MPLAB® Harmony 之基础篇（35）-- 如何从 RAM 执行代码

Microchip Technology Inc.
MCU32 产品部

一、 简介

本文将介绍如何在 MPLAB X IDE 开发环境下，让 MPLAB Harmony 3 代码从 RAM 中执行，适用于 Microchip 32 位 SAM®微控制器。文中的代码示例将以 SAM D21J18A 举例，其它基于 ARM Cortex-M 内核的微控制器方法类同。在附录中，还会介绍如何在 Keil 中让代码从 RAM 中执行。

一般 Microchip 32 位 SAM®微控制器代码都存放在 Flash 中，直接从 Flash 运行。但也有些场景，希望部分代码从 RAM 中执行。比如，把 Flash 的一部分空间作为仿真 EEPROM，当对 Flash 进行擦除或者写入时，应用程序此时就无法从 Flash 中执行代码。为了解决这个问题，许多 SAM®系列微控制器都增加了一小块独立的 RWW Flash（read while write Flash）作为仿真 EEPROM 的存储空间。对于没有 RWW Flash 支持的微控制器，如果需要在写 Flash 时不暂停程序执行，保证能及时响应中断，则可以把中断向量表和相关函数代码放到 RAM 中执行。

二、 软件平台

MPLAB® X IDE: v5.45
- MHC 3 Launcher: v3.6.2

XC32: v2.50

Harmony 3:
- mhc: v3.6.4
- csp: v3.8.2
- dev_packs: v3.8.0

注：以上的软件工具版本会持续更新，建议您使用新版本。当使用新版本时，如果遇到问题，建议您尝试使用与上面相同的版本。

在阅读下面详细步骤之前，请参考“MPLAB® Harmony 3 之基础篇（01）-- Harmony 3 开发环境搭建”文档，下载 Harmony 3 的代码仓库。

三、 如何在 MPLAB X 中从 RAM 执行代码

(一) 执行的函数放在 RAM 中 – MPLAB X

让一个函数从 RAM 中执行，需要把它定义为 RAM Function 函数。在 MPLAB X 中可以先定义“__ramfunc”关键字，并把它放在函数前，例如：

```
#define __ramfunc       __attribute__(( long_call, section(".ramfunc"), noinline ))


__ramfunc void foo_function (void)
{
      // Your code
}
```

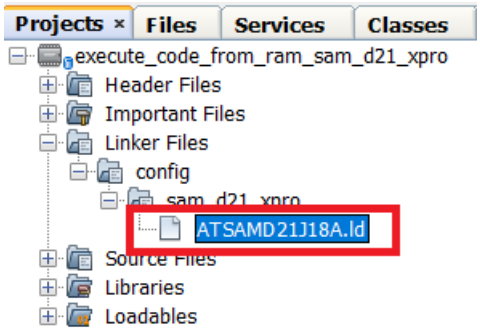
(二) 执行的中断放在 RAM 中 – MPLAB X

如果需要中断处理函数从 RAM 运行，则需要做下面的修改：

- 将中断向量表放置在 RAM 中，并正确配置 ARM 的 VTOR 寄存器
- 相应的中断处理函数也放在 RAM 中

把中断向量表从 FLASH 中重定位到 RAM，需要编辑项目使用的链接文件，以及修改在 MCU 进入 main()函数之前执行的启动代码。

首先修改链接文件。在 MPLAB X 中打开工程，一般可以从这里找到它：



文件路径：

<your project path>\firmware\src\config\sam_d21_xpro\ATSAMD21J18A.ld

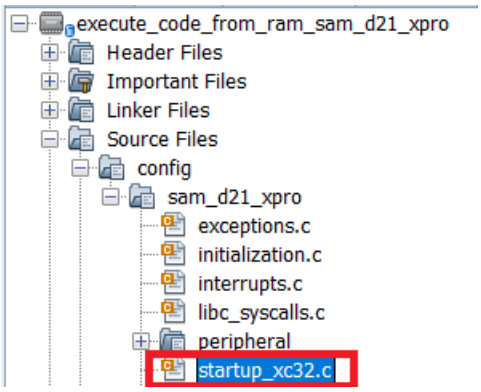
打开“ATSAMD21J18A.ld”并添加如下蓝色内容：

```
    . = ALIGN(4);
    _etext = .;

    /*
     * RAM Vector table section
     * __pic32c_data_initialization() will initialize the vector table.
     */
    .relocate_vector : AT (_etext)
    {
        . = ALIGN(4);
        _sfixed_ram_vect = .;
        KEEP(*(.ram_vectors .ram_vectors.*));
    } > DATA_REGION
```

