AN2465

SAM D21 SERCOM SPI Configuration

Introduction

This application note explains the various features of the SERCOM SPI in SAM D21 microcontrollers and its configurations with example codes and corresponding screen-shots.

For demonstration purpose two SAM D21 Xplained Pro boards will be used.

Table of Contents

1. Glossary

ASF: Atmel Software Framework

DFLL48M: 48MHz Digital Frequency Locked Loop

DI: Data Input DMA Direct Memory Access

DO: Data Output EDBG Embedded Debugger

EXT 1/2/3: Extension Header (1/2/3) in Xplained Pro Kit

GCLK: Generic Clock Controller

GPIO: General Purpose I/O-pins

I^2C: Inter-Integrated Circuit

IDE: Integrated Development Environment

LED: Light-emitting diode

MISO: Master In and Slave Out Data Line for SPI Communication

MOSI: Master Out and Slave In Data Line for SPI Communication

OSC8M: 8MHz high-accuracy internal oscillator

SCK: Serial Clock Line for SPI Communication

SERCOM: Serial communication interface

SPI: Serial communication interface

SS: Slave Select Line for SPI Communication

USART: Universal Synchronous/ Asynchronous Receiver/ Transmitter

UART: Universal Asynchronous Receiver/Transmitter

XOSC32K: External 32kHz Crystal Oscillator

2. Pre-requisites

The solutions discussed in this document require basic familiarity with:

- Atmel Studio 6.2 or above
- ASF version 3.22.0 or above
- SAM D21 Xplained Pro kit

3. SERCOM Implementation in SAM D21 Microcontrollers

Generally a microcontroller will have separate serial communication modules with different pinouts for each module. Separate dedicated peripherals and user registers will be available for each module. For example, USART will be a separate peripheral with dedicated pins for its function and I^2C will be a separate peripheral with its own dedicated pins.

In SAM D microcontrollers, all the serial peripherals are designed into a single module as serial communication interface (SERCOM). A SERCOM module can be configured either as USART, I^2C, or SPI, selectable by the user. Each SERCOM will be assigned four pads from PAD0 to PAD3. The functionality of each pad is configurable depending on the SERCOM mode used. Unused pads can be used for other purposes and the SERCOM module will not control them unless they are configured to be used by the SERCOM module.

For example, SERCOM0 can be configured as USART mode with PAD0 as transmit pad and PAD1 as receive pad. Other unused pads (PAD2 and PAD3) can be used either as GPIO pins or be assigned to some other peripherals. The assignment of SERCOM functionality for different pads is highly flexible making the SERCOM module more advantageous compared to the typical serial communication peripheral implementation.

3.1 SERCOM Overview

The serial communication interface (SERCOM) can be configured to support three different modes; I^2C, SPI, or USART. Once configured and enabled, all SERCOM resources are dedicated to the selected mode.

The SERCOM serial engine consists of a transmitter and receiver, baud-rate generator, and address matching functionality. It can be configured to use the internal generic clock or an external clock, making operation in all sleep modes possible.

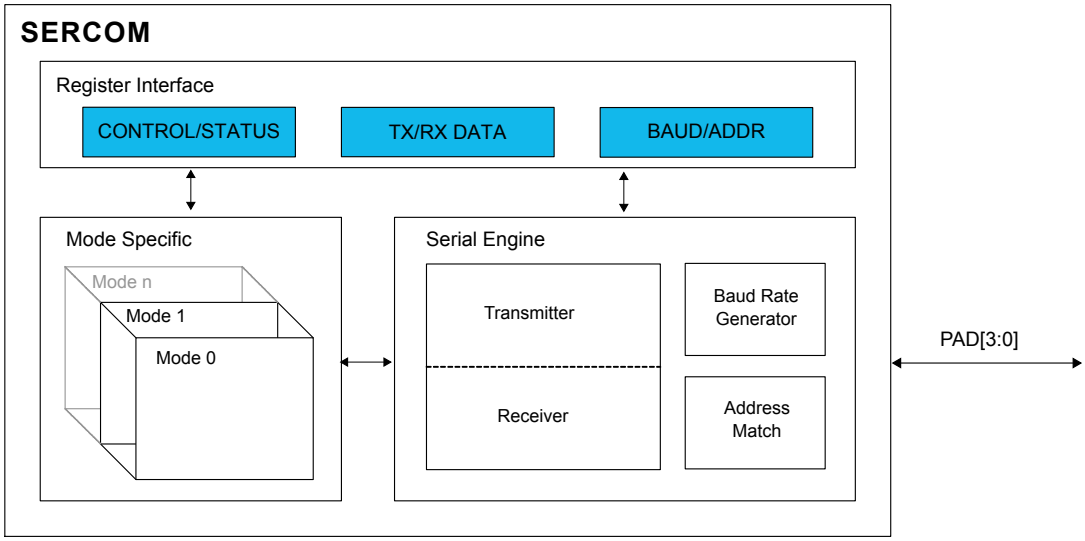
3.2 Features

- Combined interface configurable as one of the following:
  - I^2C – Two-wire serial interface (SMBus compatible)
  - SPI – Serial Peripheral Interface
  - USART – Universal Synchronous/Asynchronous Receiver/Transmitter
- Single transmit buffer and double receive buffers
- Baud-rate generator
- Address match/mask logic
- Operational in all sleep modes
- Can be used with DMA (not supported in SAM D20 MCUs)

3.3 Block Diagram

The figure below shows the block diagram of a SERCOM module. The module mainly consists of a serial engine handling the actual data transfers and mode specific IPs implementing the corresponding protocol.

Figure 3-1. SERCOM Block Diagram



3.4 SPI Implementation in SERCOM

The SPI is single buffered when transmitting and double buffered when receiving. SPI communication in slave will not happen until the SPI Slave select line (_SS) is driven low by the master. Once the slave select line goes low, data to be transmitted should be placed in the master's SPI data register. The SPI Transmit Data register (TxDATA) and SPI Receive Data register (RxDATA) share the same I/O address, referred to as the SPI Data register (DATA). Writing DATA register will update the Transmit Data register. Reading the DATA register will return the content of the Receive Data register. Writing the DATA register by software will move the data into the shift register, if no transfer is ongoing. Once the data is moved into the shift register the DRE (data register empty) interrupt flag is set in the master. This allows the software to write the next data to the DATA register. The data from the master shift register will be transmitted to the slave shift register through the MOSI line.

When data is transmitted using shift register from master through MOSI line the slave will simultaneously transmit the data written to its shift register through the MISO line. Data transmission in both master and slave is based on the common clock signal on the SCK line generated by the master.

The double buffering on the receive side of the SERCOM SPI module is implemented as a FIFO buffer containing RX buffer register and RX data register. After receiving the data in master or in slave the storage of the received data in the RX buffer register or in the RX data register is determined by an internal register pointer. Once the SERCOM SPI module is enabled the register pointer points to the RX data register. The first received data will be placed in the RX data register, which is currently pointed to by the register pointer. Once placing the data, the register pointer will now point to the RX buffer register.

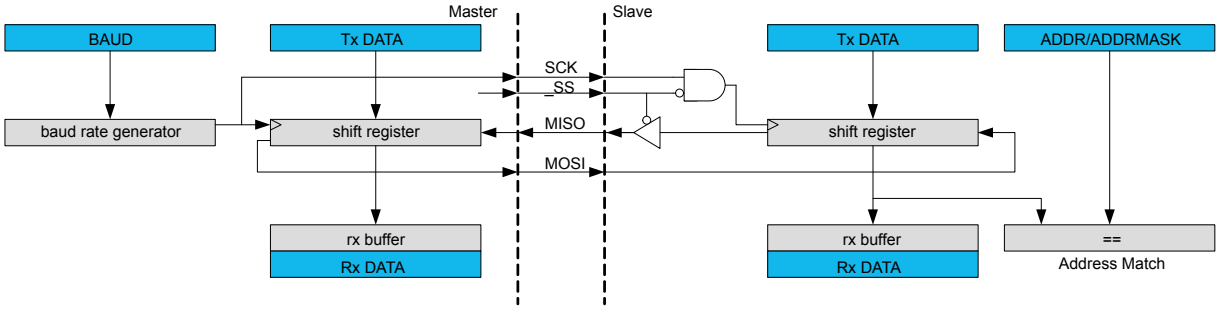
Now there can be two possible scenarios:

1. The software makes a read of the RX data register. This read will return the received data to the software and the register pointer will now point back to the RX data register.
2. The software does not make a read of the received data and the next data reception happens. Now the received data is placed in the RX buffer register and the buffer pointer remains pointing to the buffer register. In this situation there can be two more possible scenarios:
    2.1. The software makes a read of the RX data register. This will return the data in the RX data register, which is basically the data received during the previous transfer and not the current transfer. The data in the RX buffer register is now shifted into the RX data register

and the register pointer remains pointing to the RX buffer register. If the software makes one more read, then the data in the RX data register will be returned back, which is the received data of the current transfer and the register pointer will now point to the RX data register.

2.2. The software does not make a read and the next data reception happens. Now the data in the RX buffer register is transferred to the RX data register (the old data in the RX data register is lost) and the received data is placed into the RX buffer register. The register pointer remains pointing to the RX buffer register.

Figure 3-2. Full-duplex SPI Master Slave Interconnection



3.5 Clocks

The SERCOM module needs three clocks for its operation:

- SERCOM bus clock (APB clock)
- SERCOM CORE generic clock
- SERCOM SLOW generic clock

SERCOM bus clock (CLK_SERCOMx_APB) is used for reading and writing SERCOM registers by the CPU. This clock is disabled by default and can be enabled or disabled in the Power Manager (PM) module.

Two generic clocks are used by the SERCOM module; GCLK_SERCOMx_CORE and GCLK_SERCOMx_SLOW. The generic clocks are used for SERCOM's operation. All the SERCOM communication timings are based on the generic clocks.

