Atmel®

APPLICATION NOTE

AT03499: Getting Started With SAM3S Microcontrollers

AT91 ARM Cortex-M3 Based Microcontrollers

Introduction

This application note is aimed at helping the reader become familiar with the Atmel® ARM® Cortex®-M3 based SAM3S microcontroller.

It describes in detail a simple project that uses several important features present on SAM3S chips. This includes how to set up the microcontroller prior to executing the application, as well as how to add the functionalities themselves. After going through this guide, the reader should be able to successfully start a new project from scratch.

This document also explains how to set up 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™ EWARM and Keil MDK; the reader should disregard Chapter 3, Building the Project on page 19 when using these versions.

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

IMPORTANT NOTE: This application note was originally written for the SAM3S-EK evaluation kit, which demonstrates the AT91SAM3S4. The SAM3S-EK is now replaced by the SAM3S-EK2 evaluation kit, which demonstrates the AT91SAM3SD8. The kits and devices are compatible.

Table of Contents

1. Requirements

The software provided with this application note requires several components:

- The SAM3S Evaluation Kit
- A computer running Microsoft® Windows® 2000/XP or later
- An ARM cross-compiler toolchain support ARM Cortex-M3 (such as Codesourcery-2008-q3)
- AT91-ISP V1.13 or later

2. Getting Started With a Software Example

This chapter describes how to program a basic application that helps you to become familiar with the SAM3S microcontrollers. It is divided into two main sections; the first one covers the specification of the example (what it does, what peripherals are used) and the other details the programming aspect.

2.1 Specification

2.1.1 Features

The demonstration program makes two LEDs on the board blink at a fixed rate. This rate is generated by using a timer for the first LED and the second one uses a Wait function based on a 1ms tick generated by using the System Tick Timer (SysTick). The blinking can be stopped by 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 makes a good starting point for someone wanting to become familiar with the AT91SAM microcontroller series.

2.1.2 Peripherals

In order to perform the operations described in the previous section, the software example uses the following set of peripherals:

- Parallel Input/output (PIO) controller
- Timer Counter (TC)
- Universal asynchronous Receiver Transmitter (UART)
- System Tick 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. Buttons are configured to have this behavior.

The TC and SysTick are used to generate two time bases in order to obtain the LED blinking rates. They are both used in interrupt mode; the TC triggers an interrupt at a fixed rate, each time 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 is used to output debug traces on a serial line; the UART. Having the firmware send debug traces at key points of the code can greatly help the debugging process and the UART can be easily disabled from Power Management Controller (PMC), by setting the UART bit in the register PMC_PCDR0.

2.1.3 Evaluation Kit

2.1.3.1 Memories

The AT91SAM3S4 found on SAM3S-EK evaluation board features one internal 48KB SRAM memory, 256KB Flash Memory block. In addition, it provides an external bus interface made of a Static Memory Controller (SMC) and a NAND Flash Controller (NFC), enabling the connection of external memories. A 256MB NAND Flash are present on the SAM3S-EK. The Getting Started example software can be compiled and loaded on the internal Flash and the internal SRAM.

2.1.3.2 Buttons

The AT91SAM3S4 Evaluation Kit features two push buttons respectively connected to pins PB3 and PC12. When pressed, they force a logical low level on the corresponding PIO line.

The Getting Started example uses both the buttons by using the internal hardware debouncing circuitry embedded in the SAM3S. Refer to Section 2.2.8.6 on page 17 for more details.

2.1.3.3 LEDs

There are two general-purpose LEDs (one is blue, the other is green) on the SAM3S-EK, as well as a software-controllable red power LED; they are wired to pins PA19, PA20 and PC20, respectively. Setting a logical low level on these PIO lines turns the corresponding LED on.

The example application uses the two general-purpose LEDs (PA19 and PA20).

2.1.3.4 UART

On the SAM3S-EK, the UART uses the PA9 and PA10 pins for the URXD and UTXD signals, respectively.

2.2 Implementation

As previously stated, 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.

2.2.1 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 sections.

