APPLICATION NOTE

AT11628: SAM D21 SERCOM I2C Configuration

ATSAMD21J18

Introduction

This application note explains the various features of SERCOM I²C in the Atmel® SAM D21 microcontrollers and its configurations with example codes and corresponding scope shots.

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

Features

- Combined interface configurable as one of the following:
  - I²C – 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)

Table of Contents

1 Glossary

SERCOM	Serial communication interface
I^2C	Inter-Integrated Circuit
USART	Universal asynchronous receiver/transmitter
SPI	Serial communication interface
EDBG	Embedded Debugger
IDE	Integrated Development Environment
SCL	Serial Clock Line
SDA	Serial Data Line
SMBus	System Management Bus
DMA	Direct Memory Access

2 Pre-requisites

The solutions discussed in this document require basic familiarity with the following skills and technologies:

- 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 microcontrollers 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 I²C 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 either configured as USART or I²C or SPI selectable by 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 purpose and the SERCOM module will not control them unless it is 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 either used as GPIO pins or can 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

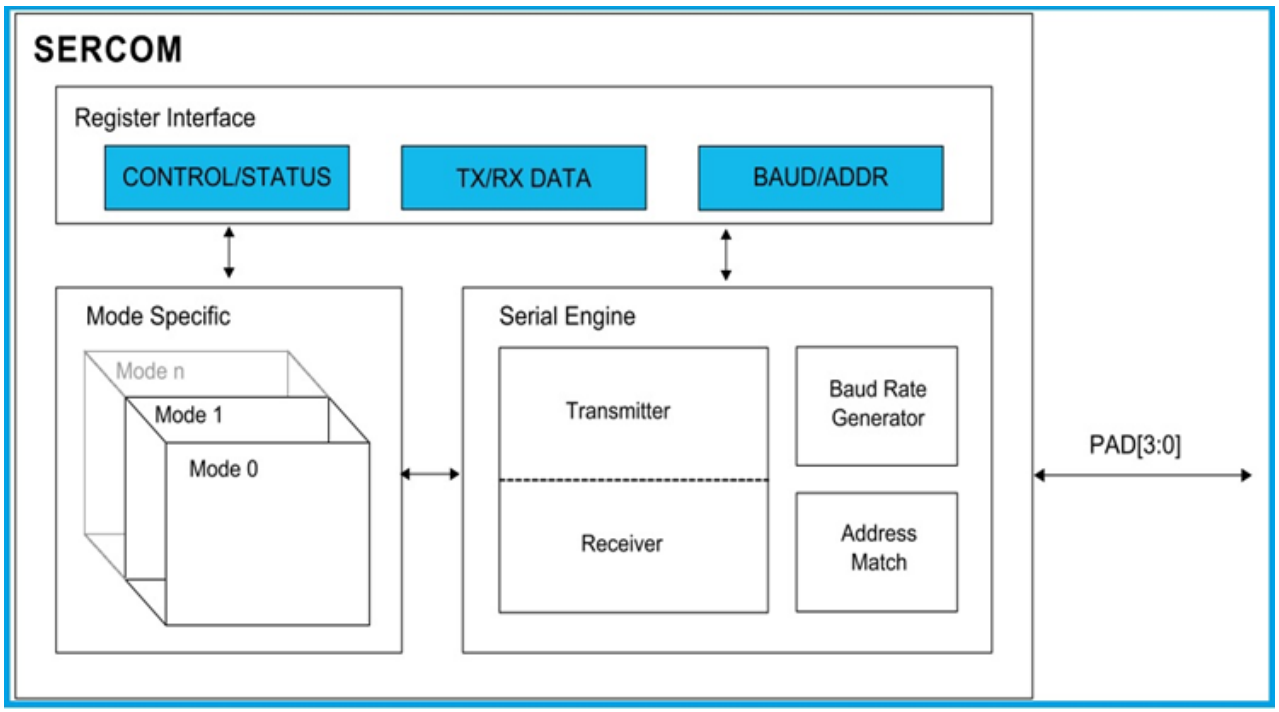
The serial communication interface (SERCOM) can be configured to support three different modes: I²C, SPI, and USART. Once configured and enabled, all SERCOM resources are dedicated to the selected mode.

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

3.2 Block Diagram

Figure 3-1 depicts 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



3.3 Clocks

SERCOM module needs below 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 Power Manager (PM) module.

Two generic clocks are used by the SERCOM module namely 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.

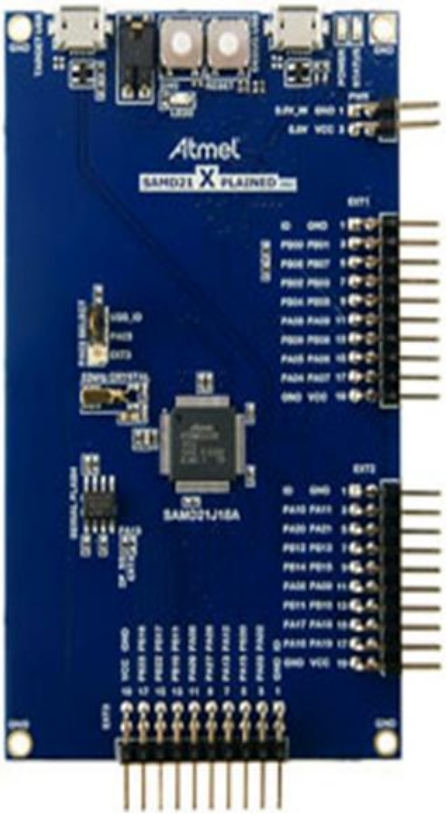
The core clock (GCLK_SERCOMx_CORE) is required to clock the SERCOM while operating as a master, while the slow clock (GCLK_SERCOMx_SLOW) is only required for certain functions like I^2C timeouts.

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 require two SAM D21 Xplained Pro boards. One board will be configured as master and other board as slave.

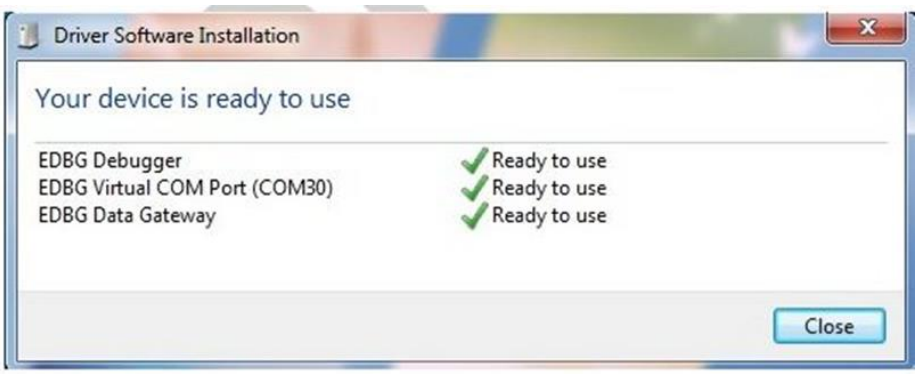
Figure 4-1. SAM D21 Xplained Pro Board



There are two USB ports on the SAM D21 Xplained Pro board – DEBUG USB and 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.

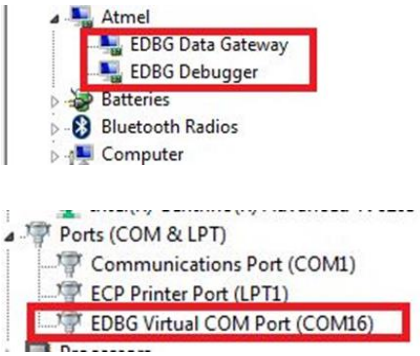
Once the kit is successfully connected for the first time, the Windows® Task bar will pop up a message as shown in Figure 4-2.

Figure 4-2. SAM D21 Xplained Pro Driver Installation



If the driver installation is proper, EDBG will be listed in the Device Manager as shown in Figure 4-3.

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 master and other for slave. GCC C ASF Board project from Atmel Studio is used for the implementation.

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

Figure 4-4. New project in Atmel Studio

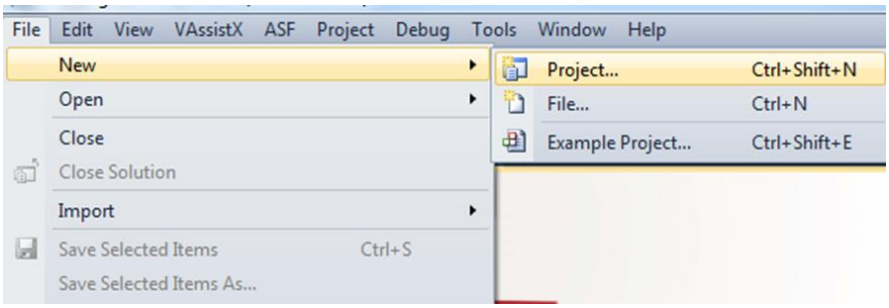
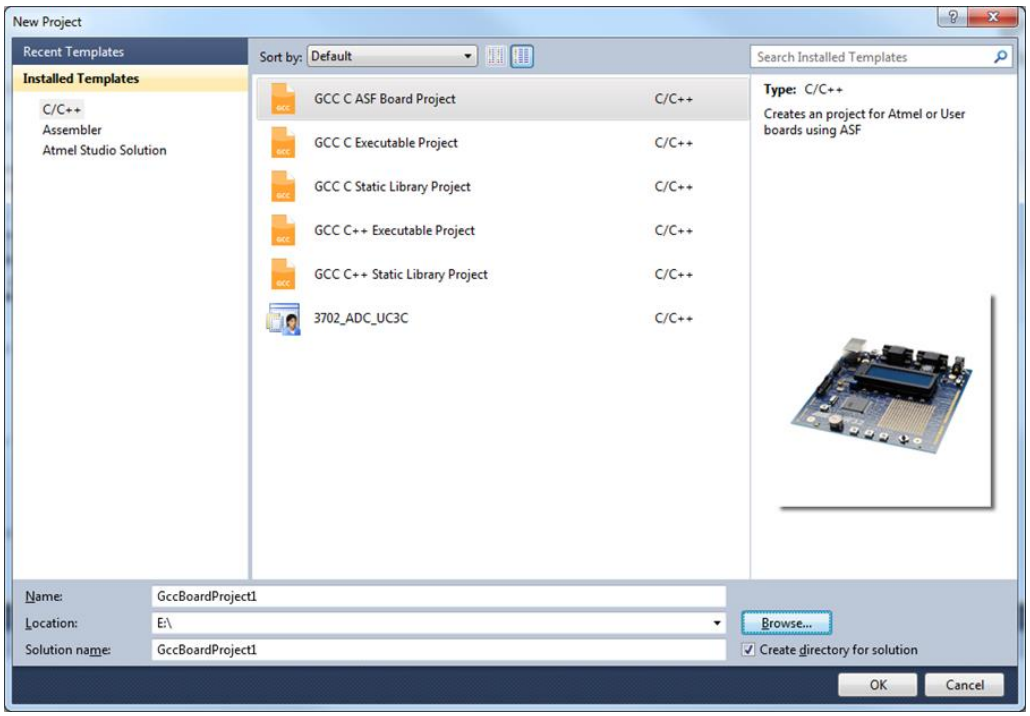
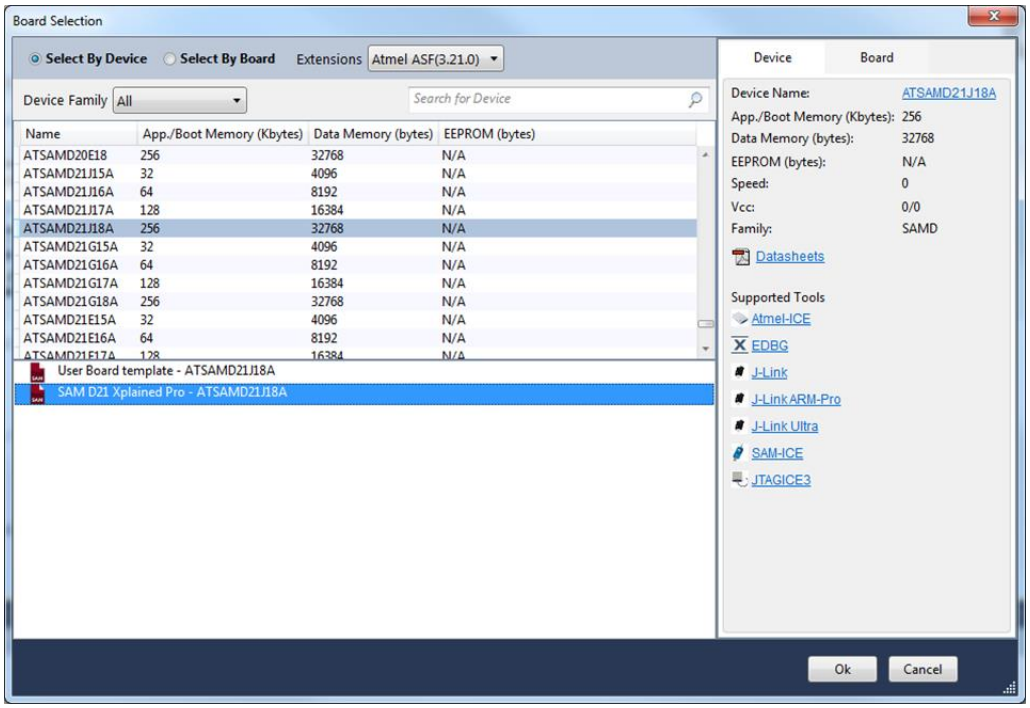


Figure 4-5. ASF Board Project



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 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 section will demonstrate the various features of the SERCOM I²C module of SAM D21 with different example codes. Following are the examples demonstrated in this application note.

