APPLICATION NOTE

AT03758: Getting Started with SAM4N

32-bit Microcontroller

Features

- Getting started with SAM4N device and tools
- Getting started with SAM4N Xplained Pro in Atmel Studio, IAR Embedded Workbench® for ARM® and SAM-BA®
- Getting started example in Atmel Software Framework (ASF)

Description

This application note provides information on how to get start with the Atmel ARM Cortex®-M4 based SAM4N microcontroller. It will provide information on how to get the datasheet, tools, software, and give a step-by-step instruction on how to load and build up a single example project to SAM4N Xplained Pro.

Table of Contents

1 Get the Device Datasheet

Web page: www.atmel.com/products/microcontrollers/arm/sam4n.aspx

Documents: SAM4N Series Datasheet (Summary, Complete) (.pdf)

- Complete version (Full datasheet)
- Summary version (short version includes product features, package, pinout, and order information)

2 Get the SAM4N Xplained Pro

Web page: www.atmel.com/tools/ATSAM4N-XPRO.aspx

Get the kit: www.store.atmel.com

Document/file:

- SAM4N Xplained Pro User Guide (.pdf)

Key Features:

- SAM4N16C microcontroller
- One mechanical reset button
- One mechanical user pushbutton
- One yellow user LED
- 32.768kHz crystal
- 12MHz crystal
- Three Xplained Pro extension headers
- Embedded Debugger
  - Auto-ID for board identification in Atmel Studio 6.1
  - One yellow status LED
  - One green board power LED
  - Programming
  - Virtual COM port (CDC)
- USB powered
- Supported with application examples in Atmel Software Framework

The SAM4N Xplained Pro User Guide introduces the SAM4N Xplained Pro and describes its development and debugging capabilities.

3 Get the Tools

The following tools are necessary for SAM4N development.

Atmel Studio 6.1 (build 2674 or above): www.atmel.com/atmelstudio

IAR Embedded Workbench for ARM 6.50.5: www.iar.com/en/Products/IAR-Embedded-Workbench/ARM/

SAM4N patch for IAR Embedded Workbench for ARM: IAR-EWARM-SAM4N-ADDON-V1.0.zip (provided with the application note)

SAM-BA v2.12: www.atmel.com/tools/ATMELSAM-BAIN-SYSTEMPROGRAMMER.aspx

SAM-BA v2.12 patch SAM4N: sam-ba_2.12_patch4n.exe (provided with the application note)

Atmel Software Framework (ASF) (the latest revision): www.atmel.com/tools/avrsoftwareframework.aspx

4 Get Started with Atmel Studio 6

4.1 Requirements

- Atmel Studio 6.1 (build 2674 or above) installed
- ASF included in Atmel Studio installation, update to the latest version
- SAM4N Xplained Pro connected to Atmel Studio through USB cable

4.2 Load the Example

- Launch Atmel Studio
- Open the example selection menu in ASF from Atmel Studio: File → New → Example Project
- Select the “SAM4, 32-bit” from Device Family drop-down list
- Select the “Applications” from Category drop-down list
- Select the “Kit” view and select SAM4N Xplained Pro
- Pick project “Getting-Started Application on SAM – SAM4N Xplained Pro” in the list and then press OK
- Accept the license agreement and press Finish. Then the Atmel Studio will open the example
- Build the project: Build → Build Solution
- Load the code in SAM4N and start debugging: Debug → Start Debugging and Break

Now the application has been programmed and the debugger stops at the beginning of main(). To execute it, click on Debug → Continue.

5 Get Started with IAR EWARM

5.1 Requirements

- ASF the latest revision standalone package installed
- IAR Embedded Workbench for ARM 6.50.5 installed
- SAM4N patch for IAR Embedded Workbench for ARM installed
- SAM4N Xplained Pro connected to IAR Embedded Workbench for ARM through USB cable

5.2 Load the Example

- Find the example iar project for SAM4N Xplained Pro in ASF standalone package and open it.
- Build the project: Project → Make
- Load the code in SAM4N and start debugging: Project → Download and Debug

Now the application has been programmed and the debugger stops at the beginning of main(). To execute it, click on Debug → Go.

