AN4296

Application note

Use STM32F3/STM32G4 CCM SRAM with IAR Embedded Workbench®, Keil® MDK-ARM, STMicroelectronics STM32CubeIDE, and other GNU-based toolchains

Introduction

This document gives a presentation of the core-coupled memory (CCM) SRAM available on STM32F3/STM32G4 microcontrollers and describes what is required to execute part of the application code from this memory region using different toolchains.

The first section provides an overview of the CCM SRAM, while the next sections describe the steps required to execute part of the application code from CCM SRAM using the following toolchains:

- IAR Systems® IAR Embedded Workbench®
- Keil® MDK-ARM
- STMicroelectronics STM32CubeIDE and other GNU-based toolchains

The procedures described throughout the document are applicable to other SRAM regions such as the CCM data RAM of some STM32F4 devices, or external SRAM.

Table 1 lists the STM32 microcontrollers used as examples for CCM SRAM.

Table 1. Applicable products

Reference		Products
STM32F3/STM32G4	STM32F3	STM32F303 line, STM32F334 line STM32F328C8 STM32F358CC, STM32F358RC, STM32F358VC STM32F398VE
	STM32G4	STM32G4 Series

1 Overview of STM32F3/STM32G4 CCM SRAM

This document applies to STM32F3/STM32G4 microcontrollers based on the Arm® Cortex®-M core.

Note: Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

1.1 Purpose

The CCM SRAM is tightly coupled with the Arm® Cortex® core, to execute the code at the maximum system clock frequency without any wait-state penalty. This also brings a significant decrease of the critical task execution time, compared to code execution from flash memory.

The CCM SRAM is typically used for real-time and computation intensive routines, like the following:

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

When the code is located in CCM SRAM 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-bus and D-bus (see the figures below) and can thus perform at 1.25 DMIPS/MHz, with a deterministic performance of 90 DMIPS in STM32F3 and 213 DMIPS in STM32G4. This also guarantees a minimal latency if the interrupt service routines are placed in the CCM SRAM.