2.2.1.1 Vector Table

The vector table contains the initialization value for the stack pointer (see Section 2.2.1.9, Initializing Stacks on page 11) on reset, and the entry point addressed for all exception handlers. The exception number (see Table 2-1 on page 6) defines the order of entries in the vector table associated with exception handler entry as illustrated in Table 2-2 on page 6.

Table 2-1. Exception Numbers.

Exception number	Exception
1	Reset
2	RESERVED
3	HardFault
4	MemManager
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 2-2. 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 Technical Reference Manual".

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

```
#include "exceptions.h"
/* Stack top */
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 */
    SMC_IrqHandler,     /* 10 SMC */
    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 */
    MCI_IrqHandler,     /* 18 MCI */
    TWI0_IrqHandler,    /* 19 TWI 0 */
    TWI1_IrqHandler,    /* 20 TWI 1 */
    SPI_IrqHandler,     /* 21 SPI */
    SSC_IrqHandler,     /* 22 SSC */
    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 */
    CRCCU_IrqHandler,   /* 32 CRC Calculation Unit */
    ACC_IrqHandler,     /* 33 Analog Comparator */
    USBD_IrqHandler,    /* 34 USB Device Port */
    IrqHandlerNotUsed   /* 35 not used */
};
```

2.2.1.2 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 2-3.

Table 2-3. Reset Exception Behavior.

Action	Description
Low-level initialization	Initialize critical peripherals
Initialize variables	Any global/static variables must be set up. 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 2.2.1.4 will introduce the above actions in detail.

2.2.1.3 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 used depends on the mode of the processor at the time of the exception. Refer to Section 2.2.1.9 on page 11 for more details about stacks.

Exception handler in the vector table corresponding with the exception number happens will be executed. If the program does not need to handle an exception, then the corresponding instruction can simply be set to an infinite loop. An example is like:

```
/**
 * 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 we add "WEAK" attribute to these exception handlers, we can re-implement them whenever we want in our applications.

For example, in exception.c, the default SysTick's exception handler is implemented like:

```
/* 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 we will use the SysTick exception in our example, we can re-implement the SysTick exception handler in our example like:

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

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

2.2.1.4 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.

2.2.1.5 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 EEFC.

After reset, the chip uses its internal 4MHz 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 (FMR) of the EEFC. For example, running the core clock at 64MHz operation requires the use of three wait states.

This example configures the core clock at 64MHz (PLLA output), using two wait states.

```
EFC->EEFC_FMR = (2 << 8);
```

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 corresponding datasheet.

2.2.1.6 Low-level Initialization: NRST

The Reset Controller (RSTC) embeds a 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 enable 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;
```

2.2.1.7 Low-level Initialization: Slow Clock

The Supply Controller (SUPC) embeds a slow clock generator, which 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 100µs).

The user can select the crystal oscillator to be the source of the slow clock, as it provides a more accurate frequency. This is made 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 | ((uint32_t)0xA5 << 24);
        timeout = 0;
        while (!(SUPC->SUPC_SR & SUPC_SR_OSCSEL_CRYST) );
    }
```

2.2.1.8 Low-level Initialization: Main Oscillator and PLL

After reset, the chip is running with 4MHz 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:

- 4/8/12MHZ RC Oscillator, which starts very quickly and is used at startup
- 3 to 20MHz 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 Main Oscillator Register (MOR) of the PMC starts the oscillator. Stabilization occurs when bit MOSCXTS of the PMC Status Register (PMC_SR) becomes set. The following piece of code performs these two operations:

```
if(!(PMC->CKGR_MOR & CKGR_MOR_MOSCSEL)) {
        PMC->CKGR_MOR = (0x37 << 16) | 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 "AT91 ARM Cortex-M3 based Microcontrollers SAM3S series Preliminary". Note that the internal slow clock of the AT91SAM3S4 is generated by using a RC oscillator. This must be taken into account as this impact the slow clock accuracy. Here is an example:

Table 2-4. Oscillator parameters.

Parameter	Value
RC oscillator frequency range in kHz	$20 \le f_{RC} \le 44$
Oscillator frequency range in MHz	$4 \le f_{OSC} \le 12$
Oscillator frequency on EK	$f_{OSC} = 12MHz$
Oscillator startup time	$1ms \le t_{STARTUP} \le 1.4ms$
Value for a 2ms startup	$OSCOUNT = \frac{42000 \times 0.0014}{8} = 8$

The second step is to switch the source of 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, selection done occurs when bit MOSCSEL of the PMC_SR becomes set:

```
PMC->CKGR_MOR = (0x37 << 16) | BOARD_OSCOUNT | CKGR_MOR_MOSCRCEN |
CKGR_MOR_MOSCXTEN | CKGR_MOR_MOSCSEL;
timeout = 0;
while (!(PMC->PMC_SR & PMC_SR_MOSCSELS) && (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, while the second one multiplies it. The *MUL* and *DIV* factors are set in the PLLA Register (PLLAR) 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. Here is an example given for the SAM3S-EK:

$f_{INPUT} = 12$

DIV = 3

MUL = (16 – 1) = 15

$f_{OUTPUT} = 12/3 \times 16 = 64MHz$

Like the main oscillator, a PLL startup time must also be provided. Again, it can be calculated by looking at the DC characteristics given in the datasheet of the corresponding microcontroller. After PLLAR is modified with the PLL configuration values, the software must wait for the PLL to become locked. This is done by monitoring the Status Register of the PMC:

```
PMC->CKGR_PLLAR = BOARD_PLLAR;
timeout = 0;
while (!(PMC->PMC_SR & PMC_SR_LOCKA) && (timeout++ < CLOCK_TIMEOUT));
```

Finally, the prescaling value of the main clock must be set, and the PLL output selected. Note that the prescaling value must be set first, to avoid having the chip run at a frequency higher than the maximum operating frequency defined in the AC characteristics. As such, this step is done using two register writes, with two loops to wait for the main clock to be ready.

```
PMC->PMC_MCKR = (BOARD_MCKR & ~PMC_MCKR_CSS) | PMC_MCKR_CSS_MAIN_CLK;
timeout = 0;
while (!(PMC->PMC_SR & PMC_SR_MCKRDY) && (timeout++ < CLOCK_TIMEOUT));

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

At this point, the chip is configured to run on the main clock at 64MHz with the PLLA at 64MHz.

2.2.1.9 Initializing Stacks

There are two stacks supported in ARMv7-M, 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 "Cortex-M3 Technical Reference Manual". SP_main locates at vector table offset "0" - see Section 2.2.1.1, Vector Table on page 5, it is selected and initialized on reset.

2.2.1.10 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) and 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 case when they have been declared with a value. Uninitialized 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 the Flash. For IAR Embedded Workbench® and Keil® MDK, refer to documents at 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 toolchain:

```
extern unsigned int _efixed;
extern unsigned int _srelocate;
extern unsigned int _erelocate;
extern unsigned int _szero;
extern unsigned int _ezero;
```

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

```
unsigned int *pSrc, *pDest;
pSrc = &_efixed;
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 not needed; however, it may 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;
}
```

2.2.1.11 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, we put the vector table in CODE space. This is made by setting the TBLBASE bit in the VTOR register to “0”, the code is like:

```
extern unsigned int _sfixed;
unsigned int *pSrc;
pSrc = (uint32_t *)&_sfixed;
SCB->VTOR = ((uint32_t)(pSrc)) & SCB_VTOR_MASK;
```

2.2.2 Debug Message Implementation

UART peripheral is used to print debug messages. Refer to Section 2.2.9, Using the UART on page 18 for detailed UART operations.

2.2.3 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, it shall be disabled in the Watchdog Mode Register (WDMR):