6 Get Started with SAM-BA

6.1 Requirements

- Atmel Studio 6.1 (build 2674 or above) installed
- ASF (the latest revision) standalone package installed
- SAM-BA v2.12 and SAM4N patch installed
- SAM4N Xplained Pro connected to SAM-BA through USB cable

6.2 Build the Binary File

- Open the Atmel Studio command line: Start → All Programs → Atmel → Atmel Studio 6.1 Command Prompt
- Find the example gcc project for SAM4N Xplained Pro in ASF standalone package
- Change the directory where the makefile is, type “make” and enter
- Then the binary file (getting-started_flash.bin) will be generated in the directory
- The binary file generated by IAR can be programmed by SAM-BA as well. How to generate binary files by IAR, Project → Options → Output Converter: Click “Generate additional output” and select “binary” from Output Format drop-down list. More details please refer to IAR C/C++ Development Guide for ARM provided by IAR Embedded Workbench for ARM.

6.3 Load the Example

- Open SAM-BA
- Select COMn (n could be 1, 2, or other number) as the connection
- Select at91sam4n16-xpro as the target board. Then press Connect
- In SAM-BA GUI, choose Flash tab
- For Send File Name, choose the binary file (getting-started_flash.bin) generated previously
- Specify the address (0x400000), then press Send File
- For Scripts, select Boot from Flash (GPNVM1), then press Execute

Now the application has been programmed. To execute it, reset the board.

7 The Getting-started Example

This chapter describes a simple example project that uses several important features present on SAM4N device.

There are four main parts in this section:

- The specification of the getting-started example
- The introduction about relevant on-chip peripherals
- The introduction about relevant on-board components
- The implementation of the example

7.1 Specification

The getting-started example makes the user LED on the board blink at a fixed rate. This rate is generated by using a timer. The blinking can be stopped and restarted by using the user pushbutton.

7.2 On-chip Peripherals

In order to perform the operations described previously, the getting-started 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)
- Power Management Controller (PMC)

LED and button 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 timebases, 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 delay function monitors this variable to provide a precise delay.

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.

7.3 On-board Components

7.3.1 Buttons

The SAM4N Xplained Pro features two push-buttons, RESET, and SW0, connected to pins nRST and PA30 respectively.

The RESET is usually used to reset the MCU, while SW0 is used for general purpose, which can force a logical low level on the corresponding PIO line when pressed.

The getting-started example uses SW0 button with the internal hardware debouncing circuitry embedded in the SAM4N.

7.3.2 LEDs

There are three LEDs on the SAM4N Xplained Pro. LED0 is used for general purpose, which is connected to PB14. POWER is the power LED and STATUS is the debugger status LED.

LED0 is used in the getting-started example.

7.3.3 COM Port

UART0 of the SAM4N is connected to the virtual COM port on SAM4N Xplained Pro.

7.4 Implementation

7.4.1 Startup

Most of the code in this program is written in C, which makes it 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.

Note: There are two versions of c-startup code in Atmel Software Framework. One is for the IAR Embedded Workbench for ARM compiler and the other is for GNU GCC compiler. This application note will focus on the details of the GCC one.

7.4.1.1 Vector Table

The vector table contains the initialization value for the stack pointer (see “Initializing Stacks”) on reset, and the entry point addressed for all exception handlers. The exception numbers (see Table 7-1) define the order of entries in the vector table associated with the exception handler entries (see Table 7-2).

Table 7-1. Exception Numbers

Exception Number	Exception
1	Reset
2	Non-Maskable Interrupt
3	Hard Fault
4	Memory Management
5	Bus Fault
6	Usage Fault
7-10	Reserved
11	SVCall
12	Debug Monitor
13	Reserved
14	PendSV
15	SysTick
16	External Interrupt 0
…	…
16 + N	External Interrupt N

Table 7-2. Vector Table Format

Word Offset	Description
0	Initial 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-M4 Technical Reference Manual".