Figure 1. STM32F3 device system architecture

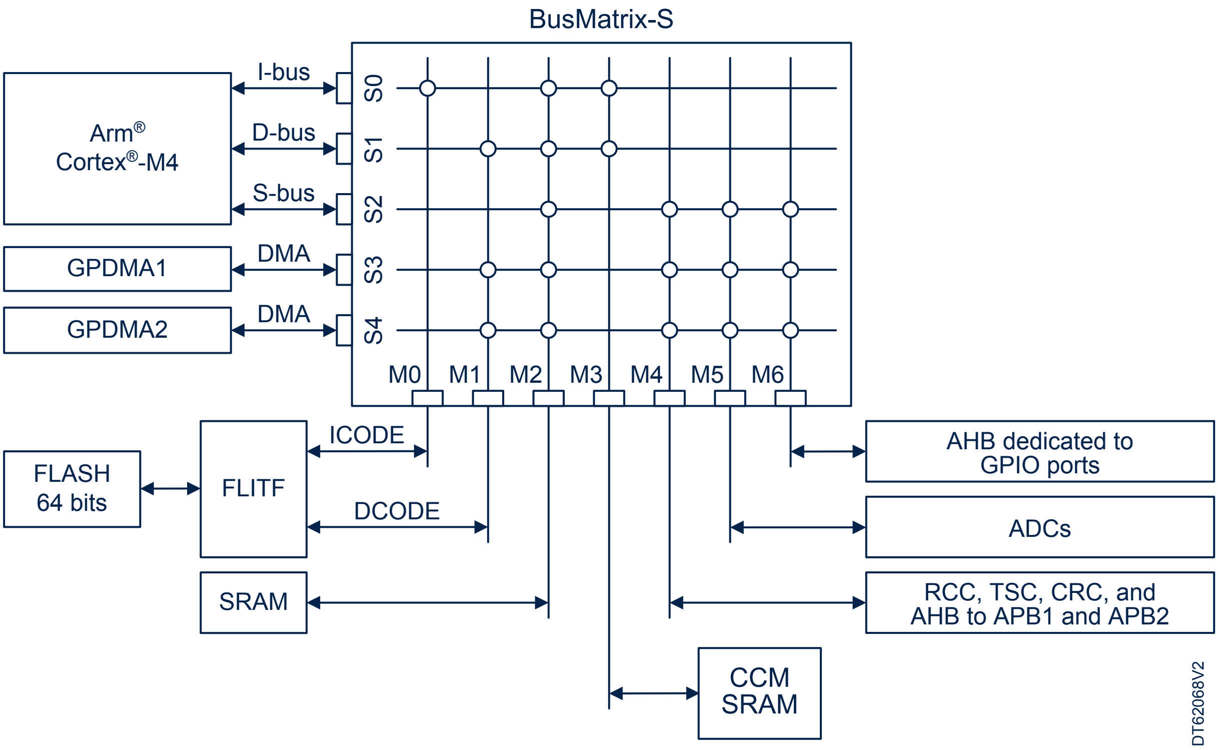
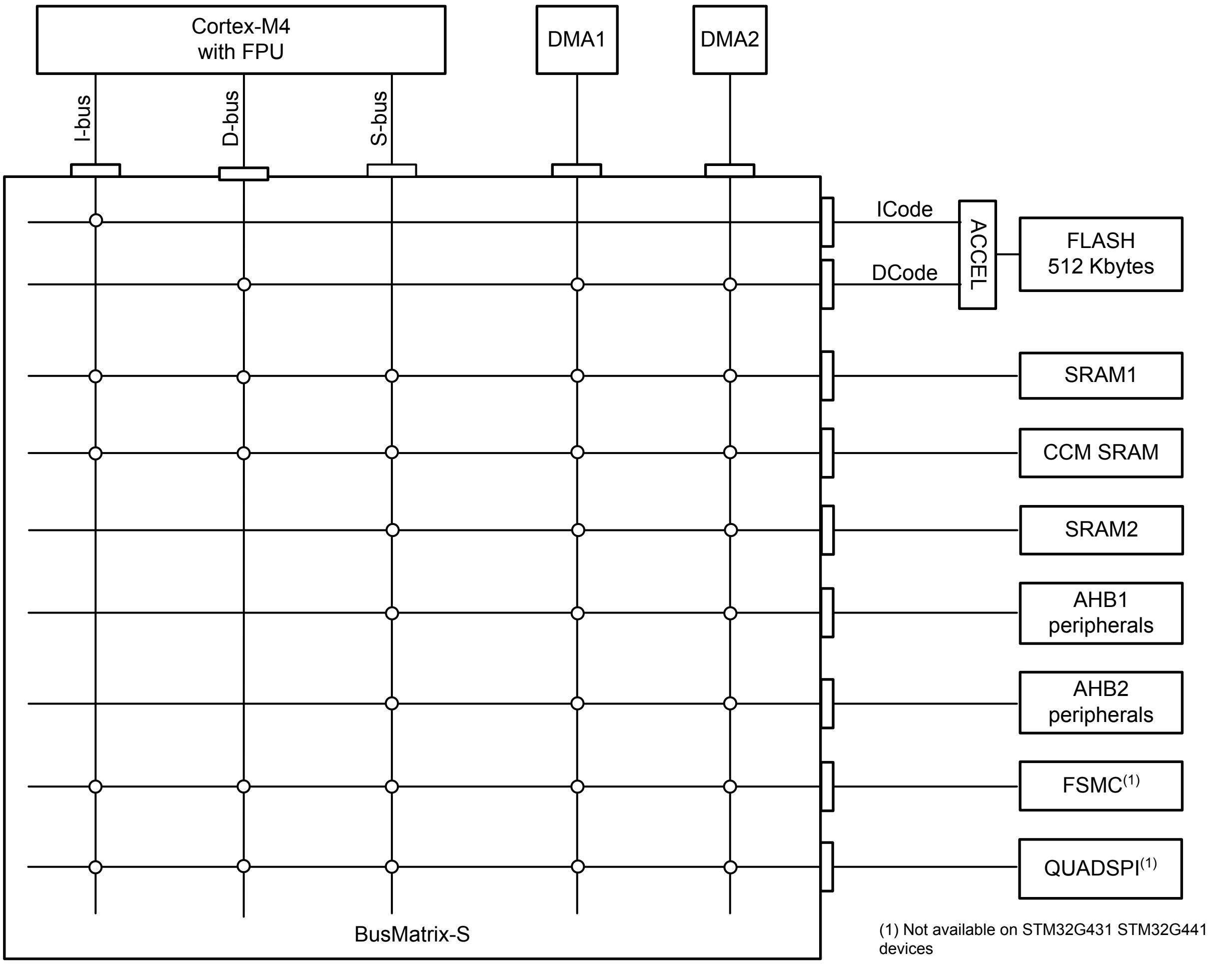


Figure 2. STM32G4 device system architecture



Example

A benchmark between the STM32F103 and STM32F303 microcontrollers using the STMicroelectronics MC library V3.4 shows that, in case of single motor control using three-shunt algorithm, the field-oriented control (FOC) total execution time for STM32F303 is 16.97 µs compared to 21.3 µs for the STM32F103 (see the note below), with FOC core and sensorless core loops running from CCM SRAM for STM32F303. This means that the STM32F303 is 20.33% faster than the STM32F103 thanks to the CCM SRAM.

