SMART ARM-based Microcontrollers

AT07347: Executing Code from RAM

APPLICATION NOTE

Preface

Executing code from RAM can be useful in cases where low power requirements demand this and in cases where low latency or fast execution is necessary. This application note explains how this can be done. There is an associated project for the SAM D20 that can be used as an example.

Table of Contents

1. Introduction

This appnote aims to show how to set up a project so that you can execute all or parts of the code in RAM.

Executing code from RAM can be of interest for power reduction. It may in some cases be possible to execute all code from RAM. This can make it possible to turn of clocks to the FLASH and NVM controller and power consumption will go down. In some cases this may help to execute code faster and accelerate entering Sleep mode.

In cases where it is necessary to execute code or interrupts as fast as possible, and FLASH is operating with wait cycles, executing from RAM may give an improvement in execution time.

It can also be useful when using FLASH as emulated EEPROM. When writing to FLASH it is necessary to do page-wise writes and at times it may be necessary to write multiple pages at the same time. When writing to FLASH it is not possible to execute code from FLASH, but it is still possible to execute code from RAM.

In cases when it is necessary to have low latency interrupts and when emulated EEPROM is being used, executing interrupt code from RAM may be necessary.

1.1. Executing Functions in RAM

Executing code from RAM can be easily done by placing the RAMFUNC attribute in front of the function that it is desirable to execute from RAM.

1.2. Executing Interrupts from RAM

If it is desirable to run interrupts from RAM due to latency when using interrupts, it is good practice to also relocate the vector table for the interrupts. Indeed if Emulated EEPROM is being used and only the interrupt handler is placed in RAM, it will still be necessary to access the vector table located in FLASH before the interrupt can be executed and latency issues will not be solved.

To relocate the vector table from FLASH to RAM it is necessary to edit the linker script that is used when compiling, and some of the code that is executed before the CPU enters the main() function.

First locate the linker script for the project. In a project started from Atmel® Studio this will normally be found by using the Solution Explorer (If it is not open go to view->Solution Explorer). It will usually be located in the following path src/ASF/sam0/utils/linker_scripts/samd20/gcc/. The name of the script will differ dependent on what chip is being used. For the SAM D20 used on the SAM D20 Xplained Pro it is "samd20j18_flash.ld"

In the file found here two lines needs to be added in the .relocate block. It should look like this:

```
.relocate : AT (_etext)
{
    . = ALIGN(4);
    _sfixed_ram_vect = .;
    KEEP(*(.ram_vectors .ram_vectors.*));
    _srelocate = .;
    *(.ramfunc .ramfunc.*);
    *(.data .data.*);
    . = ALIGN(4);
```

```
    _erelocate = .;
} > ram
```

This will create a ram_vectors define and set the value of _sfixed_ram_vect so that it points to the first part of RAM.

The next file that needs to be edited is the startup_samxxx.c file. This file can be found here: src/ASF/sam0/utils/cmsis/samxxx/source/gcc. This file contains code that is executed when the device starts up before entering main().

The first step is to declare the _sfixed_ram_vect which will point to RAM. This should be done at the top of the document together with the other declared variables:

```
extern uint32_t _sfixed_ram_vect;
```

The actual uploading of the interrupt vector and the functions with the RAMFUNC attribute gets done in the Reset_Handler() function but this needs to be updated with the changes in the linker script. The pDest variable needs to be pointed to RAM. This is done by changing the following line:

```
pDest = &_srelocate;
```

To this:

```
pDest = &_sfixed_ram_vect;
```

The next step is to use the _sfixed_ram_vect to set the address of the interrupt vector table (the VTOR). This is done in the Reset_Handler() function found in the same file. Replace the _sfixed value used in the Reset_Handler() with _sfixed_ram_vect:

```
pSrc = (uint32_t *) & _sfixed_ram_vect;
```

Now the Reset_Handler() function should look like this:

```
void Reset_Handler(void)
{
        uint32_t *pSrc, *pDest;

        /* Initialize the relocate segment */
        pSrc = &_etext;
        pDest = &_sfixed_ram_vect;

        if (pSrc != pDest) {
                for (; pDest < &_erelocate;) {
                        *pDest++ = *pSrc++;
                }
        }

        /* Clear the zero segment */
        for (pDest = &_szero; pDest < &_ezero;) {
                *pDest++ = 0;
        }

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

        /* Initialize the C library */
        __libc_init_array();
```

```
        /* Branch to main function */
        main();

        /* Infinite loop */
        while (1);
}
```

The final change is to put the whole interrupt vector into RAM at the correct address. This is done by copying the exception_table and renaming it to so that you have one interrupt table named exception_table and one called exception_table2. The attribute for exception_table2 should then be changed to the following line:

```
__attribute__ ((section(".ram_vectors")))
```

