AN4296
应用笔记

以 IAR EWARM、 Keil MDK-ARM 和基于 GNU 的工具链来使用 STM32F303/358xx CCM RAM 的概述与技巧

前言

本应用笔记的目的在于介绍适用于 STM32F303xB/xC 和 STM32F358xC 微控制器的内核耦合存储区（CCM）RAM，并描述利用不同的工具链从该存储区域执行部分应用程序代码所需的操作。

本应用笔记分为四个部分：第一部分是 STM32F3 CCM RAM 的概述，第二部分描述了利用以下工具链执行部分应用程序代码的步骤：

- IAR EWARM
- KEIL MDK-ARM™
- RIDE 和 Atollic 基于 GNU 的工具链

本文档通篇所述的步骤均适用于其它 RAM 区域，比如某些 F4 设备中的 CCM 数据 RAM 或外部 SRAM。

关于微控制器嵌入式 CCM RAM 的列表，请参考*表 1*。

表 1. 适用产品

产品系列	产品编号或产品类别
微控制器	STM32F303xB， STM32F303xC， STM32F358xC

目录

表格索引

图片索引

1 STM32F303xB/C 与 STM32F358xC CCM RAM 的概述

1.1 目的

STM32F303xB/C 和 STM32F358xC CCM RAM 与 Cortex™ 内核紧密结合。这主要为了以最高的系统时钟频率（72 MHz）来执行代码，同时避免出现等待状态。因此，与闪存的代码执行情况相比，该器件大大减少了关键任务的执行时间。

CCM RAM 一般用于实时的计算密集型程序，比如：

- 数字电源转换的控制环 （开关电源，照明）
- 矢量 3 相电机控制
- 实时 DSP 任务

当代码位于 CCM RAM 且数据保存在普通 SRAM 当中时，Cortex-M4 内核便处于最优的哈佛配置。专用的零等待状态存储器与所有 I 总线及 D 总线连接（请参考*图 1: STM32F303xB/xC 与 STM32F358xC 系统架构*），由此可实现 1.25DMIPS/MHz 的执行速度，频率可达 72 MHz，最终性能可达 90 DMIPS。如果中断服务程序处于 CCM RAM 当中，这还确保了最短的延迟时间。