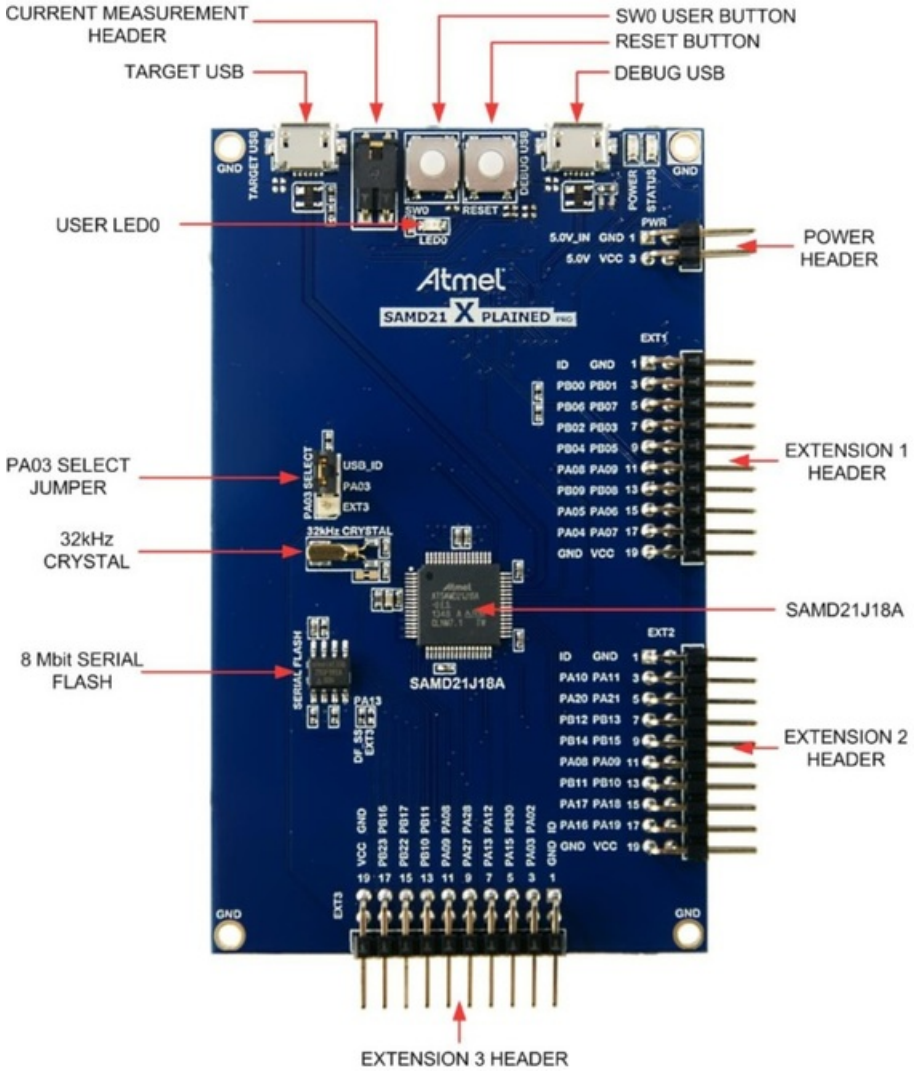
The core clock (GCLK_SERCOMx_CORE) is required to clock the SERCOM while operating as a master, while the slow clock (GCLK_SERCOMx_SLOW) is only required for certain functions like I^2C timeouts.

Note: In this application note only the SERCOM bus clock (CLK_SERCOMx_APB) and core clock (GCLK_SERCOMx_CORE) are used.

4. Hardware and Software Requirements

The application demonstration needs two SAM D21 Xplained Pro boards. One board will be configured as master and then other board as slave.

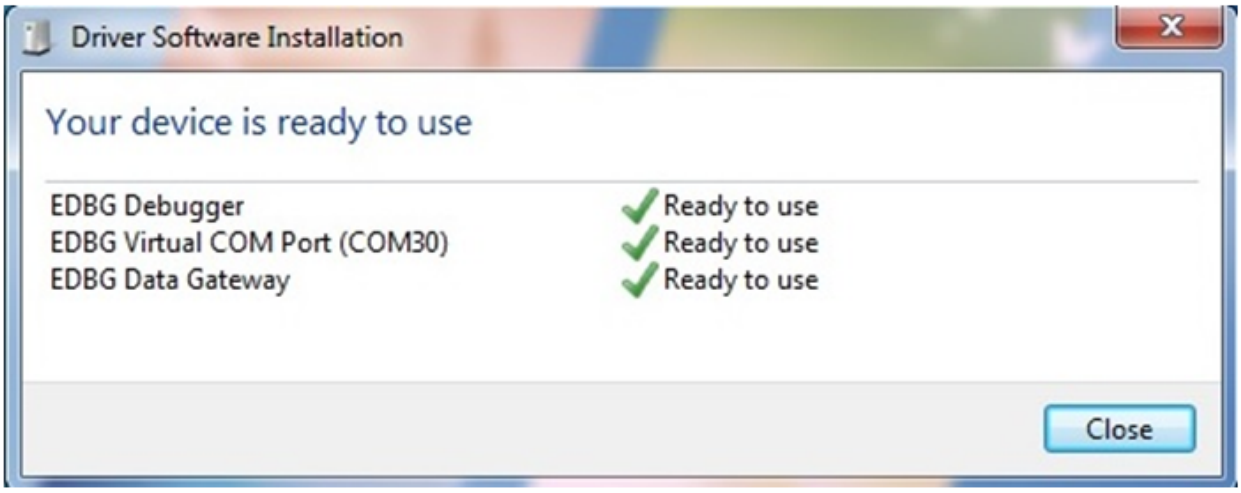
Figure 4-1. SAM D21 Xplained Pro Board



There are two USB ports on the SAM D21 Xplained Pro board; the DEBUG USB and the TARGET USB. For debugging the target SAM D21 MCU using the Embedded debugger (EDBG) a Micro-B USB cable should be connected between a host PC running Atmel Studio and the DEBUG USB port on the SAM D21 Xplained Pro board.

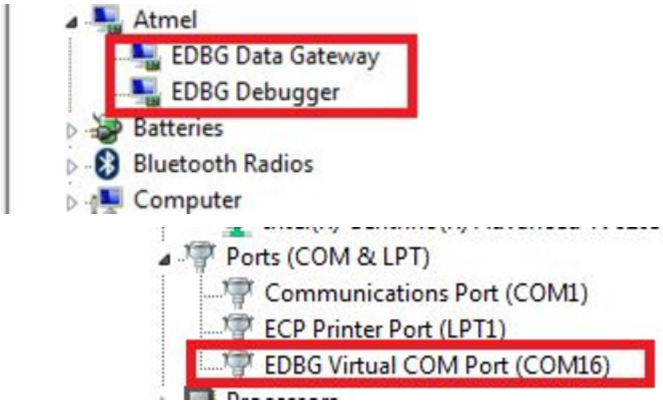
Once the kit is successfully connected for the first time, the Windows® task bar will pop up a message as shown in the figure below.

Figure 4-2. SAM D21 Xplained Pro Driver Installation



If the driver installation is proper, the EDBG will be listed in the Device Manager as shown in the figure below.

Figure 4-3. Successful EDBG Driver Installation



Application codes are tested in Atmel Studio 6.2 with ASF version 3.22.0 and above. Two projects are needed for implementing the functionalities; one for the master and the other for the slave. The GCC C ASF Board project from Atmel Studio is used for the implementation.

To create an ASF board project for the SAM D21 Xplained Pro board, go to the file menu → New → Project and select “GCC C ASF Board project” in the new project wizard.