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

The Full Vector Table in the Getting-Started Example

```
const DeviceVectors exception_table = {

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

        (void *)Reset_Handler,
        (void *)NMI_Handler,
        (void *)HardFault_Handler,
        (void *)MemManage_Handler,
```

```
        (void *)BusFault_Handler,
        (void *)UsageFault_Handler,
        (void *)(0UL),     /* Reserved */
        (void *)(0UL),     /* Reserved */
        (void *)(0UL),     /* Reserved */
        (void *)(0UL),     /* Reserved */
        (void *)SVC_Handler,
        (void *)DebugMon_Handler,
        (void *)(0UL),     /* Reserved */
        (void *)PendSV_Handler,
        (void *)SysTick_Handler,

        /* Configurable interrupts */
        (void *)SUPC_Handler,   /* 0  Supply Controller */
        (void *)RSTC_Handler,   /* 1  Reset Controller */
        (void *)RTC_Handler,    /* 2  Real Time Clock */
        (void *)RTT_Handler,    /* 3  Real Time Timer */
        (void *)WDT_Handler,    /* 4  Watchdog Timer */
        (void *)PMC_Handler,    /* 5  Power Management Controller */
        (void *)EFC_Handler,    /* 6  Enhanced Flash Controller */
        (void *)(0UL),     /* 7 Reserved */
        (void *)UART0_Handler,  /* 8  UART 0 */
        (void *)UART1_Handler,  /* 9  UART 1 */
        (void *)UART2_Handler,  /* 10 UART 2 */
        (void *)PIOA_Handler,   /* 11 Parallel I/O Controller A */
        (void *)PIOB_Handler,   /* 12 Parallel I/O Controller B */
        (void *)PIOC_Handler,   /* 13 Parallel I/O Controller C */
        (void *)USART0_Handler, /* 14 USART 0 */
        (void *)USART1_Handler, /* 15 USART 1 */
        (void *)UART3_Handler,  /* 16 UARG 3 */
        (void *)USART2_Handler, /* 17 USART 2 */
        (void *)(0UL),     /* 18 Reserved */
        (void *)TWI0_Handler,   /* 19 Two Wire Interface 0 */
        (void *)TWI1_Handler,   /* 20 Two Wire Interface 1 */
        (void *)SPI_Handler,    /* 21 Serial Peripheral Interface */
        (void *)TWI2_Handler,   /* 22 Two Wire Interface 2 */
        (void *)TC0_Handler,    /* 23 Timer/Counter 0 */
        (void *)TC1_Handler,    /* 24 Timer/Counter 1 */
        (void *)TC2_Handler,    /* 25 Timer/Counter 2 */
        (void *)TC3_Handler,    /* 26 Timer/Counter 3 */
        (void *)TC4_Handler,    /* 27 Timer/Counter 4 */
        (void *)TC5_Handler,    /* 28 Timer/Counter 5 */
        (void *)ADC_Handler,    /* 29 Analog To Digital Converter */
        (void *)DACC_Handler,   /* 30 Digital To Analog Converter */
        (void *)PWM_Handler     /* 31 Pulse Width Modulation */
};
```

7.4.1.2 Reset Exception

The handler of reset exception is responsible for starting up the application by performing the following actions:

Table 7-3. Reset Exception Actions

Action	Description
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.
Set 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 the main() application.

7.4.2 System Clock Initialization

At the very beginning of the getting-started example main(), sysclk_init() is called to initialized the system clock of SAM4N. In this function, Power Management Controller (PMC) is set according to the clock configuration file, conf_clock.h.

In the conf_clock.h, the system clock source (CONFIG_SYSCLK_SOURCE) and system clock prescaler (CONFIG_SYSCLK_PRES) must be defined. In the case of the getting-started example, since the Phase Lock Loop block (PLLA) is used to multiply the frequency of the system clock, PLLA source, factor and divider must be defined too.

Clock Configuration

```
// ===== System Clock (MCK) Source Options
#define CONFIG_SYSCLK_SOURCE        SYSCLK_SRC_PLLACK

// ===== System Clock (MCK) Prescaler Options
#define CONFIG_SYSCLK_PRES          SYSCLK_PRES_2

// ===== PLL0 (A) Options
// Use mul and div effective values here.
#define CONFIG_PLL0_SOURCE          PLL_SRC_MAINCK_8M_RC
#define CONFIG_PLL0_MUL             25
#define CONFIG_PLL0_DIV             1
```

As shown in the code above, the 8MHz Fast RC Oscillator (PLL_SRC_MAINCK_8M_RC) is selected as the PLLA source (CONFIG_PLL0_SOURCE). The factor (CONFIG_PLL0_MUL) and divider (CONFIG_PLL0_DIV) are defined as 25 and 1 respectively. PLLA (SYSCLK_SRC_PLLACK) is chosen as the system clock source (CONFIG_SYSCLK_SOURCE), the prescaler of which (CONFIG_SYSCLK_PRES) is defined as 2.