这里我们创建了一个.relocate_vector 段用来存放 RAM vector table：

- _sfixed_ram_vect：用来记录 RAM Vector 的起始地址，用于存放中断向量表（Vector table）
- KEEP(*(.ram_vectors .ram_vectors.*))：告诉编译器保留程序中的.ram_vectors 数据段，不被优化掉

然后修改启动代码。在 MPLAB X 工程里，一般可以从这里找到它：

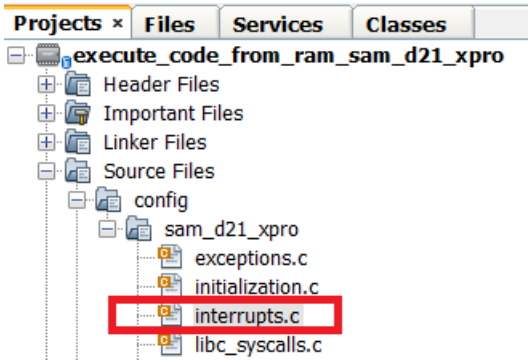


文件路径：
<your project path>\firmware\src\config\sam_d21_xpro\startup_xc32.c

打开“startup_xc32.c”并添加如下蓝色内容：

```
/* Initialize segments */
extern uint32_t __svectors;

extern uint32_t _sfixed_ram_vect;

// 这里省略部分代码

void __attribute__((optimize("-O1"), section(".text.Reset_Handler"), long_call, noreturn)) Reset_Handler(void)
{
      // 这里省略部分代码

#   ifdef SCB_VTOR_TBLOFF_Msk
      /* Set the vector-table base address in RAM */
      pSrc = (uint32_t *) & _sfixed_ram_vect;
      SCB->VTOR = ((uint32_t) pSrc & SCB_VTOR_TBLOFF_Msk);
#   endif /* SCB_VTOR_TBLOFF_Msk */

      // 这里省略部分代码
}
```

最后修改中断向量表。在 MPLAB X 工程里，一般可以从这里找到它：



文件路径：

<your project path>\firmware\src\config\sam_d21_xpro\interrupts.c

打开“interrupts.c”，将原来放在 Flash 的中断向量表分别放在 Flash 和 RAM 中：

- 定义 DeviceVectorsFlash exception_table 数组，存放在 Flash 中，只存放中断向量表的前两项，目的是程序能够从 Flash 启动。
- 定义 DeviceVectors exception_table_ram 数组，存放在 RAM 中，存放整个中断向量表。

```
/* Mutiple handlers for vector */

/* Exception Table 1 (In Flash) */
typedef struct _DeviceVectorsFlash
{
	/* Stack pointer */
	void* pvStack;
	/* Cortex-M handlers */
	void* pfnReset_Handler;
} DeviceVectorsFlash;

__attribute__ ((section(".vectors")))
const DeviceVectorsFlash exception_table = {
	/* Configure Initial Stack Pointer, using linker-generated symbols */
	.pvStack = (void*) (&_stack),
	.pfnReset_Handler                = ( void * ) Reset_Handler,
};


__attribute__ ((section(".ram_vectors")))
DeviceVectors exception_table_ram =
{
	/* Configure Initial Stack Pointer, using linker-generated symbols */
	.pvStack = (void*) (&_stack),

	.pfnReset_Handler                = ( void * ) Reset_Handler,
	.pfnNonMaskableInt_Handler       = ( void * ) NonMaskableInt_Handler,
	.pfnHardFault_Handler            = ( void * ) HardFault_Handler,
	.pfnSVCall_Handler               = ( void * ) SVCall_Handler,
	.pfnPendSV_Handler               = ( void * ) PendSV_Handler,
	.pfnSysTick_Handler              = ( void * ) SysTick_Handler,
	.pfnPM_Handler                   = ( void * ) PM_Handler,
	.pfnSYSCTRL_Handler              = ( void * ) SYSCTRL_Handler,
	.pfnWDT_Handler                  = ( void * ) WDT_Handler,
	.pfnRTC_Handler                  = ( void * ) RTC_Handler,
	.pfnEIC_Handler                  = ( void * ) EIC_Handler,
	.pfnNVMCTRL_Handler              = ( void * ) NVMCTRL_Handler,
	.pfnDMAC_Handler                 = ( void * ) DMAC_Handler,
```

```
    .pfnUSB_Handler                 = ( void * ) USB_Handler,
    .pfnEVSYS_Handler               = ( void * ) EVSYS_Handler,
    .pfnSERCOM0_Handler             = ( void * ) SERCOM0_Handler,
    .pfnSERCOM1_Handler             = ( void * ) SERCOM1_Handler,
    .pfnSERCOM2_Handler             = ( void * ) SERCOM2_Handler,
    .pfnSERCOM3_Handler             = ( void * ) SERCOM3_Handler,
    .pfnSERCOM4_Handler             = ( void * ) SERCOM4_Handler,
    .pfnSERCOM5_Handler             = ( void * ) SERCOM5_Handler,
    .pfnTCC0_Handler                = ( void * ) TCC0_Handler,
    .pfnTCC1_Handler                = ( void * ) TCC1_Handler,
    .pfnTCC2_Handler                = ( void * ) TCC2_Handler,
    .pfnTC3_Handler                 = ( void * ) TC3_Handler,
    .pfnTC4_Handler                 = ( void * ) TC4_Handler,
    .pfnTC5_Handler                 = ( void * ) TC5_Handler,
    .pfnTC6_Handler                 = ( void * ) TC6_Handler,
    .pfnTC7_Handler                 = ( void * ) TC7_Handler,
    .pfnADC_Handler                 = ( void * ) ADC_Handler,
    .pfnAC_Handler                  = ( void * ) AC_Handler,
    .pfnDAC_Handler                 = ( void * ) DAC_Handler,
    .pfnPTC_Handler                 = ( void * ) PTC_Handler,
    .pfnI2S_Handler                 = ( void * ) I2S_Handler,


};
```

