Getting Started with SAM3U Microcontrollers

1. Scope

This application note aims at helping the reader to become familiar with the Atmel ARM® Cortex®-M3 based SAM3U microcontroller.

It describes in detail a simple project that uses several important features present on SAM3U chips. This includes how to setup 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 setup and use a GNU ARM toolchain in order to compile and run a software 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 appropriate documents available from http://www.arm.com.

AT91 ARM Cortex-M3 based Microcontrollers

Application Note

11020A–ATARM–23-Aug-10

2. Requirements

The software provided with this application note comprises several components:

- The SAM3U Evaluation Kit
- A computer running Microsoft® Windows® 2000/XP
- An ARM cross-compiler toolchain support ARM® Cortex®-M3
- SAM-BA® V2.9 or later

3. Software Example

This section describes how to program a basic application. It is divided into two main sections:

- the specification of the example (what it does, which peripherals are used),
- the programming aspects.

3.1 Specification

3.1.1 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 a Wait function based on a 1 ms tick for the second LED. 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 is a good starting point to become familiar with the AT91SAM microcontroller series.

3.1.2 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)
- System Tick Timer (SysTick)
- Nested Vectored Interrupt Controller (NVIC)
- Universal Asynchronous Receiver Transmitter (UART).

LEDs and buttons on the board are connected to standard input/output pins on the chip. The pins are 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. Three different functions are used to handle PIO, TC 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. The UART can be easily disabled from the Power Management Controller (PMC) by setting the UART bit in the PMC_PCDR register.

3.1.3 Evaluation Kit

3.1.3.1 Memories

The AT91SAM3U4 located on the SAM3U-EK evaluation board features two internal SRAM memories (32KB and 16KB), and 2 x 128 KB Flash Memory block. In addition, it provides an external bus interface made of a Static Memory Controller (SMC) and of a NAND Flash Controller (NFC), enabling the connection of external memories. A 1MB PSRAM and a 256MB NAND Flash are present on the SAM3U-EK. The Getting Started software example can be compiled and loaded on the internal Flash and the internal SRAM memories.

3.1.3.2 Buttons

The AT91SAM3U4 Evaluation Kit features two push-buttons, connected to pins PA18 and PA19. When pressed, they force a logical low level on the corresponding PIO line.

The Getting Started example uses both buttons by using the internal hardware debouncing circuitry embedded in the SAM3U.

3.1.3.3 LEDs

There are two general-purpose green LEDs on the SAM3U-EK, as well as a software-controllable red power LED. They are wired to pins PB0, PB1 and PB2, respectively. Setting a logical low level on these PIO lines turns the corresponding LED on.

The application example uses the two green LEDs (PB0 and PB1).

3.1.3.4 UART

On the SAM3U-EK, the UART uses pins PA11 and PA12 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 “Initializing Stacks” on page 11) on reset, and the entry point addressed for all exception handlers. The exception num-

bers (see Table 3-1 on page 5) define the order of entries in the vector table associated with the exception handler entries (see Table 3-2 on page 5).

Table 3-1. Exception Numbers

Exception number	Exception
1	Reset
2	NMI
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

On reset, the vector table is located at CODE partition. The 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 on 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 unsigned int _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
    EFC0_IrqHandler,    // 6  EFC0
    EFC1_IrqHandler,    // 7  EFC1
    DBGU_IrqHandler,    // 8  DBGU
    HSMC4_IrqHandler,   // 9  HSMC4
    PIOA_IrqHandler,    // 10 Parallel IO Controller A
    PIOB_IrqHandler,    // 11 Parallel IO Controller B
    PIOC_IrqHandler,    // 12 Parallel IO Controller C
    USART0_IrqHandler,  // 13 USART 0
    USART1_IrqHandler,  // 14 USART 1
    USART2_IrqHandler,  // 15 USART 2
    USART3_IrqHandler,  // 16 USART 3
    MCI0_IrqHandler,    // 17 Multimedia Card Interface
    TWI0_IrqHandler,    // 18 TWI 0
    TWI1_IrqHandler,    // 19 TWI 1
    SPI0_IrqHandler,    // 20 Serial Peripheral Interface
    SSC0_IrqHandler,    // 21 Serial Synchronous Controller 0
    TC0_IrqHandler,     // 22 Timer Counter 0
    TC1_IrqHandler,     // 23 Timer Counter 1
    TC2_IrqHandler,     // 24 Timer Counter 2
    PWM_IrqHandler,     // 25 Pulse Width Modulation Controller
    ADCC0_IrqHandler,   // 26 ADC controller0
    ADCC1_IrqHandler,   // 27 ADC controller1
    HDMA_IrqHandler,    // 28 HDMA
    UDPD_IrqHandler,   // 29 USB Device High Speed UDP_HS
    IrqHandlerNotUsed   // 30 not used
};
```

The reset routine is responsible for starting up the application and then enabling interrupts. For any details on the reset exception, refer to Table 3.2.1.3 on page 8.

3.2.1.2 *Exception Handlers*

When an exception occurs, the context state is saved by the hardware onto a stack pointed to by the SP register. The stack which is used depends on the processor mode at the time of the exception.

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 simply be set to an infinite loop, as in the example below:

```
//------------------------------------------------------------------------------
// 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 when we want in our applications.

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

```
WEAK void SysTick_Handler(void)
{
    while(1);
}
```

Since we use the SysTick exception in our example, we can re-implement the SysTick exception handler as:

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

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

3.2.1.3 *Reset Exception*

After reset, SP_main (from vector table offset 0) and start PC (from vector table offset 4) are loaded in SP and PC. The reset exception handler runs. A normal reset exception handler follows the steps below:

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, value 0, to a location in SRAM. This is normally done to enable dynamic changes.
Branch to main()	Branch to main application.

The following sections detail each action.

3.2.1.4 *Low-Level Initialization*

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

- Enhanced Embedded Flash Controller (EEFC)
- Watchdog
- Slow clock
- Main oscillator and its PLL
- UART clock
- Set default master

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

3.2.1.5 *Enhanced Embedded Flash Controller (EEFC)*

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. This example configures the core clock at 48MHz (PLLA output divided by 2).

For more information about the required number of wait states depending on the operating frequency of a microcontroller, please refer to the AC Electrical Characteristics section of the corresponding datasheet.

3.2.1.6 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 shall be disabled in the Watchdog Mode Register (WDMR):

```
AT91C_BASE_WDTC->WDTC_WDMR = AT91C_WDTC_WDDIS;
```

3.2.1.7 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 crystal oscillator and 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 made by writing the SUPC Control Register (SUPC_CR) with the XTALSEL bit at 1:

```
if ((AT91C_BASE_SUPC->SUPC_SR & AT91C_SUPC_SR_OSCSEL_CRYST) !=
AT91C_SUPC_SR_OSCSEL_CRYST) {
       AT91C_BASE_SUPC->SUPC_CR = AT91C_SUPC_CR_XTALSEL_CRYSTAL_SEL | (0xA5
<< 24);
        timeout = 0;
        while (!(AT91C_BASE_SUPC->SUPC_SR & AT91C_SUPC_SR_OSCSEL_CRYST) &&
(timeout++ < CLOCK_TIMEOUT));
    }
```

3.2.1.8 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 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 consists in enabling both oscillators and waiting 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. The stabilization occurs when bit MOSCXTS of the PMC Status Register (PMC_SR) is set. The following piece of code performs these two operations:

```
if(!(AT91C_BASE_PMC->PMC_MOR & AT91C_CKGR_MOSCSEL))
    {
        AT91C_BASE_PMC->PMC_MOR = (0x37 << 16) | BOARD_OSCOUNT |
AT91C_CKGR_MOSCRCEN                          | AT91C_CKGR_MOSCXTEN;
        timeout = 0;
       while (!(AT91C_BASE_PMC->PMC_SR & AT91C_PMC_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 SAM3U series Preliminary". Note that the AT91SAM3U4 internal slow clock is generated using an RC oscilla-

tor. This must be taken into account as it impacts the slow clock accuracy, as in the example below:

– RC oscillator frequency range, in kHz: $20 \leq f_{RC} \leq 42$

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

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

– Oscillator startup time: $1 ms \leq t_{Startup} \leq 1.4 ms$

– Value for a 2ms startup: $OSCOUNT = \frac{42000 \times 0.0014}{8} = 8$

The second step consists in switching the source of the main oscillator to the crystal oscillator, waiting for the selection to be done. Writing the MOSCSEL bit of the PMC_MOR register switches the source of the main oscillator to the crystal oscillator. The selection is done when bit MOSCSEL of the PMC_SR is set:

```
AT91C_BASE_PMC->PMC_MOR = (0x37 << 16) | BOARD_OSCOUNT |
AT91C_CKGR_MOSCRCEN | AT91C_CKGR_MOSCXTEN | AT91C_CKGR_MOSCSEL;
    timeout = 0;
    while (!(AT91C_BASE_PMC->PMC_SR & AT91C_PMC_MOSCSELS) && (timeout++ <
CLOCK_TIMEOUT));
```

Once the crystal main oscillator is started and stabilized, the PLL can be configured. The PLL is made 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 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. Both constraints must be taken into account. The example below is given for the SAM3U-EK:

$f_{Input} = 12$

$DIV = 1$

$MUL = (8 - 1) = 7$

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

The PLL startup time must be provided. 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 be locked, which is done by monitoring the Status Register of the PMC:

```
AT91C_BASE_PMC->PMC_PLLAR = BOARD_PLLR;
    timeout = 0;
    while (!(AT91C_BASE_PMC->PMC_SR & AT91C_PMC_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. This step uses two register writes, with two loops to wait for the main clock to be ready.

```
AT91C_BASE_PMC->PMC_MCKR = (BOARD_MCKR & ~AT91C_PMC_CSS) |
AT91C_PMC_CSS_MAIN_CLK;
    timeout = 0;
    while (!(AT91C_BASE_PMC->PMC_SR & AT91C_PMC_MCKRDY) && (timeout++ <
CLOCK_TIMEOUT));
    AT91C_BASE_PMC->PMC_MCKR = BOARD_MCKR;
    timeout = 0;
    while (!(AT91C_BASE_PMC->PMC_SR & AT91C_PMC_MCKRDY) && (timeout++ <
CLOCK_TIMEOUT));
```

The chip is configured to run on the main clock at 48MHz with the PLLA at 96MHz divided by two.

3.2.1.9 *UART Clock*

The UART is used for print debug messages. The UART clock has to be enabled after reset, in order to save power and to be closed. The clock is enabled by setting the UART bit in the PMC_PCER:

```
AT91C_BASE_PMC->PMC_PCER = (1 << AT91C_ID_DBGU);
```

3.2.1.10 *Set Default Master*

The normal latency to connect an AHB master to an AHB slave is one cycle except for the default master of the accessed slave which is connected directly (zero cycle latency). To increase the access speed of an AHB master to the Internal SRAM, the default master has to be set. This is made by setting the DEFMSTR_TYPE bit of the MATRIX_SCRFx (where x is the slave number) register to “Fixed Default Master”. Below is an example of setting SRAM0 and SRAM1 to a default master:

```
AT91PS_HMATRIX2 pMatrix = AT91C_BASE_MATRIX;
// Set default master: SRAM0 -> Cortex-M3 System
        pMatrix->HMATRIX2_SCFG0 |= AT91C_MATRIX_FIXED_DEFMSTR_SCFG0_ARMS |
                                   AT91C_MATRIX_DEFMSTR_TYPE_FIXED_DEFMSTR;
        // Set default master: SRAM1 -> Cortex-M3 System
        pMatrix->HMATRIX2_SCFG1 |= AT91C_MATRIX_FIXED_DEFMSTR_SCFG1_ARMS |
                                   AT91C_MATRIX_DEFMSTR_TYPE_FIXED_DEFMSTR;
```

Please note that setting SRAM0 and SRAM1 to a fixed default master is application dependent, i.e. depends on other peripherals used in the end application, such as USB or NAND Flash Controller.

3.2.1.11 *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 which is used in exception entry and exit is described in the "Cortex-M3 Technical Reference Manual". SP_main locates at vector table offset “0” (see “Vector Table” on page 3). It is selected and initialized on reset.

3.2.1.12 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 one contains read/write data, i.e., data which 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'. A variable is said 'uninitialized' when the programmer has not set any particular value when declaring the variable. A variable is said 'initialized' when it has 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 on-chip internal Flash memory, the ***data*** segment must be initialized at startup. This is necessary because read/write variables are located in the SRAM or External RAM, not in the Flash memory. For IAR Embedded Workbench® and Keil® MDK, refer to documentation on 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, for debug purposes, it is safer and more useful to initialize the BSS segment by filling it with zeroes. Theoretically, this operation is not necessary. 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.