- Basic Configuration
- High speed configuration
- Address mode configuration

Note: This section 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 register level coding for SERCOM module configuration. The clock configuration will, however, use ASF functions.

WARNING

Since this application note is demonstrated using the ASF template project, build errors are expected when compiling the project with SERCOM drivers included from ASF. This because ASF SERCOM drivers will predefine all the SERCOM handlers which will make redefinition error as SERCOM handlers are defined again as per the application note.

5.1 Basic Configuration

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

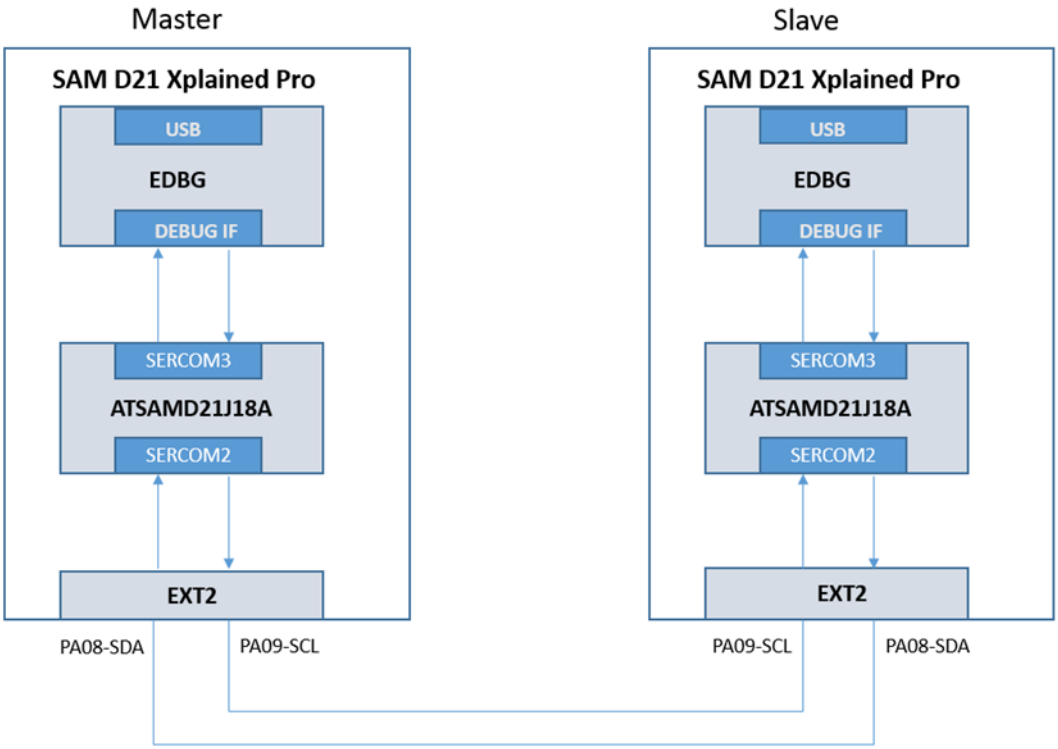
Basic configuration application performs the following actions:

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

SERCOM I²C lines (PA08 - SDA, PA09 - SCL) of the two SAM D21 Xplained Pro boards should be connected through EXT2 connector using a pair of wires as shown in Figure 5-1.

Note that the same I²C lines (PA08 and PA09) are available on all three EXT headers in SAM D21 Xplained Pro board so any of the three headers can be used for this connection.

Figure 5-1. Block Diagram



The SAM D21 Xplained Pro board has on-board pull-up resistors on I²C lines with 4.7kΩ resistance value and with reference designators R305 and R306.

Note: To get better I²C signal timings the 4.7kΩ pull-up resistors on both the master and slave SAM D21 Xplained Pro boards have been replaced by 2kΩ resistors and all the modes have been tested with this pull-up configuration.

In Basic configuration section, both master and slave communicates at the fast mode plus – 1MHz speed.

Following are the common function calls used in master and slave applications in basic configuration example:

- system_init()
- i2c_clock_init()
- i2c_pin_init()

Detailed explanation on each function will be provided in the upcoming sections. At the end of each section a complete code will be provided for reference.

The Complete Project solution for all the applications can be found in the zipped folder attachment that comes with this application note.

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, 1MHz clock from OSC8M (prescaler set to 8) is used as the clock source for 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

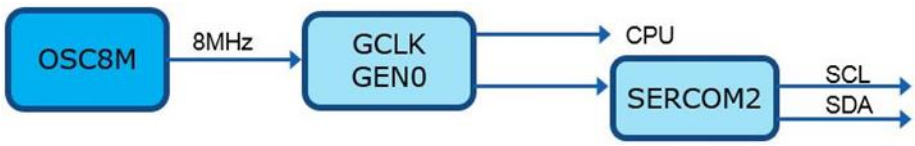
The master and slave application uses OSC8M as the clock source for Generator 0. The following lines in the conf_clocks.h will initialize the source clock for generator 0.

```
/* 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_OSC8M
#  define CONF_CLOCK_GCLK_0_PRESCALER             1
#  define CONF_CLOCK_GCLK_0_OUTPUT_ENABLE         false
```

The flash wait state can be set to 0 as the CPU will not run in maximum speed in this example.

5.1.3 Clock Flow for Master and Slave

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



5.1.4 System Initialization

`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 the Section 5.1.2. It also initializes the board hardware of SAM D21 Xplained Pro and the event system.

5.1.5 I²C Clock Initialization

`i2c_clock_init()` function configures the peripheral bus clock (APB clock) and generic clock for the SERCOM I²C module. SERCOM2 is used in both the master and slave boards.

```
void i2c_clock_init()
{
	struct system_gclk_chan_config gclk_chan_conf;
	uint32_t gclk_index = SERCOM2_GCLK_ID_CORE;
	/* Turn on module in PM */
	system_apb_clock_set_mask(SYSTEM_CLOCK_APB_APBC, PM_APBCMASK_SERCOM2);
	/* Turn on Generic clock for I2C */
	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.
- SERCOM2 core clock "`SERCOM2_GCLK_ID_CORE`" and bus clock "`SYSTEM_CLOCK_APB_APBC`" are configured

- Generic clock "SERCOM2_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 SERCOM2 module is clocked at 48MHz from DFLL48M.
- system_gclk_chan_set_config will set the generic clock channel configuration
- system_gclk_chan_enable will enable the generic clock "SERCOM2_GCLK_ID_CORE"

5.1.6 I²C Pin Initialization

i2c_pin_init() function will initialize pins PA08 and PA09 to the SERCOM-Alternate peripheral function (D).

```
void i2c_pin_init()
{
	/* PA08 and PA09 set into peripheral function D*/
	pin_set_peripheral_function(PINMUX_PA08D_SERCOM2_PAD0);
	pin_set_peripheral_function(PINMUX_PA09D_SERCOM2_PAD1);
}
```

i2c_pin_init function calls the pin_set_peripheral_function to assign I/O lines PA08 and PA09 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)));
	0x0000FFFF) << (4 * ((pinmux >> 16) & 0x01u)));
	PORT->Group[port].PINCFG[((pinmux >> 16) - (port*32))].bit.PMUXEN = 1;
}
```

The function *pin_set_peripheral_function()* will switch the GPIO functionality of an I/O pin to peripheral functionality and assigns the given peripheral function to the pin. The function takes a 32-bit pinmux value as its argument. The 32-bit pinmux value contains the pin number in its 16-bit MSB part and the peripheral function number in its 16-bit LSB part. So each 32-bit pinmux value is unique per pin per peripheral function. The function first identifies the PORT group from the pin number (MSB 16-bit) and updates the PMUX register with the peripheral number (LSB 16-bit).

SERCOM2 PAD0 will be SDA line and SERCOM2 PAD1 will be SCL line as per the SERCOM I²C pad assignment shown in Table 5-1.

Table 5-1. Signal Description

Signal name	Type	Description
PAD[0]	Digital I/O	SDA
PAD[1]	Digital I/O	SDL
PAD[2]	Digital I/O	SDA_OUT (4-wire)
PAD[3]	Digital I/O	SDC_OUT (4-wire)

Note: System initialization, I²C clock initialization, and I²C pin initialization are common for both master and slave. For the slave part the same will be applicable.

5.1.7 I²C Master Initialization

The i2c_master_init function will initialize the I²C master function by configuring the control registers, baud registers, and setting the respective interrupt enable bits.

```
void i2c_master_init()
{
	/* By setting the SPEED bit field as 0x01, I2C Master runs at Fast mode + -  1MHz,
	   By setting the SDAHOLD bit field as 0x02, SDA hold time is configured for 300-600ns,
	   By setting the RUNSTDBY bit as 0x01,Generic clock is enabled in all sleep modes,
	   any interrupt can wake up the device,
	   SERCOM2 is configured as an I2C Master by writing the MODE bitfield as 0x5 */
	SERCOM2->I2CM.CTRLA.reg = SERCOM_I2CM_CTRLA_SPEED (FAST_MODE_PLUS)     |
	                          SERCOM_I2CM_CTRLA_SDAHOLD(0x2)               |
	                          SERCOM_I2CM_CTRLA_RUNSTDBY                   |
	                          SERCOM_I2CM_CTRLA_MODE_I2C_MASTER;
	/* smart mode enabled by setting the bit SMEN as 1 */
	SERCOM2->I2CM.CTRLB.reg = SERCOM_I2CM_CTRLB_SMEN;
	/* synchronization busy */
	while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
	/* BAUDLOW is non-zero, and baud register is loaded */
	SERCOM2->I2CM.BAUD.reg = SERCOM_I2CM_BAUD_BAUD(11) | SERCOM_I2CM_BAUD_BAUDLOW(22);
	/* synchronization busy */
	while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
	/* SERCOM2 peripheral enabled by setting the ENABLE bit as 1*/
	SERCOM2->I2CM.CTRLA.reg |= SERCOM_I2CM_CTRLA_ENABLE;
	/* SERCOM Enable synchronization busy */
	while((SERCOM2->I2CM.SYNCBUSY.reg & SERCOM_I2CM_SYNCBUSY_ENABLE));
	/* bus state is forced into idle state */
	SERCOM2->I2CM.STATUS.bit.BUSSTATE = 0x1;
	/* synchronization busy */
	while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
	/* Both master on bus and slave on bus interrupt is enabled */
	SERCOM2->I2CM.INTENSET.reg = SERCOM_I2CM_INTENSET_MB | SERCOM_I2CM_INTENSET_SB;
	/* SERCOM2 handler enabled */
	system_interrupt_enable(SERCOM2_IRQn);
}
```

- The CTRLA register is used to configure the I²C speed, SDA hold time, and I²C mode. In the above function I²C speed is configured as Fast-mode plus, SDA hold time is set for 300 - 600ns. I²C is configured as master and the I²C module is made to run even in standby sleep mode.
- CTRLB register is used to write the commands and to enable Smart Mode. In the above function SMEN – Smart Mode Enable bit is set.
- For BAUD.BAUDLOW non-zero, the following formula is used to determine the SCL frequency, $F_{SCL} = f_{GCLK}/(10 + BAUD + BAUDLOW + f_{GCLK} T_{RISE})$

F_{SCL} = I²C clock frequency

f_{GCLK} = SERCOM generic clock frequency

BAUD = BAUD register value

BAUDLOW = BAUD LOW register value

T_{RISE} = Rise time for I²C in the defined mode

Rise time for the respective speed modes can be found in section Electrical Characteristics → Timing Characteristics → SERCOM I²C Mode Timing in the SAM D21 device datasheet.

In this configuration, the I²C runs at the speed of 1MHz and the worst case rise time for fast mode plus is 100ns.

From the equation:

$BAUD + BAUDLOW = f_{GCLK}/F_{SCL} – (f_{GCLK}\ T_{RISE}) – 10$

$= 48M/1M – (48M \times 100ns) – 10$

$= 33.2$

Note: For Fast-mode plus the nominal high to low SCL ratio is 1 to 2 and BAUD should be set accordingly. At a minimum, BAUD.BAUD and/or BAUD.BAUDLOW must be non-zero.

So the BAUD value is set to 11 and BAUDLOW value is set to 22.

Apart from high-speed mode, the same equation given above can be used for BAUD-BAUDLOW calculation for other speed modes like standard mode and fast mode with appropriate rise time taken from the SERCOM I²C Mode Timing section in the datasheet as mentioned earlier.

