a given application, and includes just those sections in the startup code. The linker places these sections in the correct order to create custom __rt_entry and __rt_lib_init functions as required by the user application.

The functions called by __rt_lib_init are described in *Functions called by __rt_lib_init*.

4.1 _platform_pre_stackheap_init

The standard C library does not provide this function but you can define it if you require it. You can use this function to setup hardware for example. __rt_entry calls this function, if you define it, before the code that initializes the stack and heap.

4.2 __user_setup_stackheap

This function enables you to setup and return the location of the initial stack and heap. The C library does not provide this function but you can define it if you require it. __rt_entry calls this function if you define it or if you define the legacy function __user_initial_stackheap. If you define __user_initial_stackheap, then the C library provides a default __user_setup_stackheap as a wrapper around your __user_initial_stackheap function.

4.3 _platform_post_stackheap_init

The C library does not provide this function but you can define it if you require it. You can use this function to setup hardware for example. `__rt_entry` calls this function, if you define it, after the code that initializes the stack and heap.

4.4 __rt_lib_init

This function initializes the various C library subsystems. It initializes the referenced library functions, initializes the locale and, if necessary, sets up `argc` and `argv` for `main()`. `__rt_entry` calls this function always during startup.

If you use the `__user_setup_stackheap` or `__user_initial_stackheap` functions to setup the stack pointer and heap, then the start and end address of the heap memory block are passed as arguments to `__rt_lib_init` in registers r0 and r1 respectively.

The function returns `argc` and `argv` in registers r0 and r1 respectively if the user-level `main()` requires them.

4.5 _platform_post_lib_init

The C library does not provide this function but you can define it if you require it. You can use this function to setup hardware for example. `__rt_entry` calls this function, if you define it, after the call to `__rt_lib_init` and before the call to the user-level `main()` function.

4.6 See also

Entry point to the C library

Developing Software for ARM Processors:

- *Reset and initialization*
- *Application startup*
- *Stack pointer initialization*
- *Placing the stack and heap.*

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

- *__rt_entry*
- *__user_setup_stackeheap()*
- *__rt_stackheap_init()*
- *__rt_lib_init()*
- *__rt_lib_shutdown()*
- *_sys_exit()*
- *Legacy function __user_initial_stackheap().*

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

- *Stack pointer initialization and heap bounds*

- *Initialization of the execution environment and execution of the application*
- *Legacy support for __user_initial_stackheap().*

5 Functions called by __rt_lib_init

The linker includes various initialization code sections from the internal object files to create a custom `__rt_lib_int` function. The linker places a function in `__rt_lib_init` only if it is needed by the application.

This lists the functions that `_rt_lib_init` can call. The functions are listed in the order they get called:

1. `_fp_init`
2. `_init_alloc`
3. `_rand_init`
4. `_get_lc_collate`
5. `_get_lc_ctype`
6. `_get_lc_monetary`
7. `_get_lc_numeric`
8. `_get_lc_time`
9. `_atexit_init`
10. `_signal_init`
11. `_fp_trap_init`
12. `_clock_init`
13. `_getenv_init`
14. `_initio`
15. `_ARM_get_argv`
16. `_alloca_initialize`
17. `_ARM_exceptions_init`
18. `__cpp_initialize__aeabi_`

5.1 _fp_init

This function initializes the floating-point environment by setting up the FP status word. If the user application uses VFP hardware, the function initializes the Floating-point Status and Control Register (FPSCR). If the application uses software VFP the function initializes the FP status word in memory. `__rt_lib_init` calls this function always during startup.

How fp_init is called depends on the ARM Compiler version:

- ARM Compiler v4.1

  `fp_init` is always called during startup.

- ARM Compiler 5

  `_fp_init` is called during startup unless you are using both `softfp` and an FP model without a status word (`--fpmode={ieee_no_fenv,std,fast}`). In this case, the call to `_fp_init` is completely omitted.