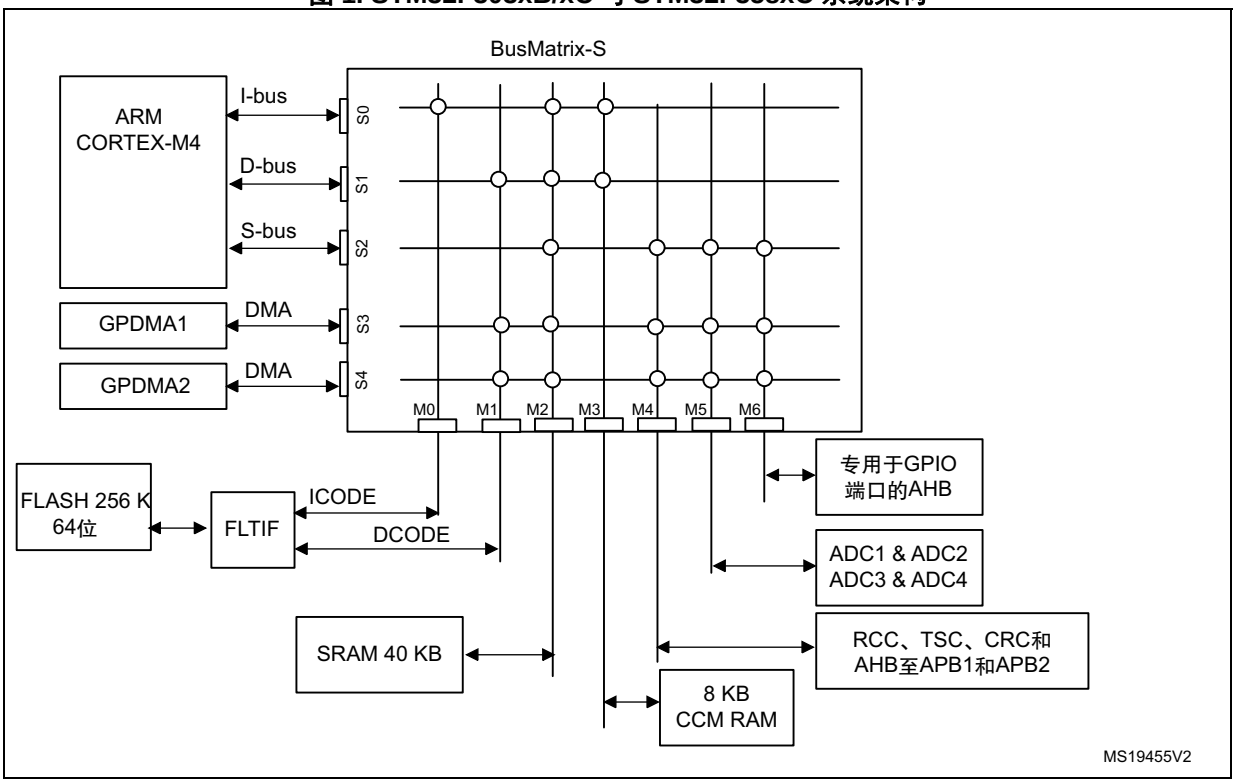
示例

STM32F103xx 与 STM32F303xx 微控制器之间的基准（使用意法半导体电机库 V3.4）表明，对于采用 3 电阻算法的单电机控制， STM32F303xx 的 FOC 总执行时间为 16.97 µs，而 STMF103xx 的总执行时间为 21.3 µs （参考以下注释）； STM32F303xx 中的 FOC 算法和无传感器算法环均从 CCM RAM 上运行。这意味着，得益于 CCM RAM， STM32F303xx 比 STM32F103xx 快 20.33%。

注： *FOC 程序以结构化的 C 语言编写，因此以上数值并不代表 STM32F103xx 和 STM32F303xx 的最快执行时间。此外，执行时间还受到所使用的编译器及其版本的影响。*

如果 CCM RAM 没有用来存储代码，可以像额外 SRAM 内存那样保存数据。然而，它无法通过 DMA 被访问。不建议将代码和数据一起放在 CCM 中，因为这样 Cortex 内核将不得不从同一个内存中获取代码和数据，有可能发生冲突。这样，内核会变为冯 · 诺依曼配置，性能从 1.25DMIPS/MHz 下降至 1DMIPS/MHz 以下。

图 1. STM32F303xB/xC 与 STM32F358xC 系统架构



1.2 STM32F303xB/C 与 STM32F358xC CCM RAM 的特性

1.2.1 CCM RAM 映射

CCM RAM 适用于 STM32F303xB/C 和 STM32F358xC 器件，起始地址为 0x1000 0000。

1.2.2 CCM RAM 重映射

与普通 SRAM 不同， CCM RAM 无法在地址 0x0000 0000 重映射。

1.2.3 CCM RAM 写保护

CCM RAM 可针对不必要的写操作获得保护，页面大小为 1KB。关于 CCM RAM 组织的说明，请参见*表 2*。

表 2. CCM RAM 组织

页面编号	起始地址	结束地址
第 0 页	0x1000 0000	0x1000 03FF
第 1 页	0x1000 0400	0x1000 07FF
第 2 页	0x1000 0800	0x1000 0BFF
第 3 页	0x1000 0C00	0x1000 0FFF

写保护通过 SYSCFG CCM SRAM(SYSCFG_RCR) 保护寄存器来实现。这是一种一次性写 “1” 操作机制，这意味着在给定 CCM RAM 页中将对应位置 “1” 而启用写保护之后，只可通过系统重置的方式将该位清零。有关详细信息，请参见产品参考手册。

1.2.4 CCM RAM 奇偶校验

STM32F303xB/C 和 STM32F358xC 微控制器会执行奇偶校验。该功能是默认禁用的，当需要使用时，可通过选项位（SRAM_PE 位）来启用。将该选项位清零后，会对 SRAM 的前 16 KB 以及 CCM RAM 的前 8 KB 进行奇偶校验。

2 从 CCM RAM 执行应用程序代码（使用 IAR EWARM 工具链）

2.1 从 CCM RAM 执行简单代码 （中断处理程序除外）

简单代码可由一个或者多个非引用自中断处理程序的函数所组成。如果代码引用自中断处理程序，则遵循*第 2.2 章节: 从 CCM RAM 执行中断处理程序*所述的步骤。