Note: FOC routines are programmed in structured C, so the values provided above do not represent the fastest possible execution for both STM32F103 and STM32F303. In addition, the execution time is also a function of the compiler used and of its version.

When the CCM SRAM is not used for code, it can hold data like an extra SRAM memory. It is not recommended to place both code and data together in the CCM, since the Cortex® core must then fetch code and data from the same memory with the risk of collisions. The core is then in the Von Neumann configuration and its performance drops from 1.25 DMIPS/MHz to below 1 DMIPS/MHz.

1.2 CCM SRAM features

The table below summarizes the CCM SRAM features on various STM32 products. More details are provided in the next sections.

Table 2. CCM SRAM main features

Feature/ products	STM32F303xB/C STM32F358xx	STM32F303x6/8 STM32F334xx STM32F328xx	STM32F303xD/E STM32F398xx	STM32G47xx STM32G48xx	STM32G431x STM32G441x
Size (Kbytes)	8	4	16	32	10
Mapping	0x1000 0000			0x1000 0000 and can be aliased at 0x2001 8000	0x1000 0000 and can be aliased at 0x2000 5800
Parity check	Yes				
Write protection	Yes, with 1-Kbyte page granularity				
Read protection	No			Yes	
Erase	No			Yes	
DMA access	No			• No if mapped at 0x1000 0000 • Yes if mapped at 0x2001 8000	• No if mapped at 0x1000 0000 • Yes if mapped at 0x2000 5800

1.2.1 CCM SRAM mapping

The CCM SRAM is mapped at address 0x1000 0000.

On the STM32G4 devices, the CCM SRAM is aliased at the address following the end of SRAM2 offering a continuous address space with the SRAM1 and SRAM2.

1.2.2 CCM SRAM remapping

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

1.2.3 CCM SRAM write protection

The CCM SRAM can be protected against unwanted write operations with a page granularity of 1 Kbyte.

The write protection is enabled through the SYSCFG CCM SRAM protection register. This is a write-1-once mechanism: once the write protection is enabled on a given CCM SRAM 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 SRAM parity check

The implemented parity check 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.

1.2.5 CCM SRAM read protection (only on STM32G4 devices)

The CCM SRAM can also be readout-protected via the RDP option byte. When protected, the CCM SRAM cannot be read or written by the JTAG or serial-wire debug port, and when the boot in the system flash memory or the boot in the SRAM is selected.

The CCM SRAM is erased when the readout protection is changed from Level 1 to Level 0.

1.2.6 CCM SRAM erase (only on STM32G4 devices)

The CCM SRAM can be erased by software by setting the CCMER bit in the CCM SRAM system configuration control and status register.

The CCM SRAM can also be erased with the system reset depending on the option bit CCMSRAM_RST in the user option bytes.

2 Execute application code from CCM SRAM using the IAR Systems® IAR Embedded Workbench® toolchain

2.1 Execute a simple code from CCM SRAM (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.

IAR Embedded Workbench® provides the possibility to place one or more functions or a whole source file in CCM SRAM. This operation requires a new section to be defined in the linker file (`.icf`) to host the code to be placed in CCM SRAM. This section is copied to CCM SRAM at startup. The required steps are listed below:

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

The figure below shows an example of code implementing these operations.