- CTRLA, CTRLB, and BAUD registers can be written only when the I²C is disabled because these registers are enable protected. So once configuring these registers the I²C is enabled.
- As CLK_SERCOMx_APB and GCLK_SERCOMx_CORE are not synchronized and some registers needs synchronization when they are accessed. CTRLA register is Write-Synchronized so the application should wait until the synchronization busy flag (SYSOP bit in SYNCBUSY register) is cleared after performing a write to this register.
- The I²C bus-state is unknown when the master is disabled. During this time writing 0x1 to BUSSTATE forces the bus state into the idle state.
- 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 SERCOM2 interrupt request line (IRQ - 11) is enabled.
- The INTENSET register is used to enable the required SERCOM interrupts. In the above function SB – Slave on bus and MB – Master on bus interrupts are enabled

5.1.8 I²C Master Transaction

The `i2c_master_transact` function is used to perform a transaction with the connected slave device.

```
void i2c_master_transact(void)
{
       i = 0;
       /* Acknowledge section is set as ACK signal by
      writing 0 in ACKACT bit */
       SERCOM2->I2CM.CTRLB.reg &= ~SERCOM_I2CM_CTRLB_ACKACT;
       while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
       /* slave address with Write(0) */
       SERCOM2->I2CM.ADDR.reg = (SLAVE_ADDR << 1) | 0;
       while(!tx_done);
       i = 0;
       /* Acknowledge section is set as ACK signal by
      writing 0 in ACKACT bit */
       SERCOM2->I2CM.CTRLB.reg &= ~SERCOM_I2CM_CTRLB_ACKACT;
       while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
       /* slave address with read (1) */
```

```
        SERCOM2->I2CM.ADDR.reg = (SLAVE_ADDR << 1) | 1;
        while(!rx_done);
        /*interrupts are cleared */
        SERCOM2->I2CM.INTENCLR.reg = SERCOM_I2CM_INTENCLR_MB | SERCOM_I2CM_INTENCLR_SB;
}
```

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

```
uint8_t i
volatile bool tx_done = false, rx_done = false
```

- In CTRLB register, ACKACT field is used to define the I²C master's acknowledge behavior after a data byte is received from the I²C slave. The acknowledge action will be executed when a command is written to CTRLB.CMD bits or after a transfer with Smart Mode is enabled.
- In the above function, ACKACT is set to 0, so ACK will be sent by master after a data byte is received
- In the application we are using 7-bit addressing mode. The slave address is written into the Address register, which initiates a transfer by sending a start condition followed by address on the I²C line.
- The slave address is shifted by 1 bit and LSB of ADDR register is written as 0 because application is going to write/transmit the data to slave
- Once placing the address in the ADDR register, the address will be kept in Data register and transferred to Slave
- The Boolean flag variable tx_done is initialized as false, so it remains in the while loop until SERCOM2 handler sets it to true indicating the completion of transfer

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

- Once transmitting the address of slave and then after receiving the ACK/NACK, the Master on Bus (MB) interrupt will be set and `SERCOM2_Handler` will be serviced

```
if (SERCOM2->I2CM.INTFLAG.bit.MB)
        {
                if (i == BUF_SIZE)
                {  /* After transferring the last byte stop condition will be sent */
                        SERCOM2->I2CM.CTRLB.bit.CMD = 0x3;
                        while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
                        tx_done = true;
                        i = 0;
                }
                else
                { /* placing the data from transmitting buffer to DATA register*/
                        SERCOM2->I2CM.DATA.reg = tx_buf[i++];
                        while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
                }
        }
```

- Buffer size has been set by macro `BUF_SIZE`
- In SERCOM2 handler it will check for Master on bus interrupt set condition and enters that loop
- In that loop, variable i is checked with the `BUF_SIZE`. If i is not equal to `BUF_SIZE` the transmit buffer data will be placed in the data register.
- Each time when the placed data is successfully transferred on the bus, the MB interrupt will be triggered
- Once i value equals the `BUF_SIZE`, STOP condition will be sent and Boolean variable `tx_done` is set to true condition

- Now the code execution will reach the `i2c_master_transact` function. `tx_done` is true so it will come out of while loop.
- Now the application is going to read the data from the slave
- Again the variable i is set to 0
- In the address register, again address value is loaded by shifting it left by one bit and LSB of ADDR register is set as 1 for read operation
- Once placing the address (Address + R) in the ADDR register, the address will be kept in data register and transferred to slave. Then slave will acknowledge the address and since it is master read operation so the slave device will send the data to master and now slave on bus interrupt condition will be set in master.

```
if (SERCOM2->I2CM.INTFLAG.bit.SB)
	{
		if (i == (BUF_SIZE-1))
		{ /* NACK should be sent before reading the last byte */
			SERCOM2->I2CM.CTRLB.reg |= SERCOM_I2CM_CTRLB_ACKACT;
			while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
			SERCOM2->I2CM.CTRLB.bit.CMD = 0x3;
			while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
			rx_buf[i++] = SERCOM2->I2CM.DATA.reg;
			while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
			rx_done = true;
		}
		else
		{
			SERCOM2->I2CM.CTRLB.reg &= ~SERCOM_I2CM_CTRLB_ACKACT;
			while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
			rx_buf[i++] = SERCOM2->I2CM.DATA.reg;
			while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
			/* sending ACK after reading each byte */
			SERCOM2->I2CM.CTRLB.bit.CMD = 0x2;
			while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
		}
	}
```

- The received data byte will be read into the read buffer `rx_buf` and acknowledgement – ACK will be sent
- Last data byte should be read only after giving the STOP condition and NACK should be sent by the Master
- In CTRLB register the Acknowledge action is set as NACK for last byte read and STOP condition is sent
- After sending the STOP condition the last byte is read from DATA register into read buffer
- Now the code execution will reach the `i2c_master_transact` function and clears the interrupt enable bits

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

The final application “Basic Configuration” in main.c file will be as below for **MASTER**:

```
#include <asf.h>

#define STANDARD_MODE_FAST_MODE  0x0
#define FAST_MODE_PLUS           0X01
#define HIGHSPEED_MODE           0X02
#define SLAVE_ADDR              0x12
```

```
#define BUF_SIZE                        3

/* Function Prototype */
void i2c_clock_init(void);
void i2c_pin_init(void);
void i2c_master_init(void);
void i2c_master_transact(void);
uint32_t calculate_baud(uint32_t, uint32_t);


uint8_t tx_buf[BUF_SIZE] = {1, 2, 3};
uint8_t rx_buf[BUF_SIZE];
uint8_t i;
volatile bool tx_done = false, rx_done = false;

/* I2C handler */
void SERCOM2_Handler(void)
	{
	/* Master on bus interrupt checking */
	if (SERCOM2->I2CM.INTFLAG.bit.MB)
	{
		if (i == BUF_SIZE)
		{   /* After transferring the last byte stop condition will be sent */
			SERCOM2->I2CM.CTRLB.bit.CMD = 0x3;
			while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
			tx_done = true;
			i = 0;
		}
		else
		{ /* placing the data from transmitting buffer to DATA register*/
			SERCOM2->I2CM.DATA.reg = tx_buf[i++];
			while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
		}
	}
	/* Slave on bus interrupt checking */
	if (SERCOM2->I2CM.INTFLAG.bit.SB)
	{
		if (i == (BUF_SIZE-1))
		{ /* NACK should be sent before reading the last byte */
			SERCOM2->I2CM.CTRLB.reg |= SERCOM_I2CM_CTRLB_ACKACT;
			while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
			SERCOM2->I2CM.CTRLB.bit.CMD = 0x3;
			while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
			rx_buf[i++] = SERCOM2->I2CM.DATA.reg;
			while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
			rx_done = true;
		}
		else
		{
			SERCOM2->I2CM.CTRLB.reg &= ~SERCOM_I2CM_CTRLB_ACKACT;
			while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
			rx_buf[i++] = SERCOM2->I2CM.DATA.reg;
			while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
			/* sending ACK after reading each byte */
			SERCOM2->I2CM.CTRLB.bit.CMD = 0x2;
			while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
		}
	}
}
```

```
/*Assigning pin to the alternate peripheral function*/
static inline void pin_set_peripheral_function(uint32_t pinmux)
{
	uint8_t port = (uint8_t)((pinmux >> 16)/32);
	PORT->Group[port].PINCFG[((pinmux >> 16) - (port*32))].bit.PMUXEN = 1;
	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)));
}

/* calculating the BAUD value using Fgclk,Fscl,Trise
    FSCL =fGCLK / (10 + BAUD +BAUDLOW + fGCLKTRISE )*/
uint32_t calculate_baud(uint32_t fgclk, uint32_t fscl)
{
	float f_temp, f_baud;
	f_temp = ((float)fgclk/(float)fscl) - (((float)fgclk/(float)1000000)*0.3);
	f_baud = (f_temp/2)-5;
	return ((uint32_t)f_baud);
}
/* SERCOM clock and peripheral bus clock initialization */
void i2c_clock_init()
{
	struct system_gclk_chan_config gclk_chan_conf;
	uint32_t gclk_index = SERCOM2_GCLK_ID_CORE;
	/* Turn on module in PM */
	system_apb_clock_set_mask(SYSTEM_CLOCK_APB_APBC, PM_APBCMASK_SERCOM2);
	/* Turn on Generic clock for I2C */
	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);
}
/* I2C pin initialization */
void i2c_pin_init()
{
	/* PA08 and PA09 set into peripheral function D*/
	pin_set_peripheral_function(PINMUX_PA08D_SERCOM2_PAD0); // SDA
	pin_set_peripheral_function(PINMUX_PA09D_SERCOM2_PAD1); // SCL
}

/* I2C master initialization */
void i2c_master_init()
{
	/* By setting the SPEED bit field as 0x01, I2C Master runs at Fast mode + -  1MHz,
	   By setting the SDAHOLD bit field as 0x02, SDA hold time is configured for 300-600ns,
	   By setting the RUNSTDBY bit as 0x01,Generic clock is enabled in all sleep modes,any
	interrupt can wake up the device,
	   SERCOM2 is configured as an I2C Master by writing the MODE bitfield as 0x5 */
	SERCOM2->I2CM.CTRLA.reg = SERCOM_I2CM_CTRLA_SPEED (FAST_MODE_PLUS)     |
	                          SERCOM_I2CM_CTRLA_SDAHOLD(0x2)               |
	                          SERCOM_I2CM_CTRLA_RUNSTDBY                   |
	                          SERCOM_I2CM_CTRLA_MODE_I2C_MASTER;
	/* smart mode enabled by setting the bit SMEN as 1 */
	SERCOM2->I2CM.CTRLB.reg = SERCOM_I2CM_CTRLB_SMEN;
	/* synchronization busy */
	while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
```

```
        /* BAUDLOW is non-zero, and baud register is loaded */
        SERCOM2->I2CM.BAUD.reg = SERCOM_I2CM_BAUD_BAUD(11) | SERCOM_I2CM_BAUD_BAUDLOW(22);
        /* synchronization busy */
        while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
        /* SERCOM2 peripheral enabled by setting the ENABLE bit as 1*/
        SERCOM2->I2CM.CTRLA.reg |= SERCOM_I2CM_CTRLA_ENABLE;
        /* SERCOM Enable synchronization busy */
        while((SERCOM2->I2CM.SYNCBUSY.reg & SERCOM_I2CM_SYNCBUSY_ENABLE));
        /* bus state is forced into idle state */
        SERCOM2->I2CM.STATUS.bit.BUSSTATE = 0x1;
        /* synchronization busy */
        while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
        /* Both master on bus and slave on bus interrupt is enabled */
        SERCOM2->I2CM.INTENSET.reg = SERCOM_I2CM_INTENSET_MB | SERCOM_I2CM_INTENSET_SB;
        /* SERCOM2 handler enabled */
        system_interrupt_enable(SERCOM2_IRQn);
}
/* I2C master Transaction */
void i2c_master_transact(void)
{
        i = 0;
        /* Acknowledge section is set as ACK signal by
       writing 0 in ACKACT bit */
        SERCOM2->I2CM.CTRLB.reg &= ~SERCOM_I2CM_CTRLB_ACKACT;
        while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
        /* slave address with Write(0) */
        SERCOM2->I2CM.ADDR.reg = (SLAVE_ADDR << 1) | 0;
        while(!tx_done);
        i = 0;
        /* Acknowledge section is set as ACK signal by
       writing 0 in ACKACT bit */
        SERCOM2->I2CM.CTRLB.reg &= ~SERCOM_I2CM_CTRLB_ACKACT;
        while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
        /* slave address with read (1) */
        SERCOM2->I2CM.ADDR.reg = (SLAVE_ADDR << 1) | 1;
        while(!rx_done);
        /*interrupts are cleared */
        SERCOM2->I2CM.INTENCLR.reg = SERCOM_I2CM_INTENCLR_MB | SERCOM_I2CM_INTENCLR_SB;
}

int main (void)
{
        system_init();

        i2c_clock_init();

        i2c_pin_init();

        i2c_master_init();

        i2c_master_transact();

        while(1);
}
```

5.1.9 I²C Slave Initialization

The `i2c_slave_init` function will initialize the I²C slave function by configuring the control registers, address register, and setting the respective interrupt enable bits.

```
void i2c_slave_init()
{
	/* By setting the SPEED bit field as 0x01, I2C communication runs at 1MHz,
	   By setting the SDAHOLD bit field as 0x02, SDA hold time is configured for 300-600ns,
	   By setting the RUNSTDBY bit as 0x01,Generic clock is enabled in all sleep modes,
	   any interrupt can wake up the device,
	   SERCOM2 is configured as an I2C Slave by writing the MODE bitfield as 0x04 */
	SERCOM2->I2CS.CTRLA.reg = SERCOM_I2CS_CTRLA_SPEED (FAST_MODE_PLUS)     |
	                          SERCOM_I2CS_CTRLA_SDAHOLD(0x2)               |
	                          SERCOM_I2CM_CTRLA_RUNSTDBY                   |
	                          SERCOM_I2CS_CTRLA_MODE_I2C_SLAVE;
	/* smart mode enabled by setting the bit SMEN as 1 */
	SERCOM2->I2CS.CTRLB.reg = SERCOM_I2CS_CTRLB_SMEN;
	/* writing the slave address into ADDR register */
	SERCOM2->I2CS.ADDR.reg = SLAVE_ADDR << 1 ;
	/* Address match interrupt, Data ready interrupt,stop received
	interrupts are enabled */
	SERCOM2->I2CS.INTENSET.reg = SERCOM_I2CS_INTENSET_PREC | SERCOM_I2CS_INTENSET_AMATCH |
	SERCOM_I2CS_INTENSET_DRDY;
	/* SERCOM2 peripheral enabled by setting the ENABLE bit as 1*/
	SERCOM2->I2CS.CTRLA.reg |= SERCOM_I2CS_CTRLA_ENABLE;
	/* SERCOM enable synchronization busy */
	while((SERCOM2->I2CS.SYNCBUSY.reg & SERCOM_I2CS_SYNCBUSY_ENABLE));
	/* SERCOM2 handler enabled */
	system_interrupt_enable(SERCOM2_IRQn);
}
```

- I²C slave CTRLA register is used to configure the I²C speed, SDA Hold time, and the I²C mode of the device. In the above function, I²C runs at fast mode plus (1MHz) and is configured to have a SDA hold time of 300 - 600ns, and device runs as I²C slave mode. SERCOM module is made to run even in standby sleep mode.
- CTRLB register is used to configure the acknowledge action, user to write commands during transaction and to enable smart mode. In the above function Smart Mode is enabled.
- ADDR register is used to hold the address of slave
- The INTENSET register is used to enable the required interrupts. In the above function, address match interrupt, stop received interrupt and data ready interrupts are enabled.
- CTRLA, CTRLB, and BAUD registers can be written only when the I²C is disabled because these registers are enable protected. So once configuring these registers, the I²C is enabled.
- Due to the asynchronicity between CLK_SERCOMx_APB and GCLK_SERCOMx_CORE, some registers must be synchronized when accessed. ENABLE bit in CTRLA register is Write-Synchronized so the application should wait until the synchronization busy flag (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 SERCOM2 interrupt request line (IRQ - 11) is enabled.