EWARM 可用于将一个 / 多个函数或整个源文件放入 CCM RAM。

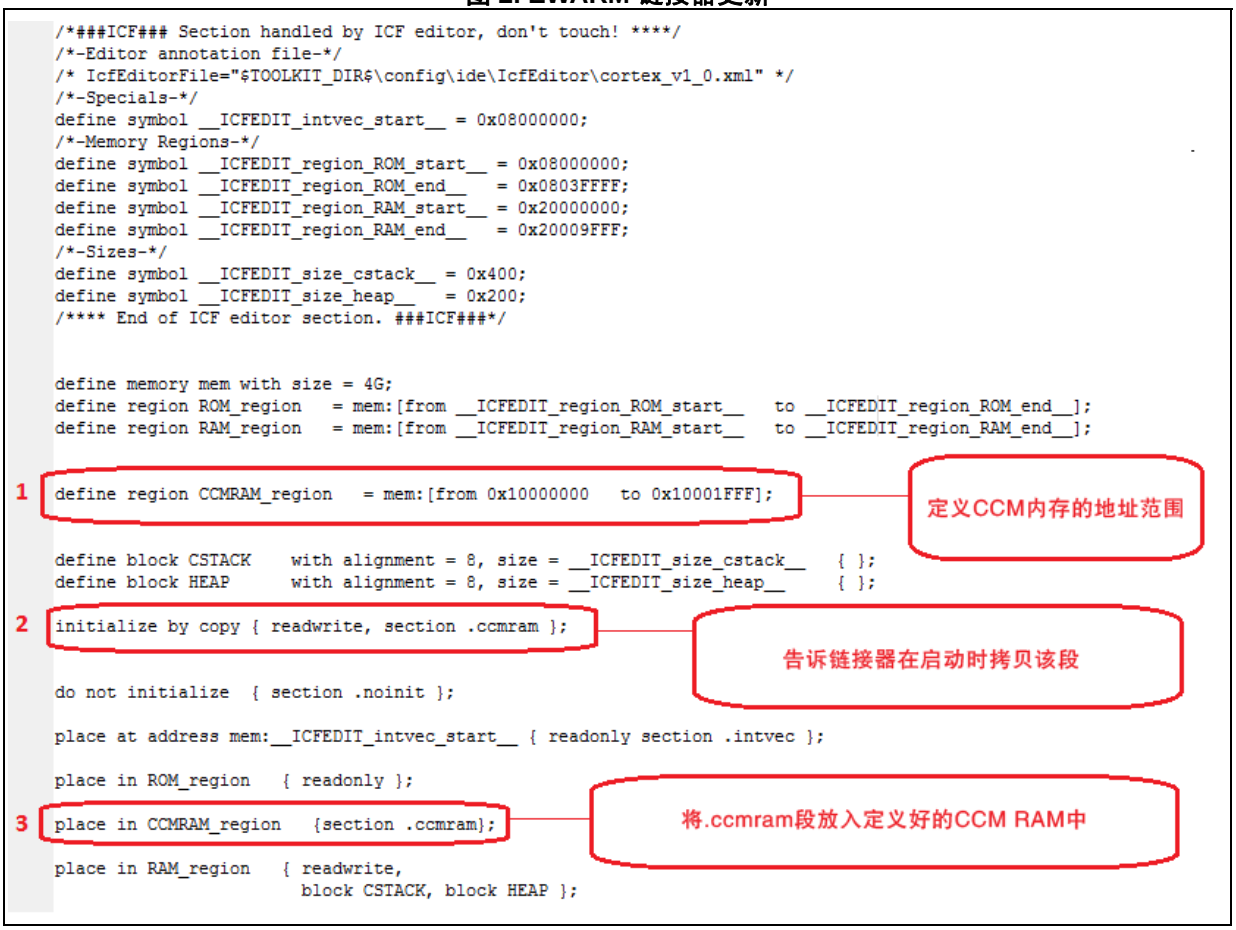
该操作需要在链接器文件 （.icf）中定义一个新的区段，用于保存需存入 CCM RAM 中的代码。该区段会在启动时复制到 CCM RAM。以下是所需的步骤：

1. 通过设定起始地址和结束地址，定义 CCM RAM 的地址区。
2. 通知链接器在启动时将名为 .ccmram 的区段从闪存复制到 CCM RAM，以避免使用闪存装载软件。
3. 向链接器说明，代码区段 .ccmram 应置入 CCM RAM 区域。

