Getting Started with SAM3N Microcontrollers

1. Introduction

This application note is intended as an aid in familiarizing the user with the Atmel ARM® Cortex® M3-based SAM3N series of microcontrollers. It describes in detail a simple project that uses several important features present on the SAM3N MCU, including how to set up the microcontroller prior to executing the application, as well as adding functionality. After examination of this application note, the reader should be able to successfully start a new project from scratch.

This application note also explains how to setup and use a GNU ARM toolchain in order to compile and run a software project.

Note that the getting started example has been ported and is included in IAR® Embedded Workbench for ARM (EWARM) and Keil® MDK-ARM development kit. When using these tools, the reader should disregard Section 4. “Building the Project”.

To use this document efficiently, the reader should be experienced in using the ARM core. For more information about the ARM core architecture, please refer to the relevant documents available from http://www.arm.com.

2. Requirements

The software provided with this application note requires several components:

- The SAM3N-EK Evaluation Kit
- A computer running Microsoft® Windows® 2000/XP
- An ARM cross-compiler toolchain supporting ARM Cortex-M3 (such as CodeSourcery-2010-q1)
- SAM-BA® V2.10 or later

AT91SAM ARM-based Flash MCU

Application Note

11097A–ATARM–07-Jan-11

3. Getting Started with a Software Example

This section describes how to program a basic application to become familiar with SAM3N microcontrollers. It is divided into two main sections:

- the specification of the example (what it does, what peripherals are used)
- details of the programming aspect

3.1 Specification

3.1.1 Features

The demonstration program causes two LEDs on the board blink at a fixed rate. This rate is generated by using a timer for the first LED and a Wait function based on a 1 ms tick generated by using the System Timer (SysTick). The blinking can be stopped using two buttons (one for each LED).

While this software may look simple, it uses several peripherals which make up the basis of an operating system. As such, it provides a good starting point for someone wanting to become familiar with the AT91SAM family of microcontrollers.

3.1.2 Peripherals

In order to perform the operations introduced above, the software example uses the following set of peripherals:

- Parallel Input/Output (PIO) controller
- Timer Counter (TC)
- Universal Asynchronous Receiver Transmitter (UART)
- Cortex M-3 System Timer (SysTick)
- Nested Vectored Interrupt Controller (NVIC)

LEDs and buttons on the board are connected to standard input/output pins of the chip and managed by a PIO controller. In addition, it is possible to have the controller generate an interrupt when the status of one of its pins changes, and buttons are configured accordingly.

The TC and SysTick are used to generate two time bases, in order to obtain the LED blinking rates. Both are used in interrupt mode.

- The TC triggers an interrupt at a fixed rate, each instance toggling the LED state (on/off).
- The SysTick triggers an interrupt every millisecond, incrementing a variable by one tick. The Wait function monitors this variable to provide a precise delay for toggling the second LED state.

Using the NVIC is required to manage interrupts. It allows the configuration of a separate interrupt handler for each source. Two different functions are used to handle PIO and SysTick interrupts.

Finally, an additional peripheral, the UART, is used to output debug traces on a serial line. Having the firmware send debug traces at key points of the code can greatly help the debugging process.

3.1.3 Evaluation Kit

3.1.3.1 Memories

The SAM3N4C located on the SAM3N-EK evaluation board features one internal 24-Kbyte SRAM memory and a 256-Kbyte Flash Memory block. The Getting Started example software can be compiled and loaded on the internal Flash and the internal SRAM.

3.1.3.2 Buttons

The SAM3N4 Evaluation Kit features two push buttons, connected to pins PA15 and PA16. When pressed, they force a logical low level on the corresponding PIO line.

The Getting Started example uses both buttons by means of the internal hardware debouncing circuitry embedded in the SAM3N. Refer to Section 3.2.8.6 “Configuring Input Debouncing” for more details.

3.1.3.3 LEDs

There are three general-purpose LEDs (blue, green and amber) on the SAM3N-EK, as well as a software-controllable red power LED. They are wired to pins PA23, PB14, PA25 and PA0, respectively. Setting a logical low level on these PIO lines turns the corresponding LED on.

The example application uses the two general-purpose LEDs (PA23 and PB14).

3.1.3.4 UART

On the SAM3N-EK, the UART uses pins PA9 and PA10 for the URXD and UTXD signals, respectively.

3.2 Implementation

As stated previously, the example defined above requires the use of several peripherals. It must also provide the necessary code for starting up the microcontroller. Both aspects are described in detail in this section, with commented source code when appropriate.

3.2.1 C-Startup

Most of the code of an embedded application is written in C. This makes the program easier to understand, more portable and modular. The C-startup code must:

- Provide vector table
- Initialize critical peripherals
- Initialize stacks
- Initialize memory segments
- Locate Vector Table Offset

These steps are described in the following paragraphs.

3.2.1.1 Vector Table

The vector table contains the initialization value for the stack pointer (see Section 3.2.1.9 “Initializing Stacks”) on reset, and the entry point addressed for all exception handlers. The exception number (see Table 3-1) defines the order of entries in the vector table associated with exception handler entry as illustrated in Table 3-2.

Table 3-1. Exception numbers

Exception number	Exception
1	Reset
2	RESERVED
3	HardFault
4	MemManage
5	BusFault
6	UsageFault
7-10	RESERVED
11	SVCall
12	Debug Monitor
13	RESERVED
14	PendSV
15	SysTick
16	External Interrupt (0)
...	...
16 + N	External Interrupt (N)

Table 3-2. Vector table format

Word offset	Description - all pointer address values
0	SP_main (reset value of the Main stack pointer)
Exception Number	Exception using that Exception Number

The vector table's current location can be determined or relocated in the CODE or SRAM partitions of the memory map using the Vector Table Offset Register (VTOR), details of the register can be found in the "Cortex-M3 TechnicalReference Manual".

In the example, a full vector table looks like this:

```
extern uint32_t _estack;
void ResetException( void ) ;

__attribute__((section(".vectors")))
IntFunc exception_table[] = {

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

    NMI_Handler,
    HardFault_Handler,
    MemManage_Handler,
```

```
    BusFault_Handler,
    UsageFault_Handler,
    0, 0, 0, 0,          /* Reserved */
    SVC_Handler,
    DebugMon_Handler,
    0,                   /* Reserved  */
    PendSV_Handler,
    SysTick_Handler,

    /* Configurable interrupts  */
    SUPC_IrqHandler,     /* 0  Supply Controller */
    RSTC_IrqHandler,     /* 1  Reset Controller */
    RTC_IrqHandler,      /* 2  Real Time Clock */
    RTT_IrqHandler,      /* 3  Real Time Timer */
    WDT_IrqHandler,      /* 4  Watchdog Timer */
    PMC_IrqHandler,      /* 5  PMC */
    EEFC_IrqHandler,     /* 6  EEFC */
    IrqHandlerNotUsed,   /* 7  Reserved */
    UART0_IrqHandler,    /* 8  UART0 */
    UART1_IrqHandler,    /* 9  UART1 */
    IrqHandlerNotUsed,   /* 10 Reserved */
    PIOA_IrqHandler,     /* 11 Parallel IO Controller A */
    PIOB_IrqHandler,     /* 12 Parallel IO Controller B */
    PIOC_IrqHandler,     /* 13 Parallel IO Controller C */
    USART0_IrqHandler,   /* 14 USART 0 */
    USART1_IrqHandler,   /* 15 USART 1 */
    IrqHandlerNotUsed,   /* 16 Reserved */
    IrqHandlerNotUsed,   /* 17 Reserved */
    IrqHandlerNotUsed,   /* 18 Reserved */
    TWI0_IrqHandler,     /* 19 TWI 0 */
    TWI1_IrqHandler,     /* 20 TWI 1 */
    SPI_IrqHandler,      /* 21 SPI */
    IrqHandlerNotUsed,   /* 22 Reserved */
    TC0_IrqHandler,      /* 23 Timer Counter 0 */
    TC1_IrqHandler,      /* 24 Timer Counter 1 */
    TC2_IrqHandler,      /* 25 Timer Counter 2 */
    TC3_IrqHandler,      /* 26 Timer Counter 3 */
    TC4_IrqHandler,      /* 27 Timer Counter 4 */
    TC5_IrqHandler,      /* 28 Timer Counter 5 */
    ADC_IrqHandler,      /* 29 ADC controller */
    DAC_IrqHandler,      /* 30 DAC controller */
    PWM_IrqHandler       /* 31 PWM */
};
```