5.1.10 I²C Slave Transaction

SERCOM2 Handler will serve the I²C slave transaction function.

```
void SERCOM2_Handler(void)
{
	/* Check for Address match interrupt */
	if(SERCOM2->I2CS.INTFLAG.bit.AMATCH)
	{
		/* clearing the Address match interrupt */
		SERCOM2->I2CS.INTFLAG.bit.AMATCH = 1;
	}

	/* Data Ready interrupt check */
	if(SERCOM2->I2CS.INTFLAG.bit.DRDY)
	{
	/* Checking for direction,
	DIR - 0 for slave read,
	DIR - 1 for slave write */
	if (SERCOM2->I2CS.STATUS.bit.DIR)
	{
		/* Slave write */
		if (i == (BUF_SIZE-1))
		{
			SERCOM2->I2CS.DATA.reg = rx_buff[i++];
			/* wait for stop condition */
			SERCOM2->I2CS.CTRLB.bit.CMD = 0x2;
			i = 0;
		}
		else
		{
			SERCOM2->I2CS.DATA.reg = rx_buff[i++];
	 /* Execute a byte read operation followed by ACK/NACK reception by master*/
			SERCOM2->I2CS.CTRLB.bit.CMD = 0x3;
		}
	}
	else
	{
		/* Slave read */
		if (i == (BUF_SIZE-1))
		{

			SERCOM2->I2CS.CTRLB.bit.ACKACT = 0;
/* Execute acknowledge action succeeded by waiting for any start     (S/Sr) condition */
			SERCOM2->I2CS.CTRLB.bit.CMD = 0x2;

		}
		else
		{
			rx_buff[i++] = SERCOM2->I2CS.DATA.reg;
			SERCOM2->I2CS.CTRLB.bit.ACKACT = 0;
/* Execute acknowledge action succeeded by reception of next byte to master*/
			SERCOM2->I2CS.CTRLB.bit.CMD = 0x3;
			}
		}
	}
	if (SERCOM2->I2CS.INTFLAG.bit.PREC)
	{
		SERCOM2->I2CS.INTFLAG.bit.PREC = 1;
```

```
                if (!SERCOM2->I2CS.STATUS.bit.DIR)
                {
                        rx_buff[i++] = SERCOM2->I2CS.DATA.reg;
                }
                i = 0;
        }

}
```

- In the SERCOM2 handler, three interrupt conditions are checked:
  - Address match interrupt
  - Data Ready interrupt
  - Stop Interrupt
- Address match interrupt flag is set when the received address matches the configured slave address. Clearing the flag by writing one to it will automatically send the configured acknowledge action when smart mode is enabled.
- Data Ready interrupt is set when I²C slave byte needs to be transmitted or received
- In I²C status register, DIR bit will be set for master read operation and cleared for master write operation
- When master transmits the buffer data, code will enter into the below loop of slave SERCOM2 handler

```
        /* Slave read */
                        if (i == (BUF_SIZE-1))
                        {

                                SERCOM2->I2CS.CTRLB.bit.ACKACT = 0;
/* Execute acknowledge action succeeded by waiting for any  start (S/Sr) condition */
                                SERCOM2->I2CS.CTRLB.bit.CMD = 0x2;
                                i = 0;
                        }
                        else
                        {
                                rx_buff[i++] = SERCOM2->I2CS.DATA.reg;
                                SERCOM2->I2CS.CTRLB.bit.ACKACT = 0;

/* Execute acknowledge action succeeded by reception of next byte to master*/
                                SERCOM2->I2CS.CTRLB.bit.CMD = 0x3;
        }
```

- Once receiving the data byte in the DATA register, it will be copied into receive buffer rx_buff
- Acknowledgement action (ACK) for each byte should be sent by the slave. After receiving, a byte slave sends the ACK by writing command value 0x3 to CMD bits in CTRLB register. Command value 0x3 is used to send acknowledge action followed by next byte reception.
- For the last data byte, command value 0x2 is written to CMD bits in CTRLB register, which waits for a STOP/Repeated START condition. The last data received should only be read after receiving a STOP condition.
- Inside STOP interrupt condition, the application will clear the STOP interrupt flag and read the last received data into the receive buffer in master write mode

```
        if (SERCOM2->I2CS.INTFLAG.bit.PREC)
                {
                        SERCOM2->I2CS.INTFLAG.bit.PREC = 1;
```

```
                if (!SERCOM2->I2CS.STATUS.bit.DIR)
                {
                        rx_buff[i++] = SERCOM2->I2CS.DATA.reg;
                }
                i = 0;
        }
```

- Below code is for the master read operation:

```
if (SERCOM2->I2CS.STATUS.bit.DIR)
{
        /* Slave write */
        if (i == (BUF_SIZE-1))
        {
                SERCOM2->I2CS.DATA.reg = rx_buff[i++];
                /* wait for stop condition */
                SERCOM2->I2CS.CTRLB.bit.CMD = 0x2;
                i = 0;
        }
        else
        {
                SERCOM2->I2CS.DATA.reg = rx_buff[i++];
        /* Execute a byte read operation followed by ACK/NACK reception by master */
                SERCOM2->I2CS.CTRLB.bit.CMD = 0x3;
        }
}
```

- Received data from the master will be in the receive buffer rx_buff
- In master read operation, receive buffer data i.e.data byte which received from the master is transmitted by slave to master by placing in the DATA register
- Once writing the data byte, the command is set to 0x3 for executing a byte read operation by master followed by ACK/NACK reception by slave
- For the last byte transfer, after writing the last data byte stop command will be placed in command register

The final application “Basic Configuration” in main.c file will be as below for **SLAVE**:

```
#include <asf.h>
#define STANDARD_MODE_FAST_MODE  0x0
#define FAST_MODE_PLUS           0X01
#define HIGHSPEED_MODE           0X02
#define SLAVE_ADDR               0x12
#define BUF_SIZE                 3

/* Function Prototype */
void i2c_clock_init(void);
void i2c_pin_init(void);
void i2c_slave_init(void);
uint8_t i = 0;
uint8_t rx_buff[BUF_SIZE];

        /* SERCOM2 I2C handler */
        void SERCOM2_Handler(void)
        {
        /* Check for Address match interrupt */
        if(SERCOM2->I2CS.INTFLAG.bit.AMATCH)
```

```
	{
		/* clearing the Address match interrupt */
		SERCOM2->I2CS.INTFLAG.bit.AMATCH = 1;
	}

	/* Data Ready interrupt check */
	if(SERCOM2->I2CS.INTFLAG.bit.DRDY)
	{
	/* Checking for direction,
	DIR - 0 for slave read,
	DIR - 1 for slave write */
	if (SERCOM2->I2CS.STATUS.bit.DIR)
	{
		/* Slave write */
		if (i == (BUF_SIZE-1))
		{
			SERCOM2->I2CS.DATA.reg = rx_buff[i++];
			/* wait for stop condition */
			SERCOM2->I2CS.CTRLB.bit.CMD = 0x2;
			i = 0;
		}
		else
		{
			SERCOM2->I2CS.DATA.reg = rx_buff[i++];
		/* Execute a byte read operation followed by ACK/NACK reception by master */
			SERCOM2->I2CS.CTRLB.bit.CMD = 0x3;
		}
	}
	else
	{
		/* Slave read */
		if (i == (BUF_SIZE-1))
		{
			SERCOM2->I2CS.CTRLB.bit.ACKACT = 0;
	/* Execute acknowledge action succeeded by waiting for any start (S/Sr) condition */
			SERCOM2->I2CS.CTRLB.bit.CMD = 0x2;
		}
		else
		{
			rx_buff[i++] = SERCOM2->I2CS.DATA.reg;
			SERCOM2->I2CS.CTRLB.bit.ACKACT = 0;
	/* Execute acknowledge action succeeded by reception of next byte    to master*/
			SERCOM2->I2CS.CTRLB.bit.CMD = 0x3;
		}
	}
	}
	if (SERCOM2->I2CS.INTFLAG.bit.PREC)
	{
		SERCOM2->I2CS.INTFLAG.bit.PREC = 1;
		if (!SERCOM2->I2CS.STATUS.bit.DIR)
		{
			rx_buff[i++] = SERCOM2->I2CS.DATA.reg;
		}
		i = 0;
	}
}

/*Assigning pin to the alternate peripheral function*/
static inline void pin_set_peripheral_function(uint32_t pinmux)
```

```
{
	uint8_t port = (uint8_t)((pinmux >> 16)/32);
	PORT->Group[port].PINCFG[((pinmux >> 16) - (port*32))].bit.PMUXEN = 1;
	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)));
}
/* SERCOM clock and peripheral bus clock initialization */
void i2c_clock_init()
{
	struct system_gclk_chan_config gclk_chan_conf;
	uint32_t gclk_index = SERCOM2_GCLK_ID_CORE;
	/* Turn on module in PM */
	system_apb_clock_set_mask(SYSTEM_CLOCK_APB_APBC, PM_APBCMASK_SERCOM2);
	/* Turn on Generic clock for I2C */
	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);
}
/* I2C pin initialization */
void i2c_pin_init()
{
    /* PA08 and PA09 set into peripheral function D*/
	pin_set_peripheral_function(PINMUX_PA08D_SERCOM2_PAD0);
	pin_set_peripheral_function(PINMUX_PA09D_SERCOM2_PAD1);
}
/* I2C Slave initialization */
void i2c_slave_init()
{
	/* By setting the SPEED bit field as 0x01, I2C communication runs at 1MHz,
	   By setting the SDAHOLD bit field as 0x02, SDA hold time is configured for 300-600ns,
	   By setting the RUNSTDBY bit as 0x01,Generic clock is enabled in all sleep modes,
	   any interrupt can wake up the device,
	   SERCOM2 is configured as an I2C Slave by writing the MODE bitfield as 0x04 */
	SERCOM2->I2CS.CTRLA.reg = SERCOM_I2CS_CTRLA_SPEED (FAST_MODE_PLUS)     |
	                          SERCOM_I2CS_CTRLA_SDAHOLD(0x2)               |
	                          SERCOM_I2CM_CTRLA_RUNSTDBY                   |
	                          SERCOM_I2CS_CTRLA_MODE_I2C_SLAVE;
	/* smart mode enabled by setting the bit SMEN as 1 */
	SERCOM2->I2CS.CTRLB.reg = SERCOM_I2CS_CTRLB_SMEN;
	/* writing the slave address into ADDR register */
	SERCOM2->I2CS.ADDR.reg = SLAVE_ADDR << 1 ;
	/* Address match interrupt, Data ready interrupt,stop received
	interrupts are enabled */
	SERCOM2->I2CS.INTENSET.reg = SERCOM_I2CS_INTENSET_PREC | SERCOM_I2CS_INTENSET_AMATCH |
	SERCOM_I2CS_INTENSET_DRDY;
	/* SERCOM2 peripheral enabled by setting the ENABLE bit as 1*/
	SERCOM2->I2CS.CTRLA.reg |= SERCOM_I2CS_CTRLA_ENABLE;
	/* SERCOM enable synchronization busy */
	while((SERCOM2->I2CS.SYNCBUSY.reg & SERCOM_I2CS_SYNCBUSY_ENABLE));
	/* SERCOM2 handler enabled */
	system_interrupt_enable(SERCOM2_IRQn);
}

	int main (void)
	{
```

```
        system_init();
        i2c_clock_init();
        i2c_pin_init();
        i2c_slave_init();
        while(1);
}
```

Scope plot for Basic configuration application – Fast mode plus (1MHz).