Figure 4-4. New Project in Atmel Studio

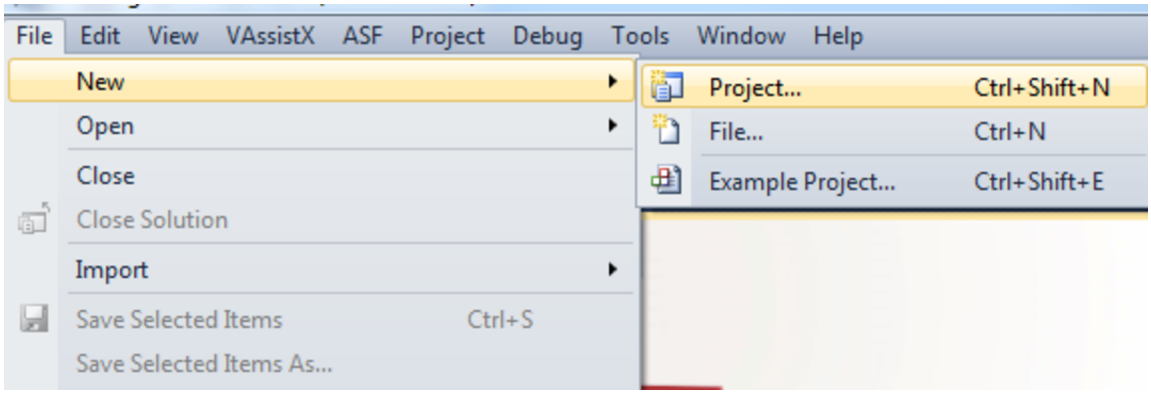
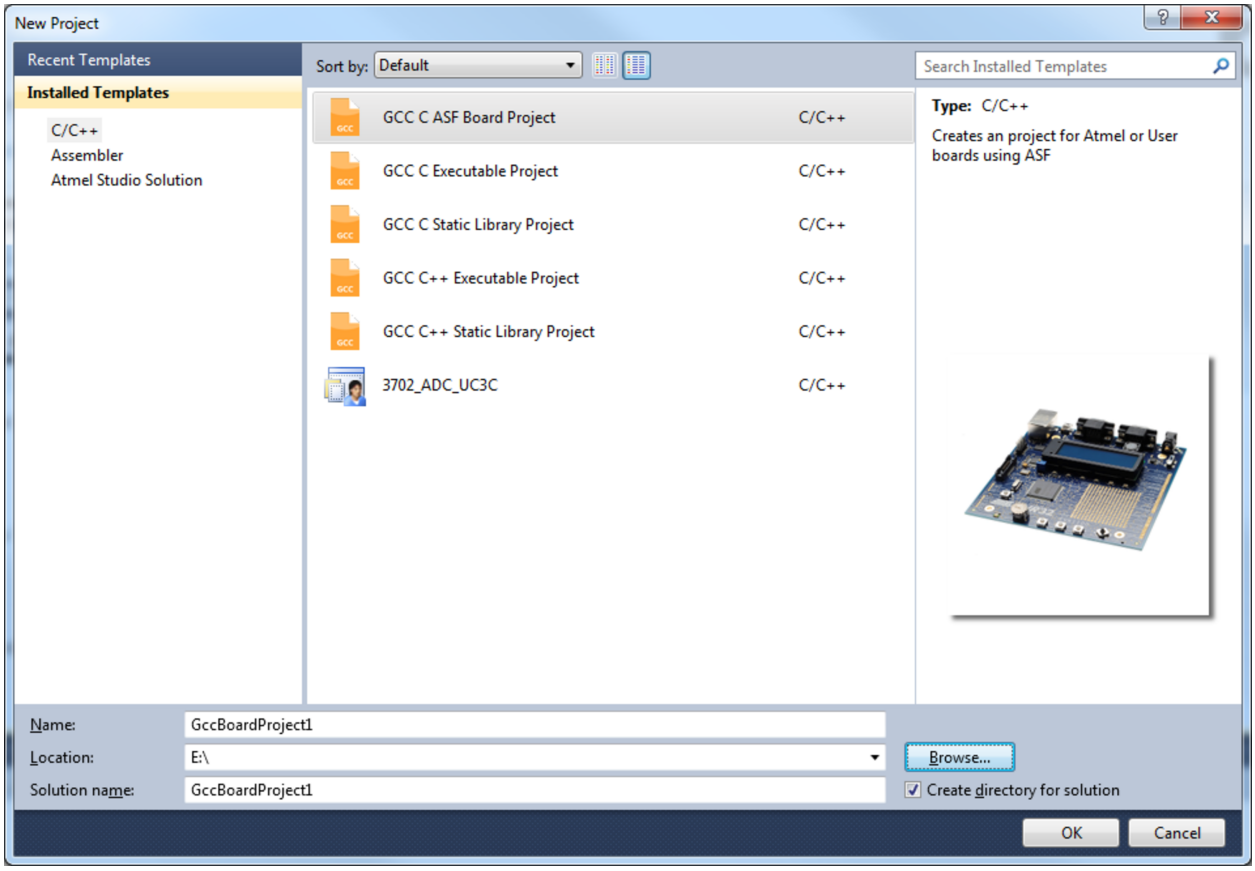
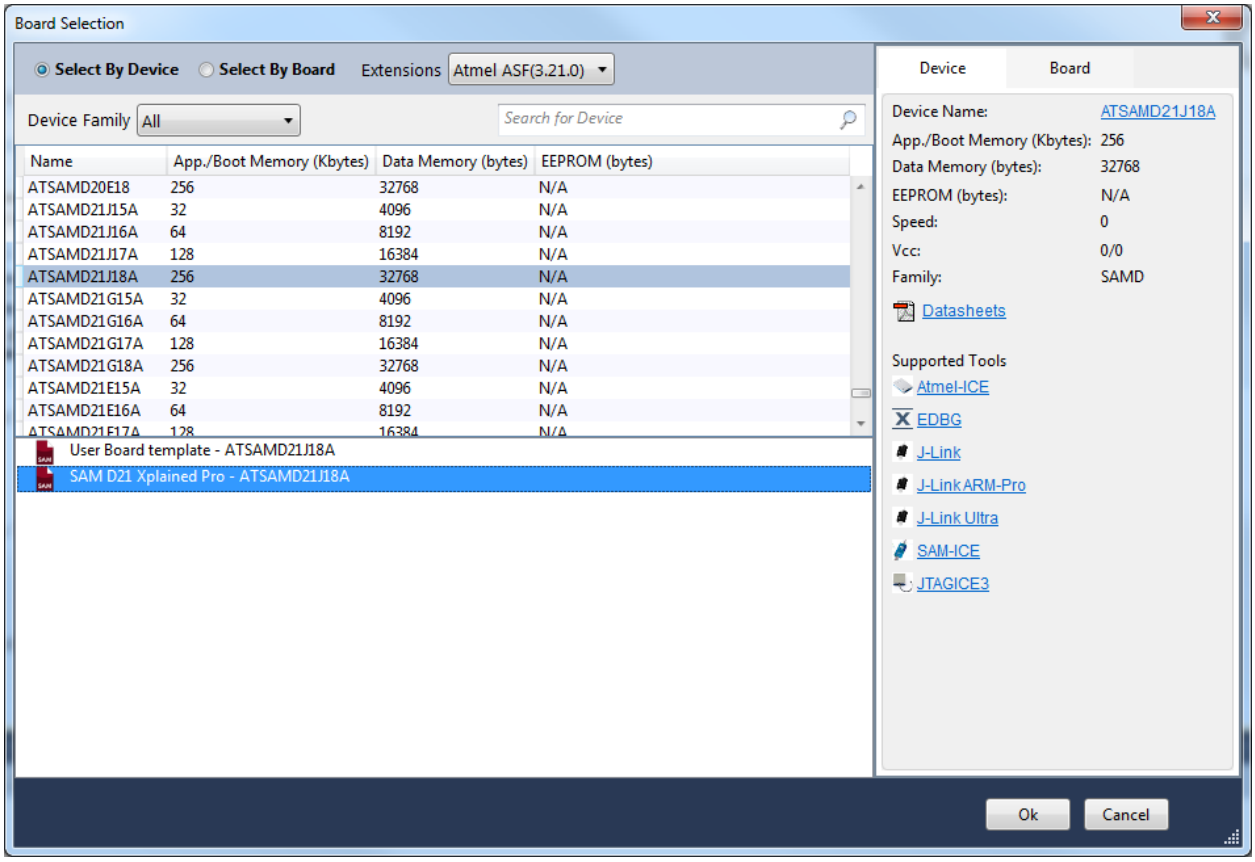


Figure 4-5. ASF Board Project



In the next window, select the device family as "SAM D", scroll down and select the device "ATSAMD21J18A" and board as "SAM D21 Xplained PRO - ATSAMD21J18A", and click on "OK" to create the new project.

Figure 4-6. Device and Board Selection



The new project by default has a minimal application that will turn ON or OFF the LED on the SAM D21 Xplained Pro based on the state of the SW0 push button. Pressing the SW0 button will turn the LED ON and releasing the button will turn the LED OFF. To verify that the SAM D21 Xplained Pro is connected correctly this application can be run and checked whether it shows the expected output.

5. Application Demonstration

This chapter will demonstrate the various features of the SERCOM SPI module of SAM D21 with different example codes. The following examples are demonstrated in this application note:

- Basic Configuration
- Slave preloading configuration
- Hardware controlled SS and SS low detection configuration

Note: This chapter assumes that the user has previous knowledge on programming/debugging a SAM D21 device using Atmel Studio IDE.

For easier understanding, the examples will use the register level coding for the SERCOM module configuration. The clock configuration will, however, use ASF functions.

5.1 Basic Configuration

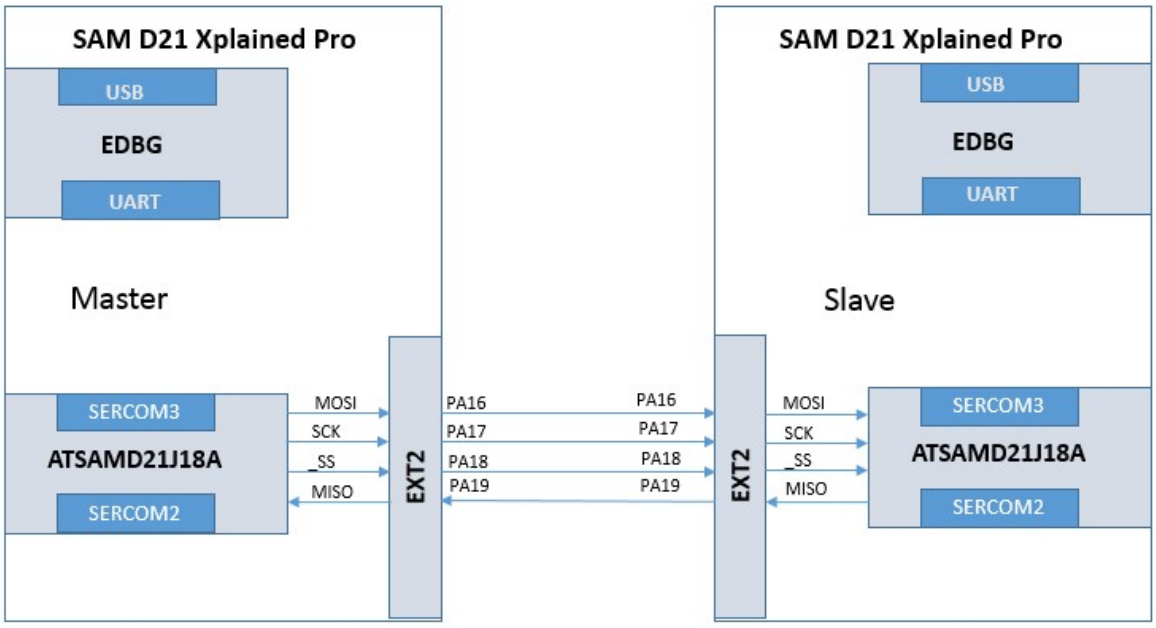
In the Basic configuration application, the master will transmit a data buffer of a few bytes to the slave and the slave will re-transmit the same data buffer to the master.

The basic configuration application performs the following actions:

- Master write (Slave read)
- Slave write (Master read)

The SERCOM SPI lines of the two SAM D21 Xplained Pro boards should be connected through the EXT2 connector using wires as shown in the figure below.

Figure 5-1. SERCOM SPI Connection Diagram



The common function calls used in both master and slave applications for the basic configuration examples are shown below.

```
system_init()
```

```
spi_clock_init()
```

Detailed explanation of each function will be provided in the upcoming sections.