Figure 3. IAR Embedded Workbench® linker update

```
/*###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__];

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__     { };

initialize by copy { readwrite, section .ccmram };

do not initialize  { section .noinit };

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

place in ROM region   { readonly };

place in CCMRAM region   {section .ccmram};

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

1 Defines the address zone for CCM SRAM.

2 'Initialize by copy ' tells the linker to copy this section at startup time.

3 Places .ccmram section at CCM SRAM defined above.

Note: This procedure is not valid for interrupt handlers.

2.1.1 Execute a source file from CCM SRAM

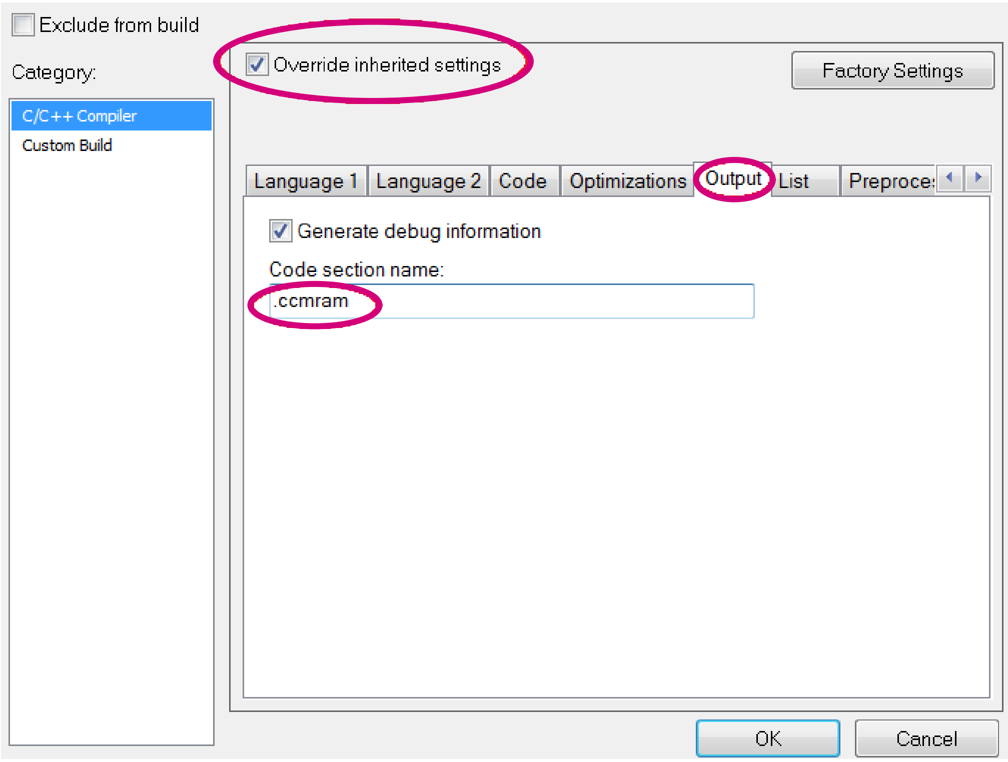
Execute a source file from CCM SRAM means that all functions declared in this file are executed from this memory area.

To place and execute a source file from CCM SRAM, use the IAR Embedded Workbench® file *Options* window as follows:

1. Add the section *.ccmram* (for example) in the linker file as defined in Section 2.1.
2. Right click on 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 the figure below).

Figure 4. IAR Embedded Workbench® file placement

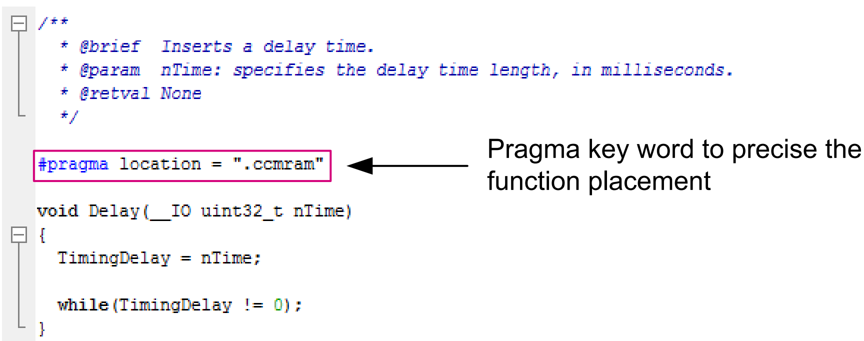


2.1.2 Executing one or more functions from CCM SRAM

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

1. Add the section `.ccmram` in the linker file as described in Section 2.1.
2. Using keyword `pragma location`, specify the function to be executed from CCM SRAM (see the figure below).

Figure 5. IAR Embedded Workbench® function placement



Note: To execute more than one function from CCM SRAM, the `pragma location` keyword must be placed above each function declaration.

2.2 Execute an interrupt handler from CCM SRAM

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

The IAR Embedded Workbench® linker protects the sections that are referenced from the startup code from being affected by an '`initialize by copy`' directive. The symbol `__vector_table` must not be used to allow copying interrupt handler sections via the '`initialize by copy`' directive. A second vector table must be created and placed in CCM SRAM.

The steps required to execute an interrupt handler from CCM SRAM are listed below and described in the next sub-sections:

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

2.2.1 Updating the linker file (.icf)

The following steps are needed to update the linker file `.icf` (see the figure below):

1. Define the address where the second vector table is located: 0x1000 0000.
2. Define the memory address area for the CCM SRAM 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 SRAM.
4. Tell the linker that the second vector table must be placed in the `.intvec_CCMRAM` section.
5. Indicate that the `.ccmram` code section must be placed in CCM SRAM.

Figure 6. IAR Embedded Workbench® 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

The following steps are needed to update the startup file:

1. Make a second vector table to be stored in CCM SRAM. For example, the `startup_stm32f30x.s` file must be modified by removing all entries except `sfe(CSTACK)` and `Reset_Handler` from the original vector table `__vector_table`.
2. Add a second vector table to be placed in CCM SRAM. It must contain all entries. As an example this table can be called `__vector_table_CCMRAM`. This vector table must be placed in the `.intvec_CCMRAM` section defined in the linker file.

Figure 7. IAR Embedded Workbench® startup file update for interrupt handler