Figure 5-3. Complete Transaction

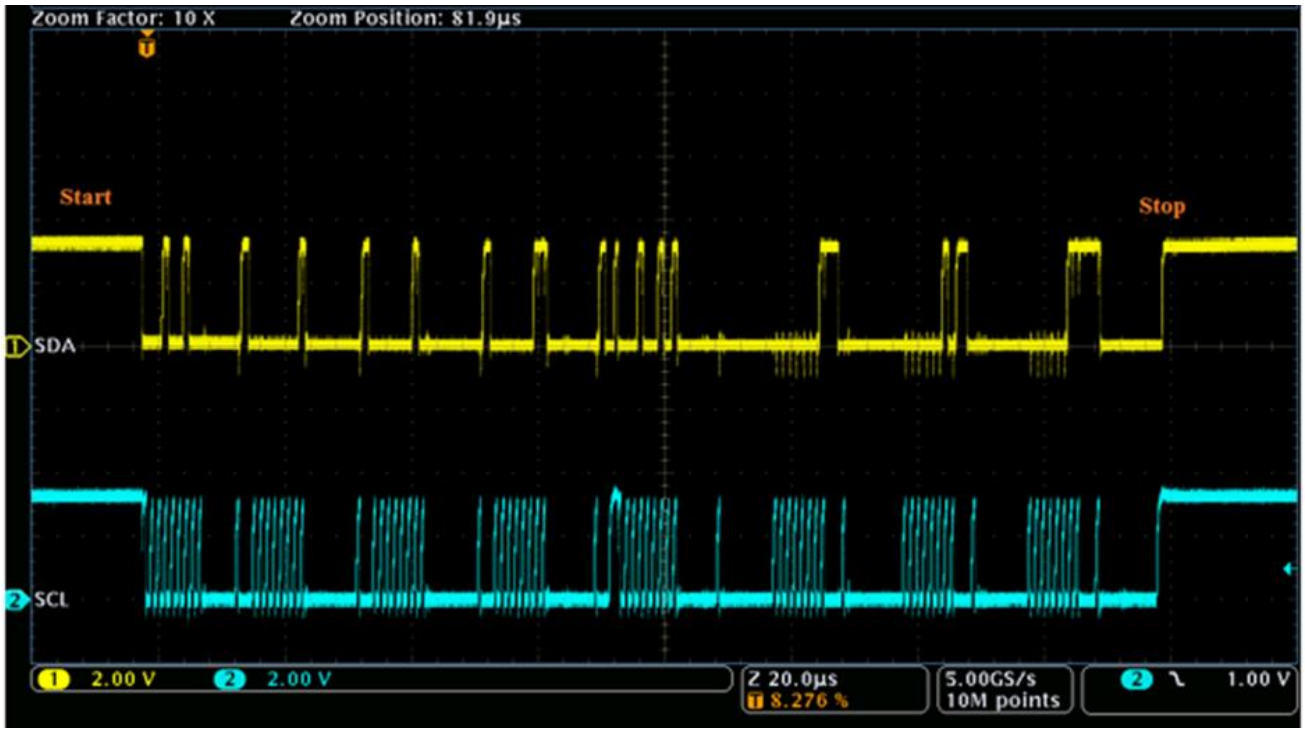


Figure 5-4. Start Condition + Address + Write Transmission

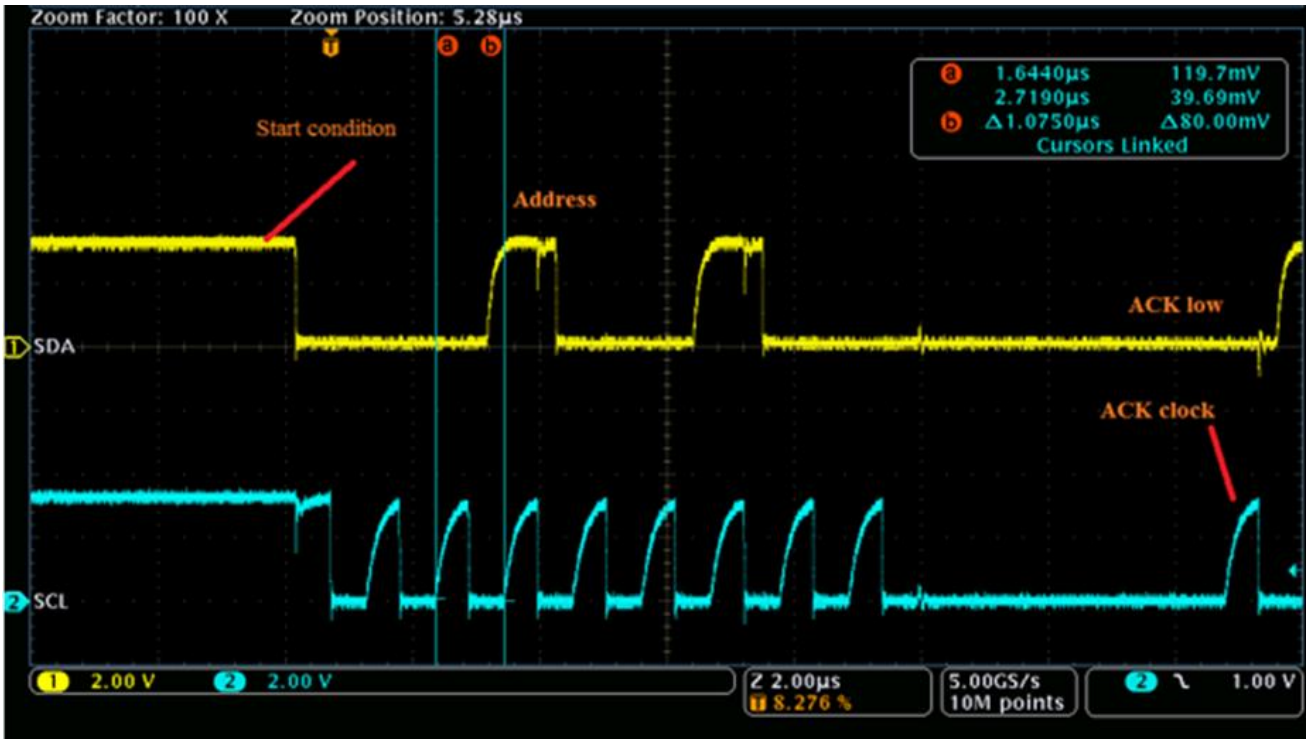
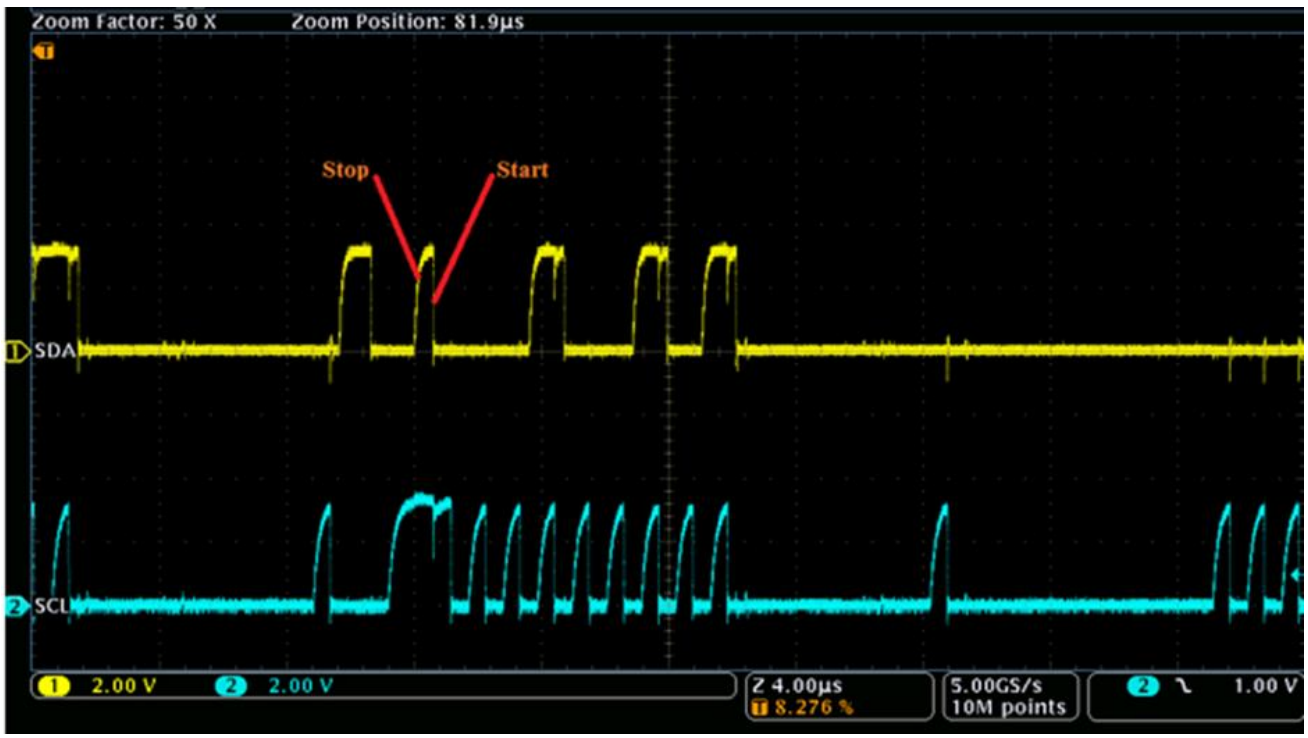


Figure 5-5. Master Stops and Starts Transaction Again for Read



The Complete Project solution for all the applications can be found in the zipped folder attachment that comes with this application note.

5.2 High Speed Configuration

In high speed configuration, SERCOM I²C communicates at an SCL clock frequency of 3.4MHz.

Unlike other speed modes the high speed mode starts the communication in full speed mode (fast mode – 400kHz) frequency in which it sends a unique code and then continues the normal transaction in high speed frequency (3.4MHz).

High speed mode application example does the following actions:

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

The example configuration is almost similar to the Basic Configuration example but with some additional steps. This section explains only the additional steps from the Basic Configuration step.

5.2.1 Master and Slave Clock in High-speed Configuration

The default ASF clock configuration in 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.

Following changes should be implemented in 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 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 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)
```

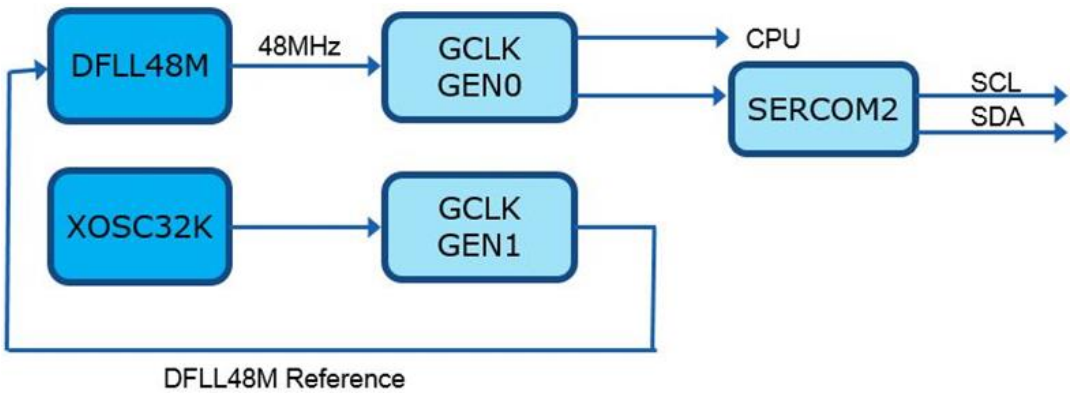
5. Set DFLL48M as clock source for GCLK Generator 0 which sources the main clock domain and also used as clock source for SERCOM module.

```
/* 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
```

5.2.2 Clock Flow for Master and Slave

Figure 5-6. Clock Flow Diagram for Master and Slave



5.2.3 System Initialization

`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 5.2.1. It also initializes the board hardware of SAM D21 Xplained Pro and the event system.

5.2.4 I²C clock Initialization

This section is the same as Section 5.1.5.

5.2.5 I²C Pin Initialization

This section is the same as Section 5.1.6.

Note: System initialization, I²C clock initialization, and I²C pin initialization are common for both master and slave. For the slave part the same will be applicable.

5.2.6 I²C Master Initialization

The `i2c_master_init` function will initialize the I²C master by configuring the control registers, baud registers, and setting the respective interrupt enable bits.