So after calling sysclk_init() with this configuration, the system clock frequency (SYSCLK) is:
SYSCLK = FAST_RC * MUL / DIV / PRES = 8MHz * 25 / 1 / 2 = 100MHz

7.4.3 Board Initialization

To control the on-board components, button, LED, and COM port in the case of the getting-started example, board_init() is called in the main(). With the conf_board.h, the corresponding pins are configured in the appropriate mode.

Board Configuration

```
/** Enable Com Port. */
#define CONF_BOARD_UART_CONSOLE
```

In board_init(), the pin connected to the user pushbutton is configured as input port and the pin connected to LED is configured as output port.

In the getting-started example, CONF_BOARD_UART_CONSOLE is predefined as above, which enables the COM port by configuring PA9 and PA10 as URXD0 and UTXD0 respectively.

7.4.4 Peripherals Configuration and Usage

7.4.4.1 UART

UART outputs the debug information via the COM port in the getting-started example. To display characters on PC terminal software correctly, several parameters must be configured before calling puts() or printf().

In SAM4N, the UART peripheral operates in asynchronous mode only and supports only 8-bit character handling (with parity) and one stop bit. No flow control is supported. So there are the baudrate and parity left to be configured.

UART Parameters

```
/** Baudrate setting */
#define CONF_UART_BAUDRATE   115200
/** Parity setting */
#define CONF_UART_PARITY     UART_MR_PAR_NO
```

In conf_uart_serial.h, the baudrate is set as 115200bps and no parity is used.

UART Configuration

```
const usart_serial_options_t uart_serial_options = {
      .baudrate = CONF_UART_BAUDRATE,
      .paritytype = CONF_UART_PARITY
};

/* Configure console UART. */
sysclk_enable_peripheral_clock(ID_UART0);
stdio_serial_init(UART0, &uart_serial_options);
```

In the above code, the peripheral clock for UART0 is enabled by calling sysclk_enable_peripheral_clock(). Then stdio_serial_init() configures the baudrate and the parity type.

7.4.4.2 SysTick

SysTick can be easily configured by calling SysTick_Config(). To generate 1ms period, the only parameter of this function should be system clock frequency / 1000.

SysTick Configuration

```
SysTick_Config(sysclk_get_cpu_hz() / 1000)
```

sysclk_get_cpu_hz() returns the current system clock frequency in Hz.

Then the SysTick interrupt will be triggered every 1ms. In the getting-started example, the SysTick interrupt handler SysTick_Handler() simply increases a global counter by 1 every time, which is used by the wait function to generate a specified period delay.

SysTick Interrupt Handler

```
volatile uint32_t g_ul_ms_ticks = 0;
void SysTick_Handler(void)
{
	g_ul_ms_ticks++;
}
```

Delay Function

```
static void mdelay(uint32_t ul_dly_ticks)
{
	uint32_t ul_cur_ticks;

	ul_cur_ticks = g_ul_ms_ticks;
	while ((g_ul_ms_ticks - ul_cur_ticks) < ul_dly_ticks);
}
```

Note: The global counter, g_ul_ms_ticks, is declared as a volatile variable. It prevents the compiler from optimizing the code casuing that the delay function does not work.

7.4.4.3 TC

SAM4N provides several 32-bit TC channels, which could be used to measure frequency, count event, generate PWM wave and so on.

In the getting-started example, the TC channel 0 is configured to generate an interrupt per a quarter of a second.