3.2.1.2 *Vectors: Reset Exception*

The reset exception handler runs after the core reads the start SP, SP_main from vector table offset 0, and the start PC from vector table offset. A normal reset exception handler follows the steps in Table 3-3.

Table 3-3. Reset exception behavior

Action	Description
Low-Level Initialization	Initialize critical peripherals
Initialize variables	Any global/static variables must be setup. This includes initializing the BSS variable to 0, and copying initial values from ROM to RAM for non-constant variables.
Switch vector table	Optionally change vector table from Code area, @0, to a location in SRAM. This is normally done to enable dynamic changes.
Branch to main()	Branch to main application

Section 3.2.1.4 “Low-Level Initialization”, gives details on the above actions.

3.2.1.3 *Vectors: Exception Handlers*

When an exception occurs, context state is saved by the hardware onto a stack pointed to by the SP register. The stack that is used depends on the mode of the processor at the time of the exception. Refer to Section 3.2.1.9 “Initializing Stacks” for more details about stacks.

In the vector table, the exception handler corresponding to the exception number will be executed. If the program does not need to handle an exception, then the corresponding instruction can be set to an infinite loop. An example follows:

```
/**
 * Default interrupt handler for Supply Controller.
 */
WEAK void SUPC_IrqHandler(void)
{
    while(1);
}
```

By default, all the exception handlers are implemented as an infinite loop. Since “WEAK” attribute is added to these exception handlers, they can be re-implemented when wanted in the applications.

For example, in exception.c, the default SysTick’s exception handler is implemented as follows:

```
/* Define WEAK attribute */
#if defined   ( __CC_ARM   )
    #define WEAK __attribute__ ((weak))
#elif defined ( __ICCARM__ )
    #define WEAK __weak
#elif defined (  __GNUC__  )
    #define WEAK __attribute__ ((weak))
#endif

WEAK void SysTick_Handler(void)
{
```

```
        while(1);
    }
```

Since the SysTick exception is used in the example, the user can re-implement the SysTick exception handler in the example as follows:

```
/**
 * Handler for Sytem Tick interrupt. Increments the timestamp counter.
 */
void SysTick_Handler(void)
{
    timestamp++;
}
```

In this way, when a SysTick exception occurs, a variable “timestamp” will be increased instead of executing an infinite loop.

3.2.1.4 *Low-Level Initialization*

The first step of the initialization process is to configure critical peripherals:

- Enhanced Embedded Flash Controller (EEFC)
- NRST reset
- Slow clock
- Main oscillator and its PLL

The following sections explain why these peripherals are considered critical, and detail the required operations to configure them properly.

3.2.1.5 *Low-Level Initialization: Enhanced Embedded Flash Controller*

Depending on the clock of the microcontroller core, one or more wait states must be configured to adapt the Flash access time and the core cycle access time. The number of wait states can be configured in the Enhanced Embedded Flash Controller (EEFC).

After reset, the chip uses its internal 4 MHz Fast RC Oscillator, so there is no need for any wait state. However, before switching to the main oscillator (in order to run at full speed), the correct number of wait states must be set. If not, the core may no longer be able to read the code from the Flash.

Configuring the number of wait states is done in the Flash Mode Register (EEFC_FMR). For example, running the core clock at 48 MHz operation requires the use of at least two wait states.

This example configures the core clock at 48 MHz (PLL output), using three wait states.

```
EFC->EEFC_FMR = EEFC_FMR_FWS(3);
```

For more information about the required number of wait states, depending on the operating frequency of a microcontroller, refer to the AC Electrical Characteristics section of the “SAM3N Series Datasheet”.

3.2.1.6 *Low-Level Initialization: NRST*

The Reset Controller (RSTC) embeds an NRST Manager that samples the NRST pin at Slow Clock speed. When the line is detected low, a User Reset is reported. The user can program the NRST Manager to disable reset when assertion of NRST occurs.

The example enables the NRST pin as the user reset trigger, to perform system reset by writing RSTC Mode Register (RSTC_MR) with the URSTEN bit at 1:

```
RSTC->RSTC_MR |= RSTC_MR_URSTEN;
```

3.2.1.7 *Low-Level Initialization: Slow clock*

The Supply Controller (SUPC) embeds a slow clock generator that is supplied with the backup power supply. As soon as the backup is supplied, both the crystal oscillator and the embedded RC oscillator are powered up, but only the embedded RC oscillator is enabled. This allows the slow clock to be valid in a short time (about 100 µs).

The user can select the crystal oscillator to be the source of the slow clock, as it provides a more accurate frequency. This is done by writing the SUPC Control Register (SUPC_CR) with the XTALSEL bit at 1:

```
if ((SUPC->SUPC_SR & SUPC_SR_OSCSEL) != SUPC_SR_OSCSEL_CRYST)
{
    SUPC->SUPC_CR = SUPC_CR_XTALSEL_CRYSTAL_SEL | SUPC_CR_KEY(0xA5u);
    timeout = 0;
    while (!(SUPC->SUPC_SR & SUPC_SR_OSCSEL_CRYST) );
}
```

3.2.1.8 *Low-Level Initialization: Main Oscillator and PLL*

After reset, the chip is running with the 4 MHz Fast RC Oscillator. The main oscillator and its Phase Lock Loop (PLL) must be configured in order to run at full speed. Both can be configured in the Power Management Controller (PMC).

The main oscillator has two sources:

- 4/8/12 MHz RC Oscillator, which starts very quickly and is used at startup
- 3 to 20 MHz Crystal Oscillator, which can be bypassed.