5.1.1 Main Clock

In SAM D21 devices, the output from GCLK Generator 0 will be used as the main clock. The Generic Clock Generator 0, also called GCLK_MAIN, is the clock feeding the Power Manager used to generate synchronous clocks. The GCLK Generator 0 can have one of the SYSCTRL oscillators as its source clock.

By default, after reset, the 1MHz clock from the OSC8M (prescaler set to 8) is used as the clock source for the GCLK Generator 0 and hence the main clock. However, as per the default ASF clock configuration, 8MHz clock from OSC8M (prescaler set to 1) is used as the clock source for GCLK Generator 0.

5.1.2 Master and Slave Clock Configuration

The default ASF clock configuration in the conf_clocks.h header file should be changed to make the device as well as the SERCOM module clocked at a maximum speed of 48MHz.

The following changes should be implemented in the conf_clocks.h file in both master and slave applications for 48MHz operation.

1. Set the flash wait-states to 1.

```
#  define CONF_CLOCK_FLASH_WAIT_STATES  1
```

2. Configure and enable the XOSC32K oscillator, which will be used as the reference clock for the DFLL48M module.

```
/* SYSTEM_CLOCK_SOURCE_XOSC32K configuration - External 32KHz crystal/clock oscillator */
#  define CONF_CLOCK_XOSC32K_ENABLE               true
#  define CONF_CLOCK_XOSC32K_EXTERNAL_CRYSTAL     SYSTEM_CLOCK_EXTERNAL_CRYSTAL
#  define CONF_CLOCK_XOSC32K_STARTUP_TIME         SYSTEM_XOSC32K_STARTUP_65536
#  define CONF_CLOCK_XOSC32K_AUTO_AMPLITUDE_CONTROL  false
#  define CONF_CLOCK_XOSC32K_ENABLE_1KHZ_OUPUT    false
    #  define CONF_CLOCK_XOSC32K_ENABLE_32KHZ_OUTPUT  true
#  define CONF_CLOCK_XOSC32K_ON_DEMAND            true
#  define CONF_CLOCK_XOSC32K_RUN_IN_STANDBY       false
```

3. Set XOSC32K as the clock source for the GCLK Generator 1.

```
/* Configure GCLK generator 1 */
#  define CONF_CLOCK_GCLK_1_ENABLE                true
#  define CONF_CLOCK_GCLK_1_RUN_IN_STANDBY        false
#  define CONF_CLOCK_GCLK_1_CLOCK_SOURCE          SYSTEM_CLOCK_SOURCE_XOSC32K
#  define CONF_CLOCK_GCLK_1_PRESCALER             1
#  define CONF_CLOCK_GCLK_1_OUTPUT_ENABLE         false
```

4. Configure and enable DFLL48M in closed loop mode using GCLK Generator 1 as reference clock generator and with appropriate multiplication factor.

```
/* SYSTEM_CLOCK_SOURCE_DFLL configuration - Digital Frequency Locked Loop */
#  define CONF_CLOCK_DFLL_ENABLE                  true
#  define CONF_CLOCK_DFLL_LOOP_MODE               SYSTEM_CLOCK_DFLL_LOOP_MODE_CLOSED
#  define CONF_CLOCK_DFLL_ON_DEMAND               false

/* DFLL closed loop mode configuration */
#  define CONF_CLOCK_DFLL_SOURCE_GCLK_GENERATOR   GCLK_GENERATOR_1
#  define CONF_CLOCK_DFLL_MULTIPLY_FACTOR         (48000000 / 32768)
#  define CONF_CLOCK_DFLL_QUICK_LOCK              true
```

```
#  define CONF_CLOCK_DFLL_TRACK_AFTER_FINE_LOCK   true
#  define CONF_CLOCK_DFLL_KEEP_LOCK_ON_WAKEUP     true
#  define CONF_CLOCK_DFLL_ENABLE_CHILL_CYCLE      true
#  define CONF_CLOCK_DFLL_MAX_COARSE_STEP_SIZE    (0x1f / 4)
#  define CONF_CLOCK_DFLL_MAX_FINE_STEP_SIZE      (0xff / 4)
```

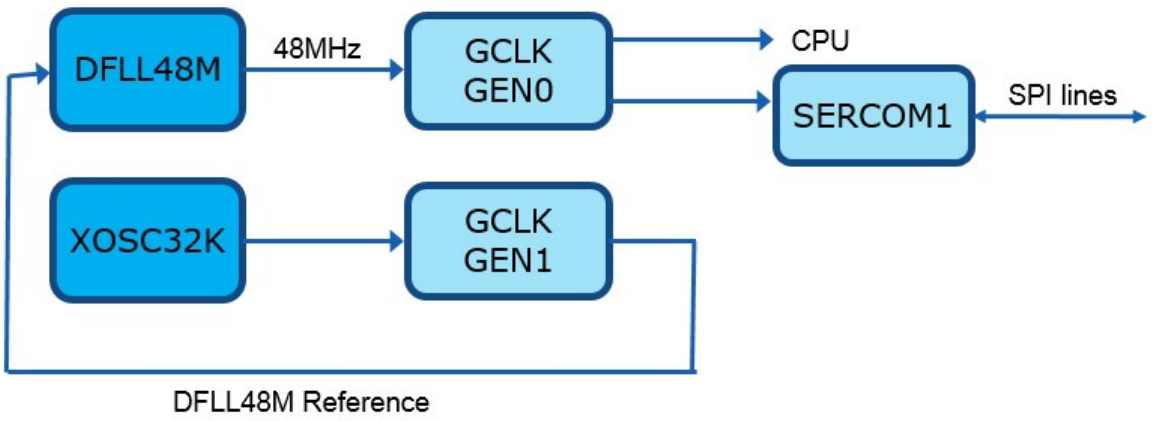
5. Set DFLL48M as clock source for GCLK Generator 0, which sources the main clock domain and is also used as clock source for the SERCOM module.

```
/* Configure GCLK generator 0 (Main Clock) */
#  define CONF_CLOCK_GCLK_0_ENABLE                true
#  define CONF_CLOCK_GCLK_0_RUN_IN_STANDBY        false
#  define CONF_CLOCK_GCLK_0_CLOCK_SOURCE          SYSTEM_CLOCK_SOURCE_DFLL

#  define CONF_CLOCK_GCLK_0_PRESCALER             1
#  define CONF_CLOCK_GCLK_0_OUTPUT_ENABLE         false
```

5.1.3 Clock Flow for Master and Slave

Figure 5-2. Clock Flow Diagram for Master and Slave



5.1.4 System Initialization

The system_init() is an ASF function used to configure the clock sources and GCLK generators as per the settings in the conf_clocks.h file. The main clock will be configured as stated in Section 5.1.1. It also initializes the board hardware of the SAM D21 Xplained Pro and the event system.

5.1.5 SPI Clock Initialization for Master

The spi_clock_init() function configures the peripheral bus clock (APB clock) and generic clock for the SERCOM SPI module. SERCOM1 is used in both the master and slave boards.

```
void spi_clock_init()
{
    struct system_gclk_chan_config gclk_chan_conf;
    uint32_t gclk_index = SERCOM1_GCLK_ID_CORE;
    /* Turn on module in PM */
    system_apb_clock_set_mask(SYSTEM_CLOCK_APB_APBC, PM_APBCMASK_SERCOM1);
    /* Turn on Generic clock for USART */
    system_gclk_chan_get_config_defaults(&gclk_chan_conf);
    //Default is generator 0. Other wise need to configure like below
    /* gclk_chan_conf.source_generator = GCLK_GENERATOR_1; */
    system_gclk_chan_set_config(gclk_index, &gclk_chan_conf);
    system_gclk_chan_enable(gclk_index);
}
```

- A structure variable gclk_chan_conf is declared. This structure is used to configure the generic clock for the SERCOM used.
- SERCOM1 core clock “SERCOM1_GCLK_ID_CORE” and bus clock “SYSTEM_CLOCK_APB_APBC” are configured
- Generic clock “SERCOM1_GCLK_ID_CORE” uses GCLK Generator 0 as source generator (generic clock source can be changed to any other GCLK Generators as per user needs). So the SERCOM1 module is clocked at 48MHz from the DFLL48M.
- system_gclk_chan_set_config will set the generic clock channel configuration
- system_gclk_chan_enable will enable the generic clock “SERCOM1_GCLK_ID_CORE”

The following sections are common for all three demonstrations in this application note including both master and slave:

- Master and slave clock configuration
- Clock flow for master and slave
- System initialization
- SPI Clock initialization.

5.1.6 SPI Master Pin Initialization

The spi_master_pin_init() function will initialize pins PA16, PA17, and PA19 to the SERCOM-Alternate peripheral function (C). It configures pin PA18 as GPIO pin. This GPIO pin is used as slave select line (_SS). In Master side, slave select line should be output. Initial state of PA18 is kept as logic high.

```
void spi_master_pin_init()
{
    /* configuring GPIO pin PA18 as output
    for slave select line */
    struct port_config pin_conf = {
        .direction = PORT_PIN_DIR_OUTPUT,
        .input_pull = PORT_PIN_PULL_NONE,
        .powersave = false };
    port_pin_set_config(PIN_PA18, &pin_conf);
    port_pin_set_output_level(PIN_PA18, true);
    /* PA16, PA17 and PA19 set into peripheral function*/
    pin_set_peripheral_function(PINMUX_PA16C_SERCOM1_PAD0);
    pin_set_peripheral_function(PINMUX_PA17C_SERCOM1_PAD1);
    pin_set_peripheral_function(PINMUX_PA19C_SERCOM1_PAD3);
}
```

The spi_master_pin_init() function calls the pin_set_peripheral_function to assign I/O lines to the SERCOM peripheral function.

```
static void pin_set_peripheral_function(uint32_t pinmux)
{
    uint8_t port = (uint8_t)((pinmux >> 16)/32);
    PORT->Group[port].PMUX[((pinmux >> 16) - (port*32))/2].reg &= ~(0xF << (4 * ((pinmux >>
16) & 0x01u)));
    PORT->Group[port].PMUX[((pinmux >> 16) - (port*32))/2].reg |= (uint8_t)((pinmux &
0x0000FFFF) << (4 * ((pinmux >> 16) & 0x01u)));
    PORT->Group[port].PINCFG[((pinmux >> 16) - (port*32))].bit.PMUXEN = 1;
}
```