```
;; Forward declaration of sections.
SECTION CSTACK:DATA:NOROOT(3)

SECTION .intvec:CODE:NOROOT(2)

EXTERN  __iar_program_start
EXTERN  SystemInit
PUBLIC  __vector_table

DATA
```

1
```
__vector_table
        DCD     sfe(CSTACK)
        DCD     Reset_Handler              ; Reset Handler
```

2
```
        SECTION .intvec_CCMRAM:CODE:ROOT(2)

        PUBLIC  __vector_table_CCMRAM

        DATA
__vector_table_CCMRAM
        DCD     sfe(CSTACK)
        DCD     Reset_Handler              ; Reset Handler
        DCD     NMI_Handler                ; NMI Handler
        DCD     HardFault_Handler          ; Hard Fault Handler
        DCD     MemManage_Handler          ; MPU Fault Handler
        DCD     BusFault_Handler           ; Bus Fault Handler
        DCD     UsageFault_Handler         ; Usage Fault Handler
        DCD     0                          ; Reserved
        DCD     0                          ; Reserved
        DCD     0                          ; Reserved
```

2.2.3 Place the interrupt handler in CCM SRAM

Place the interrupt handler to be executed in CCM SRAM as described in Executing one or more functions from CCM SRAM or the whole `stm32f_it.c` file as described in Execute a source file from CCM SRAM.

2.2.4 Remap the vector table to CCM SRAM

In the `SystemInit` function, remap the vector table to CCM SRAM by modifying the VTOR register as follows:

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

2.3 Execute a library (.a) from CCM SRAM

IAR Embedded Workbench® allows the execution of a library or a library module from CCM SRAM. The required steps are listed below:

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

Figure 8. CCM SRAM area definition

```
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__];

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__     { };
```

Defines the address zone for CCM SRAM.

2. Update the linker to copy at startup the library or the library module in CCM SRAM using the '`initialize by copy`' directive.

Figure 9. IAR Embedded Workbench® section initialization

```
initialize by copy { readwrite,ro object iar_cortexM4lf_math.a };

do not initialize  { section .noinit };
```

3. Indicate to the linker that the library must be placed in CCM SRAM.

Figure 10. IAR Embedded Workbench® library placement

```
place in ROM_region   { readonly };