The first step is to enable both oscillators and wait for the crystal oscillator to stabilize. Writing the oscillator startup time and the MOSCRCEN and MOSCXTEN bits in the PMC Main Oscillator Register (PMC_MOR), starts the oscillator. Stabilization occurs when MOSCXTS bit of the PMC Status Register (PMC_SR) is set. The following piece of code performs these two operations:

```
#define BOARD_OSCOUNT   (CKGR_MOR_MOSCXTST(0x8))

if ( !(PMC->CKGR_MOR & CKGR_MOR_MOSCSEL) )
{
   PMC->CKGR_MOR = CKGR_MOR_KEY(0x37) | BOARD_OSCOUNT | CKGR_MOR_MOSCRCEN |
CKGR_MOR_MOSCXTEN;
    timeout = 0;
    while (!(PMC->PMC_SR & PMC_SR_MOSCXTS) && (timeout++ < CLOCK_TIMEOUT));
}
```

Calculation of the correct oscillator startup time value is done by looking at the Crystal Oscillators characteristics given in the "SAM3N Series Datasheet". Note that the internal slow clock of the SAM3N4 is generated using an RC oscillator. This must be taken into account as this impacts the slow clock accuracy. Here is an example:

RC oscillator frequency range in kHz: $20 \leq f_{RC} \leq 44$

Oscillator frequency range in MHz: $4 \leq f_{Osc} \leq 12$

Oscillator frequency on EK: $f_{Osc} = 12 MHz$

Oscillator startup time: $t_{Startup} \leq 1.4 ms$

Value for a 2ms startup: $MOSCXTST = \frac{32768 \times 0.002}{8} = 8$

The second step is to switch the source of the main oscillator to the crystal oscillator and wait for the selection to be done. Writing the MOSCSEL bit of the PMC_MOR register will switch the source of the main oscillator to the crystal oscillator. The selection is done when MOSCSEL bit of the PMC_SR is set:

```
PMC->CKGR_MOR = CKGR_MOR_KEY(0x37) | BOARD_OSCOUNT | CKGR_MOR_MOSCRCEN |
CKGR_MOR_MOSCXTEN | CKGR_MOR_MOSCSEL;
timeout = 0;
while (!(PMC->PMC_SR & PMC_SR_MOSCSELS) && (timeout++ < CLOCK_TIMEOUT));
PMC->PMC_MCKR = (PMC->PMC_MCKR & ~(uint32_t)PMC_MCKR_CSS_Msk) |
PMC_MCKR_CSS_MAIN_CLK;
for ( timeout = 0; !(PMC->PMC_SR & PMC_SR_MCKRDY) && (timeout++ <
CLOCK_TIMEOUT) ; );
```

Once the crystal main oscillator is started and stabilized, the PLL can be configured. The PLL is made up of two chained blocks.

- the first one divides the input clock
- the second one multiplies it

The *MUL* and *DIV* factors are set in the PLL Register (CKGR_PLLR) of the PMC. These two values must be chosen according to the main oscillator (input) frequency and the desired main clock (output) frequency. In addition, the multiplication block has a minimum input frequency, and the master clock has a maximum allowed frequency. These two constraints have to be taken into account. The example below is given for the SAM3N-EK:

$$f_{Input} = 12$$

$$DIV = 1$$

$$MUL = (8 - 1) = 7$$

$$f_{Output} = \frac{12}{1} \times 8 = 96 MHz$$

Like the main oscillator, a PLL startup time must also be provided. Again, it can be calculated by looking at the electrical characteristics given in the "SAM3N Series Datasheet". After CKGR_PLLR is modified with the PLL configuration values, the software must wait for the PLL to be locked. This is done by monitoring the Status Register of the PMC:

```
#define BOARD_PLLR      (CKGR_PLLR_STUCKTO1 \
                        | CKGR_PLLR_MUL(0x7) \
                        | CKGR_PLLR_PLLCOUNT(0x1) \
                        | CKGR_PLLR_DIV(0x1))

PMC->CKGR_PLLR = BOARD_PLLR;
timeout = 0;
while (!(PMC->PMC_SR & PMC_SR_LOCK) && (timeout++ < CLOCK_TIMEOUT));
```

Finally, the prescale value of the main clock must be set, and the PLL output selected. Note that the prescale value must be set first, to avoid having the chip run at a frequency higher than the

maximum operating frequency defined in the SAM3N AC characteristics. As such, this step is done using two register writes, with two loops to wait for the main clock to be ready.

```
#define BOARD_MCKR      (PMC_MCKR_PRES_CLK_2 | PMC_MCKR_CSS_PLL_CLK)

PMC->PMC_MCKR = (BOARD_MCKR & (uint32_t)~PMC_MCKR_CSS_Msk) |
PMC_MCKR_CSS_MAIN_CLK;
for ( timeout = 0; !(PMC->PMC_SR & PMC_SR_MCKRDY) && (timeout++ <
CLOCK_TIMEOUT) ; );

PMC->PMC_MCKR = BOARD_MCKR ;
for ( timeout = 0; !(PMC->PMC_SR & PMC_SR_MCKRDY) && (timeout++ <
CLOCK_TIMEOUT) ; );
```

At this point, the chip is configured to run on the main clock at 48 MHz (96 MHz/2) with the PLL at 96 MHz.

3.2.1.9 *Initializing Stacks*

There are two stacks supported in ARMv7-M architecture, each with its own (banked) stack pointer register.

- the Main stack - SP_main
- the Process stack - SP_process

The stack pointer that is used in exception entry and exit is described in the Cortex-M3 TechnicalReference Manual". SP_main is located at vector table offset "0" - (see Section 3.2.1.1 "Vector Table"), it is selected and initialized on reset.

3.2.1.10 *Initializing BSS and Data Segments*

A binary file is usually divided into two segments. The first one holds the executable code of the application, as well as read-only data (declared as *const* in C). The second segment contains read/write data, i.e., data that can be modified. These two sections are called **text** and **data**, respectively.

Variables in the **data** segment are said to be either uninitialized or initialized. In the first case, the programmer has not set a particular value when declaring the variable. Conversely, variables fall in the second category when they have been declared with a value. Unintialized variables are held in a special subsection called BSS (for Block Started by Symbol).

Whenever the application is loaded in the internal Flash memory of the chip, the Data segment must be initialized at startup. This is necessary because read/write variables are located in SRAM or External RAM, not in Flash. For IAR Embedded Workbench and Keil MDK-ARM, refer to http://iar.com/website1/1.0.1.0/3/1/ and http://www.keil.com/.

Initialized data is contained in the binary file and loaded with the rest of the application in the memory. Usually, it is located right after the **text** segment. This makes it easy to retrieve the starting and ending address of the data to copy. To load these addresses faster, they are explicitly stored in the code using a compiler-specific instruction. Here is an example for the GNU tool chain:

```
extern uint32_t _sfixed;
extern uint32_t _efixed;
extern uint32_t _etext;
```

```
extern uint32_t _srelocate;
extern uint32_t _erelocate;
extern uint32_t _szero;
extern uint32_t _ezero;
extern uint32_t _sstack;
extern uint32_t _estack;
```

The actual copy operation consists of loading these values and several registers, and looping through the data:

```
uint32_t *pSrc, *pDest ;

pSrc = &_etext ;
pDest = &_srelocate ;

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

In addition, it is both safer and more useful for debug purposes to initialize the BSS segment by filling it with zeroes. Theoretically, this operation is unneeded, however, it can have several benefits. For example, it makes it easier when debugging to see which memory regions have been modified. This can be a valuable tool for spotting stack overflow and similar problems.

Initialization of the BSS looks like:

```
for ( pDest = &_szero ; pDest < &_ezero ; )
{
    *pDest++ = 0;
}
```

3.2.1.11 Locate Vector Table Offset

The Vector Table Offset Register positions the vector table in CODE or SRAM space. The default, on reset, is in CODE space. For the sake of the interrupt latency, put the vector table in CODE space. This is done by setting the TBLBASE bit in VTOR register to “0”, the code is as follows:

```
extern uint32_t _sfixed;
uint32_t*pSrc;
pSrc = (uint32_t *)&_sfixed;
SCB->VTOR = ( (uint32_t)pSrc & SCB_VTOR_TBLOFF_Msk ) ;

if ( ((uint32_t)pSrc >= IRAM_ADDR) && ((uint32_t)pSrc <
IRAM_ADDR+IRAM_SIZE) )
{
    SCB->VTOR |= 1 << SCB_VTOR_TBLBASE_Pos ;
}
```

3.2.2 Debug Message Implementation

The UART peripheral is used to print debug messages. Refer to Section 3.2.9 “Using the UART” for detailed UART operations.

3.2.3 Using the Watchdog

The Watchdog peripheral is enabled by default after a processor reset. If the application does not use it, which is the case in this example, then it is disabled in the Watchdog Mode Register (WDT_MR):

```
WDT_Disable( WDT ) ;
or
WDT->WDT_MR = WDT_MR_WDDIS;
```

Note, the Watchdog Mode Register (WDT_MR) can be written only once. The Watchdog has been disabled at the very beginning of the application.

3.2.4 Using Generic Peripherals

3.2.4.1 Initialization

Most peripherals are initialized by performing four actions

- Enabling the peripheral clock in the PMC
- Enabling the control of the peripheral on PIO pins
- Configuring the interrupt source of the peripheral in the NVIC
- Enabling the interrupt source at the peripheral level

Most peripherals are not clocked by default. This makes it possible to reduce the power consumption of the system at startup. However, it requires that the programmer explicitly enable the peripheral clock. This is done in the Power Management Controller (PMC). Exception is made for the System Controller (which comprises several different controllers), as it is continuously clocked.

For peripherals which need to use one or more pins of the chip as external inputs/outputs, it is necessary to configure the Parallel Input/Output controller first. This operation is described in more detail in Section 3.2.8 “Using the Parallel Input/Output Controller”.

Finally, if an interrupt is to be generated by the peripheral, then the source must be configured properly in the Nested Vectored Interrupt Controller. Refer to Section 3.2.5 “Using the Nested Vectored Interrupt Controller” for more information.

3.2.5 Using the Nested Vectored Interrupt Controller

3.2.5.1 Purpose

The NVIC manages all internal and external interrupts of the system. It enables the definition of one handler for each interrupt source, i.e., a function which is called whenever the corresponding event occurs. Interrupts can also be individually enabled or masked, and have several different priority levels.

In the example software, using the NVIC is required because several interrupt sources are present (see Section 3.1.2 “Peripherals”). The NVIC functions are implemented using the “Core Peripheral Access Layer” from the Cortex Microcontroller Software Interface Standard (CMSIS). For further details on CMSIS, refer to http://www.arm.com/products/CPUs/CMSIS.html.

3.2.5.2 *Initialization*

Unlike most other peripherals, the NVIC is always clocked and cannot be shut down. Therefore, there is no enable/disable bit for NVIC clock in the PMC.

For debug purposes, it is good practice to use dummy handlers (i.e., which loop indefinitely) for all interrupt sources (see Section 3.2.1.1 “Vector Table”). This way, if an interrupt is triggered before being configured, the debugger is stuck in the handler instead of jumping to a random address.

3.2.5.3 *Configuring an Interrupt*

Configuring an interrupt source requires six steps:

- Implementing interrupt handler
- Disable the interrupt in case it was enabled
- Clear any pending interrupt if any
- Configure the interrupt priority
- Enable the interrupt at the peripheral level
- Enable the interrupt at NVIC level

The first step is to re-implement the interrupt handler with the same name as the default interrupt handler in the vector table (see Section 3.2.1.1 “Vector Table”). So that when the corresponding interrupt occurs, the re-implemented interrupt handler will be executed instead of the default interrupt handler (see Section 3.2.1.3 “Vectors: Exception Handlers”).

An interrupt triggered during configuration may result in unpredictable behavior of the system. Therefore, the interrupt should be disabled. To disable the interrupt, Interrupt Clear-enable Register (ICER0) of the NVIC must be written with the interrupt source ID to mask it. Refer to the “SAM3N Series Datasheet” for a list of peripheral IDs.

Setting the Interrupt Clear-pending Register bit puts the corresponding pending interrupt in the inactive state. It is also written with the interrupt source ID to mask it.

Use the Interrupt Priority Registers to assign a priority from 0 to 15 for each interrupt. A higher level corresponds to a lower priority, so level 0 is the highest interrupt priority. The priority registers are stored with the Most Significant Bit (MSB) first. This means that bits[7:4] of the byte store the priority value, and bits[3:0] of the byte read as zero and ignore writes. For details of the priority, refer to the Cortex-M3 TechnicalReference Manual” and “SAM3N Series Datasheet”.

Finally, the interrupt source can be enabled, both on the peripheral (in a mode register usually) and in the Interrupt Set-enable Register (ISER0) of the NVIC. At this point, the interrupt is fully configured and operational.

3.2.6 Using the Timer Counter

3.2.6.1 *Purpose*

Timer Counters on SAM3N devices can perform several functions, e.g., frequency measurement, pulse generation, delay timing, Pulse Width Modulation (PWM), etc.

In this example, a single Timer Counter (TC) channel provides a fixed-period delay. An interrupt is generated each time the timer expires, toggling the associated LED on or off. This makes the LED blink at a fixed rate.

3.2.6.2 Initialization

In order to reduce power consumption, most peripherals are not clocked by default. Writing the ID of a peripheral in the Peripheral Clock Enable Register (PMC_PCER) of the Power Management Controller activates its clock. This is the first step when initializing the Timer Counter.

The Timer Counter is then disabled, in case it has been turned on by a previous execution of the program. This is done by setting the CLKDIS bit in the corresponding Channel Control Register (TC_CCR). In the example, timer channel 0 is used.

The next step is to configure the Channel Mode Register (TC_CMR). TC channels can operate in two different modes.

- Capture mode, is normally used for performing measurements on input signals.
- Waveform mode, enables the generation of pulses.

In the example, the purpose of the TC is to generate an interrupt at a fixed rate. Actually, such an operation is possible in both Capture and Waveform modes. Since no signal is being sampled or generated, there is no reason to choose one mode over the other. However, setting the TC in Waveform mode and outputting the tick on TIOA or TIOB can be helpful for debugging purpose.

Setting the CPCTRG bit of TC_CMR resets the timer and restarts its clock every time the counter reaches the value programmed in the TC Register C (TC_RC). Generating a specific delay is done by choosing the correct value for TC_ RC. It is also possible to choose between several different input clocks for the channel, which in practice makes it possible to prescale MCK. Since the timer resolution is 16 bits, using a high prescale factor may be necessary for bigger delays.

Consider the following example: the timer must generate a 500 ms delay with a 48 MHz main clock frequency. RC must be equal to the number of clock cycles generated during the delay period. Below are the results with different prescaling factors:

$$Clock = \frac{MCK}{2},\ RC = 24000000 \times 0.5 = 12000000$$

$$Clock = \frac{MCK}{8},\ RC = 6000000 \times 0.5 = 3000000$$

$$Clock = \frac{MCK}{128},\ RC = 375000 \times 0.5 = 187500$$

$$Clock = \frac{MCK}{1024},\ RC = 46875 \times 0.5 = 23437.5$$

$$Clock = 32kHz,\ RC = 32768 \times 0.5 = 16384$$

Since the maximum value for RC is 65535, it is clear from these results that using MCK divided by 1024 or the internal slow clock is necessary for generating long (about 1s) delays. In the example, a 250 ms delay is used. This means that the slowest possible input clock is selected in the CMR, and the corresponding value written in RC. The following two operations configure a 250 ms period by selecting the slow clock and dividing its frequency by 4:

```
TC0->TC_CHANNEL[0].TC_CMR = TC_CMR0_TCCLKS_TIMER_CLOCK5
                          | TC_CMR0_CPCTRG;
TC0->TC_CHANNEL[0].TC_RC = SLOW_CLOCK >> 2;
```

The last initialization step is to configure the interrupt whenever the counter reaches the value programmed in TC_ RC. At the TC level, this is easily done by setting the CPCS bit of the Interrupt Enable Register. Refer to Section 3.2.5.3 “Configuring an Interrupt” for more information on configuring interrupts in the NVIC.

3.2.6.3 *Interrupt Handler*

The first action to do in the interrupt handler is to acknowledge the pending interrupt from the peripheral. Otherwise, the latter continues to assert the interrupt line. In the case of a Timer Counter channel, acknowledgement is done by reading the corresponding Status Register (SR).

Special care must be taken to avoid having the compiler optimize away a dummy read to this register. In C, this is accomplished by declaring a *volatile* local variable and setting it to the register content. The *volatile* keyword tells the compiler to never optimize accesses (read/write) to a variable.

The rest of the interrupt handler is straightforward. It toggles the state (on or off) of one of the blinking LEDs. Refer to Section 3.2.8 “Using the Parallel Input/Output Controller” for more details on how to control LEDs with the PIO controller.

3.2.7 Using the System Timer (SysTick)

3.2.7.1 *Purpose*

The primary goal of the System Timer (SysTick) is to generate periodic interrupts. This is most often used to provide the base tick of an operating system. The SysTick can select its clock source, in this software example, the core clock (Master Clock) is selected as the SysTick input clock. The SysTick has a 24-bit counter. The start value to load into the SysTick Current Value Register (VAL) is called reload value, and is stored in the SysTick Reload Value Register (LOAD). Each time the counter in VAL reaches 0, an interrupt is generated, and the value stored in LOAD is loaded into VAL.

The getting started example uses the SysTick to provide a 1 ms time base. Each time the SysTick interrupt is triggered, a 32-bit counter is incremented. A Wait function uses this counter to provide a precise way for an application to suspend itself for a specific amount of time.

3.2.7.2 *Initialization*

Since the SysTick is part of the System Controller, it is continuously clocked. As such, there is no need to enable its peripheral clock in the PMC.

The first step is to disable the SysTick and select the clock source by setting the SysTick Control and Status Register (SysTick_CTRL):

```
SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk;
```

Before starting SysTick, the value in the SysTick Current Value Register (VAL) should be cleared and reload value should be stored in the SysTick Reload Value Register (LOAD). Given the SysTick source clock is MCK, in order to generate a 1ms interrupt, the reload value should be MCK/1000. A partial example follows:

```
reloadValue = BOARD_MCK/1000;
SysTick->VAL &= ~AT91C_NVIC_STICKCURRENT;
SysTick->LOAD = reloadValue;
```

The SysTick then can be enabled with the SysTick interrupt enabled by setting CTRL:

```
SysTick->CTRL  = SysTick_CTRL_CLKSOURCE_Msk |
```

```
                                SysTick_CTRL_TICKINT_Msk   |
                                SysTick_CTRL_ENABLE_Msk;
```

The application operation is executed by invoking the CMSIS function, SysTick_Config(), as shown below:

```
if ( SysTick_Config( BOARD_MCK / 1000 ) )
{
    printf("-F- Systick configuration error\n\r" ) ;
}
```

The function also enables SysTick interrupt.

3.2.7.3 *Interrupt Handler*

By default, the interrupt handler of the SysTick is implemented as an infinite loop. The application should re-implement it (see Section 3.2.1.3 “Vectors: Exception Handlers”).

In the handler, a 32-bit counter is increased when SysTick interrupt occurs.

Using a 32-bit counter may not always be appropriate, depending on how long the system should stay up and on the tick period. In the example, a 1 ms tick overflows the counter after about 50 days which may not be enough for a real application. In that case, a larger counter can be implemented.

3.2.7.4 *Wait Function*

Using the global counter, a wait function taking a number of milliseconds as its parameter is very easy to implement.

When called, the function first saves the current value of the global counter in a local variable. It adds the requested number of milliseconds which has been given as an argument. Then, it loops until the global counter becomes equal to or greater than the computed value.

For proper implementation, the global counter must be declared with the *volatile* keyword in C. Otherwise, the compiler might decide that being in a empty loop prevents the modification of the counter; obviously, this is not the case since it can be altered by the interrupt handler.

3.2.8 Using the Parallel Input/Output Controller

3.2.8.1 *Purpose*

Most pins on SAM3N microcontrollers can either be used by a peripheral function (e.g. USART, SPI, etc.) or used as generic input/outputs. All those pins are managed by one or more **Parallel Input/Output (PIO)** controllers.

A PIO controller enables the programmer to configure each pin as used by the associated peripheral or as a generic I/O. As a generic I/O, the pin level can be read/written using several registers of the PIO controller. Each pin can also have an internal pull-up activated individually.

In addition, the PIO controller can detect a status change on one or more pins, optionally triggering an interrupt whenever this event occurs. Note that the generated interrupt is considered **internal** by the NVIC, so it must be configured as level-sensitive (see Section 3.2.5.3 “Configuring an Interrupt”).

In this example, the PIO controller manages two LEDs and two buttons. The buttons are configured to trigger an interrupt when pressed (as defined in Section 3.1.1 “Features”).

3.2.8.2 Initialization

There are two steps for initializing the PIO controller.

1. Its peripheral clock must be enabled in the PMC
2. its interrupt source can be configured in the NVIC

3.2.8.3 Configuring LEDs

The two PIOs connected to the LEDs must be configured as outputs, in order to turn them on or off. First, the PIOC control must be enabled in PIO Enable Register (PIO_PER) by writing the value corresponding to a logical OR between the two LED IDs.

PIO direction is controlled using two registers: Output Enable Register (PIO_OER) and Output Disable Register (PIO_ODR). Since both PIOs must be outputs, the same value written in PIO_PER must be written in PIO_OER.

Note that there are individual internal pull-ups on each PIO pin. These pull-ups are enabled by default. Since they are useless for driving LEDs, they should be disabled to reduce power consumption. This is done through the Pull Up Disable Register (PIO_PUDR) of PIOA.

Here is the code for LED configuration:

```
/* Configure the pins as outputs */
PIOA->PIO_OER = (LED_A | LED_B);
/* Enable PIOC control on the pins*/
PIOA->PIO_PER = (LED_A | LED_B);
/* Disable pull-ups */
PIOA->PIO_PUDR = (LED_A | LED_B);
```

3.2.8.4 Controlling LEDs

LEDs are turned on or off by changing the level on the PIOs to which they are connected. After those PIOs have been configured, their output values can be changed by writing the pin IDs in the Set Output Data Register (PIO_SODR) and the Clear Output Data Register (PIO_CODR) of the PIO controller.

In addition, a register indicates the current level on each pin, Output Data Status Register (PIO_ODSR). It can be used to create a toggle function, i.e. when the LED is ON according to PIO_ODSR, then it is turned off, and vice-versa.