```
WDT->WDT_MR = WDT_MR_WDDIS;
```

Note: The watchdog should be initialized only once just before the application starts to run.

2.2.4 Using Generic Peripherals

2.2.4.1 Initialization

Most peripherals are initialized by performing the following 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 2.2.8, Using the Parallel Input/output Controller on page 16.

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 2.2.5, Using the Nested Vectored Interrupt Controller on page 12 for more information.

2.2.5 Using the Nested Vectored Interrupt Controller

2.2.5.1 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 2.1.2, Peripherals on page 4). The NVIC functions are implemented using the “Core Peripheral Access Layer” from the Cortex Microcontroller Software Interface Standard (CMSIS). For further details of the CMSIS, refer to http://www.arm.com/products/processors/cortex-m/cortex-microcontroller-software-interface-standard.php

2.2.5.2 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 2.2.1.1, Vector Table on page 5). This way, if an interrupt is triggered before being configured, the debugger is stuck in the handler instead of jumping to a random address.

2.2.5.3 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 2.2.1.1, Vector Table on page 5). So that when the corresponding interrupt occurs, the re-implemented interrupt handler will be executed instead of the default interrupt handler (see Section 2.2.1.3, Vectors: Exception Handlers on page 8).

An interrupt triggering at the same time may result in unpredictable behavior of the system. To disable the interrupt, the Interrupt Clear-Enable Register (ICER) of the NVIC must be written with the interrupt source ID to mask it. Refer to the corresponding 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. Details of the priority, refer to "Cortex-M3 Technical Reference Manual" and "AT91 ARM Cortex-M3 based Microcontrollers SAM3S series Preliminary".

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

2.2.6 Using the Timer Counter

2.2.6.1 Purpose

Timer Counters on AT91SAM chips can perform several functions, e.g., frequency measurement, pulse generation, delay timing, Pulse Width Modulation (PWM), etc.

In this example, a single Timer Counter channel is going to provide 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.

2.2.6.2 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 (PCER) of the Power Management Controller (PMC) activates its clock. This is the first step when initializing the Timer Counter.

The TC 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 (CCR). In the example, timer channel 0 is used.

The next step is to configure the Channel Mode Register (CMR). TC channels can operate in two different modes. The first one, which is referred to as the Capture mode, is normally used for performing measurements on input signals. The second one, the 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 the Capture and Waveform mode. 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 the CMR resets the timer and restarts its clock every time the counter reaches the value programmed in the TC Register C. Generating a specific delay is thus done by choosing the correct value for 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 500ms delay with a 64MHz main clock frequency. RC must be equal to the number of clock cycles generated during the delay period; here are the results with different prescaling factors:

Table 2-5. Clock prescaling.

Clock	Frequency
MCK/2	RC = 32000000 x 0.5 = 16000000
MCK/8	RC = 8000000 x 0.5 = 4000000
MCK/128	RC = 500000 x 0.5 = 250000
MCK/1024	RC = 62500 x 0.5 = 23437.5
32 kHz	RC = 32768 x 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 250ms 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 250ms 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 RC. At the TC level, this is easily done by setting the CPCS bit of the Interrupt Enable Register. Refer to Section 2.2.5.3, Configuring an Interrupt on page 13 for more information on configuring interrupts in the NVIC.

2.2.6.3 Interrupt Handler

The first action to do in the 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, acknowledging 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 done 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 simply toggles the state (on or off) of one of the blinking LEDs. Refer to Section 2.2.8.1, Purpose on page 16 for more details on how to control LEDs with the PIO controller.

2.2.7 Using the System Tick Timer

2.2.7.1 Purpose

The primary goal of the System Tick 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, we select core clock (Master Clock) as the SysTick input clock. The SysTick has a 24-bit counter. The start value to load into the Current Value Register (CVR) is called reload value, and is stored in Reload Value Register (RVR). Each time the counter in CVR reaches 0, an interrupt is generated, and the value stored in RVR is loaded into the CVR.

The getting started example uses the SysTick to provide a 1ms 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.

2.2.7.2 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 Control and Status Register (CSR):

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

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

```
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 CSR:

```
SysTick->CTRL  = SysTick_CTRL_CLKSOURCE_Msk |
                   SysTick_CTRL_TICKINT_Msk    |
                   SysTick_CTRL_ENABLE_Msk;
```

Currently the application operation is executed by invoke CMSIS function SysTick_Config(), it is like:

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

The function also enables SysTick interrupt.