Taking the SAM D20 as an example, there should now be a table looking like the below code that will be placed in RAM. It is important that all interrupt vectors or at the very least a placeholder for it is in the list. If the list is misaligned the incorrect interrupt handler will be triggered. The actual placement of code and the interrupt vector will be done in the Reset_Handler().

```
__attribute__ ((section(".ram_vectors")))
const DeviceVectors exception_table_2 = {

        (void*) (&_estack),
        (void*) Reset_Handler,
        (void*) NMI_Handler,
        (void*) HardFault_Handler,
        (void*) (0UL), /* Reserved */
        (void*) (0UL), /* Reserved */
        (void*) (0UL), /* Reserved */
        (void*) (0UL), /* Reserved */
        (void*) (0UL), /* Reserved */
        (void*) (0UL), /* Reserved */
        (void*) (0UL), /* Reserved */
        (void*) SVC_Handler,
        (void*) (0UL), /* Reserved */
        (void*) (0UL), /* Reserved */
        (void*) PendSV_Handler,
        (void*) SysTick_Handler,

        /* Configurable interrupts */
        (void*) PM_Handler,       /*  0 Power Manager */
        (void*) SYSCTRL_Handler, /*  1 System Control */
        (void*) WDT_Handler,      /*  2 Watchdog Timer */
        (void*) RTC_Handler,      /*  3 Real-Time Counter */
        (void*) EIC_Handler,      /*  4 External Interrupt Controller */
        (void*) NVMCTRL_Handler, /*  5 Non-Volatile Memory Controller */
        (void*) EVSYS_Handler,    /*  6 Event System Interface */
        (void*) SERCOM0_Handler, /*  7 Serial Communication Interface 0 */
        (void*) SERCOM1_Handler, /*  8 Serial Communication Interface 1 */
        (void*) SERCOM2_Handler, /*  9 Serial Communication Interface 2 */
        (void*) SERCOM3_Handler, /* 10 Serial Communication Interface 3 */
        (void*) SERCOM4_Handler, /* 11 Serial Communication Interface 4 */
        (void*) SERCOM5_Handler, /* 12 Serial Communication Interface 5 */
        (void*) TC0_Handler,      /* 13 Basic Timer Counter 0 */
        (void*) TC1_Handler,      /* 14 Basic Timer Counter 1 */
        (void*) TC2_Handler,      /* 15 Basic Timer Counter 2 */
        (void*) TC3_Handler,      /* 16 Basic Timer Counter 3 */
        (void*) TC4_Handler,      /* 17 Basic Timer Counter 4 */
        (void*) TC5_Handler,      /* 18 Basic Timer Counter 5 */
        (void*) TC6_Handler,      /* 19 Basic Timer Counter 6 */
```

```
        (void*) TC7_Handler,      /* 20 Basic Timer Counter 7 */
        (void*) ADC_Handler,      /* 21 Analog Digital Converter */
        (void*) AC_Handler,       /* 22 Analog Comparators */
        (void*) DAC_Handler,      /* 23 Digital Analog Converter */
        (void*) PTC_Handler       /* 24 Peripheral Touch Controller */
};
```

The vector table that was placed in FLASH can now be made smaller as it only needs to contain the Reset_Handler() and the _estack pointer. To do this we need a new typedef placed in the startup_samxxx.c file.

```
typedef struct _DeviceVectorsFlash
{
    /* Stack pointer */
    void* pvStack;

    /* Cortex-M handlers */
    void* pfnReset_Handler;
} DeviceVectorsFlash;
```

Now the new typdef can be used to create the part of the vector table that needs to be placed in FLASH. The "__attribute__ ((section(".vectors")))" attribute will tel the linker to place this in the first part of FLASH and as flash is not volatile there is no need to handle this in the Reset_Handler() function.

```
__attribute__ ((section(".vectors")))
const DeviceVectorsFlash exception_table = {

        /* Configure Initial Stack Pointer, using linker-generated symbols */
        (void*) (&_estack),

        (void*) Reset_Handler,
};
```

For other devices with different interrupts the lists will not be the same so the list should be made as described.

Now the interrupt table will be placed in RAM. If a interrupt routine should be placed in RAM, use RAMFUNC as an attribute on the interrupt handler function. If the interrupt handler calls other functions they to will have to be placed in RAM.

1.3. Executing all Code from RAM

If it is desirable to place all code in RAM, a linker script is available for this purpose. This can be found in ASF, however currently not trough the ASF bundled with Atmel Studio. This will be found in the ASF standalone package found on atmel.com/asf. There is one linker script for each device so select the correct linker script depending on the RAM and FLASH size of the device being used. The linker scripts will be found in the following path for the SAM D20 "\sam0\utils\linker_scripts\samd20\gcc" in ASF.

1.4. Example Code

The example code used for this application note uses the code found in ASF for the external interrupt controller, for more in depth information on how this example runs see the callback example. The change is that the linker script and the startup code has been modified as described in section Executing

Interrupts from RAM. The interrupt itself is executed from FLASH as neither the handler or the functions being called from the handler have the RAMFUNC attribute.

2. Revision History

Doc. Rev.	Date	Comments
42249B	04/2016	New template and some minor corrections
42249A	03/2014	Initial document release

Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA **T:** (+1)(408) 441.0311 **F:** (+1)(408) 436.4200 | **www.atmel.com**