当然，如果希望某个中断处理函数能从 RAM 执行，需要把它申明为 RAM function。以 SysTick_Handler 中断处理举例：

```
__ramfunc
void SysTick_Handler(void)
{
    // 你的中断处理代码
}
```

(三) Flash 的擦除和写入函数放在 RAM 中执行 – MPLAB X

以 SAM D21J18A 举例，当对 Flash 进行擦除（执行 Erase Row 命令）或者写入（执行 Write Page 命令）时，如果下一条执行指令是存放在 Flash 中，则 MCU 会暂停执行直至 Flash 擦除或者写入操作完成。为了能够在对 Flash 进行擦除或者写入操作时，可以继续执行代码，我们可以对 Harmony 3 的 NVMCTRL PLIB 的相关函数稍作修改，并定义为 RAM function。Harmony 3 自动生成的代码在：<your project path>\firmware\src\config\sam_d21_xpro\peripheral\nvmctrl\plib_nvmctrl.c ， 以 Flash RowErase 和 PageWrite 函数为例， 修改后的代码如下：

```
/**
 *    \brief NVMCTRL Erase a row, execute code from RAM
 */
__ramfunc
bool NVMCTRL_RowErase_from_ram( uint32_t address )
{
	/* Set address and command */
	NVMCTRL_REGS->NVMCTRL_ADDR = address >> 1;

	NVMCTRL_REGS->NVMCTRL_CTRLA = NVMCTRL_CTRLA_CMD_ER_Val |
NVMCTRL_CTRLA_CMDEX_KEY;

	/* Wait NVMCTRL ready */
	while (!(NVMCTRL_REGS->NVMCTRL_INTFLAG & NVMCTRL_INTFLAG_READY_Msk))
	{
	}

	return true;
}



/**
 *    \brief NVMCTRL write a page, execute code from RAM
 */
__ramfunc
bool NVMCTRL_PageWrite_from_ram( uint32_t *data, const uint32_t address )
{
	uint32_t i = 0;
	uint32_t * paddress = (uint32_t *)address;

	/* writing 32-bit data into the given address */
	for (i = 0; i < (NVMCTRL_FLASH_PAGESIZE/4); i++)
	{
		*paddress++ = data[i];
	}

	 /* Set address and command */
	NVMCTRL_REGS->NVMCTRL_ADDR = address >> 1;

	NVMCTRL_REGS->NVMCTRL_CTRLA = NVMCTRL_CTRLA_CMD_WP_Val |
NVMCTRL_CTRLA_CMDEX_KEY;

	/* Wait NVMCTRL ready */
```

```
    while (!(NVMCTRL_REGS->NVMCTRL_INTFLAG & NVMCTRL_INTFLAG_READY_Msk))
    {
    }

    return true;
}
```

四、 总结

本文说明如何从 RAM 执行代码，包括将函数定义为 RAM function 和将中断向量表以及中断处理函数放到 RAM 中。这样就为单一的程序从 Flash 中执行方式提供了一种有益的补充，用户可以根据自己的项目需求灵活应用。