5.1.7 SPI Master Initialization

The spi_master_init function will initialize the SPI master function by configuring the control registers, baud registers, and enabling the SERCOM interrupt.

```
void spi_master_init()
{
    /*PAD3 for MISO,PAD0-MOSI,PAD1-SCK,PAD2-CS */
    SERCOM1->SPI.CTRLA.reg = SERCOM_SPI_CTRLA_DIPO(0x3) | SERCOM_SPI_CTRLA_MODE_SPI_MASTER;
    /* synchronization busy */
```

```
    while(SERCOM1->SPI.SYNCBUSY.bit.CTRLB);
    /* SPI receiver is enabled */
    SERCOM1->SPI.CTRLB.reg = SERCOM_SPI_CTRLB_RXEN;
    /* synchronization busy */
    while(SERCOM1->SPI.SYNCBUSY.bit.CTRLB);
    /*baud register value corresponds to the SPI speed */
    SERCOM1->SPI.BAUD.reg = (system_gclk_chan_get_hz(SERCOM1_GCLK_ID_CORE)/(2*SPI_SPEED)) -
1;
    /* SERCOM1 peripheral enabled */
    SERCOM1->SPI.CTRLA.reg |= SERCOM_SPI_CTRLA_ENABLE;
    /* synchronization busy */
    while(SERCOM1->SPI.SYNCBUSY.reg & SERCOM_SPI_SYNCBUSY_ENABLE);
    /* SERCOM1 interrupt handler mapped to callback handler 'SERCOM1_App_Handler' */
    _sercom_set_handler (SERCOM_INTRANCE_INDEX, (sercom_handler_t) SERCOM1_App_Handler);
    /* SERCOM1 handler enabled */
    system_interrupt_enable(SERCOM1_IRQn);
}
```

- The CTRLA register is used to configure the data order, SPI mode, and the SPI lines to the PAD. In the above function SPI is configured as master. In Master operation **DI** is **MISO** and **DO** pin is **MOSI**. Tables "SERCOM SPI Signals" and "SPI Pin Configuration" in the SAM D21 data sheet gives the settings to configure the SPI pin functionalities to the PAD. In the above function DIPO field bit is configured as 0x3 so PAD3 is **DI**. The DOPO field is not configured in the above function so the reset value of 0x0 will be present in that field. As DOPO is 0x0, PAD0 is **DO** pin and PAD1 is SCK pin.
- The CTRLB register is used to enable the SPI receiver mode, character size, and address mode. In the above function the SPI receiver mode is enabled.
- The following formula is used to determine the BAUD value to be loaded in the BAUD register:

$f_{BAUD} = f_{REF}/2(BAUD+1)$

f_{REF} = SERCOM generic clock frequency

f_{BAUD} = SPI clock frequency

BAUD = BAUD register value

From the equation,

$BAUD = (f_{REF}/(2 * fBAUD)) - 1$

In the application the SERCOM runs at generator 0 frequency and the SPI clock is 50kHz.

system_gclk_chan_get_hz function will retrieve the SERCOM generic clock frequency.

- CTRLA, CTRLB, and BAUD registers can be written only when the SPI is disabled because these registers are enable protected. So once configuring these registers the SPI is enabled.
- Due to the asynchronicity between CLK_SERCOMx_APB and GCLK_SERCOMx_CORE, some registers must be synchronized when accessed. The CTRLA and CTRLB register is Write-Synchronized so the application should wait until the synchronization busy flag (CTRLB bit and ENABLE bit in SYNCBUSY register) is cleared after performing a write to this register.
- Each peripheral has a dedicated interrupt line, which is connected to the **Nested Vector Interrupt Controller** in the Cortex®-M0+ core. In the above function the SERCOM1 interrupt request line (IRQ - 10) is enabled.

Note: In the register the bit fields, which are not configured, will hold its reset value. The reset value can be found in the register description.

5.1.8 SPI Master Transaction

The spi_master_send function is used to perform a transaction with the connected slave device.

```
void spi_master_send()
{
    i = 0;
    j = 0;
    tx_done = false;
    /* Slave_select line is made into low
    to start the communication */
    port_pin_set_output_level(PIN_PA18, false);
    //delay_ms(1);
    /*Data regsiter empty and receive complete interrupt is enabled */
    SERCOM1->SPI.INTENSET.reg = SERCOM_SPI_INTENSET_DRE | SERCOM_SPI_INTENSET_RXC;
    while (!tx_done);
    /* Slave_select line is made into low
    to start the communication */
    port_pin_set_output_level(PIN_PA18, true);
              SERCOM1->SPI.CTRLA.reg &= ~SERCOM_SPI_CTRLA_ENABLE;
           while(SERCOM1->SPI.SYNCBUSY.reg & SERCOM_SPI_SYNCBUSY_ENABLE);
}
```

- In the master application a global variable for iteration count and Boolean variable to indicate transmission done status are used

```
uint8_t i = 0, j = 0;
volatile bool tx_done;
```

- SPI master must initiate transaction by pulling the slave select line (_SS) low. In master application pin PA18 is connected with the slave, so once pulling this pin low the SPI transaction starts.
- The port_pin_set_output_level function sets the output value of a pin with the level given at its argument
- The INTENSET register is used to enable the interrupt. In the above function data register empty and receive complete interrupts are enabled.
- The Boolean flag variable tx_done is initialized as false, so it remains in the while loop until the SERCOM1 handler sets it to true indicating the completion of transaction

```
while(!tx_done);
```

- In the SERCOM1 handler, two interrupt conditions are checked:
  - Data Ready interrupt
  - Receive complete interrupt

```
void SERCOM1_Handler()
{
    /* Data register empty flag set */
    if (SERCOM1->SPI.INTFLAG.bit.DRE && SERCOM1->SPI.INTENSET.bit.DRE) {
        SERCOM1->SPI.DATA.reg = tx_buffer[i++];
        if (i == 5) {
            SERCOM1->SPI.INTENCLR.reg = SERCOM_SPI_INTENCLR_DRE;
        }
    }
    /* receive complete interrupt */
    if (SERCOM1->SPI.INTFLAG.bit.RXC && SERCOM1->SPI.INTENSET.bit.RXC) {
        rx_buffer[j++] = SERCOM1->SPI.DATA.reg;
        if (j == 5) {
            SERCOM1->SPI.INTENCLR.reg = SERCOM_SPI_INTENCLR_RXC;
            tx_done = true;
        }
    }
}
```

- Data Ready interrupt is set when SPI data byte is transmitted from data register. Data byte will then move into shift register.
- Receive complete interrupt is set when there are unread data in the RX data register

- When master transmits the data byte from tx_buffer, code will enter into the below loop of slave SERCOM1 handler

```
/* Data register empty flag set */
    if (SERCOM1->SPI.INTFLAG.bit.DRE && SERCOM1->SPI.INTENSET.bit.DRE) {
        SERCOM1->SPI.DATA.reg = tx_buffer[i++];
        if (i == 5) {
            SERCOM1->SPI.INTENCLR.reg = SERCOM_SPI_INTENCLR_DRE;
        }
    }
```

- Inside the SERCOM1_Handler handler function, after the data transmission, the Data Register Empty (DRE) flag is checked in both Interrupt Flag Status and Clear register (INTFLAG) and Interrupt Enable Set register (INTENSET). Similarly after data reception, the Receive Complete flag (RXC) is checked in both Interrupt Flag Status and Clear register (INTFLAG) and Interrupt Enable Set register (INTENSET). If both are set then tx_buffer data will be placed in the data register.
- Buffer size has been set by macro BUF_SIZE
- Once the iteration variable *i* reaches the value of 5 it means all data is transferred and the data register empty interrupt is cleared
- When the master transmits the data through MOSI lines, it enters into the slave shift register and the shift register content will be transferred to the master through the MISO line. The initial value of the slave shift register will be zero and this value will reach the master.
- Once the master receives the slave shift register value zero as stated in the above point, the master application code will enter into the below loop of slave SERCOM1 handler
- For each master transfer it receives the data byte from the slave shift register

```
if (SERCOM1->SPI.INTFLAG.bit.RXC && SERCOM1->SPI.INTENSET.bit.RXC) {
        rx_buffer[j++] = SERCOM1->SPI.DATA.reg;
        if (j == 5) {
            SERCOM1->SPI.INTENCLR.reg = SERCOM_SPI_INTENCLR_RXC;
            tx_done = true;
        }
    }
```

- In receive complete interrupt checking, both the RXC interrupt flag and the RXC interrupt set should be checked
- Received data bytes are read from the data register and stored in the receive buffer rx_buffer
- Once the iteration variable *j* reaches the value of 5 it means all the data bytes are received and the receive complete interrupt is cleared
- Now the Boolean variable tx_done is set to true and the control jump backs to the spi_master_send and comes out of the tx_done loop
- The GPIO pin PA18 is set to logic high to stop the SPI communication and the SERCOM is disabled by writing low in the ENABLE bit of the CTRLA register

Note: CTRLA, CTRLB, ADDR, and DATA registers are write synchronized so the SYSOP bit in the SYNCBUSY register should be checked after writing these registers.

The final application “Basic Configuration for Master” will be in the Zip attachment to this application note.

5.1.9 SPI Clock Initialization for Slave

The explanation for this section is the same as for the SPI Clock Initialization for Master section.

