AN4296
Application note

Overview and tips for using STM32F303/328/334/358xx CCM RAM with IAR EWARM, Keil MDK-ARM and GNU-based toolchains

Introduction

The purpose of this application note is to give a presentation of the core coupled memory (CCM) RAM available on STM32F303xB/xC and STM32F358xC microcontrollers and describe what is required to execute part of the application code from this memory region using different toolchains.

This application note is divided into four sections: the first section gives an overview of the STM32F3 CCM RAM, while the next sections describe the steps required to execute part of the application code from CCM RAM using the following toolchains:

- IAR EWARM
- KEIL MDK-ARM™
- RIDE and Atollic GNU based toolchain

The procedures described throughout the document are applicable to other RAM regions such as the CCM data RAM of some F4 devices, or external SRAM.

Refer to *Table 1* for the list microcontrollers embedding CCM RAM.

Table 1. Applicable products

Product family	Part numbers or product categories
Microcontrollers	STM32F303xB, STM32F303xC, STM32F358xC STM32F303x6/x8, STM32F328x8, STM32F334x4/x6/x8

Contents

List of tables

List of figures

1 Overview of STM32F303xB/C and STM32F358xC CCM RAM

1.1 Purpose

The STM32F303xB/C and STM32F358xC CCM RAM is tightly coupled with the Cortex™ core. it is primarily intended to execute code at maximum system clock frequency (72 MHz) without any wait state penalty. It thus allows to significantly decrease critical task execution time, compared to code execution from Flash memory.

CCM RAM is typically used for real-time and computation intensive routines, such as:

- Digital power conversion control loops (switch mode power supplies, lighting)
- Field-oriented 3-phase motor control
- Real-time DSP tasks

When code is located in CCM RAM and data stored in the regular SRAM, the Cortex-M4 core is in the optimum Harvard configuration. A dedicated zero-wait state memory is connected to each of its I- and D-bus (refer to *Figure 1: STM32F303xB/xC and STM32F358xC system architecture*) and can thus perform at 1.25DMIPS/MHz up to 72 MHz, with a deterministic performance of 90 DMIPS. This also guarantees a minimal latency if interrupt service routines are placed in the CCM RAM.

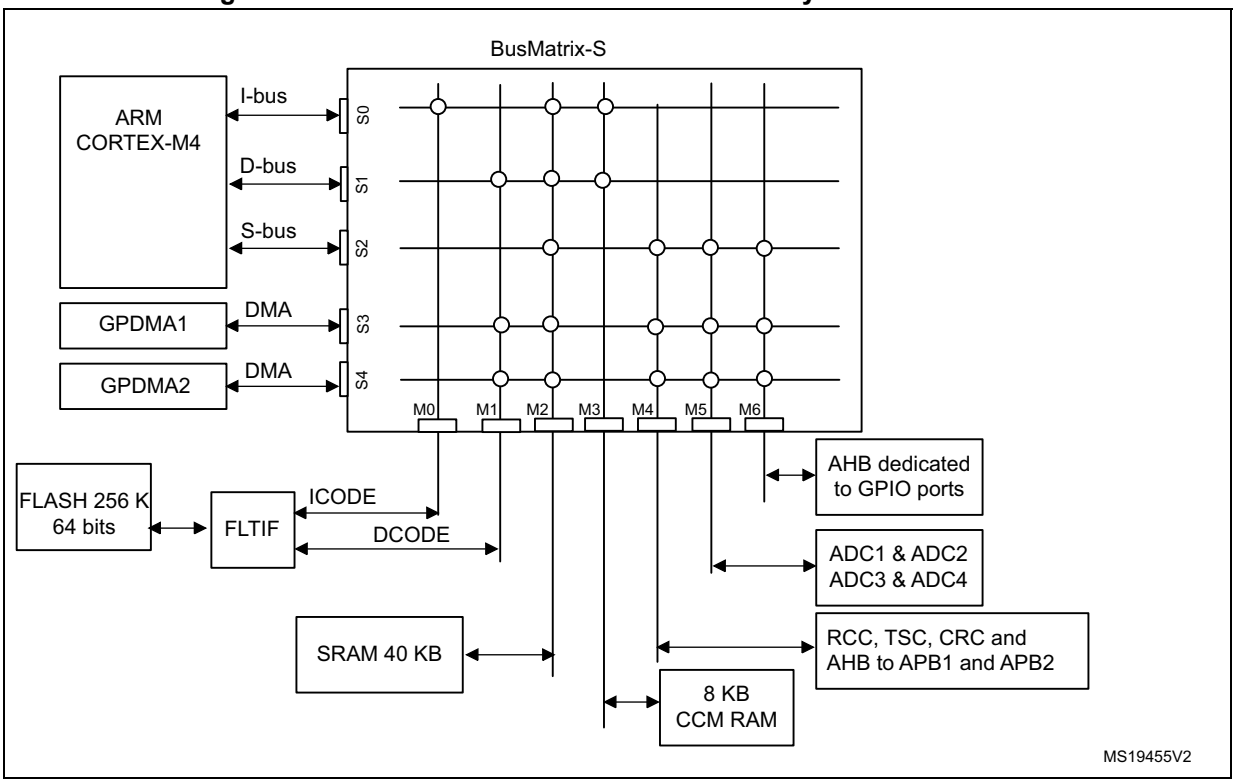
Example

A benchmark between the STM32F103xx and STM32F303xx microcontrollers using STMicroelectronics MC library V3.4 shows that in case of single motor control using 3 shunt algorithm, the FOC total execution time for STM32F303xx is 16.97 µs compared to 21.3 µs in STMF103xx (see note below); with FOC core and sensorless core loops running from CCM RAM for STM32F303xx. This means that the STM32F303xx is 20.33 % faster than the STM32F103xx thanks to the CCM RAM.

Note: FOC routines are programmed in structured C, so the values provided above do not represent the fastest possible execution both for STM32F103xx and STM32F303xx. In addition, the execution time is also function of the compiler used and of its version.