2.2.7.3 Interrupt Handler

By default, the interrupt handler of the SysTick is implemented as an infinite loop. The application should re-implement it (see Section 2.2.1.3, Vectors: Exception Handlers on page 8).

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 1ms tick overflows the counter after about 50 days. This may not be enough for a real application. In that case, a larger counter can be implemented.

2.2.7.4 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 simply 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 an empty loop prevents the modification of the counter. Obviously, this is not the case since it can be altered by the interrupt handler.

2.2.8 Using the Parallel Input/output Controller

2.2.8.1 Purpose

Most pins on AT91SAM microcontrollers can either be used by a peripheral function (e.g. USART, SPI, etc.) or used as generic inputs/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. In the second case, the level of the pin 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 2.2.5.3, Configuring an Interrupt on page 13).

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 2.1.1, Features on page 4).

2.2.8.2 Initialization

There are two steps for initializing the PIO controller. First, its peripheral clock must be enabled in the PMC. After that, its interrupt source can be configured in the NVIC.

2.2.8.3 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 (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 (OER) and Output Disable Register (ODR). Since in this case the two PIOs must be output, the same value as before shall be written in OER.

Note: 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, as this reduces power consumption. This is done through the Pull Up Disable Register (PUDR) of the 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_PPUDR = (LED_A | LED_B);
```

2.2.8.4 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 (SODR) and the Clear Output Data Register (CODR) of the PIO controller.

In addition, a register indicates the current level on each pin (Output Data Status Register, ODSR). It can be used to create a toggle function, i.e. when the LED is ON according to 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;
```

2.2.8.5 Configuring Buttons

As stated previously, the two PIOs connected to the switches on the board shall be 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 PER), they are configured as inputs by writing their IDs in ODR. Conversely to the LEDs, it is necessary to keep the pull-ups enabled.

AT91SAM3S4 has internal hardware debouncing filter to remove intermediate noise states from input, as described in Section 2.2.8.6, Configuring Input Debouncing on page 17.

Enabling interrupts on the two pins is done simply in the Interrupt Enable Register (IER). However, the PIO controller interrupt must be configured as described in Section 2.2.5.3, Configuring an Interrupt on page 13.

2.2.8.6 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 Input Filter Enable Register (PIO_IFER).

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

The debouncing filter is selected by writing to Debouncing Input Filter Select Register (PIO_DIFSR). The Glitch or Debouncing Input Filter Selection Status Register (PIO_IFDGSR) holds current selection status.

```
pio->PIO_DIFSR = 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, a debounce time to 100ms or noise cutting off frequency 10Hz can be obtained with a 32768Hz Slow Clock frequency by writing a value of 1637 in SCDR.

2.2.8.7 Interrupt Handler

The interrupt handler for the PIO controller must first check which button has been pressed. PDSR indicates the level on each pin, so it can show if each button is currently pressed or not. Alternatively, the Interrupt Status Register (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 2.1.1, Features on page 4), 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 ISR is read by the software.

2.2.9 Using the UART

2.2.9.1 Purpose

The UART provides a two-pin Universal Asynchronous Receiver and Transmitter (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.

The UART is used in the example to output a single string of text whenever the application starts. It is configured with a baud rate of 115200, eight bits of data, no parity, one stop bit and no flow control.

2.2.9.2 Initialization

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

```
PMC->PMC_PCER0 = (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 (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 (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 very 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 a previous execution of the application. Setting bits RSTRX and RSTTX in the Control Register (CR) of the UART resets and disables the received and transmitter, respectively. Setting all bits of the Interrupt Disable Register (IDR) disable 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 (BRGR). The following values are possible:

Table 2-6. 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:

$$CD = \frac{MCK}{16 \times Baud\ rate}$$

For example, an 115200 baud rate can be obtained with a 48MHz master clock frequency by writing a value of 26 in CD. Obviously, there is a slight deviation from the desired baud rate; these values yield a true rate of 115384 bauds. However, it is a mere 1.6% error, so it does not have any impact in practice.