A BSS initialization example is given below:

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

3.2.1.13 *Locate Vector Table Offset*

The Vector Table Offset Register positions the vector table in the Flash or the SRAM memory. Default on reset is the Flash memory. For the sake of the interrupt latency, the vector table is put in the Flash memory by setting TBLBASE bit in VTOR register to “0”, as in the example below:

```
extern unsigned int _sfixed;
unsigned int *pSrc;
pSrc = (unsigned int *)&_sfixed;
AT91C_BASE_NVIC->NVIC_VTOFFR = ((unsigned int)(pSrc)) | (0x0 << 7);
```

3.2.2 Generic Peripheral Usage

3.2.2.1 *Initialization*

Most peripherals are initialized by performing four actions:

1. Enabling the peripheral clock in the PMC
2. Enabling the control of the peripheral on PIO pins
3. Configuring the interrupt source of the peripheral in the NVIC
4. 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 enables the peripheral clock in the PMC. Exception is for the System Controller, which comprises several different controllers, as it is continuously clocked.

For peripherals which need to use one or more on-chip pins as external inputs/outputs, it is necessary to configure the Parallel Input/Output controller first. This operation is detailed in Section 3.2.6 on page 17.

Finally, if an interrupt is to be generated by the peripheral, then the source must be configured properly in the Nested Vectored Interrupt Controller. For more information. refer to Section 3.2.3 on page 13.

3.2.3 Using the Nested Vectored Interrupt Controller

3.2.3.1 *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 software example, using the NVIC is required because several interrupt sources are present (see Section 3.1.2 on page 2). 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/CPUs/CMSIS.html.

3.2.3.2 *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., handlers which loop indefinitely) for all interrupt sources (see Section 3.2.1.1 on page 3). This way, if an interrupt is

triggered before being configured, the debugger is stuck in the handler instead of jumping to a random address.

3.2.3.3 *Configuring an Interrupt*

Configuring an interrupt source requires six steps:

1. Implementing an interrupt handler
2. Disabling the interrupt in case it was enabled
3. Clearing any pending interrupt, if any
4. Configuring the interrupt priority
5. Enabling the interrupt at the peripheral level
6. Enabling the interrupt at the NVIC level.

The first step consists in to re-implementing the interrupt handler with the same name as the default interrupt handler in the vector table (see Section 3.2.1.1 on page 3). Thus, when the corresponding interrupt occurs, the re-implemented interrupt handler is executed instead of the default interrupt handler (see Section 3.2.1.2 on page 6).

An interrupt triggering at the same time may result in an 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. For a list of peripheral IDs, refer to the corresponding datasheet.

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 to each interrupt. A higher level corresponds to a lower priority. 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] store the priority value, and bits [3:0] read as zero and ignore writes. For any priority details, refer to the "Cortex-M3 Technical Reference Manual" and "AT91 ARM Cortex-M3 based Microcontrollers SAM3U series Preliminary".

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

3.2.4 Using the Timer Counter

3.2.4.1 *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.

3.2.4.2 *Initialization*

In order to reduce the power consumption, most peripherals are not clocked by default. Writing the ID of a peripheral in the Peripheral Clock Enable Register (PCER) of the 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 consists in configuring the Channel Mode Register (CMR). TC channels can operate in two different modes:

- the Capture mode, normally used to perform measurements on input signals.
- The Waveform mode, which 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 the CMR resets the timer and restarts its clock every time the counter reaches the value programmed in the TC_RC Register. Choosing the correct RC value to generate a specific delay. You can choose between several different input clocks for the channel which, in practice, allows to prescale the 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. Results with different prescaling factors are given below:

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

$$Clock = \frac{MCK}{8},\ RC = 6000000 \times 0,\ 5 = 3000000$$

$$Clock = \frac{MCK}{128},\ RC = 375000 \times 0,\ 5 = 187500$$

$$Clock = \frac{MCK}{1024},\ RC = 46875 \times 0,\ 5 = 23437.5$$

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

Since the maximum RC value 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, which means that the slowest possible input clock is selected in the CMR, and the corresponding value written in the RC. The following two operations configure a 250 ms period by selecting the slow clock and dividing its frequency by 4:

```
AT91C_BASE_TC0->TC_CMR = AT91C_TC_CLKS_TIMER_DIV5_CLOCK
                       | AT91C_TC_CPCTRG;
AT91C_BASE_TC0->TC_RC = AT91B_SLOW_CLOCK >> 2;
```

The last initialization step consists in configuring the interrupt whenever the counter reaches the value programmed in the RC. At the TC level, this is easily done by setting the CPCS bit of the Interrupt Enable Register. For more information on configuring interrupts in the NVIC, see Section 3.2.3.3 on page 14.

3.2.4.3 *Interrupt Handler*

The first action to perform 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. For more details on how to control the LEDs with the PIO controller, see Section 3.2.6 on page 17.

3.2.5 Using the System Tick Timer

3.2.5.1 *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, core clock (Master Clock) is 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 the Reload Value Register (RVR). Each time the CVR counter reaches 0, an interrupt is generated, and the value stored in the RVR is loaded into the CVR.

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.

3.2.5.2 *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 consists in disabling the SysTick and selecting the clock source by setting the Control and Status Register (CSR):

```
AT91C_BASE_NVIC->NVIC_STICKCSR = AT91C_NVIC_STICKCLKSOURCE;
```

Before starting the SysTick, the CVR value should be cleared and the reload value should be stored in the RVR. Given the SysTick source clock is MCK, in order to generate a 1ms interrupt, the reload value should be MCK/1000, as in the example below:

```
reloadValue = BOARD_MCK/1000;
AT91C_BASE_NVIC->NVIC_STICKCVR &= ~AT91C_NVIC_STICKCURRENT;
AT91C_BASE_NVIC->NVIC_STICKRVR = reloadValue;
```

The SysTick can be enabled writing to the NVIC_STICKCSR register:

```
AT91C_BASE_NVIC->NVIC_STICKCSR =  AT91C_NVIC_STICKCLKSOURCE
                                | AT91C_NVIC_STICKENABLE
                                | AT91C_NVIC_STICKINT;
```

3.2.5.3 *Interrupt Handler*

By default, the SysTick interrupt handler is implemented as an infinite loop. The application should re-implement it (see Section 3.2.1.2 on page 6).

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

Using a 32-bit counter may not be appropriate, depending on:

- how long the system should stay up, and
- the tick period.

In the example, a 1ms 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.

3.2.5.4 *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 a 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.

3.2.6 Using the Parallel Input/Output Controller

3.2.6.1 *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, and 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.

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 on page 3).

3.2.6.2 *Initialization*

There are two steps to initialize the PIO controller:

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

3.2.6.3 *Configuring LEDs*

Both 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 the PIO Enable Register (PER) by writing the value corresponding to a logical OR between the two LED IDs.

The PIO direction is controlled using two registers: Output Enable Register (OER) and Output Disable Register (ODR). Since both PIOs must be outputs, the same values as before shall be written in the 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 to reduce the power consumption, through the Pull Up Disable Register (PUDR) of the PIOC.

Here is the code for LED configuration:

```
/* Configure the pins as outputs */
AT91C_BASE_PIOA->PIO_OER = (LED_A | LED_B);

/* Enable PIOC control on the pins*/
AT91C_BASE_PIOA->PIO_PER = (LED_A | LED_B);

/* Disable pull-ups */
AT91C_BASE_PIOA->PIO_PPUDR = (LED_A | LED_B);
```

3.2.6.4 *Controlling LEDs*

LEDs are turned on or off by changing the PIO level to which they are connected. Once the 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 the ODSR, then it is turned off, and vice-versa.

```
/* Turn LED off */
AT91C_BASE_PIOA->PIO_SODR = LED_A;

/* Turn LED on */
AT91C_BASE_PIOA->PIO_CODR = LED_A;
```

3.2.6.5 *Configuring Buttons*

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

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

Enabling interrupts on both pins is done in the Interrupt Enable Register (IER). However, the PIO controller interrupt must be configured as described in Section 3.2.3.3 on page 14.