When the CCM RAM is not used for code, it can hold data like an extra SRAM memory. However it cannot be accessed through DMA. It is not recommended to place both code and data together in the CCM, since the Cortex core will have to fetch code and data from the same memory with the risk of collisions. The core would then be in the Von Neuman configuration, and its performance would drop from 1.25DMIPS/MHz to below 1DMIPS/MHz.

Figure 1. STM32F303xB/xC and STM32F358xC system architecture



1.2 STM32F303xB/C and STM32F358xC CCM RAM features

1.2.1 CCM RAM mapping

The CCM RAM is available on the STM32F303xB/C and STM32F358xC devices, starting from 0x1000 0000 address.

1.2.2 CCM RAM remapping

Unlike regular SRAM, the CCM RAM cannot be remapped at address 0x0000 0000.

1.2.3 CCM RAM write protection

The CCM RAM can be protected against unwanted write operations with a page granularity of 1 Kbyte. Refer to *Table 2* for a description of CCM RAM organization.

Table 2. CCM RAM organization

Page number	Start address	End address
Page 0	0x1000 0000	0x1000 03FF
Page 1	0x1000 0400	0x1000 07FF
Page 2	0x1000 0800	0x1000 0BFF
Page 3	0x1000 0C00	0x1000 0FFF

The write protection is enabled through the SYSCFG CCM SRAM protection register (SYSCFG_RCR). This is a write '1' once mechanism, which means that once the write protection is enabled on a given CCM RAM page by programming the corresponding bit to ‘1’, it can be cleared only through a system reset. For more details refer to the product reference manual.

1.2.4 CCM RAM parity check

A parity check is implemented on STM32F303xB/C and STM32F358xC microcontrollers. It is disabled by default and can be enabled by the user when needed through an option bit (SRAM_PE bit). When this option bit is cleared, the parity check is enabled for the first 16 Kbytes of SRAM and for the 8-Kbyte CCM RAM.

2 Execute application code from CCM RAM using the IAR EWARM toolchain

2.1 Executing a simple code from CCM RAM (except for interrupt handler)

A simple code can be composed of one or more functions that are not referenced from an interrupt handler. If the code is referenced from an interrupt handler, follow the steps described in *Section 2.2: Executing an interrupt handler from CCM RAM*.

EWARM provides the possibility to place one or more functions or a whole source file in CCM RAM.

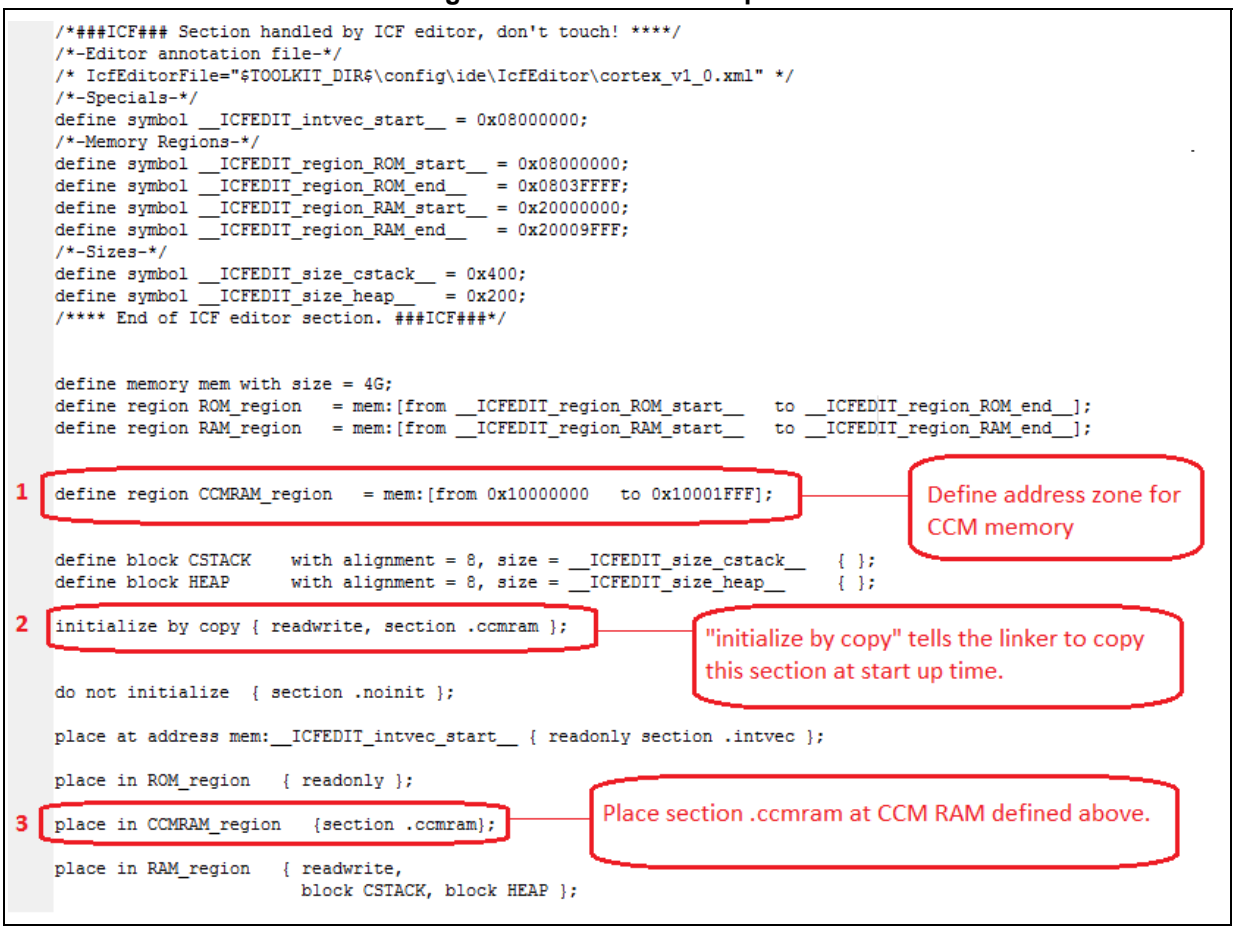
This operation requires a new section to be defined in the linker file (.icf) to host the code to be placed in CCM RAM. This section is copied to CCM RAM at startup. The required steps are the following:

1. Define the address area for the CCM RAM by indicating the start and end addresses.
2. Tell the linker to copy at startup the section named .ccmram from Flash memory to CCM RAM.
3. Indicate to the linker that the code section .ccmram should be placed in the CCM RAM region.