五、 附录：如何在 Keil 中从 RAM 执行代码

MPLAB® Harmony 3 可以通过 MHC 生成 Keil 项目，请参考“MPLAB® Harmony 3 之基础篇（31） -- 使用 MHC 来创建 IAR 或 Keil 项目”文档。对应 Keil 的项目，如何让代码从 RAM 中运行呢？道理和“如何在 MPLAB X 中从 RAM 执行代码”是一样的，只是链接脚本和 RAM function 的定义方式稍有区别，具体方法见下文。

(一)执行的函数放在 RAM 中 – Keil

在 Keil 中，可以通过“__attribute__ ((section(".ramfunc")))”把一个函数定义为 RAM Function，例如：

```
__attribute__ ((section(".ramfunc")))
void foo_function (void)
{
    // Your code
}
```

(二)执行的中断放在 RAM 中 – Keil

如果需要中断处理函数从 RAM 运行，则需要做下面的修改：

- 将中断向量表放置在 RAM 中，并正确配置 ARM 的 VTOR 寄存器
- 相应的中断处理函数也放在 RAM 中

把中断向量表从 FLASH 中重定位到 RAM，需要编辑项目使用的链接文件，以及修改在 MCU 进入 main()函数之前执行的启动代码。

首先修改链接文件，文件路径：
<your project path>\firmware\src\config\sam_d21_xpro\sam_d21_xpro.sct

打开“sam_d21_xpro.sct”并修改如下蓝色内容：

```
LR_IROM1 0x00000000 0x40000    {       ; load region size_region
    ER_IROM1 0x00000000 0x40000    {    ; load address = execution address
     *.o (RESET, +First)
     *(InRoot$$Sections)
     .ANY (+RO)
     .ANY (+XO)
    }
    RW_VECTOR 0x20000000 0x100       ; Vector table in RAM
    {
      interrupts.o(.vector_table_ram, +First)
    }
    RW_IRAM1 0x20000100 0x8000 - 0x1200 - 0x100    ; Entire RAM minus vector table in RAM, the
stack and heap block
    {
      *.o (.ramfunc)
      .ANY (+RW +ZI)
    }

    ARM_LIB_STACKHEAP 0x20000000 + 0x8000 - 0x1200 ALIGN 8 EMPTY    0x1200 ; Stack heap block
    {

    }

}
```

这里我们创建了一个 RW_VECTOR 段（0x100 字节）用来存放 RAM vector table。

然后修改启动代码，文件路径：
<your project path>\firmware\src\config\sam_d21_xpro\startup_keil.c

打开“startup_ keil.c”并添加如下蓝色内容：

```
/* Initialize segments */
extern const DeviceVectors __Vectors;
extern const DeviceVectors __Vectors_RAM;

extern char Image$$RW_VECTOR$$Base ;
extern char Image$$RW_VECTOR$$Length ;
extern char Load$$RW_VECTOR$$Base ;

// 这里省略部分代码

/**
 * \brief This is the code that gets called on processor reset.
```

```
 * To initialize the device, and call the main() routine.
 */
void __attribute__((section(".text.Reset_Handler"))) Reset_Handler(void)
{
	uint32_t *pSrc;
	uint32_t *pDest;
	uint32_t length;

	/* Call the optional application-provided _on_reset() function. */
	if (_on_reset)
	{
		_on_reset();
	}

	/* Initialize the Vector table relocate segment */
	pSrc = (uint32_t *)&Load$$RW_VECTOR$$Base;
	pDest = (uint32_t *)&Image$$RW_VECTOR$$Base;
	length = (uint32_t)&Image$$RW_VECTOR$$Length / 4;

	if (pSrc != pDest) {
		while (length) {
			*pDest++ = *pSrc++;
			length--;
		}
	}

#	ifdef SCB_VTOR_TBLOFF_Msk
	/*	Set the vector-table base address in RAM */
	pSrc = (uint32_t *) &__Vectors_RAM;
	SCB->VTOR = ((uint32_t) pSrc & SCB_VTOR_TBLOFF_Msk);
#	endif /* SCB_VTOR_TBLOFF_Msk */

	/* Call the optional application-provided _on_bootstrap() function. */
	if (_on_bootstrap)
	{
		_on_bootstrap();
	}

	/* Execute entry point to the C library initialization routine,
	    which eventually executes application's main function */
	__main();

	/* Infinite loop */
```

```
    while (1) {}
}
```

最后修改中断向量表，文件路径：
<your project path>\firmware\src\config\sam_d21_xpro\interrupts.c