place in CCMRAM_region   {section .text object iar_cortexM4lf_math.a};
```

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

The example in the figure below shows how to place `arm_abs_f32.o` (a module of `iar_cortexM4l_math.a` library) in CCM SRAM.

Figure 11. IAR Embedded Workbench® 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 SRAM using the Keil® MDK-ARM toolchain

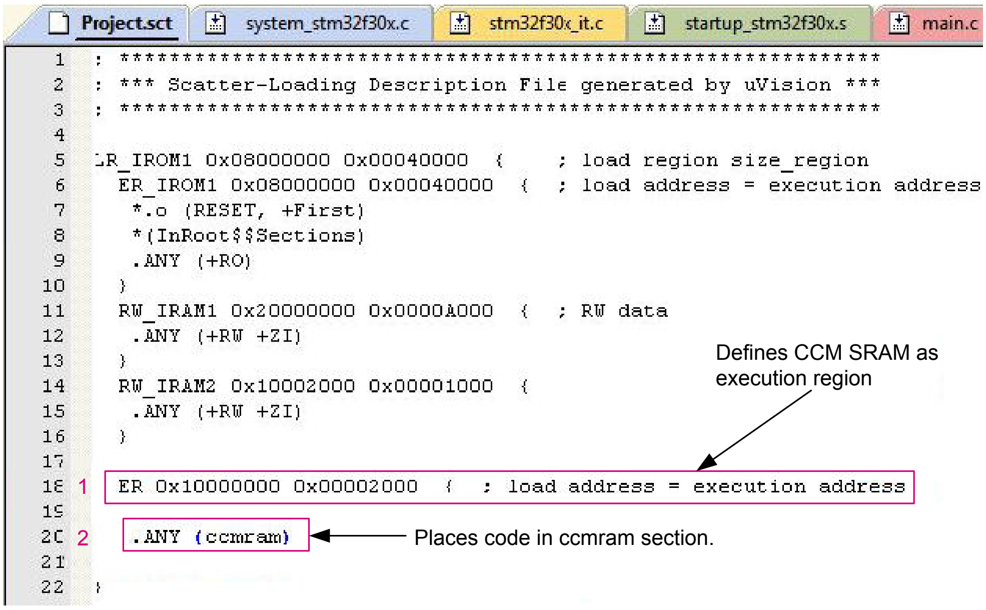
MDK-ARM features make it possible to execute simple functions or interrupt handlers from CCM SRAM. The following sections explain how to use these features to execute code from CCM SRAM.

3.1 Execute a function or an interrupt handler from CCM SRAM

The steps required to execute a function or an interrupt handler from CCM SRAM are listed below:

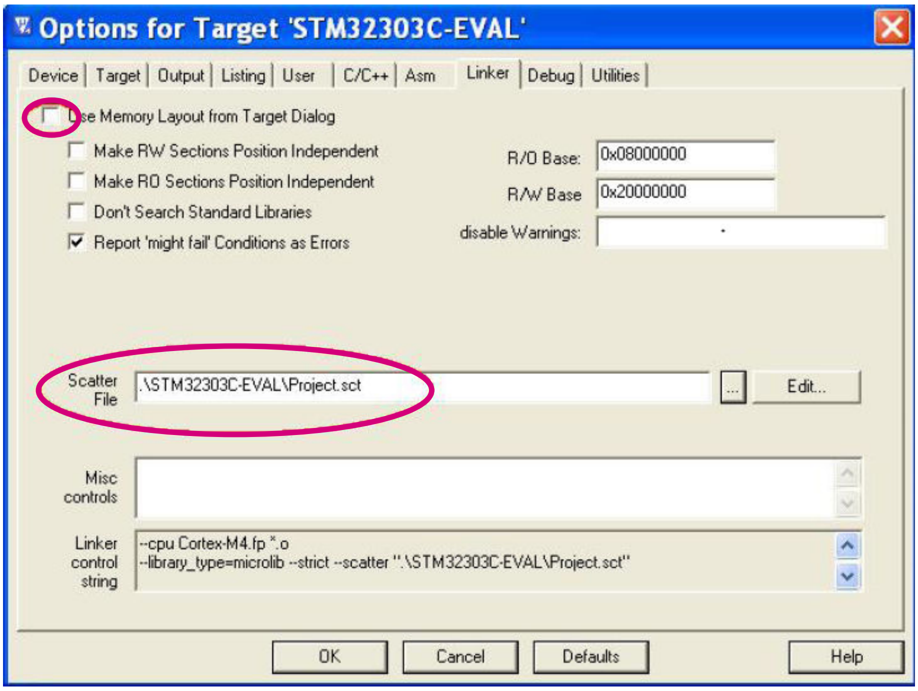
1. Define a new region (`ccmram`) in the scatter file by indicating the start and end addresses of CCM SRAM.
2. Indicate to the linker that the sections with the `ccmram` attribute must be placed in the CCM SRAM region.

Figure 12. MDK-ARM scatter file



3. Refer to the modified scatter file for the project options.

Figure 13. MDK-ARM *Options* menu



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

Figure 14. 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 SRAM, the attribute keyword must be placed above each function declaration.

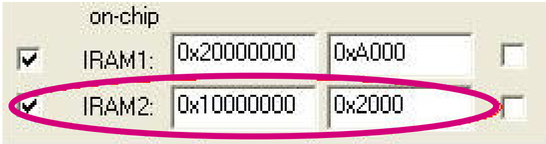
3.2 Execute a source file from CCM SRAM

Executing a source file from CCM SRAM means that all functions declared in this file are executed from the CCM SRAM region.

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

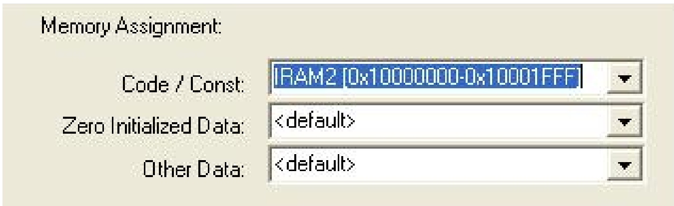
1. Define the CCM SRAM as a memory area in the project option window ([**Project**]>[**option**]>[**target**]).

Figure 15. MDK-ARM target memory



2. Right click on the file to place it in CCM SRAM and select [**Options**].
3. Select the CCM SRAM region in the [**Memory assignment**] menu.

Figure 16. MDK-ARM file placement



3.3 Execute a library or a library module from CCM SRAM

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

1. Define CCM SRAM as a memory area as shown in the figure below.
2. Right click on the library from the workspace and select [**Options**].
3. Place the complete library or a module from a library in CCM SRAM.

Figure 17. MDK-ARM library placement

4 Execute application code from CCM SRAM using STM32CubeIDE with GNU-based toolchain

STM32CubeIDE and GNU-based toolchains allow executing simple functions or interrupt handlers from CCM SRAM. The following sections explain how to use these features to execute code from CCM SRAM.

4.1 Execute a function or an interrupt handler from CCM SRAM

The steps required to execute a function or an interrupt handler from CCM SRAM are listed below:

1. Define a new region (`ccmram`) in the linker file (`.ld`) by defining the start address and the size of the CCM SRAM region.

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

/* Highest address of the user mode stack */
_estack = ORIGIN(RAM) + LENGTH(RAM);  /* end of "RAM" Ram type memory */

_Min_Heap_Size = 0x200;               /* required amount of heap  */
_Min_Stack_Size = 0x400;              /* required amount of stack */

/* Memories definition */
MEMORY
{
  RAM    (xrw)   : ORIGIN = 0x20000000, LENGTH = 112K
  FLASH  (rx)    : ORIGIN = 0x8000000,  LENGTH = 512K
  CCMRAM (xrw)   : ORIGIN = 0x10000000, LENGTH = 8K
}
```