5.2 _init_alloc

This function sets up the data structures used by malloc, free, and other related functions. The function takes 2 parameters. The first parameter, in register r0, is the start of the heap memory block (heapbase), and the second parameter, in register r1, is the end of the heap memory block (heaptop). If these heap bound parameters are not passed as parameters to `__rt_lib_init`, then `__rt_lib_init` loads them using the symbols `__heap_base` and `__heap_limit`, or a special scatter load region, see *Functions called by __rt_entry*. `__rt_lib_init` calls this function if the application uses the heap.

5.3 _rand_init

This function initializes the random number generator to its default starting state. `__rt_lib_init` calls this function if the application uses `rand()`.

5.4 _get_lc_collate

This function obtains a pointer to the default block of data containing settings for the `LC_COLLATE` locale category. It inserts the pointer into the C library's stored locale pointer variable. `__rt_lib_init` calls this function if the application calls any function whose behavior depends on this locale setting.

5.5 _get_lc_ctype

This function obtains a pointer to the default block of data containing settings for the `LC_CTYPE` locale category. It inserts the pointer into the C library's stored locale pointer variable. `__rt_lib_init` calls this function if the application calls any function whose behavior depends on this locale setting.

5.6 _get_lc_monetary

This function obtains a pointer to the default block of data containing settings for the `LC_MONETARY` locale category. It inserts the pointer into the C library's stored locale pointer variable. `__rt_lib_init` calls this function if the application calls any function whose behavior depends on this locale setting.

5.7 _get_lc_numeric

This function obtains a pointer to the default block of data containing settings for the `LC_NUMERIC` locale category. It inserts the pointer into the C library's stored locale pointer variable. `__rt_lib_init` calls this function if the application calls any function whose behavior depends on this locale setting.

5.8 _get_lc_time

This function obtains a pointer to the default block of data containing settings for the `LC_TIME` locale category. It inserts the pointer into the C library's stored locale pointer variable. `__rt_lib_init` calls this function if the application calls any function whose behavior depends on this locale setting.

5.9 _atexit_init

This function sets up the C library's storage for the function pointers that are passed to `atexit()`. In a multithreaded application, it also sets up the mutex that protects the storage against concurrent accesses. `__rt_lib_init` calls this function if the application uses `atexit()`.

5.10 _signal_init

This function sets up the storage that contains the current handler for each signal number. In a multithreaded application, it also sets up the mutex that protects this storage against concurrent accesses. `__rt_lib_init` calls this function if the application uses `signal()`.

5.11 _fp_trap_init

This function sets up the library's storage that contains the current handler for each type of floating-point exception. In a multithreaded application, it also sets up the mutex that protects this storage against concurrent accesses. `__rt_lib_init` calls this function if the application uses trapped floating-point exceptions, for example if you use either of the following:

- `--fpmode=ieee_full`
- `--fpmpde=ieee_fixed`.

5.12 _clock_init

This function reads the current value of the timer used by `clock()`. This is stored as the start time of the program. Subsequent calls to `clock()` returns the time elapsed since the program start time. These are the default implementations of `clock()` and `_clock_init`. You can re-implement them differently. `__rt_lib_init` calls this function if the application uses `clock()`.

5.13 _getenv_init

The standard C library does not provide this function but you can define it if you require it. This function enables `getenv()` to retrieve any needed data. `__rt_lib_init` calls this function if you define it.

5.14 _initio

This function sets up the stdio internal state. This includes initializing the list of open files, and calling `_sys_open()` to open the three standard streams. `__rt_lib_init` calls this function if the application uses stdio.

5.15 __ARM_get_argv

This function gets the argc and argv values for passing to `main()`. The function returns `argc` and `argv` in registers r0 and r1 respectively. The function might return two more arguments in registers r2 and r3. `__rt_lib_init` calls this function if `main()` is declared with arguments.

`__ARM_get_argv` calls `_sys_command_string` to obtain the argument list as a single string. It then breaks this string into separate strings for each word.