3.2.6.6 *Configuring Input Debouncing*

Optional input debouncing filters are independently programmable on each I/O line. The debouncing filter 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).

AT91C_BASE_PIOC->PIO_IFER = BUTTON_1 | BUTTON_2;

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 the current selection status.

AT91C_BASE_PIOC->PIO_DIFSR = BUTTON_1 | BUTTON_2;

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 DIV value, given the Slow Clock frequency and the desired cut-off frequency:

Tdiv_slclk = ((DIV+1)x 2)x Tslow_clock

For example, a 100 ms debounce time or a 10 Hz noise cut-off frequency can be obtained with a 32,768 Hz Slow Clock frequency by writing a value of 1637 in SCDR.

3.2.6.7 Interrupt Handler

At first, the interrupt handler for the PIO controller must check which button has been pressed. The PDSR indicates each pin level, showing if each button is pressed or not. Alternatively, the Interrupt Status Register (ISR) reports which PIOs had their status changed since the last read of the register.

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

As described in Section 3.1.1 on page 3, 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 Timer Counter is modified, the TC clock is either stopped or restarted by the interrupt handler as well.

Note: The interrupt must be acknowledged in the PIOC. This is implicitly done when the ISR is read by the software. Since the ISR value is used in several operations, there is no need to worry about the compiler inadvertently.

3.2.7 Using the UART

3.2.7.1 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 115,200 bps, 8 bits of data, no parity, one stop bit and no flow control.

3.2.7.2 Initialization

Writing the UART ID in the PCER of the PMC activates its clock. This is the first step when initializing the UART, which is done in the low_level initialization (see Section 3.2.1.9 on page 11). 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 a peripheral control on those pins. However, some PIOs are shared between two different peripherals; Peripheral AB Select Register (ABSR) is used to switch control between the two.

The very next action to perform is to disable the receiver and transmitter logic, as well as the interrupts. This enables a 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 UART Control Register (CR) resets and disables the receiver and transmitter, respectively. Setting all bits of the Interrupt Disable Register (IDR) disables all interrupts coming from the Debug Unit.

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), with the following values:

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 CD value, given the microcontroller operating frequency and the desired baud rate:

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

For example, a 115,200 baud rate can be obtained with a 48MHz master clock frequency by writing a CD value of 26. Obviously, there is a slight deviation from the desired baudrate; these values yield a true rate of 115,384 bauds. However, it is a mere 1.6% error, with no impact in practice.

The Mode Register (MR) has two configurable values:

- The Channel Mode in which the UART is operating. Several modes are available for testing purpose. In the example, only the normal mode is of any 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. It enables a faster data transfer and reduces the processor overhead by taking care of most of the transmission and reception operations. The PDC (Peripheral DMA Controller) is not used in this example, and should be disabled by setting bits RXTDIS and TXTDIS in the PDC Transfer Control Register (PTCR) of the UART.

At that step, the UART is fully configured. The last one consists in enabling the transmitter. Transmitter enabling is done by setting bit TXEN in the Control Register. It is useless to enable the receiver, which is not being used in this demo application.

3.2.7.3 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 (THR) to start the transfer.

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

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

3.2.7.4 *String Print Function*

A *printf()* function is defined in the application example. 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 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 to build the project and load the binary file are:

- an ARM cross-compiler toolchain
- SAM-BA® V2.9 or later (available at www.atmel.com).

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

4.1 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 from release 2008-q3.

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

Another software, not included into the CodeSourcery package, is also required: *make* utility. Get it by installing the unxutils package available at unxutils.sourceforge.net.

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. it is divided into two parts, dedicated to:

- variables setting,
- 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.

```
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 the trace level used for compilation.

```
OUTPUT=getting-started-project-$(BOARD)-$CHIP)
```

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

```
AT91LIB = ../at91lib
EXT_LIBS= ../external_libs
```

Defines the library path.

```
INCLUDES += -I$(AT91LIB)/boards/$(BOARD)
INCLUDES += -I$(AT91LIB)/peripherals
INCLUDES += -I$(AT91LIB)/components
INCLUDES += -I$(AT91LIB)
INCLUDES += -I$(EXT_LIBS)
```

Defines the paths for header files.