5.1.10 SPI Slave Pin Initialization

The spi_slave_pin_init() function will initialize pins PA16, PA17, PA18, and PA19 to the SERCOM-Alternate peripheral function (C).

```
void spi_slave_pin_init()
{
   /* pin18 pull-resistor is set into high */
    PORT->Group[0].PINCFG[18].reg = PORT_PINCFG_INEN | PORT_PINCFG_PULLEN;
    PORT->Group[0].OUTSET.reg = (1u << 18);
    pin_set_peripheral_function(PINMUX_PA16C_SERCOM1_PAD0); //MOSI
    pin_set_peripheral_function(PINMUX_PA17C_SERCOM1_PAD1); //SCK
    pin_set_peripheral_function(PINMUX_PA18C_SERCOM1_PAD2); //SS
    pin_set_peripheral_function(PINMUX_PA19C_SERCOM1_PAD3); //MISO
}
```

The spi_slave_pin_init() function calls the pin_set_peripheral_function to assign I/O lines to the SERCOM peripheral function.

```
static void pin_set_peripheral_function(uint32_t pinmux)
{
    uint8_t port = (uint8_t)((pinmux >> 16)/32);
    PORT->Group[port].PMUX[((pinmux >> 16) - (port*32))/2].reg &= ~(0xF << (4 * ((pinmux >>
16) & 0x01u)));
    PORT->Group[port].PMUX[((pinmux >> 16) - (port*32))/2].reg |= (uint8_t)((pinmux &
0x0000FFFF) << (4 * ((pinmux >> 16) & 0x01u)));
    PORT->Group[port].PINCFG[((pinmux >> 16) - (port*32))].bit.PMUXEN = 1;
}
```

- Slave select line (_SS) is connected with pin PA18, which is SERCOM1 PAD [2]. This line will be connected to the SPI master and once the line is pulled low by the master the SPI communication starts.
- During idle condition when master is not pulling the slave select line (_SS) the low level noise signal can appear in the slave select line, which results in malfunction. To avoid this, pin PA18 input buffer is enabled and internal pull resistor is enabled.

  ```
  PORT->Group[0].PINCFG[18].reg = PORT_PINCFG_INEN | PORT_PINCFG_PULLEN;
  ```

- Pull resistor value is tied to logic high by setting its value in OUTSET register as one.

  ```
  PORT->Group[0].OUTSET.reg = (1u << 18);
  ```

5.1.11 SPI Slave Initialization

The spi_slave_init function will initialize the SPI slave function by configuring the control registers and enabling the SERCOM handler.

```
void spi_slave_init()
{
    /* MOSI(DI)-PAD0,SCK-PAD1,SS-PAD2,MISO(DO)-PAD3
    SPI in slave mode */
    SERCOM1->SPI.CTRLA.reg = SERCOM_SPI_CTRLA_DOPO(0x2) |
                    SERCOM_SPI_CTRLA_MODE_SPI_SLAVE ;
    /* SPI receiver enabled */
    SERCOM1->SPI.CTRLB.reg = SERCOM_SPI_CTRLB_RXEN ;
    /* synchronization busy */
    while(SERCOM1->SPI.SYNCBUSY.bit.CTRLB);
    /* SERCOM1 enabled */
    SERCOM1->SPI.CTRLA.reg |= SERCOM_SPI_CTRLA_ENABLE;
    /* synchronization busy */
    while(SERCOM1->SPI.SYNCBUSY.reg & SERCOM_SPI_SYNCBUSY_ENABLE);
    /* SERCOM1 interrupt handler mapped to callback handler 'SERCOM1_App_Handler' */
    _sercom_set_handler (SERCOM_INTRANCE_INDEX, (sercom_handler_t) SERCOM1_App_Handler);
    /* SERCOM1 handler enabled */
    system_interrupt_enable(SERCOM1_IRQn);
}
```

- The CTRLA register is used to configure the data order, SPI mode, and the SPI lines to the PAD. In the above function the SPI is configured as Slave. In Slave operation **DI** is **MOSI** and **DO** pin is **MISO**. The tables "SERCOM SPI Signals" and "SPI Pin Configuration" in the SAM D21 data sheet gives the settings to configure the SPI pin functionalities to the PAD. In the above function the DOPO field bit is configured as 0x2 so PAD3 is **DO**, PAD1 is SCK, and PAD2 is slave select line. The DIPO field is not configured in the above function so the reset value of 0x0 will be present in that field. As DIPO is 0x0, PAD0 is **DI** pin.
- CTRLA, CTRLB can be written only when the SPI is disabled because these registers are enable protected. So once configuring these registers the SPI is enabled.
- Due to the asynchronicity between CLK_SERCOMx_APB and GCLK_SERCOMx_CORE, some registers must be synchronized when accessed. The CTRLA and CTRLB register is Write-Synchronized so the application should wait until the synchronization busy flag (CTRLB bit and ENABLE bit in SYNCBUSY register) is cleared after performing a write to this register.
- Each peripheral has a dedicated interrupt line, which is connected to the **Nested Vector Interrupt Controller** in the Cortex-M0+ core. In the above function the SERCOM1 interrupt request line (IRQ - 10) is enabled.

5.1.12 SPI Slave Transaction

The spi_slave_rx_data function is used to perform a transaction with the connected master device.

```
void spi_slave_rx_data()
{
    i = 0;
    SERCOM1->SPI.DATA.reg = tx_buffer[0];
    j = 1;
    SERCOM1->SPI.INTENSET.reg = SERCOM_SPI_INTENSET_RXC | SERCOM_SPI_INTENSET_DRE;
    while(!rx_done);
}
```

- In slave application a global variable for iteration count and Boolean variable to indicate reception done status are used. The Boolean variable flag is set as false.

```
uint8_t i = 0, j = 0;
volatile bool rx_done;
```

- Data byte received from the master will be in the receive buffer rx_buffer. At the same time the slave shift register content will be transmitted to the master through the MISO line so the data byte from the transmit buffer tx_buffer is placed in slave shift register.
- The first data byte is placed in the TX DATA register from the transmit buffer and the Boolean variable used for transmit is initialized as 1
- The INTENSET register is used to enable the interrupt. In the above function data register empty and receive complete interrupts are enabled.
- The Boolean flag variable rx_done is initialized as false, so it remains in the while loop until the SERCOM1 handler sets it to true indicating the completion of reception

```
while(!rx_done);
```

- In the SERCOM1 handler, two interrupt conditions are checked:
  - Data Ready interrupt
  - Receive complete interrupt
-

```
void SERCOM1_Handler()
{
    /* Data register empty flag set */
    if(SERCOM1->SPI.INTFLAG.bit.DRE && SERCOM1->SPI.INTENSET.bit.DRE)
    {
        SERCOM1->SPI.DATA.reg = tx_buffer[j++];
```

```
            if (j == 5)
                SERCOM1->SPI.INTENCLR.reg = SERCOM_SPI_INTENCLR_DRE;
        }
        /* receive complete interrupt */
        if(SERCOM1->SPI.INTFLAG.bit.RXC && SERCOM1->SPI.INTENSET.bit.RXC)
        {
            rx_buffer[i++] = SERCOM1->SPI.DATA.reg;
            if(i == 5){
                SERCOM1->SPI.INTENCLR.reg = SERCOM_SPI_INTENCLR_RXC;
                rx_done = true;
            }
        }
    }
```

- When the master transmits the data byte, it enters the slave shift register, and from there it will be placed in the RX DATA register. Then the below part of SERCOM1 handler executes:

```
if(SERCOM1->SPI.INTFLAG.bit.RXC && SERCOM1->SPI.INTENSET.bit.RXC)
    {
        rx_buffer[i++] = SERCOM1->SPI.DATA.reg;
        if(i == 5){
            SERCOM1->SPI.INTENCLR.reg = SERCOM_SPI_INTENCLR_RXC;
            rx_done = true;
        }
    }
```

  The RX DATA register content will be read till the iteration count reach its maximum value.

- For each data reception the slave will transmit its shift register content to the master. In normal configuration the first data byte from the slave shift register is zero and the rest of the data byte will be transmitted from the transmit buffer tx_buffer. The code below here will do this part.

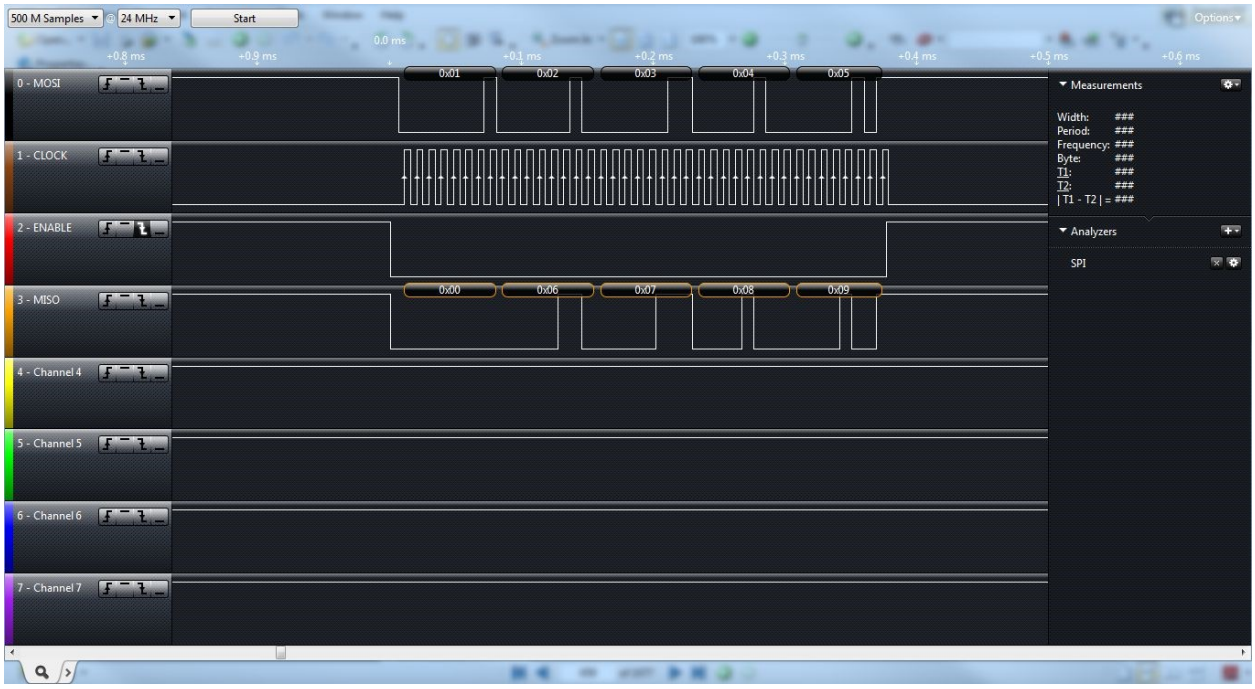
```
if(SERCOM1->SPI.INTFLAG.bit.DRE && SERCOM1->SPI.INTENSET.bit.DRE)
    {
        SERCOM1->SPI.DATA.reg = tx_buffer[j++];
        if (j == 5)
            SERCOM1->SPI.INTENCLR.reg = SERCOM_SPI_INTENCLR_DRE;
}
```

- As the first data byte sent from slave shift register is zero, the last byte of the buffer cannot be read by the master
- Once receiving the complete data byte from the master, the Boolean variable is set into true and the code reaches the while loop in the main function

The final application “Basic Configuration for Slave” will be in the Zip attachment to this application note.