Timer Counter Configuration

```
uint32_t ul_div;
uint32_t ul_tcclks;
uint32_t ul_sysclk = sysclk_get_cpu_hz();

/* Configure PMC */
pmc_enable_periph_clk(ID_TC0);

/** Configure TC for a 4Hz frequency and trigger on RC compare. */
tc_find_mck_divisor(4, ul_sysclk, &ul_div, &ul_tcclks, ul_sysclk);
tc_init(TC0, 0, ul_tcclks | TC_CMR_CPCTRG);
tc_write_rc(TC0, 0, (ul_sysclk / ul_div) / 4);

/* Configure and enable interrupt on RC compare */
NVIC_EnableIRQ((IRQn_Type) ID_TC0);
tc_enable_interrupt(TC0, 0, TC_IER_CPCS);

/** Start the counter */
tc_start(TC0, 0);
```

Before any configuration, TC peripheral clock is enabled. Two necessary parameters; the TC divider and the tick value for the compare register (RC is used in the example), must be calculated to initialize the TC and the compare register. Then the program enables the TC channel 0 interrupt and the compare interrupt. In the end, it starts TC channel 0 and the counter starts ticking.

In the TC channel 0 interrupt handler, the COM port outputs “2 ” every time.

Interrupt Handler for TC Channel 0

```
volatile uint32_t ul_dummy;

/* Clear status bit to acknowledge interrupt */
ul_dummy = tc_get_status(TC0, 0);

/* Avoid compiler warning */
UNUSED(ul_dummy);

printf("2 ");
```

7.4.4.4 PIO

Besides toggling LED, in the getting-started example, PIO retrieves the button input. When a button is pressed, the level of the corresponding pin is changed. PIO detects the change and triggers an interrupt.

PIO Configuration for one button (one pin)

```
/* Configure Pushbutton 1 */
pmc_enable_periph_clk(PIN_PUSHBUTTON_1_ID);
pio_set_debounce_filter(PIN_PUSHBUTTON_1_PIO, PIN_PUSHBUTTON_1_MASK, 10);
/* Interrupt on rising edge */
pio_handler_set(PIN_PUSHBUTTON_1_PIO, PIN_PUSHBUTTON_1_ID,
            PIN_PUSHBUTTON_1_MASK, PIN_PUSHBUTTON_1_ATTR,
Button1_Handler);
NVIC_EnableIRQ((IRQn_Type) PIN_PUSHBUTTON_1_ID);
pio_handler_set_priority(PIN_PUSHBUTTON_1_PIO,
            (IRQn_Type) PIN_PUSHBUTTON_1_ID, IRQ_PRIOR_PIO);
pio_enable_interrupt(PIN_PUSHBUTTON_1_PIO, PIN_PUSHBUTTON_1_MASK);
```

The PIO peripheral clock is enabled at first so that the configuration above can take effect.

Usually in an application with the button inputs, there are some glitches on the input lines of the buttons. In PIO of SAM4N, the debouncing filter can be set to reject these unwanted pulses. In the getting-started example, if the period of a glitch is less than 10 slow clock cycles (slow clock frequency is 32768Hz in this case), the glitch will be ignored by PIO.

There is a specified handler for a specified button pressing. Before enabling the PIO interrupt and any pin interrupt, a handler, Button1_Handler, is set by calling pio_handler_set(). Also the condition to trigger a pin interrupt is chosen here.

In the getting-started example, the user pushbutton SW0 controls LED0. When SW0 is pressed after reset, LED0 stops blinking. Then if SW0 is pressed again, LED0 starts blinking.

Button Pressing Process

```
static void ProcessButtonEvt(uint8_t uc_button)
{
      if (uc_button == 0) {
            g_b_led0_active = !g_b_led0_active;
            if (!g_b_led0_active) {
                  ioport_set_pin_level(LED0_GPIO, IOPORT_PIN_LEVEL_HIGH);
            }
      }
```

```
}

static void Button1_Handler(uint32_t id, uint32_t mask)
{
      if (PIN_PUSHBUTTON_1_ID == id && PIN_PUSHBUTTON_1_MASK == mask) {
            ProcessButtonEvt(0);
      }
}
```

8 Revision History

Doc Rev.	Date	Comments
42169B	09/2014	Updated some sentences in description section.
42169A	08/2013	Initial document release.

Atmel | Enabling Unlimited Possibilities®

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