Refer to *Figure 2: EWARM linker update* for an example of code implementing these operations.

Note: This procedure is not valid for interrupt handlers.

Figure 2. EWARM linker update



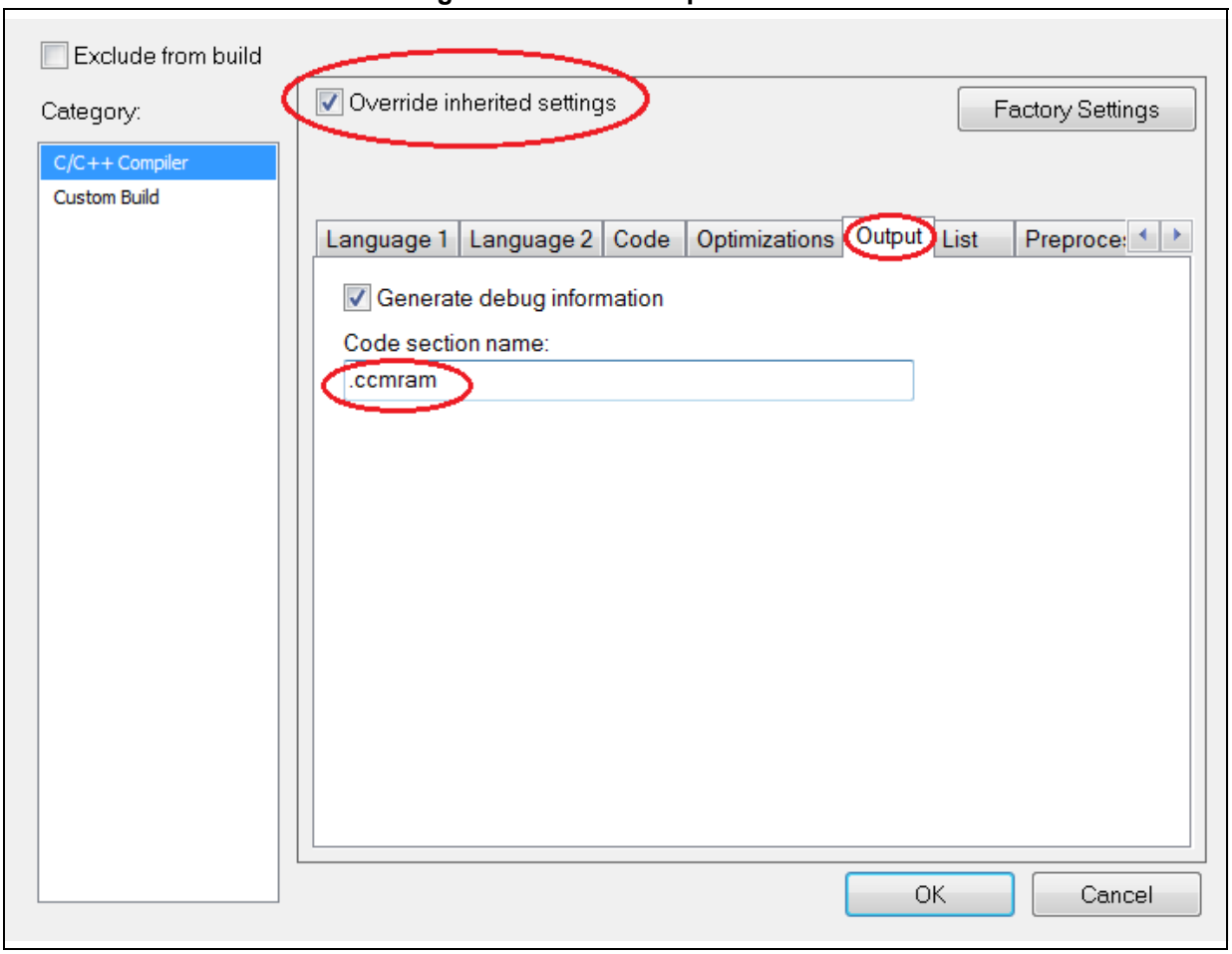
2.1.1 Executing a source file from CCM RAM

Executing a source file from CCM RAM means that all functions declared in this file will be executed from this memory area.

To place and execute a source file from CCM RAM, use the EWARM file **Options** window:

1. Add the section .ccmram (for example) in the linker file as defined in *Section 2.1*.
2. Right click the file name from the workspace window.
3. Select **options** from the displayed menu.
4. Check **override inherited settings** from the displayed window
5. Select the **output** tab, and type the name of the section already defined in the linker file ('.ccmram' in this example) in the **Code section name** field (see *Figure 3: EWARM file placement*).