```
/* Turn LED off */
PIOA->PIO_SODR = LED_A;
/* Turn LED on */
PIOA->PIO_CODR = LED_A;
```

3.2.8.5 Configuring Buttons

As stated previously, the two PIOs connected to the switches on the board are inputs. Also, a “state change” interrupt is configured for both buttons. This triggers an interrupt when a button is pressed or released.

After the PIO control has been enabled on the PIOs (by writing PIO_PER), they are configured as inputs by writing their IDs in PIO_ ODR. Conversely to the LEDs, it is necessary to keep the pull-ups enabled.

SAM3N4 has an internal hardware debouncing filter to remove intermediate noise states from inputs, as described in Section 3.2.8.6 “Configuring Input Debouncing”.

Enabling interrupts on the two pins is done in the Interrupt Enable Register (PIO_IER). However, the PIO controller interrupt must be configured as described in Section 3.2.5.3 “Configuring an Interrupt”.

3.2.8.6 *Configuring Input Debouncing*

Optional input debouncing filters are independently programmable on each I/O line. It can filter a pulse of less than 1/2 Period of a Programmable Divided Slow Clock.

The input filter is enabled by writing to the Input Filter Enable Register (PIO_IFER).

```
pio->PIO_IFER = mask;
```

The debouncing filter is selected by writing to the Input Filter Slow Clock Enable Register (PIO_IFSCER). The Input Filter Slow Clock Status Register (PIO_IFSCSR) holds current selection status.

```
pio->PIO_IFSCER = mask;
```

For the debouncing filter, the Period of the Divided Slow Clock is performed by writing in the DIV field of the Slow Clock Divider Register (PIO_SCDR). The following formula can be used to compute the value of DIV, given the Slow Clock frequency and the desired cut off frequency:

$$DIV = \frac{SlowClock}{2 \times CutOffFrequency} - 1$$

For example, 100 ms debounce time, or 10 Hz noise cut-off frequency can be obtained with a 32768 Hz Slow Clock frequency by writing a value of 1637 in PIO_SCDR.

3.2.8.7 *Interrupt Handler*

The interrupt handler for the PIO controller must first check which button has been pressed. The Pin Data Status Register (PIO_PDSR) indicates the level on each pin, so it can show if each button is currently pressed or not. Alternatively, the Interrupt Status Register (PIO_ISR) reports which PIOs have had their status changed since the last read of the register.

In the example software, the two are combined to detect a state change interrupt as well as a particular level on the pin. This corresponds to either the press or the release action on the button.

As said in the application description (Section 3.1.1 “Features”), each button enables or disables the blinking of one LED. Two variables are used as boolean values, to indicate if either LED is blinking. When the status of the LED which is toggled by the SysTick is modified, the value is modified in the interrupt handler as well.

Note: The interrupt must be acknowledged in the PIOA. This is done implicitly when PIO_ISR is read by the software.

3.2.9 Using the UART

3.2.9.1 *Purpose*

The Universal Asynchronous Receiver and Transmitter (UART) provides a two-pin UART as well as several other debug functionalities. The UART is ideal for outputting debug traces on a terminal, or as an In-System Programming (ISP) communication port. Other features include chip identification registers, management of debug signals from the ARM core, and so on.

In the example, the UART is used to output a single string of text whenever the application starts. It is configured with a baudrate of 115200 bps, 8 bits of data, no parity, one stop bit and no flow control.

3.2.9.2 Initialization

Writing the UART ID in the Peripheral Clock Enable Register (PMC_PCER) of the Power Management Controller activates its clock. This is the first step when initializing the UART. This is done once before the debug console is used, as follows:

```
PMC_EnablePeripheral(ID_UART0);
or
PMC->PMC_PCER = 1 << ID_UART0;
```

It is also necessary to configure its two pins (UTXD and URXD) in the PIO controller. Writing both pin IDs in the PIO Disable Register (PIO_PDR) of the corresponding PIO controller enables peripheral control on those pins. However, some PIOs are shared between two different peripherals, Peripheral ABCD Select Register (PIO_ABCDSR) is used to switch control between the two.

```
PIOA->PIO_ABCDSR[0] &= ~(UTXD|URXD);
PIOA->PIO_ABCDSR[1] &= ~(UTXD|URXD);
PIOA->PIO_PDR = UTXD | URXD;
```

The next action to perform, is to disable the receiver and transmitter logic, as well as disable interrupts. This enables smooth reconfiguration of the peripheral in case it had already been initialized during an execution of the application. Setting bits RSTRX and RSTTX in the UART Control Register (UART_CR) resets and disables the receiver and transmitter, respectively. Setting all bits of the Interrupt Disable Register (UART_IDR) disables all interrupts coming from the UART.

The baud rate clock must now be set up. The input clock is equal to MCK divided by a programmable factor. The Clock Divisor value is held in the Baud Rate Generate Register (UART_BRGR). The following values are possible:

Table 3-4. Possible Values for the Clock Divisor field of BRGR

Value	Comment
0	Baud rate clock is disabled
1	Baud rate clock is MCK divided by 16
2 to 65535	Baud rate clock is MCK divided by (CD x 16)

The following formula can be used to compute the value of CD given the microcontroller operating frequency and the desired baud rate:

$$CD = \frac{MCK}{16 \times Baudrate}$$

For example, a 115200 baud rate can be obtained with a 48 MHz master clock frequency by writing a value of 26 in CD. Obviously, there is a slight deviation from the desired baudrate; these values yield a true rate of 115384 bauds. However, it is a mere 1.6% error, with no impact in practice.

The Mode Register (UART_MR) has two configurable values:

- The Channel Mode in which the UART is operating. Several modes are available for testing purposes. In the example, only the normal mode is of interest. Setting the CHMODE field to a null-value selects the normal mode.
- The Parity bit. Even, odd, mark and space parity calculations are supported. In the example, no parity bit is being used (PAR value of 1xx).

The UART features its own Peripheral DMA Controller (PDC). It enables faster data transfer and reduces the processor overhead by taking care of most of the transmission and reception operations. The PDC is not used in this example, so it should be disabled by setting bits RXTDIS and TXTDIS in the PDC Transfer Control Register (PERIPH_PTCR).

At this point the UART is fully configured. The last step is to enable the transmitter. The receiver is not used in this demo application, so it is useless (but not harmful) to enable it as well. The transmitter is enabled by setting TXEN bit in the UART Control Register.

3.2.9.3 Sending a Character

Transmitting a character on the UART line is simple, provided that the transmitter is ready. Write the character value in the Transmit Holding Register (UART_THR) to start the transfer.

Two bits in the UART Status Register (UART_SR) indicate the transmitter state. The TXEMPTY bit indicates if the transmitter is enabled and sending characters. If it is set, no character is being currently sent on the UART line.

The second meaningful bit is TXRDY. When TXRDY is set, the transmitter has finished copying the value of UART_THR in its internal shift register which is used to send data. In practice, this means that UART_THR can be written when TXRDY is set, regardless of the value of TXEMPTY. When TXEMPTY rises, the whole transfer is finished.

3.2.9.4 String Print Function

A *printf()* function is defined in the example application. It takes a string pointer as an argument, and sends it across the UART.

Its operation is quite simple. C-style strings are simple byte arrays terminated by a null (0) value. Thus, the function just loops and outputs all the characters of the array until a zero is encountered.

4. Building the Project

The development environment for this project is a PC running Microsoft Windows OS.

The required software tools for building the project and loading the binary file are:

- an ARM cross-compiler toolchain
- SAM-BA 2.10 or later (available at www.atmel.com).

The connection between the PC and the board is achieved with a JTAG cable.

4.1 ARM Compiler Toolchain

The CodeSourcery GNU ARM compiler toolchain (www.codesourcery.com).is used to generate the binary file to be downloaded into the target. CodeSourcery supports the Cortex-M3 since release 2008-q3.

This toolchain provides an ARM assembler, a compiler, and linker tools. Useful debug programs are also included.

Another software, the *make* utility, but not included in the CodeSourcery package, is also required. The GnuWin32 project (http://gnuwin32.sourceforge.net/) provides native ports of GNU tools for Windows, including makefile.

4.1.1 Makefile

The makefile contains rules indicating how to assemble, compile and link the project source files to create a binary file ready to be downloaded on the target.

The makefile is divided into two parts:

- Variable settings
- Rules implementation

4.1.1.1 Variables

The first part of the makefile contains variables (uppercase), used to set up some environment parameters, such as the compiler toolchain prefix and program names, and options to be used with the compiler.