```
void i2c_master_init()
{
	/* By setting the SPEED bit field as 0x02, I2C Master runs at High speed - 3.4 MHz,
	   By setting the SDAHOLD bit field as 0x02, SDA hold time is configured for 300-600ns,
	   By setting the RUNSTDBY bit as 0x01,Generic clock is enabled in all sleep modes,
	   any interrupt can wake up the device,
	   SERCOM2 is configured as an I2C Master by writing the MODE bitfield as 0x5,
	   SCL stretch only after ACK bit is selected by setting the SCLSM  bit as 1
	*/
	SERCOM2->I2CM.CTRLA.reg = SERCOM_I2CM_CTRLA_SPEED (HIGHSPEED_MODE)   |
	                          SERCOM_I2CM_CTRLA_SDAHOLD(0x2)             |
	                          SERCOM_I2CM_CTRLA_RUNSTDBY                 |
	                          SERCOM_I2CM_CTRLA_SCLSM                    |
	                          SERCOM_I2CM_CTRLA_MODE_I2C_MASTER;
	/* smart mode enabled */
	SERCOM2->I2CM.CTRLB.reg = SERCOM_I2CM_CTRLB_SMEN;
	/* synchronization busy */
	while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
	/* loading the BAUD register */
	SERCOM2->I2CM.BAUD.reg = SERCOM_I2CM_BAUD_HSBAUD(4) | SERCOM_I2CM_BAUD_HSBAUDLOW(8) |
	SERCOM_I2CM_BAUD_BAUD(48) | SERCOM_I2CM_BAUD_BAUDLOW(48);
	/* synchronization busy */
	while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
   /* SERCOM2 peripheral enabled by setting the ENABLE bit as 1*/
	SERCOM2->I2CM.CTRLA.reg |= SERCOM_I2CM_CTRLA_ENABLE;
	/* SERCOM Enable synchronization busy */
	while((SERCOM2->I2CM.SYNCBUSY.reg & SERCOM_I2CM_SYNCBUSY_ENABLE));
	/* bus state is forced into idle state */
	SERCOM2->I2CM.STATUS.bit.BUSSTATE = 0x1;
	/* SERCOM2 handler enabled */
	system_interrupt_enable(SERCOM2_IRQn);
}
```

- The CTRLA register is used to configure the I²C speed, SDA hold time, and I²C mode. In the above function I²C runs at high speed mode – 3.4MHz, SDA hold time is set for 300 - 600ns. I²C is configured as master and the SERCOM module is made to run even in standby sleep mode.
- CTRLB register is used to write commands and to enable Smart Mode. In the above function SMEN – Smart Mode Enable is set.
- Equation for calculating the SCL frequency is:

  $F_{SCL} = F_{GCLK}/2 + HSBAUD + HSBAUDLOW$

  F_{SCL} = I²C clock frequency

  F_{GCLK} = SERCOM generic clock frequency

  HSBAUD = BAUD register value of high speed

  HSBAUDLOW = BAUD LOW register value of high speed

From the equation:

$HSBAUD + HSBAUDLOW = (F_{GCLK} / F_{SCL}) - 2$

$= (48M/3.4M) - 2$

$=12.11 \sim= 12$

HSBAUD = 4

HSBAUDLOW = 8

Note: For High-speed the nominal high to low SCL ratio is 1 to 2 and HSBAUD should be set accordingly. At a minimum, BAUD.BAUD and/or BAUD.BAUDLOW must be non-zero. Since the high speed transfer starts in full speed mode, the BAUD and BAUDLOW bits should be set to values corresponding to 400kHz apart from configuring the HSBAUD and HSBAUDLOW bits.

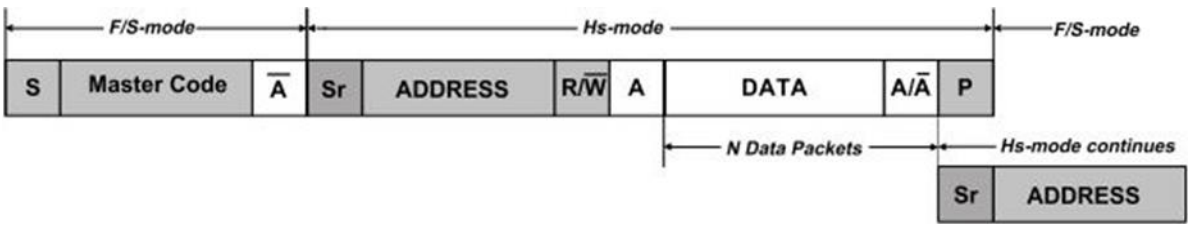
The nominal high to low SCL ratio is 1 to 2 is not applicable for 100 and 400 KHz, so the BAUD and BAUDLOW value should be equal.

- First, a master code (0000 1nnn where nnn is a unique master code) is transmitted in full speed mode, followed by a NACK reception since no slave should acknowledge this code. Arbitration is performed only during the full speed Master Code phase. The master code is transmitted by writing the master code to the address register (ADDR) with the high-speed bit (ADDR.HS) written to zero.
- In BAUD register HSBAUD, HSBAUDLOW, BAUD, and BAUDLOW values are loaded
- CTRLA, CTRLB, and BAUD registers can be written only when the I²C is disabled because these registers are enable protected. So once configuring these registers, the I²C is enabled.
- Due to the asynchronicity between CLK_SERCOMx_APB and GCLK_SERCOMx_CORE, some registers must be synchronized when accessed. The ENABLE bit in CTRLA register is Write-Synchronized so the application has to wait until the SYSOP bit in SYNCBUSY register is cleared after writing to the ENABLE bit.
- The I²C bus-state is unknown when the master is disabled. During this time writing 0x1 to BUSSTATE forces the bus state into the idle state.
- 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 SERCOM2 interrupt request line (IRQ - 11) is enabled.

5.2.7 I²C Master Transaction

The `i2c_master_transact()` function will transfer the master unique code in fast mode followed by a NACK reception since no slave should acknowledge this code.

Figure 5-7. High Speed Transfer




```
void i2c_master_transact(void)
{
	i = 0;
	/* Master unique code + NACK signal  */
	SERCOM2->I2CM.CTRLB.reg |= SERCOM_I2CM_CTRLB_ACKACT;
	while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
	SERCOM2->I2CM.ADDR.reg = UNIQUE_CODE;
	while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
	/* wait till MB flag set */
	while(!SERCOM2->I2CM.INTFLAG.bit.MB);
	/* clearing the MB interrupt */
	SERCOM2->I2CM.INTFLAG.bit.MB = 1;
	/* ACK signal */
```

```
        SERCOM2->I2CM.CTRLB.reg &= ~SERCOM_I2CM_CTRLB_ACKACT;
        /* synchronization busy */
        while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
        /* slave address+w, High speed transfer enabled */
        SERCOM2->I2CM.ADDR.reg = (SLAVE_ADDR << 1) | 0 | SERCOM_I2CM_ADDR_HS;
        /* synchronization busy */
        while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
        /* MB and SB interrupts are set */
        SERCOM2->I2CM.INTENSET.reg = SERCOM_I2CM_INTENSET_MB | SERCOM_I2CM_INTENSET_SB;
        while(!tx_done);
        /* MB and SB interrupts are cleared */
        SERCOM2->I2CM.INTENCLR.reg = SERCOM_I2CM_INTENCLR_MB | SERCOM_I2CM_INTENCLR_SB;
        i = 0;
        /* Master unique code + NACK signal  */
        SERCOM2->I2CM.CTRLB.reg |= SERCOM_I2CM_CTRLB_ACKACT;
        while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP)
        /* Master unique code + NACK signal  */;
        SERCOM2->I2CM.ADDR.reg = UNIQUE_CODE;
        while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
        /* wait till MB flag set */
        while(!SERCOM2->I2CM.INTFLAG.bit.MB);
        /* clearing the MB interrupt */
        SERCOM2->I2CM.INTFLAG.bit.MB = 1;
        /* ACK signal */
        SERCOM2->I2CM.CTRLB.reg &= ~SERCOM_I2CM_CTRLB_ACKACT;
        /* synchronization busy */
        while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
        /* slave address+R, High speed transfer enabled */
        SERCOM2->I2CM.ADDR.reg = (SLAVE_ADDR << 1) | 1 | SERCOM_I2CM_ADDR_HS;
        /* synchronization busy */
        while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
        //while(!SERCOM2->I2CM.INTFLAG.bit.MB);
        //SERCOM2->I2CM.INTFLAG.bit.MB = 1;
        /* MB and SB interrupts are set */
        SERCOM2->I2CM.INTENSET.reg = SERCOM_I2CM_INTENSET_MB | SERCOM_I2CM_INTENSET_SB;
        while(!rx_done);
        /* MB and SB interrupts are cleared */
        SERCOM2->I2CM.INTENCLR.reg = SERCOM_I2CM_INTENCLR_MB | SERCOM_I2CM_INTENCLR_SB;
}
```

- CTRLB register is used to configure the commands, acknowledge action, Smart mode enabling. In the above function, NACK action is set for the Master unique code.
- In the Application code we have the Master code of 0x0A. Format of master code is 0000 1nnn where nnn is unique master code.
- Master code is written into the ADDR register with HS bit in the ADDR register set to 0
- MB interrupt flag will be set once the master code is transmitted
- NACK will be received as acknowledge action is set as NACK in the CTRLB register
- After receiving the NACK signal, the acknowledge action should be set as ACK, and address of slave is written in ADDR register
- The Address of slave is left shifted by 1 bit and LSB of ADDR register is set as 0 for master write
- HS bit in ADDR register should be enabled for high speed mode
- Once placing the address in the ADDR register, the address will be kept in Data register and transferred to Slave
- MB and SB interrupts are enabled

- The Boolean flag variable is initialized as false, so it while remain in the while loop and reach the SERCOM2 handler

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

- Once transmitting the Address of slave, the Master on bus interrupt will be set and SERCOM2_Handler will be serviced
- SERCOM2_Handler is the same as Basic configuration
- Once writing the data bytes, the application stops the I²C transaction by using stop command
- Before starting the master read operation, the master unique code should be used as before
- In ADDR register, LSB should be set as 1 for master read operation

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

The final application “High speed Configuration” in main.c file will be as below for **MASTER**.

```
#include <asf.h>


#define STANDARD_MODE_FAST_MODE  0x0
#define FAST_MODE_PLUS           0X01
#define HIGHSPEED_MODE           0X02
#define SLAVE_ADDR               0x12
#define BUF_SIZE                  3
#define UNIQUE_CODE              0x0A

/* Function Prototype */
void i2c_clock_init(void);
void i2c_pin_init(void);
void i2c_master_init(void);
void i2c_master_transact(void);


uint8_t tx_buf[BUF_SIZE] = {1, 2, 3};
uint8_t rx_buf[BUF_SIZE];
uint8_t i;
volatile bool tx_done = false, rx_done = false, addr_transmitted = false;
uint32_t read_flag;

/* I2C handler */
void SERCOM2_Handler(void)
{
	/* Master on bus interrupt checking */
	if (SERCOM2->I2CM.INTFLAG.bit.MB)
		{
			if (i == BUF_SIZE)
			{
			/* After transferring the last byte stop condition will be sent */
			SERCOM2->I2CM.CTRLB.bit.CMD = 0x3;
			while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
			tx_done = true;
			i = 0;
		}
		else
		{
			/* placing the data from transmitting buffer to DATA register*/
			SERCOM2->I2CM.DATA.reg = tx_buf[i++];
			while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
		}
```

```
	}
	/* Slave on bus interrupt checking */
	if (SERCOM2->I2CM.INTFLAG.bit.SB)
	{
	if (i == (BUF_SIZE-1))
		{
			/* NACK should be sent before reading the last byte */
			SERCOM2->I2CM.CTRLB.reg |= SERCOM_I2CM_CTRLB_ACKACT;
			while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
			SERCOM2->I2CM.CTRLB.bit.CMD = 0x3;
			while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
			rx_buf[i++] = SERCOM2->I2CM.DATA.reg;
			while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
			rx_done = true;
		}
		else
		{
			SERCOM2->I2CM.CTRLB.reg &= ~SERCOM_I2CM_CTRLB_ACKACT;
			while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
			rx_buf[i++] = SERCOM2->I2CM.DATA.reg;
			while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
			/* sending ACK after reading each byte */
			SERCOM2->I2CM.CTRLB.bit.CMD = 0x2;
			while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
		}
	}
}

/*Assigning pin to the alternate peripheral function*/
static inline void pin_set_peripheral_function(uint32_t pinmux)
{
	uint8_t port = (uint8_t)((pinmux >> 16)/32);
	PORT->Group[port].PINCFG[((pinmux >> 16) - (port*32))].bit.PMUXEN = 1;
	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)));
}

/* SERCOM clock and peripheral bus clock initialization */
void i2c_clock_init()
{
	struct system_gclk_chan_config gclk_chan_conf;
	uint32_t gclk_index = SERCOM2_GCLK_ID_CORE;
	/* Turn on module in PM */
	system_apb_clock_set_mask(SYSTEM_CLOCK_APB_APBC, PM_APBCMASK_SERCOM2);
	/* Turn on Generic clock for I2C */
	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);
}

/* I2C pin initialization */
void i2c_pin_init()
{
	/* PA08 and PA09 set into peripheral function D*/
	pin_set_peripheral_function(PINMUX_PA08D_SERCOM2_PAD0);
```

```
	pin_set_peripheral_function(PINMUX_PA09D_SERCOM2_PAD1);
}

/* I2C master initialization */
void i2c_master_init()
{
	/* By setting the SPEED bit field as 0x02, I2C Master runs at High speed - 3.4 MHz,
	   By setting the SDAHOLD bit field as 0x02, SDA hold time is configured for 300-600ns,
	   By setting the RUNSTDBY bit as 0x01,Generic clock is enabled in all sleep modes,
	   any interrupt can wake up the device,
	   SERCOM2 is configured as an I2C Master by writing the MODE bitfield as 0x5,
	   SCL stretch only after ACK bit is selected by setting the SCLSM  bit as 1
	*/
	SERCOM2->I2CM.CTRLA.reg = SERCOM_I2CM_CTRLA_SPEED (HIGHSPEED_MODE)   |
	                          SERCOM_I2CM_CTRLA_SDAHOLD(0x2)             |
	                          SERCOM_I2CM_CTRLA_RUNSTDBY                 |
	                          SERCOM_I2CM_CTRLA_SCLSM                    |
	                          SERCOM_I2CM_CTRLA_MODE_I2C_MASTER;
	/* smart mode enabled */
	SERCOM2->I2CM.CTRLB.reg = SERCOM_I2CM_CTRLB_SMEN;
	/* synchronization busy */
	while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
	/* loading the BAUD register */
	SERCOM2->I2CM.BAUD.reg = SERCOM_I2CM_BAUD_HSBAUD(4) | SERCOM_I2CM_BAUD_HSBAUDLOW(8) |
	SERCOM_I2CM_BAUD_BAUD(48) | SERCOM_I2CM_BAUD_BAUDLOW(48);
	/* synchronization busy */
	while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
    /* SERCOM2 peripheral enabled by setting the ENABLE bit as 1*/
	SERCOM2->I2CM.CTRLA.reg |= SERCOM_I2CM_CTRLA_ENABLE;
	/* SERCOM Enable synchronization busy */
	while((SERCOM2->I2CM.SYNCBUSY.reg & SERCOM_I2CM_SYNCBUSY_ENABLE));
	/* bus state is forced into idle state */
	SERCOM2->I2CM.STATUS.bit.BUSSTATE = 0x1;
	/* SERCOM2 handler enabled */
	system_interrupt_enable(SERCOM2_IRQn);
}