关于执行这些操作的代码示例，请参考*图 2: EWARM 链接器更新*。

注： *该方法不适用于中断处理程序。*

图 2. EWARM 链接器更新



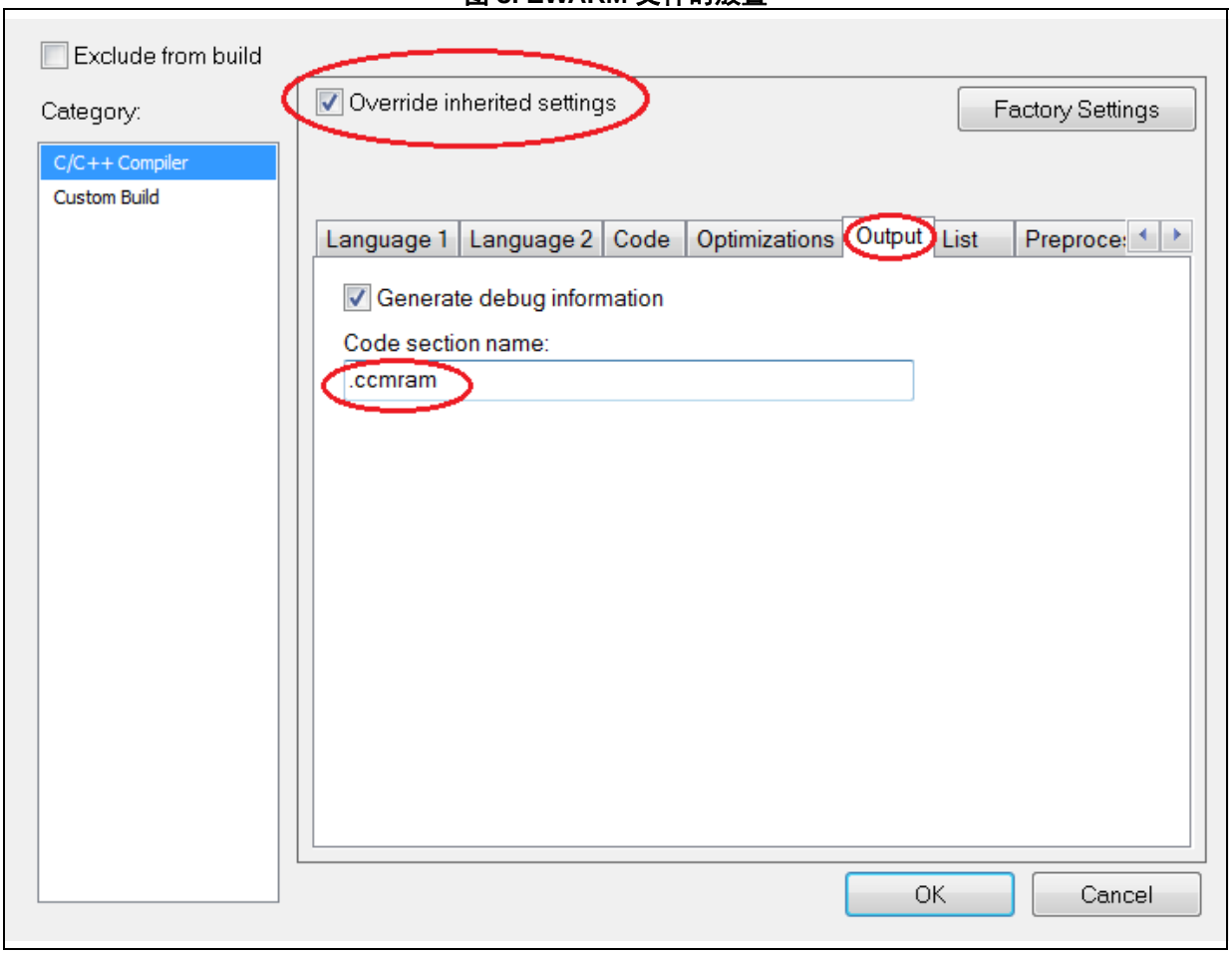
2.1.1 从 CCM RAM 执行源文件

从 CCM RAM 执行源文件意味着该文件所声明的全部函数均从该内存区域中执行。

如需放置与执行来自 CCM RAM 的源文件，请使用 EWARM 文件**选项**窗口：

1. 在如*第 2.1 章节*所定义的链接器文件中添加 （示例） .ccmram 区段。
2. 右键点击项目窗口中的文件名。
3. 在出现的菜单上选择**选项**。
4. 勾选窗口中的**覆盖继承设定**
5. 选择**输出**选项卡，然后在**代码段名称**字段中输入链接器文件中已设定的区段名称 （本例中为 “.ccmram”）（请参见*图 3: EWARM 文件的放置*）。