Figure 3. EWARM file placement

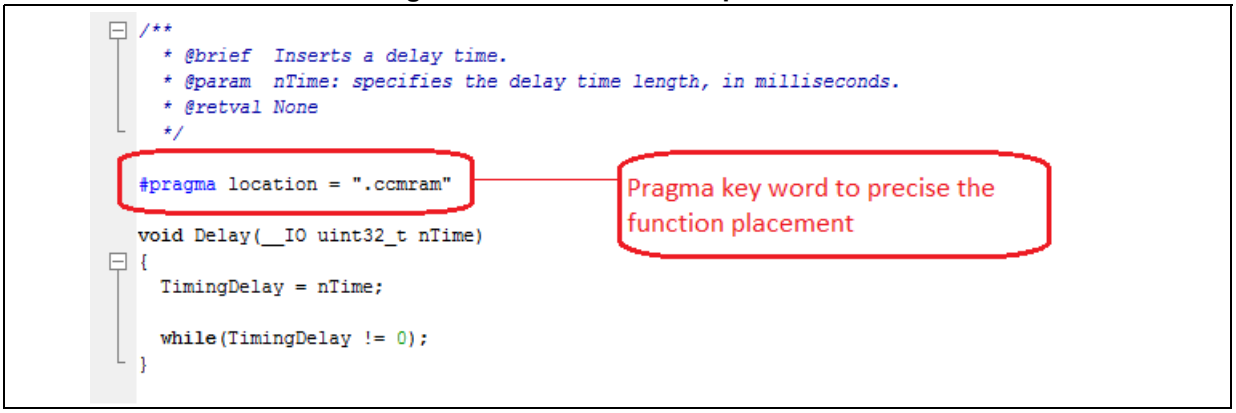


2.1.2 Executing one or more functions from CCM RAM

The steps required to execute a function from CCM RAM are the following:

1. Add the section .ccmram in the linker file as described in *Section 2.1*.
2. Using the key word **pragma location**, specify the function to be executed from CCM RAM (see *Figure 4: EWARM function placement*).

Figure 4. EWARM function placement



*Note: To execute more than one function from CCM RAM, the **pragma** location keyword should be placed above each function declaration.*

2.2 Executing an interrupt handler from CCM RAM

The vector table is implemented as an array named *__vector_table* and referenced in the startup code.

EWARM linker protects the sections that are referenced from the startup code from being affected by an 'initialize by copy' directive. So, you should not use the symbol *__vector_table* to allow copying interrupt handler sections via the 'initialize by copy' directive.

As a consequence, you should make a second vector table and place it in CCM RAM.

The steps required to execute an interrupt handler from CCM RAM are the following:

1. Update the linker file (.icf).
2. Update the startup file.
3. Place the interrupt handler in CCM RAM.
4. Remap the vector table to CCM RAM.

2.2.1 Updating the linker file (.icf)

To update the linker file:

1. Define the address where the second vector table will be located: 0x1000 0000.
2. Define the memory address area for the CCM RAM by specifying the start and end addresses.
3. Tell the linker to copy at startup the section named .ccmram and the second vector table section ‘.intvec_CCMRAM’ from Flash memory to CCM RAM.
4. Tell the linker that the second vector table should be placed in the intvec_CCMRAM section.
5. Indicate that the .ccmram code section should be placed in CCM RAM.

Figure 5. EWARM linker update for interrupt handler

```
/*###ICF### Section handled by ICF editor, don't touch! ****/
/*-Editor annotation file-*/
/* IcfEditorFile="$TOOLKIT_DIR$\config\ide\IcfEditor\cortex_v1_0.xml" */
/*-Specials-*/
define symbol __ICFEDIT_intvec_start__ = 0x08000000;
/*-Memory Regions-*/
define symbol __ICFEDIT_region_ROM_start__    = 0x08000000;
define symbol __ICFEDIT_region_ROM_end__      = 0x0803FFFF;
define symbol __ICFEDIT_region_RAM_start__    = 0x20000000;
define symbol __ICFEDIT_region_RAM_end__      = 0x20009FFF;
/*-Sizes-*/
define symbol __ICFEDIT_size_cstack__ = 0x1000;
define symbol __ICFEDIT_size_heap__   = 0x0000;
/**** End of ICF editor section. ###ICF###*/

1 define symbol CCMRAM_intvec_start = 0x10000000;

define memory mem with size = 4G;
define region ROM_region      = mem:[from __ICFEDIT_region_ROM_start__   to __ICFEDIT_region_ROM_end__];
define region RAM_region      = mem:[from __ICFEDIT_region_RAM_start__   to __ICFEDIT_region_RAM_end__];

2 define region CCMRAM_region   = mem:[from 0x10000000   to 0x10001FFF];

define block CSTACK    with alignment = 8, size = __ICFEDIT_size_cstack__   { };
define block HEAP      with alignment = 8, size = __ICFEDIT_size_heap__     { };

3 initialize by copy { readwrite, section .intvec_CCMRAM, section .ccmram, ro object stm32f30_it.o };

do not initialize  { section .noinit };

place at address mem:__ICFEDIT_intvec_start__ { readonly section .intvec };

4 place at address mem: CCMRAM_intvec_start { section .intvec_CCMRAM };

5 place in CCMRAM_region   { section .ccmram };

place in ROM_region   { readonly };
place in RAM_region   { readwrite,
                        block CSTACK, block HEAP };
```

2.2.2 Updating the startup file

To update the startup file:

1. Make a second vector table to be stored in CCM RAM. The startup_stm32f30x.s file should be modified by removing all entries except for sfe(CSTACK) and Reset_Handler from the original vector table '__vector_table'.
2. Add a second vector table to be placed in CCM RAM. It should contain all entries. As an example you can call it '__vector_table_CCMRAM'. This vector table must be placed in the intvec_CCMRAM section defined in the linker file.