2. Instruct the linker that code sections with the `ccmram` attribute must be placed in CCM SRAM. Insert the following code into the linker script. It is important to insert it before `.text` in the script.

```
/* The startup code into "FLASH" Rom type memory */
  .isr_vector :
  {
    . = ALIGN(4);
    KEEP(*(.isr_vector)) /* Startup code */
    . = ALIGN(4);
  } >FLASH

  /*--- New CCMRAM linker section definition ---*/
  _siccmram = LOADADDR(.ccmram);
  /* CCMRAM section */
  .ccmram :
  {
    . = ALIGN(4);
    _sccmram = .;         /* define a global symbols at ccmram start */
    *(.ccmram)
    *(.ccmram*)
    . = ALIGN(4);
    _eccmram = .;         /* define a global symbols at ccmram end */
  } >CCMRAM AT> FLASH
  /*--- End of CCMRAM linker section definition ---*/

  /* The program code and other data into "FLASH" Rom type memory */
  .text :
  {
    . = ALIGN(4);
    *(.text)              /* .text sections (code) */
    *(.text*)             /* .text* sections (code) */
    *(.glue_7)            /* glue arm to thumb code */
    *(.glue_7t)           /* glue thumb to arm code */
    *(.eh_frame)

    KEEP (*(.init))
    KEEP (*(.fini))

    . = ALIGN(4);
    _etext = .;           /* define a global symbols at end of code */
  } >FLASH
```