打开“interrupts.c”，将原来放在 Flash 的中断向量表分别放在 Flash 和 RAM 中：

- 定义__Vectors 数组，存放在 Flash 中，只存放中断向量表的前两项，目的是程序能够从 Flash 启动。
- 定义__Vectors_RAM 数组，存放在 RAM 中，存放整个中断向量表。

```
/* Mutiple handlers for vector */

extern unsigned int Image$$ARM_LIB_STACKHEAP$$ZI$$Limit;

/* Exception Table 1 (In Flash) */
typedef struct _DeviceVectorsFlash
{
    /* Stack pointer */
    void* pvStack;
    /* Cortex-M handlers */
    void* pfnReset_Handler;
} DeviceVectorsFlash;

__attribute__ ((section("RESET")))
const DeviceVectorsFlash __Vectors = {
    /* Configure Initial Stack Pointer, using linker-generated symbols */
    .pvStack                = (void *)(&Image$$ARM_LIB_STACKHEAP$$ZI$$Limit),
    .pfnReset_Handler       = (void*) Reset_Handler,
};


/* Exception Table 2 (In RAM) */
__attribute__ ((section(".vector_table_ram")))
DeviceVectors __Vectors_RAM =
{
    /* Configure Initial Stack Pointer, using linker-generated symbols */
    .pvStack = (void *)(&Image$$ARM_LIB_STACKHEAP$$ZI$$Limit),

    .pfnReset_Handler              = ( void * ) Reset_Handler,
    .pfnNonMaskableInt_Handler     = ( void * ) NonMaskableInt_Handler,
    .pfnHardFault_Handler          = ( void * ) HardFault_Handler,
    .pfnSVCall_Handler             = ( void * ) SVCall_Handler,
    .pfnPendSV_Handler             = ( void * ) PendSV_Handler,
```

```
    .pfnSysTick_Handler                 = ( void * ) SysTick_Handler,
    .pfnPM_Handler                      = ( void * ) PM_Handler,
    .pfnSYSCTRL_Handler                 = ( void * ) SYSCTRL_Handler,
    .pfnWDT_Handler                     = ( void * ) WDT_Handler,
    .pfnRTC_Handler                     = ( void * ) RTC_Handler,
    .pfnEIC_Handler                     = ( void * ) EIC_Handler,
    .pfnNVMCTRL_Handler                 = ( void * ) NVMCTRL_Handler,
    .pfnDMAC_Handler                    = ( void * ) DMAC_Handler,
    .pfnUSB_Handler                     = ( void * ) USB_Handler,
    .pfnEVSYS_Handler                   = ( void * ) EVSYS_Handler,
    .pfnSERCOM0_Handler                 = ( void * ) SERCOM0_Handler,
    .pfnSERCOM1_Handler                 = ( void * ) SERCOM1_Handler,
    .pfnSERCOM2_Handler                 = ( void * ) SERCOM2_Handler,
    .pfnSERCOM3_Handler                 = ( void * ) SERCOM3_Handler,
    .pfnSERCOM4_Handler                 = ( void * ) SERCOM4_Handler,
    .pfnSERCOM5_Handler                 = ( void * ) SERCOM5_Handler,
    .pfnTCC0_Handler                    = ( void * ) TCC0_Handler,
    .pfnTCC1_Handler                    = ( void * ) TCC1_Handler,
    .pfnTCC2_Handler                    = ( void * ) TCC2_Handler,
    .pfnTC3_Handler                     = ( void * ) TC3_Handler,
    .pfnTC4_Handler                     = ( void * ) TC4_Handler,
    .pfnTC5_Handler                     = ( void * ) TC5_Handler,
    .pfnTC6_Handler                     = ( void * ) TC6_Handler,
    .pfnTC7_Handler                     = ( void * ) TC7_Handler,
    .pfnADC_Handler                     = ( void * ) ADC_Handler,
    .pfnAC_Handler                      = ( void * ) AC_Handler,
    .pfnDAC_Handler                     = ( void * ) DAC_Handler,
    .pfnPTC_Handler                     = ( void * ) PTC_Handler,
    .pfnI2S_Handler                     = ( void * ) I2S_Handler,


};
```

当然，如果希望某个中断处理函数能从 RAM 执行，需要把它申明为 RAM function。以 SysTick_Handler 中断处理举例：

```
__attribute__ ((section(".ramfunc")))
void SysTick_Handler(void)
{
    // 你的中断处理代码
}
```

(三) Flash 的擦除和写入函数放在 RAM 中执行 – Keil

参考章节：Flash 的擦除和写入函数放在 RAM 中执行 – MPLAB X，方法是一样的。