Figure 6. EWARM startup file update for interrupt handler

```
49          ;; Forward declaration of sections.
50          SECTION CSTACK:DATA:NOROOT(3)
51
52          SECTION .intvec:CODE:NOROOT(2)
53
54          EXTERN  __iar_program_start
55          EXTERN  SystemInit
56          PUBLIC  __vector_table
57
58          DATA
59
(1) __vector_table
            DCD     sfe(CSTACK)
62          DCD     Reset_Handler             ; Reset Handler
63
64
65          SECTION .intvec_CCMRAM:CODE:ROOT(2)
66
67          PUBLIC  __vector_table_CCMRAM
68
(2)         DATA
    __vector_table_CCMRAM
71          DCD     sfe(CSTACK)
72          DCD     Reset_Handler             ; Reset Handler
73          DCD     NMI_Handler               ; NMI Handler
74          DCD     HardFault_Handler         ; Hard Fault Handler
75          DCD     MemManage_Handler         ; MPU Fault Handler
76          DCD     BusFault_Handler          ; Bus Fault Handler
77          DCD     UsageFault_Handler        ; Usage Fault Handler
78          DCD     0                         ; Reserved
79          DCD     0                         ; Reserved
80          DCD     0                         ; Reserved
```

2.2.3 Place the interrupt handler in CCM RAM

Place the interrupt handler to be executed in CCM RAM as described in *Section 2.1.2* or the whole stm32f_it.c file as described in *Section 2.1.1*.

2.2.4 Remap the vector table to CCM RAM

In SystemInit function, remap the vector table to CCM RAM by modifying the VTOR register as following:

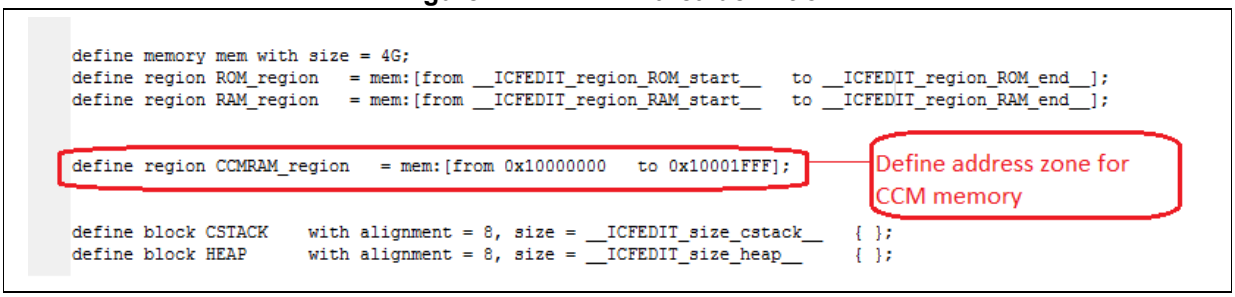
```
SCB->VTOR = 0x10000000 | VECT_TAB_OFFSET;
```

2.3 Executing a library (.a) from CCM RAM

EWARM allows executing a library or a library module from CCM RAM. The actions to be executed are the following:

1. Define the memory address area corresponding to the CCM RAM by specifying the start and end addresses.

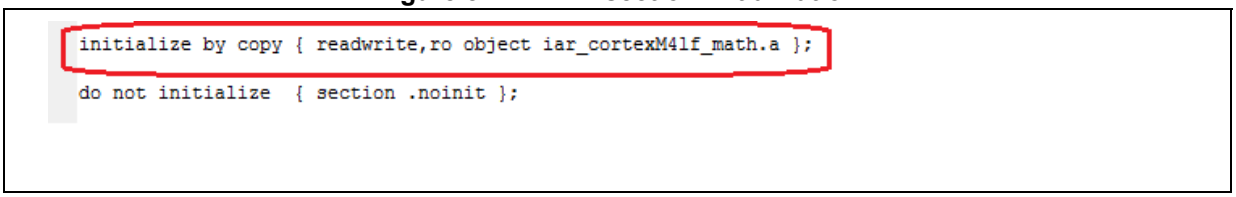
Figure 7. CCM RAM area definition



2. Update the linker to copy at startup the library or the library module in CCM RAM using the “initialize by copy” directive.

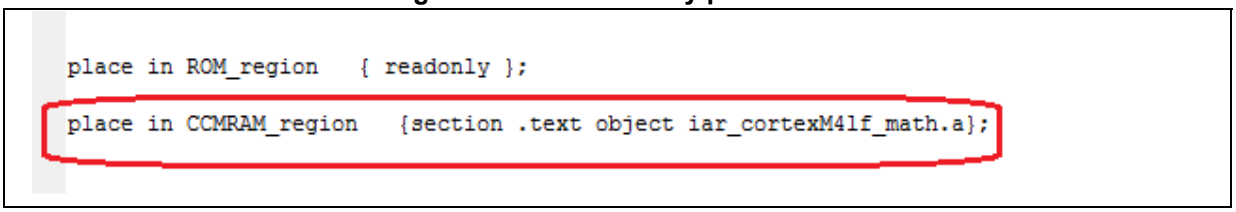
   Example:

Figure 8. EWARM section initialization



3. Indicate to the linker that the library should be placed in CCM RAM:

Figure 9. EWARM library placement



To execute a library module from CCM RAM, follow steps 1, 2 and 3 using the library module name.

The example below shows how to place arm_abs_f32.o (a module of iar_cortexM4l_math.a library) in CCM RAM:

Figure 10. EWARM library module placement

```
/*###ICF### Section handled by ICF editor, don't touch! ****/
/*-Editor annotation file-*/
/* IcfEditorFile="$TOOLKIT_DIR$\config\ide\IcfEditor\cortex_v1_0.xml" */
/*-Specials-*/
define symbol __ICFEDIT_intvec_start__ = 0x08000000;
/*-Memory Regions-*/
define symbol __ICFEDIT_region_ROM_start__ = 0x08000000;
define symbol __ICFEDIT_region_ROM_end__   = 0x0803FFFF;
define symbol __ICFEDIT_region_RAM_start__ = 0x20000000;
define symbol __ICFEDIT_region_RAM_end__   = 0x20009FFF;
/*-Sizes-*/
define symbol __ICFEDIT_size_cstack__ = 0x400;
define symbol __ICFEDIT_size_heap__   = 0x200;
/**** End of ICF editor section. ###ICF###*/
define memory mem with size = 4G;
define region ROM_region   = mem:[from __ICFEDIT_region_ROM_start__   to __ICFEDIT_region_ROM_end__];
define region RAM_region   = mem:[from __ICFEDIT_region_RAM_start__   to __ICFEDIT_region_RAM_end__];

1 define region CCMRAM_region   = mem:[from 0x10000000   to 0x10001FFF];

define block CSTACK    with alignment = 8, size = __ICFEDIT_size_cstack__   { };
define block HEAP      with alignment = 8, size = __ICFEDIT_size_heap__     { };

2 initialize by copy { readwrite, ro object arm_abs_f32.o };

do not initialize  { section .noinit };

place at address mem:__ICFEDIT_intvec_start__ { readonly section .intvec  };
place in ROM_region   { readonly };

3 place in CCMRAM_region   {section .text object arm_abs_f32.o };

place in RAM_region   { readwrite,
                        block CSTACK, block HEAP };
```