图 3. EWARM 文件的放置

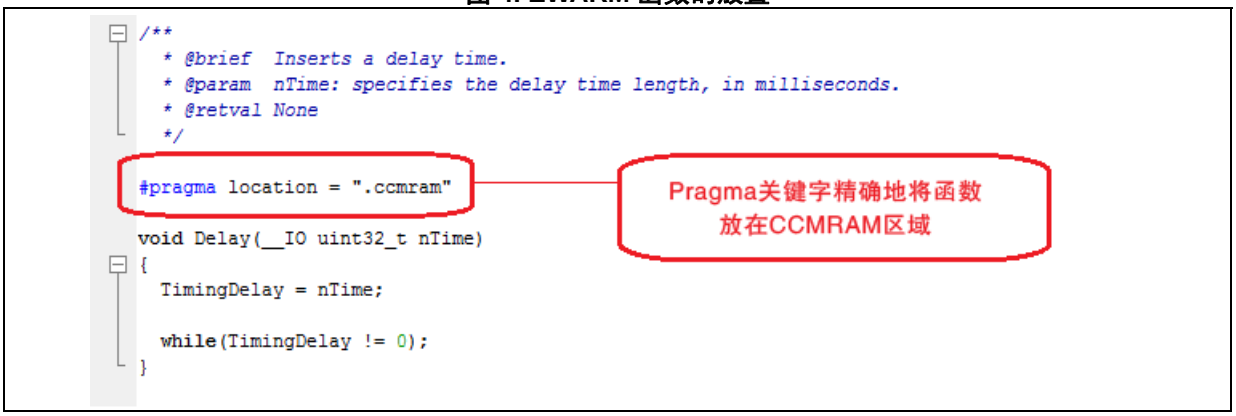


2.1.2 从 CCM RAM 执行一个或多个函数

从 CCM RAM 执行函数的步骤如下所述：

1. 在链接器文件中添加如*第 2.1 章节*所述的 .ccmram 区段。
2. 使用关键字**程序位置**，指定需从 CCM RAM 执行的函数 （参见*图 4: EWARM 函数的放置*）。

图 4. EWARM 函数的放置



注： *为从 CCM RAM 执行多个函数，应在各个函数声明上方设置* ***pragma*** *位置关键字。*

2.2 从 CCM RAM 执行中断处理程序

向量表以名为 *__vector_table* 的数组来实现，且在启动代码中引用。

EWARM 链接器保护从启动代码中引用的区段，以免受到 “ 复制初始化 ” 指令的影响。因此，不应使用 *__vector_table* 符号通过 “ 复制初始化 ” 指令来复制中断处理程序区段。

为此，应制作第二个向量表，并将其放入 CCM RAM。

从 CCM RAM 执行中断处理程序的步骤如下所述：

1. 更新链接器文件 (.icf)。
2. 更新启动文件。
3. 将中断处理程序放入 CCM RAM。
4. 将向量表重新映射至 CCM RAM。

2.2.1 更新链接器文件 (.icf)

链接器文件更新步骤：

1. 定义第二个向量表所在的地址：0x1000 0000。
2. 通过指定起始地址和结束地址，定义 CCM RAM 的内存地址区。
3. 向链接器指明，在启动时将名为 .ccmram 的区段和第二个向量表区段 “.intvec_CCMRAM” 从闪存复制到 CCM RAM，以避免使用闪存装载软件。
4. 向链接器指明，第二个向量表应放在 intvec_CCMRAM 区段内。
5. 指明 .ccmram 代码段将放在 CCM RAM 中。

图 5. 适用于中断处理程序的 EWARM 链接器更新

```
/* 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###*/

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 };

place in ROM_region   { readonly };
```

2.2.2 更新启动文件

启动文件更新步骤：

1. 制作将保存在 CCM RAM 中的第二个向量表。应删除原始向量表 “__vector_table” 中除 sfe（CSTACK）和 Reset_Handler 之外的所有条目，以修改 startup_stm32f30x.s 文件。
2. 添加需放入 CCM RAM 中的第二个向量表。表中应包含所有条目。例如，可以调用 “__vector_table_CCMRAM”。该向量表必须放在链接器文件所定义的 intvec_CCMRAM 区段中。