The figure below is a screen shot of the basic configuration transaction between master and slave.

Figure 5-3. Basic Configuration Transaction



5.2 Slave Preloading Configuration

5.2.1 Master Side

In the “Basic configuration” application the first data byte received by the SPI master will be the slave shift register with the default value. To receive the complete receive buffer data byte from the slave, an additional dummy write needs to be performed by the master.

This scenario can be avoided by using the slave preloading in the slave side. By doing slave preloading the first byte of the slave shift register will be data byte of TX DATA register of slave.

In this application section only the changes from the basic configuration will be explained.

The following section of basic configuration of the SPI master will be applicable for this application.

- SPI Master pin initialization
- SPI Master initialization
- SERCOM Handler

5.2.2 SPI Master Transaction

The spi_master_send function is used to perform transactions with the connected slave device.

```
void spi_master_send()
{
    i = 0;
    j = 0;
    tx_done = false;
    /* Slave_select line is made into low
    to start the communication */
    port_pin_set_output_level(PIN_PA18, false);
    /* delay added for preloading */
    delay_ms(1);
    /*Data register empty and receive complete interrupt is enabled */
    SERCOM1->SPI.INTENSET.reg = SERCOM_SPI_INTENSET_DRE | SERCOM_SPI_INTENSET_RXC;
    while (!tx_done);
```

```
    /* Slave_select line is made into low
    to start the communication */
    port_pin_set_output_level(PIN_PA18, true);
    /* SERCOM1 peripheral disabled */
    SERCOM1->SPI.CTRLA.reg &= ~SERCOM_SPI_CTRLA_ENABLE;
    /* synchronization busy */
    while(SERCOM1->SPI.SYNCBUSY.reg & SERCOM_SPI_SYNCBUSY_ENABLE);
}
```

In Master Side, the following steps should be implemented for the preloading configuration.

- As said in section **Preloading of the Slave Shift Register** in the SAM D21 data sheet, “Preloading can be used to preload data to the shift register, while _SS is high and eliminate sending a dummy character when starting a transaction”.
- The point above here should be taken care of in the master section by keeping the slave select line high when the slave first data byte is loaded into the slave TX DATA register with slave preloading configuration.
- After that the slave select line should be kept low for the SPI communication. In this application during debugging the debug break point should be kept at the below line of code, and then the slave should run to get the proper preloading working.

```
tx_done = false;
```

- As said in section **Preloading of the Slave Shift Register** in the SAM D21 data sheet, "In order to guarantee enough set-up time before the first SCK edge, enough time must be given between _SS going low and the first SCK sampling edge”.
- The point above here should be taken care of in the master section by adding the delay of at least four cycles before starting the SPI communication.
- In the function above, the line below is implemented for these settings.

```
delay_ms(1);
```

The final application “Preloading configuration for Master” will be in the Zip attachment to this application note.

5.2.3 Slave Section

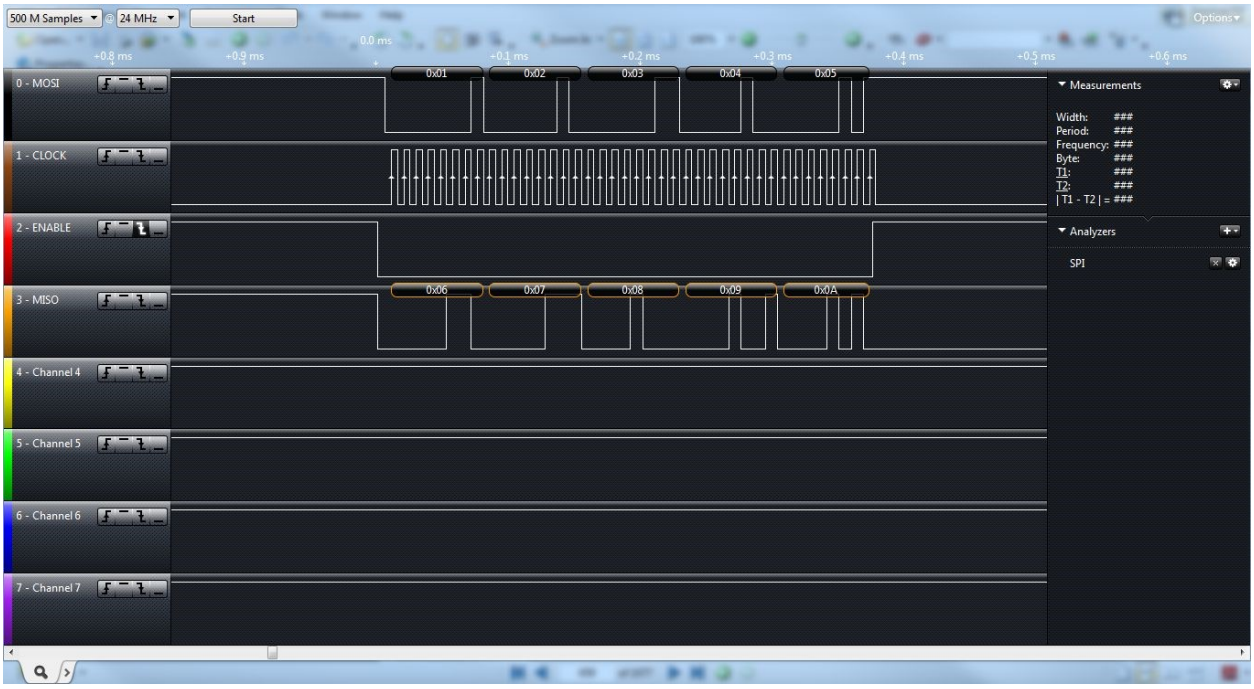
This application is the same as the Basic configuration section except that the preloading bit has to be set in the CTRLB register.

```
SERCOM1->SPI.CTRLB.reg = SERCOM_SPI_CTRLB_RXEN | SERCOM_SPI_CTRLB_PLOADEN;
```

The final application “Preloading configuration for Slave” will be in the Zip attachment to this application note.

The figure shown below is a screen-shot of the Slave Preloading Configuration transaction between master and slave.

Figure 5-4. Slave Preloading Configuration

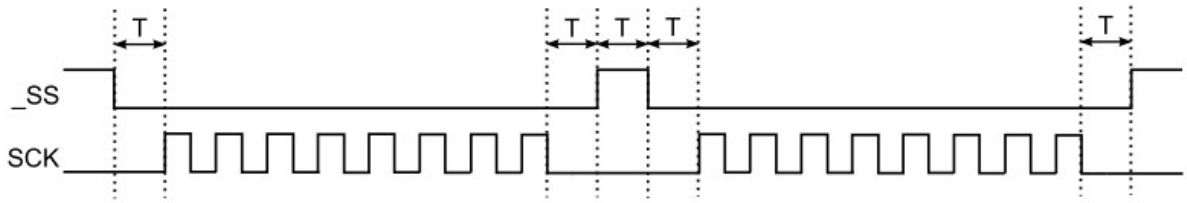


5.3 Hardware Controlled SS and SS Low Detection Configuration

Normally, in an SPI application the slave select line must to be pulled low by the master to start the communication. After the transaction has completed the slave select line will be released by the master. The software application must take care of this.

In this configuration the _SS pin is driven low for a minimum of one baud cycle before the transmission starts and stays low for a minimum of one baud cycle after the transmission has completed. If back-to-back frames are transmitted, the _SS pin will always be driven high for a minimum of one baud cycle between the frames.

Figure 5-5. Hardware Controlled_SS



This section is the same as the Basic configuration section. Below are the changes to be done in the master side.

- The Hardware slave select (_SS) control is enabled by setting the MSSEN bit in the SPI Master CTRLB register.

```
SERCOM1->SPI.CTRLB.reg = SERCOM_SPI_CTRLB_RXEN | SERCOM_SPI_CTRLB_MSSEN;
```

- The Hardware slave select line, which is connected with the slave, should be assigned to the alternate peripheral function C.