3 Execute application code from CCM RAM using the KEIL MDK-ARM toolchain

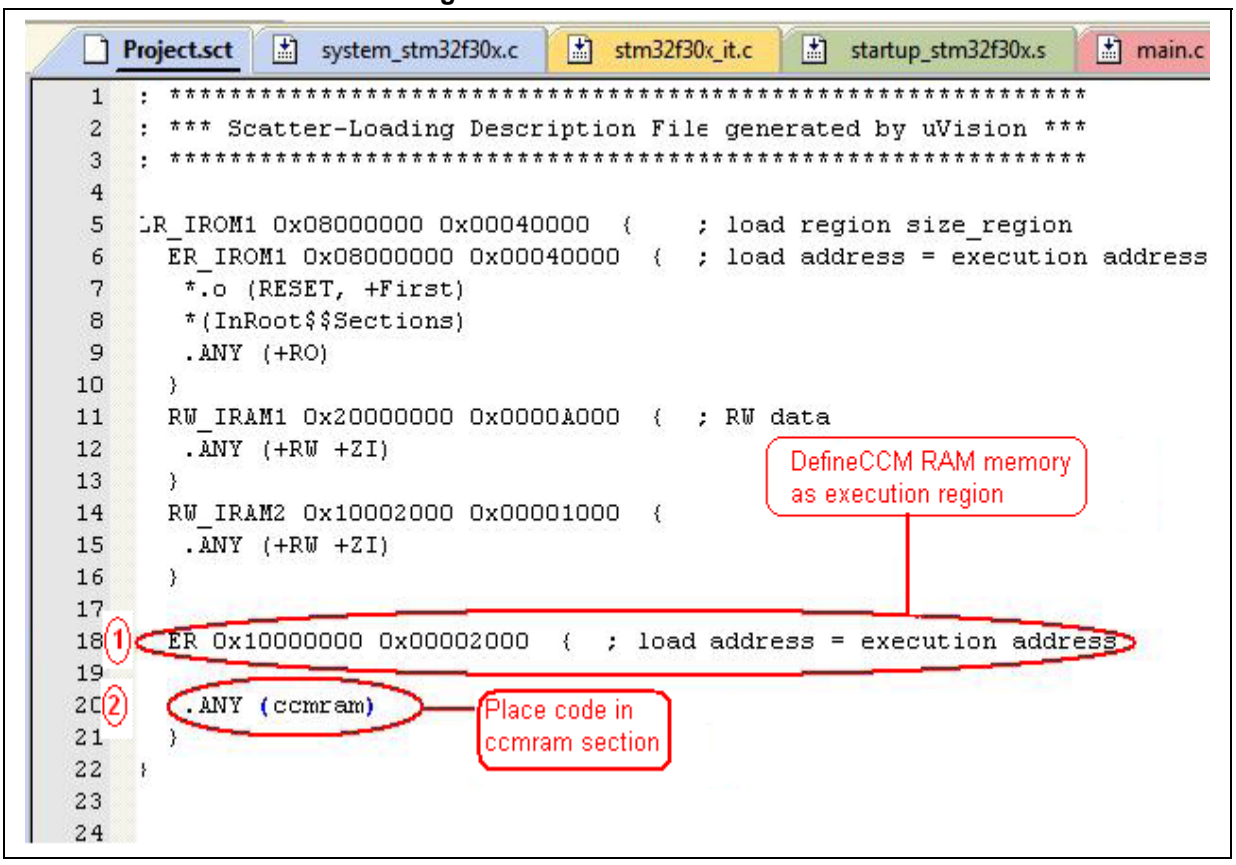
MDK-ARM features make it possible to execute simple functions or interrupt handlers from CCM RAM. The following sections explain how to use these features to execute code from CCM RAM.

3.1 Executing a function or an interrupt handler from CCM RAM

The steps required to execute a function or an interrupt handler from CCM RAM are the following:

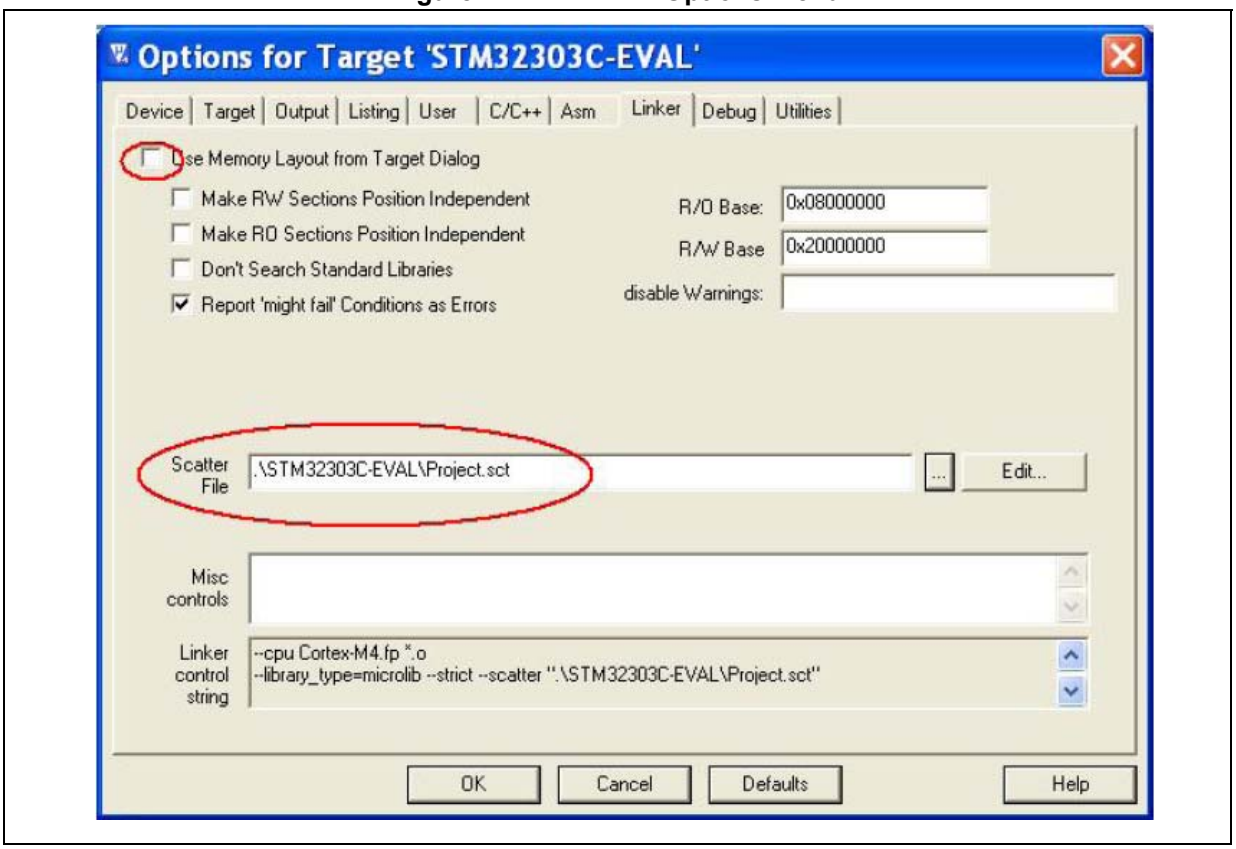
1. Define a new region (ccmram) in the scatter file by indicating the start and end addresses of the CCM RAM area.
2. Indicate to the linker that the sections with ccmram attribute must be placed in the CCM RAM region.

Figure 11. MDK-ARM scatter file



3. Refer to the modified scatter file for the project options (see *Figure 11*):

Figure 12. MDK-ARM Options menu



4. Place the part of code to be executed from CCM RAM in the ccmram section defined above. This is done by adding the attribute key word above the function declaration.

Figure 13. MDK-ARM function placement

```
/**
  * @brief  This function handles SysTick Handler.
  * @param  None
  * @retval None
  */
__attribute__((section ("ccmram")))
void SysTick_Handler(void)
{
  TimingDelay_Decrement();
}
```

Note: To execute more than one function from CCM RAM, the attribute keyword should be placed above each function declaration:

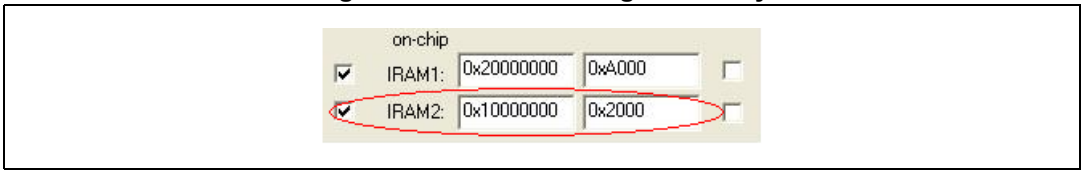
3.2 Executing a source file from CCM RAM

Executing a source file from CCM RAM means that all functions declared in this file will be executed from the CCM RAM region.

Follow the steps below to execute a file from CCM RAM:

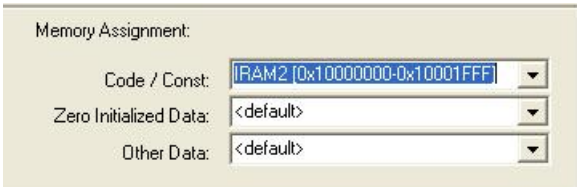
1. Define the CCM RAM as a memory area in the **project option** window (Project>option>target):

Figure 14. MDK-ARM target memory



2. Right click the file to place it in CCM RAM and select **options**
3. Select the CCM RAM region in the **memory assignment** menu:

Figure 15. MDK-ARM file placement



3.3 Executing a library or a library module from CCM RAM

Follow the steps below to execute a library or a library module from CCM RAM:

1. Define the CCM RAM as a memory area as shown in *Figure 16: MDK-ARM library placement*.
2. Right click the library from the workspace and select **options**.
3. You can either place the complete library or a module from a library in CCM RAM.

Figure 16. MDK-ARM library placement

4 Execute application code from CCM RAM using a GNU-based toolchain

GNU-based toolchains allow executing simple functions or interrupt handlers from CCM RAM. The following sections explain how to use these features to execute code from CCM RAM.