5.16 __alloca_initialize

This function sets up the alloca list pointer to NULL. `__rt_lib_init` calls this function if the RVCT heap based alloca is used.

5.17 __ARM_exceptions_init

This function sets up the C++ exception handling state. `__rt_lib_init` calls this function if the application uses C++ exceptions.

5.18 __cpp_initialize__aeabi_

This function calls the constructors of top-level C++ objects. `__rt_lib_init` calls this function if the application has top-level C++ objects.

5.19 See also

Functions called by __rt_entry

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

- *__rt_fp_status_addr()*
- *Using malloc() when exploiting the C library*
- *Using a heap implementation from bare machine C*
- *Definition of locale data blocks in the C library*
- *C++ initialization, construction and destruction*
- *Initialization of the execution environment and execution of the application*
- *Exceptions system initialization*
- *Assembler macros that tailor locale functions in the C library.*

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

- *_getenv_init()*
- *getenv()*
- *_clock_init()*
- *_findlocale()*
- *_sys_command_string()*
- *__rt_lib_init.*

Linker Reference:

- *--ref_cpp_init, --no_ref_cpp_init.*

6 Appendix

This appendix provides a summary of the various functions that might be called during the startup of an application. It also shows when the function is included in the startup code.

Table 1 Summary of startup functions

Symbol name	Description	Inclusion in startup code
`__alloca_initialize`	Sets up the alloca list pointer to NULL	When using heap based alloca
`__ARM_exceptions_init`	Sets up exception handling state	When using C++ exceptions
`__ARM_get_argv`	Gets the `argc` and `argv` values for `main()`	If `main()` is defined with arguments
`_atexit_init`	Sets up storage for function pointers	When using `atexit()`
`_clock_init`	Reads current value of timer used by `clock()`	When using `clock()`
`__cpp_initialize__aeabi_`	Calls top-level C++ constructors	When using top-level C++ objects
`_fp_init`	Initializes the floating-point environment	Always
`_fp_trap_init`	Sets up storage for floating-point exception handlers	When using trapped floating-point exceptions
`_get_lc_collate`	Stores the pointer to the data block containing the `LC_COLLATE` settings	When using functions that depend on the collate locale setting
`_get_lc_ctype`	Stores the pointer to the data block containing the `LC_CTYPE` settings	When using functions that depend on the ctype locale setting
`_get_lc_monetary`	Stores the pointer to the data block containing the `LC_MONETARY` settings	When using functions that depend on the monetary locale setting
`_get_lc_numeric`	Stores the pointer to the data block containing the `LC_NUMERIC` settings	When using functions that depend on the numeric locale setting
`_get_lc_time`	Stores the pointer to the data block containing the `LC_TIME` settings	When using functions that depend on the time locale setting
`_getenv_init`	Enables `getenv()` to initialize itself	If you define `_getenv_init`
`_init_alloc`	Sets up the data structures used by malloc, free and other related functions	When using the heap
`_initio`	Sets up the stdio internal state	When using stdio
`_rand_init`	Initializes the random number generator	When using `rand()`
`_platform_post_lib_init`	Enables initialization after `__rt_lib_init`	If you define `_platform_post_lib_init`
`_platform_post_stackheap_init`	Enables initialization after stack initialization	If you define `_platform_post_stackheap_init`
`_platform_pre_stackheap_init`	Enables initialization before stack initialization	If you define `_platform_pre_stackheap_init`
`__rt_entry`	Sets up the run-time environment, and then calls `main()`	Always
`__rt_lib_init`	Calls the necessary C library initialization functions	Always

Table 2 Summary of startup functions (continued)

Symbol name	Description	Inclusion in startup code
`__scatterload`	Copies code and data from load region to execute region	Always
`_signal_init`	Sets up storage for signal handlers	When using `signal()`
`_user_setup_stackheap`	Sets up stack and heap	If you define `_user_setup_stackheap`