```
CHIP  = sam3n4
BOARD = sam3n_ek
```

- Defines the chip and board names.

```
MEMORIES = flash
```

- Defines the available memory targets for the board.

Note: There is only flash target available on SAM3N-EK board because SRAM memory size is too small for the program.

```
TRACE_LEVEL = 4
```

- Defines trace level used for compilation

```
OPTIMIZATION = -Os
```

- Level of optimization used during compilation (-Os optimizes for size).

```
OUTPUT=getting_started_$(BOARD)_$(CHIP)
```

- Defines the outfile name (with board and chip names).

```
BIN = bin
OBJ = obj
```

- Defines output directory.

```
CROSS_COMPILE=arm-none-eabi-
```

- Defines the cross-compiler toolchain prefix.

```
LIBRARIES = ../../../../libraries
CHIP_LIB = $(LIBRARIES)/libchip_sam3n
BOARD_LIB = $(LIBRARIES)/libboard_sam3n-ek

LIBS = -Wl,--start-group -lgcc -lc -lchip_$(CHIP)_gcc_dbg -
lboard_$(BOARD)_gcc_dbg -Wl,--end-group
```

- Defines the library name and archives linker options:
  - -Wl,--start-group:archive group start
  - -lgcc: use gcc library.
  - -lc: use standard C library.
  - -lchip_$(CHIP)_gcc_dbg:use chip library.
  - -lboard_$(BOARD)_gcc_dbg:use board library.
  - -Wl,--end-group:archive group end

```
LIB_PATH = -L$(CHIP_LIB)/lib
LIB_PATH += -L$(BOARD_LIB)/lib
LIB_PATH+=-L=/lib/thumb2
LIB_PATH+=-L=/../lib/gcc/arm-none-eabi/4.4.1/thumb2
```

- Defines the library path.

```
INCLUDES  = -I$(CHIP_LIB)
INCLUDES += -I$(BOARD_LIB)
INCLUDES += -I$(LIBRARIES)
```

- Paths for header files.

```
CC = $(CROSS_COMPILE)gcc
LD = $(CROSS_COMPILE)ld
SIZE = $(CROSS_COMPILE)size
STRIP = $(CROSS_COMPILE)strip
OBJCOPY = $(CROSS_COMPILE)objcopy
GDB = $(CROSS_COMPILE)gdb
NM = $(CROSS_COMPILE)nm
```

- Names of cross-compiler toolchain binutils (compiler, symbol list extractor, etc.).

```
CFLAGS += -Wall
CFLAGS += -Dprintf=iprintf
CFLAGS += --param max-inline-insns-single=500 -mcpu=cortex-m3 -mthumb -
ffunction-sections
CFLAGS += -g $(OPTIMIZATION) $(INCLUDES) -D$(CHIP) -
DTRACE_LEVEL=$(TRACE_LEVEL)
```

- Compiler options:
  - -Wall: displays all warnings.
  - -Dprintf=iprintf: uses only integer printf function to reduce application size
  - --param max-inline-insns-single=500: Sets 500 the maximum number of instructions in a single function that the tree inliner will consider for inlining.
  - -mcpu = cortex-m3: type of ARM CPU core.
  - -mthumb: generate code for Thumb instruction set.
  - -ffunction-sections: place each function item into its own section in the output file if the target supports arbitrary sections.
  - -g: generate debugging information for GDB usage.
  - $(OPTIMIZATION): set optimization option.
  - $(INCLUDES): set paths for include files.
  - -D$(CHIP): define chip names used for compilation.
  - -DTRACE_LEVEL=$(TRACE_LEVEL): define trace level used for compilation

```
ASFLAGS = -mcpu=cortex-m3 -mthumb -Wall -g $(OPTIMIZATION) $(INCLUDES) -
D$(CHIP) -D__ASSEMBLY__
```

- Assembler options:
  - -D__ASSEMBLY__ : defines the __ASSEMBLY__ symbol, which is used in header files to distinguish inclusion of the file in assembly code or in C code.

```
LDFLAGS= -mcpu=cortex-m3 -mthumb -Wl,--cref -Wl,--check-sections -Wl,--gc-
sections -Wl,--entry=ResetException -Wl,--unresolved-symbols=report-all -
Wl,--warn-common -Wl,--warn-section-align -Wl,--warn-unresolved-symbols
```

- Linker options:
  - -Wl,--cref: Pass the --cref option to the linker (generates cross-reference in map file if this one is requested).
  - -Wl,--gc-sections: Pass the --gc-sections option to the linker.
  - -Wl,--entry=ResetException: Use ResetException as the explicit symbol for beginning execution of the program.

```
C_OBJECTS=main.o
```

- List of all object file names.

For more detailed information about GNU Compiler Collection (GCC) options, refer to GCC documentation available at gcc.gnu.org.

4.1.1.2 *Rules*

The second part of the makefile contains rules. Each rule is a line composed of a target name, and the files needed to create this target.

The following rules create the one object file from the corresponding source files. The option -c tells GCC to run the compiler and assembler, but not the linker.

```
main.o: main.c
     $(CC) -c $(CFLAGS) main.c -o main.o
```

The “all” rule is the default rule used by the make command when none is specified on the command line. It describes how to compile source files and link object files together. The first line calls the linker with the defined flags, and linker files used with -T option. This generates an elf format file, which is converted to a binary file without any debug information, by using the objcopy program. An example of Flash configuration is given below:

```
flash: $$(ASM_OBJECTS_$(1)) $$(C_OBJECTS_$(1))
  @$(CC) $(LIB_PATH) $(LDFLAGS) $(LD_OPTIONAL) -
T"$(BOARD_LIB)/resources/gcc/$(CHIP)/$$@.ld" -Wl,-Map,$(OUTPUT)-$$@.map -o
$(OUTPUT)-$$@.elf $$^ $(LIBS)
  $(NM) $(OUTPUT)-$$@.elf >$(OUTPUT)-$$@.elf.txt
  $(OBJCOPY) -O binary $(OUTPUT)-$$@.elf $(OUTPUT)-$$@.bin
  $(SIZE) $$^ $(OUTPUT)-$$@.elf
```

4.1.2 Linker File

This file describes the order in which the linker must put the different memory sections into the binary file.

4.1.2.1 *Header*

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
```

Set the object file format to elf32-littlearm.

```
OUTPUT_ARCH(arm)
```

Specify the machine architecture.