4.1 Executing a function or an interrupt handler from CCM RAM

The steps required to execute a function or an interrupt handler from CCM RAM are the following:

1. Define a new region (ccmram) in the linker file (.ld) by defining the start address and the size of CCM RAM region (see *Figure 17: GNU linker update*)

Figure 17. GNU linker update

```
/* Entry Point */
ENTRY(Reset_Handler)

/* Highest address of the user mode stack */
_estack = 0x2000a000;    /* end of 40K RAM on AHB bus*/

/* Generate a link error if heap and stack don't fit into RAM */
_Min_Heap_Size = 0;      /* required amount of heap  */
_Min_Stack_Size = 0x400; /* required amount of stack */

/* Specify the memory areas */
MEMORY
{
  FLASH (rx)      : ORIGIN = 0x08000000, LENGTH = 256K
  RAM (xrw)       : ORIGIN = 0x20000000, LENGTH = 40K
  MEMORY_B1 (rx)  : ORIGIN = 0x60000000, LENGTH = 0K
  CCMRAM (xrw)    : ORIGIN = 0x10000000, LENGTH = 8K
}
```

Define address zone for CCM RAM region

2. Tell the linker that code sections with ccmram attribute must be placed in CCM RAM (see *Figure 18: GNU linker section definition*).

Figure 18. GNU linker section definition

```
  *(.data)           /* .data sections */
  *(.data*)          /* .data* sections */

  . = ALIGN(4);
  _edata = .;        /* define a global symbol at data end */
} >RAM AT> FLASH

_siccmram = LOADADDR(.ccmram);

/* CCM-RAM section
*
* IMPORTANT NOTE!
* If initialized variables will be placed in this section,
* the startup code needs to be modified to copy the init-values.
*/
.ccmram :
{
  . = ALIGN(4);
  _sccmram = .;       /* create a global symbol at ccmram start */
  *(.ccmram)
  *(.ccmram*)

  . = ALIGN(4);
  _eccmram = .;       /* create a global symbol at ccmram end */
} >CCMRAM AT> FLASH

/* Uninitialized data section */
. = ALIGN(4);
.bss :
```

3. Modify the startup file to initialize data to place in CCM RAM at startup time (see code lines in red):

```
.section  .text.Reset_Handler
.weak  Reset_Handler
.type  Reset_Handler, %function
Reset_Handler:
/* Copy the data segment initializers from flash to SRAM and CCMRAM */
  movs  r1, #0
b  LoopCopyDataInit
CopyDataInit:
ldr  r3, =_sidata
ldr  r3, [r3, r1]
str  r3, [r0, r1]
adds  r1, r1, #4
LoopCopyDataInit:
ldr  r0, =_sdata
ldr  r3, =_edata
adds  r2, r0, r1
cmp  r2, r3
bcc  CopyDataInit
movs  r1, #0
```

```
b  LoopCopyDataInit1
CopyDataInit1:
ldr  r3, =_siccmram
ldr  r3, [r3, r1]
str  r3, [r0, r1]
adds  r1, r1, #4
LoopCopyDataInit1:
ldr  r0, =_sccmram
ldr  r3, =_eccmram
adds  r2, r0, r1
cmp  r2, r3
bcc  CopyDataInit1
ldr  r2, =_sbss
b  LoopFillZerobss
/* Zero fill the bss segment. */
FillZerobss:
movs  r3, #0
str  r3, [r2], #4
LoopFillZerobss:
ldr  r3, = _ebss
cmp  r2, r3
bcc  FillZerobss
/* Call the clock system intitialization function.*/
bl  SystemInit
/* Call the application's entry point.*/
bl  main
bx  lr
```

4. Place the part of code to be executed from CCM RAM in the .ccmram section by adding the attribute key word in the function prototype:

Figure 19. GNU function placement

```
void NMI_Handler(void);
void HardFault_Handler(void);
void MemManage_Handler(void);
void BusFault_Handler(void);
void UsageFault_Handler(void);
void SVC_Handler(void);
void DebugMon_Handler(void);
void PendSV_Handler(void);

void SysTick_Handler(void) __attribute__((section (".ccmram")));
```

4.2 Executing a file from CCM RAM

Executing a source file from CCM RAM means that all functions declared in this file will be executed from CCM RAM.

To execute a file from CCM RAM, follow the sequence below:

1. Add the .ccmram section in the linker file as defined in *Section 4.1*.
2. Place your file in CCM RAM as shown below:

Figure 20. GNU file placement

```
_siccmram = LOADADDR(.ccmram);

 /* CCM-RAM section
 *
 * IMPORTANT NOTE!
 * If initialized variables will be placed in this section,
 * the startup code needs to be modified to copy the init-values.
 */
 .ccmram :
 {
   . = ALIGN(4);
   _sccmram = .;       /* create a global symbol at ccmram start */
   *(.ccmram)
   *(.ccmram*)

     stm32f30x_it.o(*)

   . = ALIGN(4);
   _eccmram = .;       /* create a global symbol at ccmram end */


} >CCMRAM AT> FLASH
```

4.3 Executing a library from CCM RAM

Follow the steps below to execute a library from CCM RAM:

1. Add the .ccmram section in the linker file as defined in *Section 4.1*.
2. Place your libray in CCM RAM as shown below:

Figure 21. GNU library placement

```
/* CCM-RAM section
*
* IMPORTANT NOTE!
* If initialized variables will be placed in this section,
* the startup code needs to be modified to copy the init-values.
*/
.ccmram :
{
  . = ALIGN(4);
  _sccmram = .;       /* create a global symbol at ccmram start */
  *(.ccmram)
  *(.ccmram*)

    mylib.a(*)

  . = ALIGN(4);
  _eccmram = .;       /* create a global symbol at ccmram end */


} >CCMRAM AT> FLASH
```

5 Revision history

Table 3. Document revision history

Date	Revision	Changes
23-Jul-2013	1	Initial release.
25-Mar-2014	2	Changed STM32F313xC into STM32F358xC. Reworked *Section 1: Overview of STM32F303xB/C and STM32F358xC CCM RAM*.
02-Sep-2014	3	Added STM32F303x6/x8, STM32F328x8, STM32F334x4/x6/x8 in *Table 1: Applicable products*. Updated step 2 in *Section 2.1: Executing a simple code from CCM RAM (except for interrupt handler)*, step 3 in *Section 2.2.1: Updating the linker file (.icf)* and updated *Figure 5: EWARM linker update for interrupt handler*. Updated *Figure 11: MDK-ARM scatter file*.