图 6. 适用于中断处理程序的 EWARM 启动文件更新

```
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 将中断处理程序放入 CCM RAM

将*第 2.1.2 章节*所述的需执行的中断处理程序或*第 2.1.1 章节*中所述的整个 stm32f_it.c 文件放入 CCM RAM。

2.2.4 将向量表重新映射至 CCM RAM

在 SystemInit 函数中，按以下方式修改 VTOR 寄存器，将向量表重新映射至 CCM RAM：

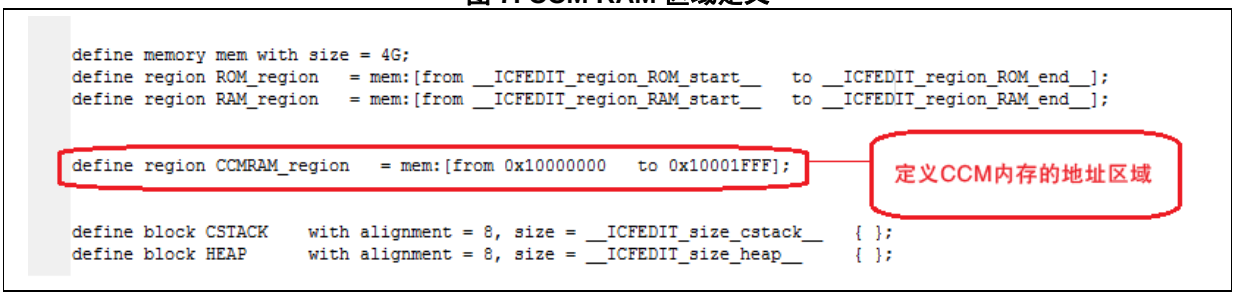
```
SCB->VTOR = 0x10000000 | VECT_TAB_OFFSET;
```

2.3 从 CCM RAM 执行库 （.a）

EWARM 允许从 CCM RAM 执行库或库模块。以下是需执行的动作：

1. 通过指定起始地址和结束地址，定义对应于 CCM RAM 的内存地址区。

图 7. CCM RAM 区域定义



2. 更新链接器，在启动时利用 “ 复制初始化 ” 指令将库或库模块复制至 CCM RAM。

示例：

图 8. EWARM 区段初始化

```
initialize by copy { readwrite,ro object iar_cortexM4lf_math.a };
do not initialize  { section .noinit };
```

3. 向链接器指明，库应放入 CCM RAM：

图 9. EWARM 库的放置

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

如需从 CCM RAM 执行库模块，请使用库模块名称，按照步骤 1、 2、 3 进行操作。

下述例子显示了如何将 arm_abs_f32.o （iar_cortexM4l_math.a 库中的一个模块）放入 CCD RAM：

图 10. EWARM 库模块的放置

```
/*###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 从 CCM RAM 执行应用程序代码（使用 KEIL MDK-ARM 工具链）

MDK-ARM 的功能可从 CCM RAM 执行简单函数或中断处理程序。以下内容解释了如何使用这些功能从 CCM RAM 执行代码。

3.1 从 CCM RAM 执行函数或中断处理程序

从 CCM RAM 执行函数或中断处理程序的步骤如下所述：

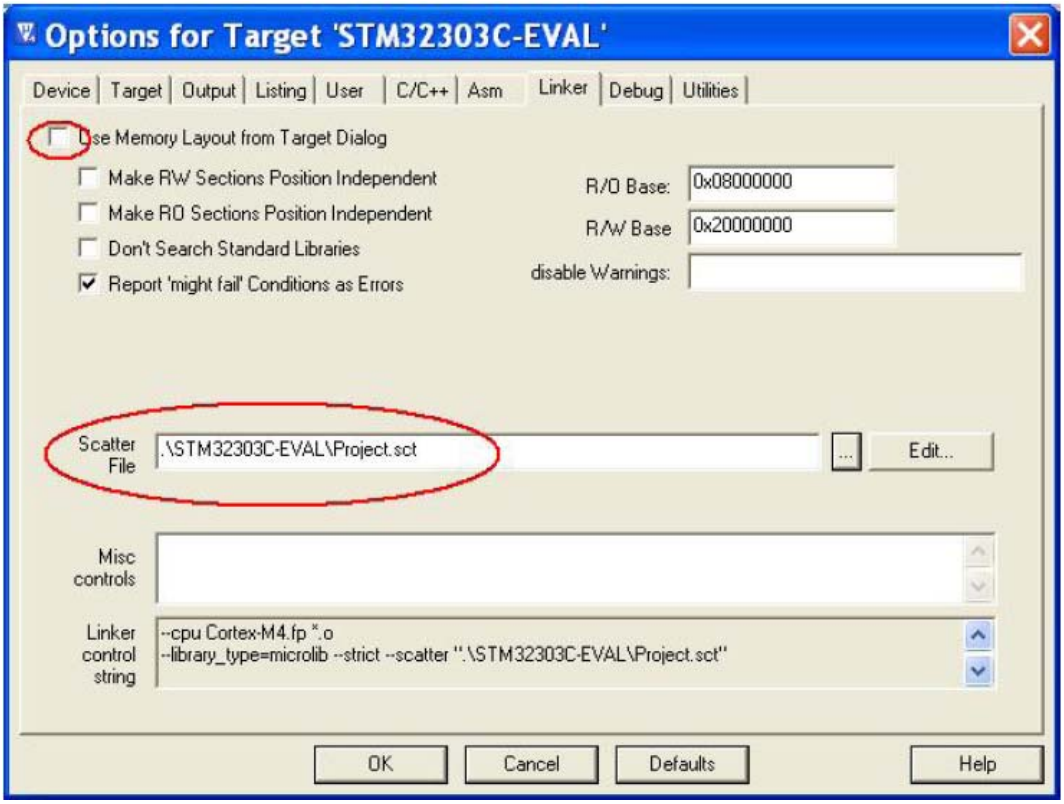
1. 通过指定 CCM RAM 区域的起始和结束地址，在分散文件中定义新的区域 （ccmram）。
2. 向链接器指明，带有 ccmram 属性的区段必须放在 CCM RAM 区域中。