```
OPTIMIZATION = -Os
```

Defines the 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
```

Defines the names of cross-compiler toolchain binutils (compiler, symbol list extractor, etc.).

```
CFLAGS = -mcpu=cortex-m3 -mthumb -Wall -mlong-calls -ffunction-sections -g
$(OPTIMIZATION) $(INCLUDES) -D$(CHIP) -DTRACE_LEVEL=$(TRACE_LEVEL)
```

Defines the compiler options, such as:

- -mcpu = cortex-m3: type of ARM CPU core.
- -mthumb: generates code for Thumb instruction set.
- -Wall: displays all warnings.
- -mlong-calls: generates code that uses long call sequences.
- -ffunction-sections: places each function item into its own section in the output file if the target supports arbitrary sections.
- -g: generates debugging information for GDB usage.
- -Os: optimizes for size.
- $(INCLUDES): sets paths for include files.
- -D$(CHIP): defines chip names used for compilation.
- -DTRACE_LEVEL=$(TRACE_LEVEL): defines a trace level used for compilation

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

Defines the 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
```

Defines the 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
....
```

Defines the list of all object file names.

For detailed information about gcc options, refer to gcc documentation (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 of the files needed to create this target.

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

```
all: sram flash
```

The following rules create 3 object files from the 3 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: a program running in Flash and a 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 the 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 SRAM configuration example is given below:

```
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
```

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")
```

– Sets the object file format to elf32-littlearm.

`OUTPUT_ARCH(arm)`

– Specify the machine architecture.

`ENTRY(ResetException)`

– Sets the symbol 'ResetException' as the entry point of the program.

4.1.2.2 *Section Organization*

_vect_start defines the vector table position.

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

The SECTION part deals with the different code sections 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

```
_vect_start = 0x20000000;

/* Memory Spaces Definitions */
MEMORY
{
   sram0 (W!RX) : ORIGIN = 0x20000000, LENGTH = 0x00008000 /* Sram0, 32K */
   sram1 (W!RX) : ORIGIN = 0x20080000, LENGTH = 0x00004000 /* Sram1, 16K */
}
SECTIONS
{
  .fixed : {
    _sfixed = .;
    KEEP(*(.vectors))
    *(.text*)
    *(.ramfunc)
    *(.rodata*)
    *(.glue_7)
    *(.glue_7t)
    . = ALIGN(4);
    _efixed = . ;
  } > sram0

  .relocate : AT ( _efixed) ) {
    _srelocate = .;
    *(.data)
    _erelocate = .;
```

```
  } > sram1

    .bss (NOLOAD) : {
    . = ALIGN(4);
    _szero = .;
    *(.bss)
    . = ALIGN(4);
    _ezero = .;
  } > sram1

  _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 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 *.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. There is no empty space between these two sections.

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 SAM-BA Tool offers an easy way to download files into AT91 products on Atmel Evaluation Kits through a USB, COM or J-TAG link.

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

- Shut down the board.
- Set the JP-2 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 SAM-BA.exe to launch SAM-BA.

Follow the steps below to download code into the internal SRAM:

- Open SAM-BA, select the SRAM window pane.

  In the SRAM window pane:

  – Make sure the "Address" value is 0x20000000.
  – Select the binary file "getting-started-project-at91sam3u-ek-at91sam3u4-sram.bin" by clicking on the folder icon.
  – Click on the "Send File" button.

  Then, in the shell frame of SAM-BA:

  - Enter the command line "go 0x20000000".
- Close the SAM-BA application.

Follow the steps below to download code into the internal Flash:

- Open SAM-BA, select the "Flash 0" window pane.

  In the "Flash 0" window pane:

  - Make sure the "Address" value is 0x80000.

  From the drop-down "Scripts" list:

  - Select "Enable flash access", then click Execute.
  - Select "flash 0 for boot", then click Execute.
  - Select the binary file "getting-started-project-at91sam3u-ek-at91sam3u4-flash.bin" by clicking on the folder icon.
  - Click on the "Send File" button.
- Close SAM-BA, power down and power up the board: the code starts running.

4.3 Debug Support

When debugging the project 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.

Revision History

Doc. Rev	Comments	Change Request Ref.
11020A	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/AT91SAM

Technical Support
AT91SAM Support
Atmel techincal support

Sales Contacts
www.atmel.com/contacts/

Literature Requests
www.atmel.com/literature