The Mode Register (MR) has two configurable values. The first one is the Channel Mode in which the UART is operating. Several modes are available for testing purpose. In this example, only the normal mode is of interest. Setting the CHMODE field to a null-value selects the normal mode.

It is also possible to configure a parity bit in the Mode Register. 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. It enables faster transfer of data 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 (PTCR) of the UART.

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

2.2.9.3 Sending a Character

Transmitting a character on the UART line is simple; writing the character value in the Transmit Holding Register (THR) starts the transfer. However, the transmitter must be ready at this time.

Two bits in the UART Status Register (SR) indicate the transmitter state. Bit TXEMPTY 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 this bit is set, the transmitter has finished copying the value of THR in its internal shift register it uses for sending the data. In practice, this means that THR can be written when TXRDY is set, regardless of the value of TXEMPTY. When TXEMPTY rises, the whole transfer is finished.

2.2.9.4 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.

3. Building the Project

The development environment for this getting started 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
- AT91-ISP v1.13 or later (available at http://www.atmel.com)

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

3.1 ARM Compiler Toolchain

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

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

We also require software that is not included into the CodeSourcery package: the *make* utility. We get it by installing the unxutils package available at unxutils.sourceforge.net.

3.1.1 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, one for variables settings, and the other for rules implementation.

3.1.1.1 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.

Table 3-1. Makefile variables.

Parameter	Description
`CROSS_COMPILE=arm-none-eabi-`	Defines the cross-compiler toolchain prefix

`CHIP  = at91sam3u4`
 	
`BOARD = at91sam3u-ek`
	Defines the chip and board names

`TRACE_LEVEL = 4`	Defines trace level used for compilation
`OUTPUT=getting-started-$(BOARD)-$CHIP)`	Output file name (with board and chip names)

`LIBRARIES = ../libraries`
 	
`CHIP_LIB = $(LIBRARIES)/board`
 	
`DRIVERS = $(LIBRARIES)/drivers`
	Defines the library path

`INCLUDES  = -I$(CHIP_LIB)`
 	
`INCLUDES += -I$(DRIVERS)`
 	
`INCLUDES += -I$(LIBRARIES)`
	Paths for header files

`OPTIMIZATION = -Os`	Level of optimization used during compilation (-Os optimizes for size)

`CC=$(CROSS_COMPILE)gcc`
 	
`SIZE=$(CROSS_COMPILE)size`
 	
`STRIP = $(CROSS_COMPILE)strip`
 	
`OBJCOPY=$(CROSS_COMPILE)objcopy`
	Names of cross-compiler toolchain binutils (compiler, symbol list extractor, etc.)

`CFLAGS += -mcpu=cortex-m3 -mthumb -Wall -mlong-calls -ffunction-sections`
 	
`CFLAGS += -g $(OPTIMIZATION) $(INCLUDES) -D$(CHIP) -DTRACE_LEVEL=$(TRACE_LEVEL)`
	Compiler options:
	-mcpu = cortex-m3: type of ARM CPU core
	-mthumb: generate code for Thumb instruction set
	-Wall: displays all warnings
	-mlong-calls: generate code that uses long call sequences
	-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
	-Os : optimize for size
	$(INCLUDES): set paths for include files
	-D$(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 = -g $(OPTIMIZATION) -nostartfiles -mcpu=cortex-m3 -mthumb -Wl,--gc-sections`	Linker options: -nostartfile: Do not use the standard system startup files when linking -Wl,--gc-sections: Pass the --gc-sections option to the linker

`C_OBJECTS=main.o`
 	
`C_OBJECTS+= board_lowlevel.o`
	List of all object file names

For more detailed information about gcc options, refer to the gcc documentation (gcc.gnu.org).