/* I2C master Transaction */
void i2c_master_transact(void)
{
	i = 0;
	/* Master unique code + NACK signal  */
	SERCOM2->I2CM.CTRLB.reg |= SERCOM_I2CM_CTRLB_ACKACT;
	while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
	SERCOM2->I2CM.ADDR.reg = UNIQUE_CODE;
	while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
	/* wait till MB flag set */
	while(!SERCOM2->I2CM.INTFLAG.bit.MB);
	/* clearing the MB interrupt */
	SERCOM2->I2CM.INTFLAG.bit.MB = 1;
	/* ACK signal */
	SERCOM2->I2CM.CTRLB.reg &= ~SERCOM_I2CM_CTRLB_ACKACT;
	/* synchronization busy */
	while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
	/* slave address+w, High speed transfer enabled */
	SERCOM2->I2CM.ADDR.reg = (SLAVE_ADDR << 1) | 0 | SERCOM_I2CM_ADDR_HS;
	/* synchronization busy */
	while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
	/* MB and SB interrupts are set */
```

```
        SERCOM2->I2CM.INTENSET.reg = SERCOM_I2CM_INTENSET_MB | SERCOM_I2CM_INTENSET_SB;
        while(!tx_done);
        /* MB and SB interrupts are cleared */
        SERCOM2->I2CM.INTENCLR.reg = SERCOM_I2CM_INTENCLR_MB | SERCOM_I2CM_INTENCLR_SB;
        i = 0;
        /* Master unique code + NACK signal  */
        SERCOM2->I2CM.CTRLB.reg |= SERCOM_I2CM_CTRLB_ACKACT;
        while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP)
        /* Master unique code + NACK signal  */;
        SERCOM2->I2CM.ADDR.reg = UNIQUE_CODE;
        while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
        /* wait till MB flag set */
        while(!SERCOM2->I2CM.INTFLAG.bit.MB);
        /* clearing the MB interrupt */
        SERCOM2->I2CM.INTFLAG.bit.MB = 1;
        /* ACK signal */
        SERCOM2->I2CM.CTRLB.reg &= ~SERCOM_I2CM_CTRLB_ACKACT;
        /* synchronization busy */
        while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
        /* slave address+R, High speed transfer enabled */
        SERCOM2->I2CM.ADDR.reg = (SLAVE_ADDR << 1) | 1 | SERCOM_I2CM_ADDR_HS;
        /* synchronization busy */
        while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
        //while(!SERCOM2->I2CM.INTFLAG.bit.MB);
        //SERCOM2->I2CM.INTFLAG.bit.MB = 1;
        /* MB and SB interrupts are set */
        SERCOM2->I2CM.INTENSET.reg = SERCOM_I2CM_INTENSET_MB | SERCOM_I2CM_INTENSET_SB;
        while(!rx_done);
        /* MB and SB interrupts are cleared */
        SERCOM2->I2CM.INTENCLR.reg = SERCOM_I2CM_INTENCLR_MB | SERCOM_I2CM_INTENCLR_SB;
}

int main (void)
{
        system_init();

        i2c_clock_init();

        i2c_pin_init();

        i2c_master_init();

        i2c_master_transact();

        while(1);

}
```

5.2.8 I²C Slave Initialization

The `i2c_slave_init` function will initialize the I²C slave by configuring the control registers, address register, and setting the respective interrupt enable bits.

```
void i2c_slave_init()
{
        /* By setting the SPEED bit field as 0x02, I2C Slave runs at High speed - 3.4 MHz,
           By setting the RUNSTDBY bit as 0x01,Generic clock is enabled in all sleep modes,
```

```
        any interrupt can wake up the device,
        SERCOM2 is configured as an I2C Slave by writing the MODE bitfield as 0x04,
        SCL stretch only after ACK bit is selected by setting the SCLSM  bit as 1,
        SERCOM2 is configured as an I2C Slave by writing the MODE bitfield as 0x04
    */
    SERCOM2->I2CS.CTRLA.reg = SERCOM_I2CS_CTRLA_SPEED (HIGHSPEED_MODE)   |
                              SERCOM_I2CS_CTRLA_RUNSTDBY                 |
                              SERCOM_I2CS_CTRLA_SCLSM                    |
                              SERCOM_I2CS_CTRLA_MODE_I2C_SLAVE;
    /* smart mode enabled by setting the bit SMEN as 1 */
    SERCOM2->I2CS.CTRLB.reg = SERCOM_I2CS_CTRLB_SMEN;
    //while(SERCOM2->I2CS.STATUS.bit.SYNCBUSY);
    /* writing the slave address into ADDR register */
    SERCOM2->I2CS.ADDR.reg = SLAVE_ADDR << 1 ;
    /* Address match interrupt, Data ready interrupt,stop received
    interrupts are enabled */
    SERCOM2->I2CS.INTENSET.reg = SERCOM_I2CS_INTENSET_PREC | SERCOM_I2CS_INTENSET_AMATCH |
    SERCOM_I2CS_INTENSET_DRDY;
    /* SERCOM2 peripheral enabled */
    SERCOM2->I2CS.CTRLA.reg |= SERCOM_I2CS_CTRLA_ENABLE;
    /* SERCOM enable synchronization busy */
    while((SERCOM2->I2CS.SYNCBUSY.reg & SERCOM_I2CS_SYNCBUSY_ENABLE));
    /* SERCOM2 handler enabled */
    system_interrupt_enable(SERCOM2_IRQn);
}
```

- I²C slave CTRLA register is used to configure the I²C speed, clock stretch, and the I²C mode of the device. In the above function, I²C runs at high speed, clock stretch only after ACK bit, and device runs as I²C slave mode. SERCOM module is made to run even in standby sleep mode.
- CTRLB register is used to configure the acknowledge action, command bits to write commands and enabling smart mode. In the above function the Smart Mode is enabled.
- ADDR register is used to hold the address of slave
- The INTENSET register is used to enable the required interrupts. In the above function, address match interrupt, stop received interrupt, and Data ready interrupt are enabled.
- CTRLA, CTRLB, and BAUD registers can be written only when the I²C is disabled because these registers are enable protected. So once configuring these registers, the I²C is enabled.
- Due to the asynchronicity between CLK_SERCOMx_APB and GCLK_SERCOMx_CORE, some registers must be synchronized when accessed. ENABLE bit in CTRLA register is Write-Synchronized so the application has to wait until SYSOP bit in SYNCBUSY register is cleared after writing to the ENABLE bit.
- 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 SERCOM2 interrupt request line (IRQ - 11) is enabled.

5.2.9 I²C Slave Transaction

SERCOM2 Handler will serve the I²C slave transaction function.

- In high speed mode, SCLSM bit should be set one, which is handled by application in the I²C slave initialization function
- For master reads (slave write), an address and data interrupt will be issued simultaneously with SCLSM =1

- So for this purpose inside the SERCOM handler application will check for the direction flag inside the AMATCH condition. If DIR is set (Master Read – Slave Write) a Boolean flag ‘slave_write_mode’ will be set to true.
- In the same AMATCH condition inside the SERCOM handler application will check for the flag ‘slave_write_mode’ flag. If it is true, the application will write the first data byte from the buffer ‘rx_buff’ followed by clearing the AMATCH interrupt flag. Otherwise the application will simply clear the AMATCH interrupt flag without performing any data read. Note that the same buffer ‘rx_buff’ will be used for both slave transmission and reception.
- Inside the DRDY condition in SERCOM handler application will check for the Boolean flag ‘slave_write_mode’ status. If it is true (Master Read – Slave Write), the application will write the data from the buffer ‘rx_buff’ for (BUF_SIZE-1) times since the first data byte was already transmitted in previous step. When DRDY is hit for the last time (iteration count is equal to BUF_SIZE) the I²C lines are released by writing command register with value 0x2.
- Otherwise, if the Boolean flag is false (Master Write – Slave Read), the application will read the received data into buffer and send acknowledgment for each byte received by writing command register with value 0x3. When the iteration count reaches BUF_SIZE-1 application will release the I²C lines by writing the command register with value 0x2 followed by reception of STOP condition.
- After reception of STOP condition (PREC condition inside SERCOM handler) application will clear the stop interrupt flag PREC and then reads the last received byte into buffer

```
void SERCOM2_Handler(void)
{
	/* Check for Address match interrupt */
	if(SERCOM2->I2CS.INTFLAG.bit.AMATCH)
	{
		/* Checking for direction,
		DIR - 0 for slave read,
		DIR - 1 for slave write */
		if (SERCOM2->I2CS.STATUS.bit.DIR)
			slave_write_mode = true;
		else
			slave_write_mode = false;
		if (slave_write_mode)
		{
			SERCOM2->I2CS.DATA.reg = rx_buff[i++];
			/* clearing the Address match interrupt */
			SERCOM2->I2CS.INTFLAG.bit.AMATCH = 1;
			SERCOM2->I2CS.CTRLB.bit.CMD = 0x3;
		}
		else
		{
			/* clearing the Address match interrupt */
			SERCOM2->I2CS.INTFLAG.bit.AMATCH = 1;
		}
	}

	/* Data Ready interrupt check */
	if(SERCOM2->I2CS.INTFLAG.bit.DRDY)
	{
	if (slave_write_mode)
	{
		if (i == BUF_SIZE)
		{
			//SERCOM2->I2CS.DATA.reg = rx_buff[i++];
```

```
                        /* wait for stop condition */
                        SERCOM2->I2CS.CTRLB.bit.CMD = 0x2;
                        i = 0;
                }
                else
                {
                        SERCOM2->I2CS.DATA.reg = rx_buff[i++];
        /* Execute a byte read operation followed by ACK/NACK reception by master */
                        SERCOM2->I2CS.CTRLB.bit.CMD = 0x3;
                }
        }
        else
        {
                if (i == BUF_SIZE-1)
                {
                        SERCOM2->I2CS.CTRLB.bit.ACKACT = 0;
/* Execute acknowledge action succeeded by waiting for any start (S/Sr) condition */
                        SERCOM2->I2CS.CTRLB.bit.CMD = 0x2;
                        //i = 0;
                }
                else
                {
                        rx_buff[i++] = SERCOM2->I2CS.DATA.reg;
                        SERCOM2->I2CS.CTRLB.bit.ACKACT = 0;
        /* Execute acknowledge action succeeded by reception of next byte to master*/
                        SERCOM2->I2CS.CTRLB.bit.CMD = 0x3;
                }
        }
        }
        /* Stop received interrupt check */

        if (SERCOM2->I2CS.INTFLAG.bit.PREC)
        {
                SERCOM2->I2CS.INTFLAG.bit.PREC = 1;
                if (!slave_write_mode)
                {
                        rx_buff[i++] = SERCOM2->I2CS.DATA.reg;
                }
                i = 0;
        }
}
```

The final application “High speed Configuration” in main.c file will be as below for **SLAVE**.