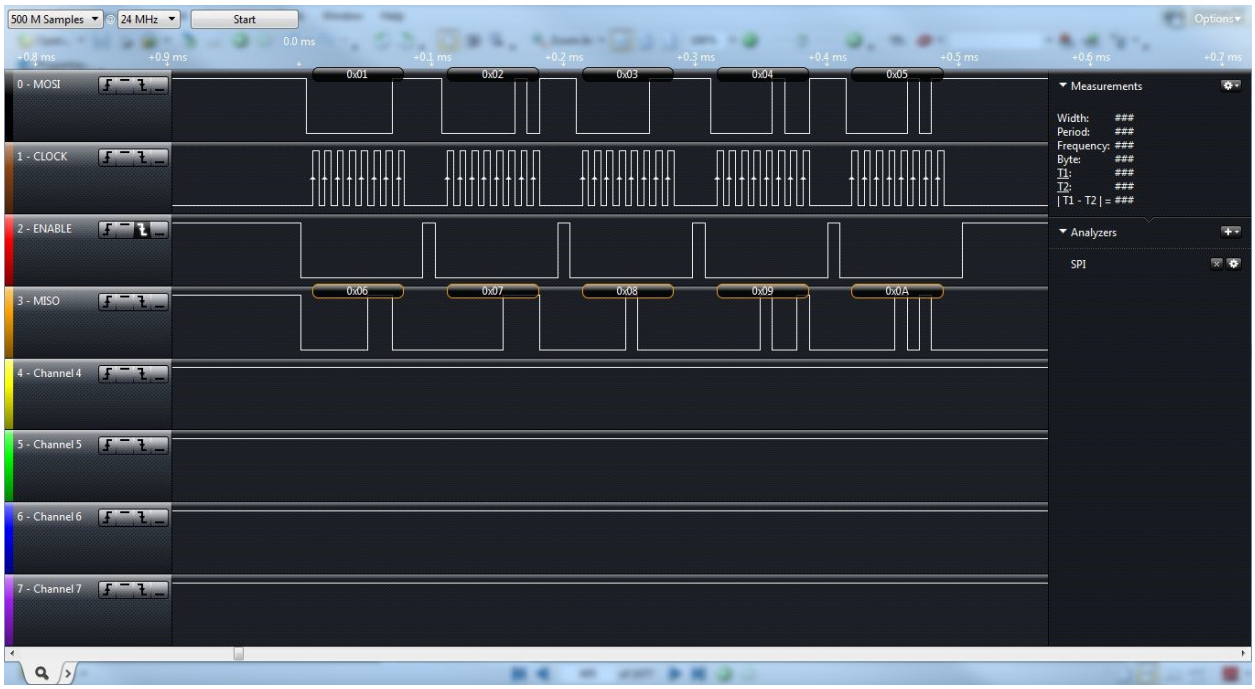
```
pin_set_peripheral_function(PINMUX_PA18C_SERCOM1_PAD2);
```

In the slave section there is no change from the preloading section. The slave preloading section of the SPI slave is used for this application.

The final application “Hardware controlled SS and SS low detection configuration” for both master and slave is part of the zip-file attachment for this application note.

The figure shown below is a screen-shot of the Hardware controlled SS and SS low detection configuration transaction between master and slave.

Figure 5-6. Hardware Controlled SS and SS Low Detection Configuration



6. References

SAM D21 Device Data Sheet - http://ww1.microchip.com/downloads/en/devicedoc/atmel-42181-sam-d21_datasheet.pdf.

SAM D21 Xplained Pro user guide and schematics link - http://www.microchip.com/developmenttools/productdetails.aspx?partno=atsamd21-xpro.

7. Revision History

Doc Rev.	Date	Comments
A	06/2017	Initial document release.

The Microchip Web Site

Microchip provides online support via our web site at http://www.microchip.com/. This web site is used as a means to make files and information easily available to customers. Accessible by using your favorite Internet browser, the web site contains the following information:

- **Product Support** – Data sheets and errata, application notes and sample programs, design resources, user's guides and hardware support documents, latest software releases and archived software
- **General Technical Support** – Frequently Asked Questions (FAQ), technical support requests, online discussion groups, Microchip consultant program member listing
- **Business of Microchip** – Product selector and ordering guides, latest Microchip press releases, listing of seminars and events, listings of Microchip sales offices, distributors and factory representatives

Customer Change Notification Service

Microchip's customer notification service helps keep customers current on Microchip products. Subscribers will receive e-mail notification whenever there are changes, updates, revisions or errata related to a specified product family or development tool of interest.

To register, access the Microchip web site at http://www.microchip.com/. Under "Support", click on "Customer Change Notification" and follow the registration instructions.

Customer Support

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Field Application Engineer (FAE)
- Technical Support

Customers should contact their distributor, representative or Field Application Engineer (FAE) for support. Local sales offices are also available to help customers. A listing of sales offices and locations is included in the back of this document.

Technical support is available through the web site at: http://www.microchip.com/support

Microchip Devices Code Protection Feature

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.

- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as “unbreakable.”

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip’s code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Legal Notice

Trademarks

ISBN: 978-1-5224-1822-1

Quality Management System Certified by DNV

ISO/TS 16949

Microchip received ISO/TS-16949:2009 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.

Worldwide Sales and Service

AMERICAS	ASIA/PACIFIC	ASIA/PACIFIC	EUROPE
Corporate Office 2355 West Chandler Blvd. Chandler, AZ 85224-6199 Tel: 480-792-7200 Fax: 480-792-7277 Technical Support: http://www.microchip.com/support Web Address: www.microchip.com **Atlanta** Duluth, GA Tel: 678-957-9614 Fax: 678-957-1455 **Austin, TX** Tel: 512-257-3370 **Boston** Westborough, MA Tel: 774-760-0087 Fax: 774-760-0088 **Chicago** Itasca, IL Tel: 630-285-0071 Fax: 630-285-0075 **Dallas** Addison, TX Tel: 972-818-7423 Fax: 972-818-2924 **Detroit** Novi, MI Tel: 248-848-4000 **Houston, TX** Tel: 281-894-5983 **Indianapolis** Noblesville, IN Tel: 317-773-8323 Fax: 317-773-5453 Tel: 317-536-2380 **Los Angeles** Mission Viejo, CA Tel: 949-462-9523 Fax: 949-462-9608 Tel: 951-273-7800 **Raleigh, NC** Tel: 919-844-7510 **New York, NY** Tel: 631-435-6000 **San Jose, CA** Tel: 408-735-9110 Tel: 408-436-4270 **Canada - Toronto** Tel: 905-695-1980 Fax: 905-695-2078	**Asia Pacific Office** Suites 3707-14, 37th Floor Tower 6, The Gateway Harbour City, Kowloon **Hong Kong** Tel: 852-2943-5100 Fax: 852-2401-3431 **Australia - Sydney** Tel: 61-2-9868-6733 Fax: 61-2-9868-6755 **China - Beijing** Tel: 86-10-8569-7000 Fax: 86-10-8528-2104 **China - Chengdu** Tel: 86-28-8665-5511 Fax: 86-28-8665-7889 **China - Chongqing** Tel: 86-23-8980-9588 Fax: 86-23-8980-9500 **China - Dongguan** Tel: 86-769-8702-9880 **China - Guangzhou** Tel: 86-20-8755-8029 **China - Hangzhou** Tel: 86-571-8792-8115 Fax: 86-571-8792-8116 **China - Hong Kong SAR** Tel: 852-2943-5100 Fax: 852-2401-3431 **China - Nanjing** Tel: 86-25-8473-2460 Fax: 86-25-8473-2470 **China - Qingdao** Tel: 86-532-8502-7355 Fax: 86-532-8502-7205 **China - Shanghai** Tel: 86-21-3326-8000 Fax: 86-21-3326-8021 **China - Shenyang** Tel: 86-24-2334-2829 Fax: 86-24-2334-2393 **China - Shenzhen** Tel: 86-755-8864-2200 Fax: 86-755-8203-1760 **China - Wuhan** Tel: 86-27-5980-5300 Fax: 86-27-5980-5118 **China - Xian** Tel: 86-29-8833-7252 Fax: 86-29-8833-7256	**China - Xiamen** Tel: 86-592-2388138 Fax: 86-592-2388130 **China - Zhuhai** Tel: 86-756-3210040 Fax: 86-756-3210049 **India - Bangalore** Tel: 91-80-3090-4444 Fax: 91-80-3090-4123 **India - New Delhi** Tel: 91-11-4160-8631 Fax: 91-11-4160-8632 **India - Pune** Tel: 91-20-3019-1500 **Japan - Osaka** Tel: 81-6-6152-7160 Fax: 81-6-6152-9310 **Japan - Tokyo** Tel: 81-3-6880- 3770 Fax: 81-3-6880-3771 **Korea - Daegu** Tel: 82-53-744-4301 Fax: 82-53-744-4302 **Korea - Seoul** Tel: 82-2-554-7200 Fax: 82-2-558-5932 or 82-2-558-5934 **Malaysia - Kuala Lumpur** Tel: 60-3-6201-9857 Fax: 60-3-6201-9859 **Malaysia - Penang** Tel: 60-4-227-8870 Fax: 60-4-227-4068 **Philippines - Manila** Tel: 63-2-634-9065 Fax: 63-2-634-9069 **Singapore** Tel: 65-6334-8870 Fax: 65-6334-8850 **Taiwan - Hsin Chu** Tel: 886-3-5778-366 Fax: 886-3-5770-955 **Taiwan - Kaohsiung** Tel: 886-7-213-7830 **Taiwan - Taipei** Tel: 886-2-2508-8600 Fax: 886-2-2508-0102 **Thailand - Bangkok** Tel: 66-2-694-1351 Fax: 66-2-694-1350	**Austria - Wels** Tel: 43-7242-2244-39 Fax: 43-7242-2244-393 **Denmark - Copenhagen** Tel: 45-4450-2828 Fax: 45-4485-2829 **Finland - Espoo** Tel: 358-9-4520-820 **France - Paris** Tel: 33-1-69-53-63-20 Fax: 33-1-69-30-90-79 **France - Saint Cloud** Tel: 33-1-30-60-70-00 **Germany - Garching** Tel: 49-8931-9700 **Germany - Haan** Tel: 49-2129-3766400 **Germany - Heilbronn** Tel: 49-7131-67-3636 **Germany - Karlsruhe** Tel: 49-721-625370 **Germany - Munich** Tel: 49-89-627-144-0 Fax: 49-89-627-144-44 **Germany - Rosenheim** Tel: 49-8031-354-560 **Israel - Ra'anana** Tel: 972-9-744-7705 **Italy - Milan** Tel: 39-0331-742611 Fax: 39-0331-466781 **Italy - Padova** Tel: 39-049-7625286 **Netherlands - Drunen** Tel: 31-416-690399 Fax: 31-416-690340 **Norway - Trondheim** Tel: 47-7289-7561 **Poland - Warsaw** Tel: 48-22-3325737 **Romania - Bucharest** Tel: 40-21-407-87-50 **Spain - Madrid** Tel: 34-91-708-08-90 Fax: 34-91-708-08-91 **Sweden - Gothenberg** Tel: 46-31-704-60-40 **Sweden - Stockholm** Tel: 46-8-5090-4654 **UK - Wokingham** Tel: 44-118-921-5800 Fax: 44-118-921-5820