3.1.1.2 Rules

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

The first rule, ‘all’, is the default rule used by the make command if none is specified in command line.

```
all: sram flash
```

The following rules create the three object files from the three 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 last rules describe how to compile source files and link object files together to generate one binary file per configuration; program running in Flash and program running in SRAM. It describes how to compile source files and link object files together. The first line calls the linker with the previously 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. Example of SRAM configuration is given:

```
sram: $(C_OBJECTS)
  $(CC) $(LDFLAGS) -T”$(AT91LIB)/boards/$(BOARD)/$(CHIP)/sram.lds -o $(OUTPUT)-sram.elf
  $(OBJCOPY) -O binary $(OUTPUT)-sram.elf $(OUTPUT)-sram.bin
```

3.1.2 Linker File

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

3.1.2.1 Header

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

Set the object file format to elf32-littlearm.

```
OUTPUT_ARCH(arm)
```

Specify the machine architecture.

```
ENTRY(ResetException)
```

Set the symbol ‘`ResetException`’ as the entry point of the program.

3.1.2.2 Section Organization

The *MEMORY* part defines the start and size of each memory 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

```
/* Memory Spaces Definitions */
MEMORY
{
        sram (W!RX) : ORIGIN = 0x20000000, LENGTH = 0x00008000 /* SRAM, 48K */
}
```

```
SECTIONS
{
        .fixed : {
                . = ALIGN(4);
                _sfixed = .;
                KEEP(*(.vectors))
                *(.text*)
                *(.ramfunc)
                *(.rodata*)
                *(.glue_7)
                *(.glue_7t)
                . = ALIGN(4);
                _efixed = . ;
        } > sram

        .relocate : AT ( _efixed) ) {
                _srelocate = .;
                *(.data)
                _erelocate = .;
        } > sram

                .bss (NOLOAD) : {
                . = ALIGN(4);
                _szero = .;
                *(.bss)
                . = ALIGN(4);
                _ezero = .;
        } > sram
        /* Stack in the end of SRAM */
        _estack = 0x20083ffc;
}
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 file are placed here, and finally the *_efixed* symbol is set and aligned on a 4-byte address.

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

In the *.data* section, the *AT (*`_efixed`*)* 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 hole between these two sections.

3.2 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 AT91 products on Atmel Evaluation Kits through a USB, COM or J-TAG link. Target programming is done here via SAM-BA® tools.

Follow the steps below to launch the SAM-BA:

- Shut down the board
- Set the JP-3 jumper on the board with power up to erase the internal Flash
- Plug the USB cable between the PC and the board and wait for a few seconds
- Shut down the board and remove the jumper
- Power on the board and plug the USB cable between the PC and the board
- Execute the SAM-BA.exe to launch SAM-BA

Follow the steps below to download code into SRAM:

- Download the binary “getting-started-at91sam3s-ek-at91sam3s4-sram.bin” into the SRAM
- Running the binary from SAM-BA
- Shut down SAM-BA

Follow the steps below to download code into Flash:

- Execute “Enable flash” to enable flash access
- Download the binary “getting-started-at91sam3s-ek-at91sam3s4-flash.bin” into the flash
- Execute “Boot from flash”
- Running the binary from SAM-BA
- Shut down SAM-BA

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

3.3 Debug Support

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, simply comment out (with a ‘#’) the “OPTIM = -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.

4. Revision History

Doc. Rev.	Date	Comments
42152A	07/2013	Initial document release

Atmel | Enabling Unlimited Possibilities®

Atmel Corporation	**Atmel Asia Limited**	**Atmel Munich GmbH**	**Atmel Japan G.K.**
1600 Technology Drive	Unit 01-5 & 16, 19F	Business Campus	16F Shin-Osaki Kangyo Building
San Jose, CA 95110	BEA Tower, Millennium City 5	Parkring 4	1-6-4 Osaki, Shinagawa-ku
USA	418 Kwun Tong Road	D-85748 Garching b. Munich	Tokyo 141-0032
Tel: (+1)(408) 441-0311	Kwun Tong, Kowloon	GERMANY	JAPAN
Fax: (+1)(408) 487-2600	HONG KONG	**Tel:** (+49) 89-31970-0	**Tel:** (+81)(3) 6417-0300
www.atmel.com	**Tel:** (+852) 2245-6100	**Fax:** (+49) 89-3194621	**Fax:** (+81)(3) 6417-0370
	Fax: (+852) 2722-1369		