图 11. MDK-ARM 分散文件

```
/*###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. 关于项目选项，请参考已修改的分散文件 （参见*图 11*）：

图 12. MDK-ARM 选项菜单



4. 将需要从 CCM RAM 执行的代码放入前文定义的 ccmram 区段。这一步通过在函数声明上方添加属性关键字来完成。

图 13. MDK-ARM 函数的放置

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

注：　为从 CCM RAM 执行多个函数，应在各个函数声明上方设置属性关键字：

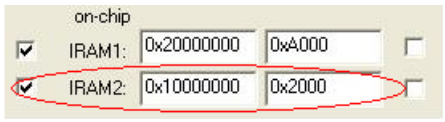
3.2 从 CCM RAM 执行源文件

从 CCM RAM 执行源文件意味着该文件所声明的全部函数均从 CCM RAM 区域执行。

请按照以下步骤从 CCM RAM 执行文件：

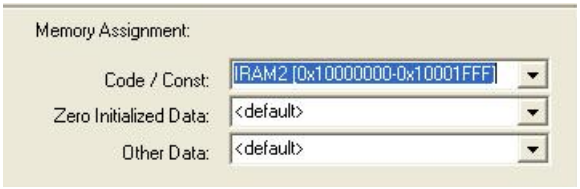
1. 在**项目选项**窗口中设定 CCM RAM 作为内存区域 （项目 > 选项 > 目标）：

图 14. MDK-ARM 目标内存



2. 右键点击文件，将其放入 CCM RAM，然后选择**选项**
3. 选择**内存分配**菜单中的 CCM RAM 区域：

图 15. MDK-ARM 文件的放置



3.3 从 CCM RAM 执行库或库模块

请按照以下步骤从 CCM RAM 执行库或库模块：

1. 设定 CCM RAM 作为内存区，如*图 16: MDK-ARM 库的放置*所示。
2. 右键点击工作空间中的库，然后选择**选项**。
3. 既可将完整的库放入 CCM RAM，也可仅放入库中的一个模块。

图 16. MDK-ARM 库的放置

4 从 CCM RAM 执行应用程序代码（使用基于 GNU 的工具链）

基于 GNU 的工具链能够从 CCM RAM 执行简单函数或中断处理程序。以下内容解释了如何使用这些功能从 CCM RAM 执行代码。

4.1 从 CCM RAM 执行函数或中断处理程序

从 CCM RAM 执行函数或中断处理程序的步骤如下所述：

1. 通过设定 CCM RAM 区域的起始地址与大小，在链接器文件 （.ld）中定义新的区域（ccmram）（参见*图 17: GNU 链接器更新*）

图 17. GNU 链接器更新

```
/* 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
}
```

定义CCMRAM区域地址

2. 向链接器指明，带有 ccmram 属性的代码段必须放在 CCM RAM 中 （参见*图 18: GNU 链接器区段定义*）。

图 18. GNU 链接器区段定义