3. Modify the startup file to initialize data and code to place in CCM SRAM at startup time.

```
/* Copy the data segment initializers from flash to SRAM */
  ldr r0, =_sdata
  ldr r1, =_edata
  ldr r2, =_sidata
  movs r3, #0
  b  LoopCopyDataInit

CopyDataInit:
  ldr r4, [r2, r3]
  str r4, [r0, r3]
  adds r3, r3, #4

LoopCopyDataInit:
  adds r4, r0, r3
  cmp r4, r1
  bcc CopyDataInit

/* Copy from flash to CCMRAM */
  ldr r0, =_sccmram
  ldr r1, =_eccmram
  ldr r2, =_siccmram
  movs r3, #0
  b  LoopCopyCcmInit

CopyCcmInit:
  ldr r4, [r2, r3]
  str r4, [r0, r3]
  adds r3, r3, #4

LoopCopyCcmInit:
  adds r4, r0, r3
  cmp r4, r1
  bcc CopyCcmInit
/* End of copy to CCMRAM */

/* Zero fill the bss segment. */
  ldr r2, =_sbss
  ldr r4, =_ebss
  movs r3, #0
  b LoopFillZerobss

FillZerobss:
  str  r3, [r2]
  adds r2, r2, #4

LoopFillZerobss:
  cmp r2, r4
  bcc FillZerobss

/* Call the clock system initialization function.*/
    bl  SystemInit
/* Call static constructors */
    bl __libc_init_array
/* Call the application's entry point.*/
    bl  main

LoopForever:
    b  LoopForever
```

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

```
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 Execute a file from CCM SRAM

Executing a source file from CCM SRAM means that all functions declared in this file are executed from CCM SRAM.

To execute a file from CCM SRAM, follow the steps listed below:

1. Add the `.ccmram` section in the linker file as defined in Execute a function or an interrupt handler from CCM SRAM.
2. Place the file in CCM SRAM as shown below. The startup file also needs to be updated to copy code from the flash memory to CCM SRAM as described in Execute a function or an interrupt handler from CCM SRAM. This example shows how to execute file `myTestCCM.o` from CCM SRAM:

```
/*--- New CCMRAM linker section definition ---*/
_siccmram = LOADADDR(.ccmram);
/* CCMRAM section */
.ccmram :
{
  . = ALIGN(4);
  _sccmram = .;      /* define a global symbols at ccmram start */
  *(.ccmram)
  *(.ccmram*)
  *myTestCCM.o(.text .text*)
  . = ALIGN(4);
  _eccmram = .;      /* define a global symbols at ccmram end */
} >CCMRAM AT> FLASH
/*--- End of CCMRAM linker section definition ---*/
```

4.3 Execute a library from CCM SRAM

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

1. Add the `.ccmram` section in the linker file as defined in Execute a function or an interrupt handler from CCM SRAM.
2. Place the library in CCM SRAM as shown below. The startup file also needs to be updated to copy code from the flash memory to CCM SRAM as described in Execute a function or an interrupt handler from CCM SRAM. This example shows how to execute library `myLib.a` from CCM SRAM:

```
/*--- New CCMRAM linker section definition ---*/
_siccmram = LOADADDR(.ccmram);
/* CCMRAM section */
.ccmram :
{
  . = ALIGN(4);
  _sccmram = .;      /* define a global symbols at ccmram start */
  *(.ccmram)
  *(.ccmram*)
  *myTestCCM.o(.text .text*)
  *myLib.a:*(.text .text*)
  . = ALIGN(4);
  _eccmram = .;      /* define a global symbols at ccmram end */
} >CCMRAM AT> FLASH
/*--- End of CCMRAM linker section definition ---*/
```

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*.
16-Apr-2019	4	Updated: • Title of the document • *Introduction* • CCM RAM replaced by CCM SRAM in the whole document • *Figure 1. STM32F3 devices system architecture* Added: • *Figure 2. STM32G4 devices system architecture* • *Table 2. CCM SRAM main features* • *Section 1.2.5 CCM SRAM read protection (only on STM32G4 devices)* • *Section 1.2.6 CCM SRAM erase (only on STM32G4 devices)* Removed *Table 2. CCM RAM organization*.
08-Feb-2021	5	Updated: • Title of the document • *Section 4* description and code examples to authorize code execution in CCM SRAM when using a GNU-based toolchain such as STM32CubeIDE
16-Apr-2025	6	Updated: • Device reference in Table 2. CCM SRAM main features • Figure 1. STM32F3 device system architecture Removed STM32F328K8, STM32F328R8, STM32F398RE, and STM32F398ZE from Table 1. Applicable products.

Contents

List of tables

List of figures