4.1.2.2 *Section Organization*

The MEMORY part defines the start and size of each memory that the application will use.

The *SECTION* part deals with the different sections of code used in the project. It tells the linker where to put the sections it finds while parsing all the project object files.

.vectors: exception vector table and IRQ handler

.text: code

.data: initialized data

.bss: uninitialized data

.stack: stack space definition

```
/* Memory Spaces Definitions */
MEMORY
{
```

```
  rom (rx)  : ORIGIN = 0x00400000, LENGTH = 0x00040000 /* flash, 256K */
  ram (rwx) : ORIGIN = 0x20000000, LENGTH = 0x00006000 /* sram, 24K */
}

/* The stack size used by the application. NOTE: you need to adjust  */
STACK_SIZE = 2048;

/* Section Definitions */
SECTIONS
{
    .text :
    {
        . = ALIGN(4);
        _sfixed = .;
        KEEP(*(.vectors .vectors.*))
        *(.text .text.* .gnu.linkonce.t.*)
        *(.glue_7t) *(.glue_7)
        *(.rodata .rodata* .gnu.linkonce.r.*)
        *(.ARM.extab* .gnu.linkonce.armextab.*)

        /* Support C constructors, and C destructors in both user code
           and the C library. This also provides support for C++ code. */
        . = ALIGN(4);
        KEEP(*(.init))
        . = ALIGN(4);
        __preinit_array_start = .;
        KEEP (*(.preinit_array))
        __preinit_array_end = .;

        . = ALIGN(4);
        __init_array_start = .;
        KEEP (*(SORT(.init_array.*)))
        KEEP (*(.init_array))
        __init_array_end = .;

        . = ALIGN(0x4);
        KEEP (*crtbegin.o(.ctors))
        KEEP (*(EXCLUDE_FILE (*crtend.o) .ctors))
        KEEP (*(SORT(.ctors.*)))
        KEEP (*crtend.o(.ctors))

        . = ALIGN(4);
        KEEP(*(.fini))

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

```
    KEEP (*(.fini_array))
    KEEP (*(SORT(.fini_array.*)))
    __fini_array_end = .;

    KEEP (*crtbegin.o(.dtors))
    KEEP (*(EXCLUDE_FILE (*crtend.o) .dtors))
    KEEP (*(SORT(.dtors.*)))
    KEEP (*crtend.o(.dtors))

    . = ALIGN(4);
    _efixed = .;            /* End of text section */
} > rom

/* .ARM.exidx is sorted, so has to go in its own output section.  */
PROVIDE_HIDDEN (__exidx_start = .);
.ARM.exidx :
{
  *(.ARM.exidx* .gnu.linkonce.armexidx.*)
} > rom
PROVIDE_HIDDEN (__exidx_end = .);

. = ALIGN(4);
_etext = .;

.relocate : AT (_etext)
{
    . = ALIGN(4);
    _srelocate = .;
    *(.ramfunc .ramfunc.*);
    *(.data .data.*);
    . = ALIGN(4);
    _erelocate = .;
} > ram

/* .bss section which is used for uninitialized data */
.bss (NOLOAD) :
{
    . = ALIGN(4);
    _sbss = . ;
    _szero = .;
    *(.bss .bss.*)
    *(COMMON)
    . = ALIGN(4);
    _ebss = . ;
    _ezero = .;
} > ram
```

```
        /* stack section */
        .stack (NOLOAD):
        {
             . = ALIGN(8);
            _sstack = .;
            . = . + STACK_SIZE;
            . = ALIGN(8);
            _estack = .;
        } > ram

        . = ALIGN(4);
        _end = . ;
    }
```

In the *.text* section, the *_sfixed* symbol is set in order to retrieve this address at runtime, then all *.text*, and *.rodata* sections as well as *.vectors section* found in all object files, are placed here, the *_efixed* symbol is set and aligned on a 4-byte address.

The same operation is done with the *.relocate* and *.bss* sections.

In the *.relocate* section, the *AT (_etext)* command specifies that the load address (the address in the binary file after link step) of this section is just after the *.text* section. Thus there is no empty space between these two sections.

4.1.3 Libraries

The SAM3N software package includes the following libraries:

- libchip_sam3n: chip specific library
- libboard_sam3n-ek: board specific library
- libqtouch: qtouch library
- libfreertos: freertos library

Note: libchip_sam3n and libboard_sam3n-ek should be compiled previously to generate the libraries which are used by the Getting Started example.

4.2 Loading the Code

Once the build step is completed, one .bin file is available and ready to be loaded into the board.

The AT91-ISP solution offers an easy way to download files into AT91SAM products on Atmel Evaluation Kits through a USB, COM or JTAG link. Target programming is done via SAM-BA tools.

Follow the steps below to launch the SAM-BA program:

- Shut down the board.
- Set the JP3 jumper on the board with power up to erase the internal Flash.
- Shut down the board and remove the jumper.
- Plug the JTAG cable between the PC and the board and power up the board.
- Execute SAM-BA.exe to launch SAM-BA.

Follow the steps below to download code into Flash:

- Execute “Enable Flash” to enable Flash access.
- Download the binary “getting_started_sam3n_ek_sam3n4-flash.bin” into the flash.
- Execute “Boot from flash”.
- Shut down SAM-BA and power off the board.
- Power on the board to run the binary.

The code then starts running, and the LEDs are now controlled by two push buttons.

4.3 Debug Support

A SEGGER GDB Server should be launched prior to debugging. A GDB command file named "sam3n_ek_flash.gdb" is provided in the package for easy debugging in Flash. It connects to GDB Server, then loads the application. In order to simplify command line debugging using GDB, type “make debug_flash” in the CMD shell to start the compilation, then launch the debug session and load the application.

When debugging the Getting Started example with GDB, it is best to disable compiler optimizations. Otherwise, the source code will not correctly match the actual execution of the program. To do that, comment out (with a ‘#’) the “OPTIMIZATION = -Os” line of the makefile and rebuild the project.

For more information on debugging with GDB, refer to the Atmel application note GNU-Based Software Development and to the GDB manual available on gcc.gnu.org.

Revision History

Doc. Rev	Comments	Change Request Ref.
11097A 09-Dec-10	First issue	

Headquarters

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: 1(408) 441-0311
Fax: 1(408) 487-2600

International

Atmel Asia
Unit 1-5 & 16, 19/F
BEA Tower, Millennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
Hong Kong
Tel: (852) 2245-6100
Fax: (852) 2722-1369

Atmel Europe
Le Krebs
8, Rue Jean-Pierre Timbaud
BP 309
78054 Saint-Quentin-en-
Yvelines Cedex
France
Tel: (33) 1-30-60-70-00
Fax: (33) 1-30-60-71-11

Atmel Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
Tel: (81) 3-3523-3551
Fax: (81) 3-3523-7581

Product Contact

Web Site
www.atmel.com
www.atmel.com/AT91SAM

Technical Support
AT91SAM Support
Atmel techincal support

Sales Contacts
www.atmel.com/contacts/

Literature Requests
www.atmel.com/literature