```
  *(.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. 修改启动文件，从而在启动时对数据进行初始化并放入 CCM RAM （参见红色的代码行）：

```
.section  .text.Reset_Handler
.weak  Reset_Handler
.type  Reset_Handler, %function
Reset_Handler:
/* 将数据段初始化器从闪存复制至 SRAM 和 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
/* 将 bss 段清零。 */
FillZerobss:
movs  r3, #0
str  r3, [r2], #4
LoopFillZerobss:
ldr  r3, = _ebss
cmp  r2, r3
bcc  FillZerobss
/* 调用时钟系统初始化函数。 */
bl  SystemInit
/* 调用应用程序的入口点。 */
bl  main
bx  lr
```

4. 通过在函数原型中添加属性关键字，将需要从 CCM RAM 执行的代码放入 .ccmram 区段：

图 19. GNU 函数的放置

```
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 从 CCM RAM 执行文件

从 CCM RAM 执行源文件意味着该文件所声明的全部函数均从 CCM RAM 执行。

如需从 CCM RAM 执行文件，请遵循以下步骤：

1. 在如*第 4.1 章节*所定义的链接器文件中添加 .ccmram 区段。
2. 将文件放入 CCM RAM，如下所示：

图 20. GNU 文件的放置

```
_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 从 CCM RAM 执行库

请按照以下步骤从 CCM RAM 执行库：

1. 在如*第 4.1 章节*所定义的链接器文件中添加 .ccmram 区段。
2. 将库放入 CCM RAM，如下所示：

图 21. GNU 库的放置

```
/* 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 修订历史

表 3. 文档修订历史

日期	修订	变更
2013 年 7 月 23 日	1	初始版本。
2014 年 3 月 25 日	2	将 STM32F313xC 改为 STM32F358xC。 已重写*第 1 章节: STM32F303xB/C 与 STM32F358xC CCM RAM 的概述*。

请仔细阅读：

中文翻译仅为方便阅读之目的。该翻译也许不是对本文档最新版本的翻译，如有任何不同，以最新版本的英文原版文档为准。

本文档中信息的提供仅与 ST 产品有关。意法半导体公司及其子公司 （“ST”）保留随时对本文档及本文所述产品与服务进行变更、更正、修改或改进的权利，恕不另行通知。

所有 ST 产品均根据 ST 的销售条款出售。

买方自行负责对本文所述 ST 产品和服务的选择和使用，ST 概不承担与选择或使用本文所述 ST 产品和服务相关的任何责任。

无论之前是否有过任何形式的表示，本文档不以任何方式对任何知识产权进行任何明示或默示的授权或许可。如果本文档任何部分涉及任何第三方产品或服务，不应被视为 ST 授权使用此类第三方产品或服务，或许可其中的任何知识产权，或者被视为涉及以任何方式使用任何此类第三方产品或服务或其中任何知识产权的保证。

除非在 ST 的销售条款中另有说明，否则，ST 对 ST 产品的使用和 / 或销售不做任何明示或默示的保证，包括但不限于有关适销性、适合特定用途 （及其依据任何司法管辖区的法律的对应情况），或侵犯任何专利、版权或其他知识产权的默示保证。

意法半导体的产品不得应用于武器。此外，意法半导体产品也不是为下列用途而设计并不得应用于下列用途：（A）对安全性有特别要求的应用，例如，生命支持、主动植入设备或对产品功能安全有要求的系统；（B）航空应用；（C）汽车应用或汽车环境，且 / 或 （D）航天应用或航天环境。如果意法半导体产品不是为前述应用设计的，而采购商擅自将其用于前述应用，即使采购商向意法半导体发出了书面通知，采购商仍将独自承担因此而导致的任何风险，意法半导体的产品设计规格明确指定的汽车、汽车安全或医疗工业领域专用产品除外。根据相关政府主管部门的规定，ESCC、QML 或 JAN 正式认证产品适用于航天应用。

经销的 ST 产品如有不同于本文档中提出的声明和 / 或技术特点的规定，将立即导致 ST 针对本文所述 ST 产品或服务授予的任何保证失效，并且不应以任何形式造成或扩大 ST 的任何责任。

意法半导体集团公司

澳大利亚 - 比利时 - 巴西 - 加拿大 - 中国 - 捷克共和国 - 芬兰 - 法国 - 德国 - 中国香港 - 印度 - 以色列 - 意大利 - 日本 - 马来西亚 - 马耳他 - 摩洛哥 - 菲律宾 - 新加坡 - 西班牙 - 瑞典 - 瑞士 - 英国 - 美国

www.st.com