```
#include <asf.h>
#define STANDARD_MODE_FAST_MODE  0x0
#define FAST_MODE_PLUS           0X01
#define HIGHSPEED_MODE           0X02
#define SLAVE_ADDR              0x12
#define BUF_SIZE                   3

/* Function Prototype */
void i2c_clock_init(void);
void i2c_pin_init(void);
void i2c_slave_init(void);

uint8_t i = 0;
```

```
uint8_t rx_buff[BUF_SIZE];
volatile bool slave_write_mode = false;

/* I2C handler */
void SERCOM2_Handler(void)
{
	/* Check for Address match interrupt */
	if(SERCOM2->I2CS.INTFLAG.bit.AMATCH)
	{
		/* Checking for direction,
		DIR - 0 for slave read,
		DIR - 1 for slave write */
		if (SERCOM2->I2CS.STATUS.bit.DIR)
			slave_write_mode = true;
		else
			slave_write_mode = false;
		if (slave_write_mode)
		{
			SERCOM2->I2CS.DATA.reg = rx_buff[i++];
			/* clearing the Address match interrupt */
			SERCOM2->I2CS.INTFLAG.bit.AMATCH = 1;
			SERCOM2->I2CS.CTRLB.bit.CMD = 0x3;
		}
		else
		{
			/* clearing the Address match interrupt */
			SERCOM2->I2CS.INTFLAG.bit.AMATCH = 1;
		}
	}

	/* Data Ready interrupt check */
	if(SERCOM2->I2CS.INTFLAG.bit.DRDY)
	{
		if (slave_write_mode)
		{
			if (i == BUF_SIZE)
			{
				//SERCOM2->I2CS.DATA.reg = rx_buff[i++];
				/* wait for stop condition */
				SERCOM2->I2CS.CTRLB.bit.CMD = 0x2;
				i = 0;
			}
			else
			{
				SERCOM2->I2CS.DATA.reg = rx_buff[i++];

/* Execute a byte read operation followed by ACK/NACK reception by master */
				SERCOM2->I2CS.CTRLB.bit.CMD = 0x3;
			}
		}
		else
		{
		if (i == BUF_SIZE-1)
		{
			SERCOM2->I2CS.CTRLB.bit.ACKACT = 0;
/* Execute acknowledge action succeeded by waiting for any start (S/Sr) condition */
			SERCOM2->I2CS.CTRLB.bit.CMD = 0x2;
			//i = 0;
		}
```

```
                else
                {
                        rx_buff[i++] = SERCOM2->I2CS.DATA.reg;
                        SERCOM2->I2CS.CTRLB.bit.ACKACT = 0;
/* Execute acknowledge action succeeded by reception of next byte to master*/
                        SERCOM2->I2CS.CTRLB.bit.CMD = 0x3;
                }
        }
        }
        /* Stop received interrupt check */
        if (SERCOM2->I2CS.INTFLAG.bit.PREC)
        {
                SERCOM2->I2CS.INTFLAG.bit.PREC = 1;
                if (!slave_write_mode)
                {
                        rx_buff[i++] = SERCOM2->I2CS.DATA.reg;
                }
                i = 0;
        }
}

/*Assigning pin to the alternate peripheral function*/
static inline void pin_set_peripheral_function(uint32_t pinmux)
{
        uint8_t port = (uint8_t)((pinmux >> 16)/32);
        PORT->Group[port].PINCFG[((pinmux >> 16) - (port*32))].bit.PMUXEN = 1;
        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)));
}
/* SERCOM clock and peripheral bus clock initialization */
void i2c_clock_init()
{
        struct system_gclk_chan_config gclk_chan_conf;
        uint32_t gclk_index = SERCOM2_GCLK_ID_CORE;
        /* Turn on module in PM */
        system_apb_clock_set_mask(SYSTEM_CLOCK_APB_APBC, PM_APBCMASK_SERCOM2);
        /* Turn on Generic clock for I2C */
        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);
}
/* I2C pin initialization */
void i2c_pin_init()
{
        /* PA08 and PA09 set into peripheral function D*/
        pin_set_peripheral_function(PINMUX_PA08D_SERCOM2_PAD0);
        pin_set_peripheral_function(PINMUX_PA09D_SERCOM2_PAD1);
}
/* I2C Slave initialization */
void i2c_slave_init()
{
        /* By setting the SPEED bit field as 0x02, I2C Slave runs at High speed - 3.4 MHz,
           By setting the RUNSTDBY bit as 0x01,Generic clock is enabled in all sleep modes,
           any interrupt can wake up the device,
           SERCOM2 is configured as an I2C Slave by writing the MODE bitfield as 0x04,
```

```
            SCL stretch only after ACK bit is selected by setting the SCLSM  bit as 1,
            SERCOM2 is configured as an I2C Slave by writing the MODE bitfield as 0x04
        */
        SERCOM2->I2CS.CTRLA.reg = SERCOM_I2CS_CTRLA_SPEED (HIGHSPEED_MODE)   |
                                  SERCOM_I2CS_CTRLA_RUNSTDBY                 |
                                  SERCOM_I2CS_CTRLA_SCLSM                    |
                                  SERCOM_I2CS_CTRLA_MODE_I2C_SLAVE;
        /* smart mode enabled by setting the bit SMEN as 1 */
        SERCOM2->I2CS.CTRLB.reg = SERCOM_I2CS_CTRLB_SMEN;
        //while(SERCOM2->I2CS.STATUS.bit.SYNCBUSY);
        /* writing the slave address into ADDR register */
        SERCOM2->I2CS.ADDR.reg = SLAVE_ADDR << 1 ;
        /* Address match interrupt, Data ready interrupt,stop received
        interrupts are enabled */
        SERCOM2->I2CS.INTENSET.reg = SERCOM_I2CS_INTENSET_PREC | SERCOM_I2CS_INTENSET_AMATCH |
        SERCOM_I2CS_INTENSET_DRDY;
        /* SERCOM2 peripheral enabled */
        SERCOM2->I2CS.CTRLA.reg |= SERCOM_I2CS_CTRLA_ENABLE;
        /* SERCOM enable synchronization busy */
        while((SERCOM2->I2CS.SYNCBUSY.reg & SERCOM_I2CS_SYNCBUSY_ENABLE));
        /* SERCOM2 handler enabled */
        system_interrupt_enable(SERCOM2_IRQn);
}
int main (void)
{
        system_init();

        i2c_clock_init();

        i2c_pin_init();

        i2c_slave_init();

        while(1);

}
```

The upcoming figures depicts the scope plots of I²C communication in High speed configuration mode – (3.4MHz).

Figure 5-8. Complete Transaction

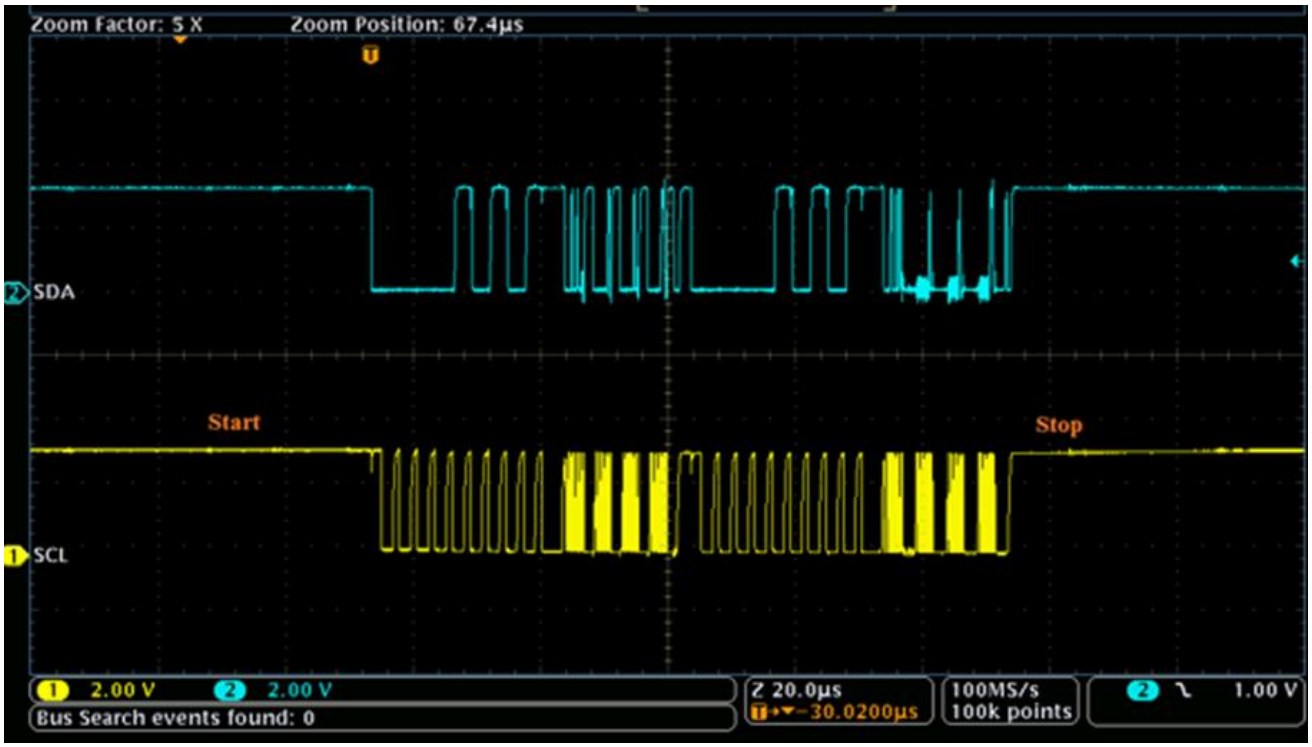


Figure 5-9. Transaction Showing Unique Code Sent in Full Speed Mode Followed by Normal I²C Transfer in High-speed Mode

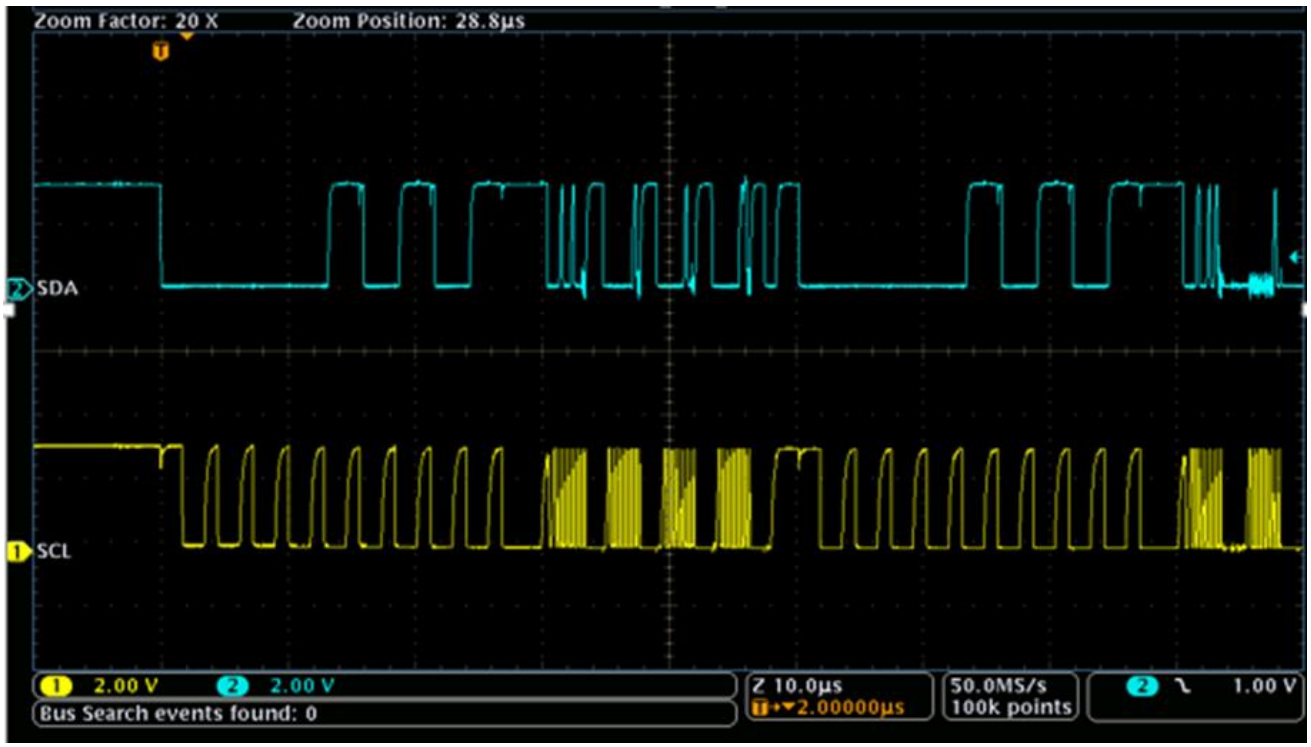


Figure 5-10. SCL Timing

5.3 Address Match and Mode Configuration

This section explains about the various address matching modes supported by the SERCOM I²C slave. Three address matching modes are supported as given below.

1. Address match and mask mode (ADDR bits are used to hold address value and ADDRMASK bits are used to hold mask bits in ADDR register). Respective bit positions set in ADDRMASK bit field represents those bits in ADDR register that will not be compared with respective incoming address bits.
2. Two unique addresses mode (ADDR bits hold first address and ADDRMASK bits hold second address). The slave will respond only to these two addresses.
3. Range of addresses (ADDR bits hold the upper limit of the range and ADDRMASK holds the lower limit of the range). The slave will respond for an address within this range including the upper and lower limit values.

For demonstration purpose the first mode (Address match and mask mode) is used as explained in upcoming sections.

5.3.1 Address Match and Mask Mode

In the Address match and mask mode, an address written to the Address bits in the Address register (ADDR.ADDR) with a mask written to the Address Mask bits in the Address register (ADDR.ADDRMASK) will yield an address match.

The bits set in the ADDRMASK will not be compared for the address match and the rest of the bits are compared and acknowledged if these bits are matched.

If the ADDR.ADDRMASK is set for all the bits then all addresses are accepted.

The main purpose of this mode is to emulate single I²C slave as multiple slaves.

5.3.2 Master and Slave Clock

The master and slave application uses OSC8M as the clock source for Generator 0.

This section is the same as Section 5.1.2.

5.3.3 Clock Flow for Master and slave

This section is the same as Section 5.1.3.

5.3.4 System Initialization

This section is the same as Section 5.1.4.

5.3.5 I²C clock Initialization

This section is the same as Section 5.1.5.

5.3.6 I²C Pin Initialization

This section is the same as Section 5.1.6.

Note: System initialization, I²C clock initialization and I²C pin initialization are common for both master and slave. For the slave part the same will be applicable.

5.3.7 I²C Master Initialization

The `i2c_master_init` function will initialize the I²C master function by configuring the control registers, baud registers, and setting the respective interrupt enable bits.

```
void i2c_master_init()
{
	/* By setting the SPEED bit field as 0x00, I2C Master runs at standard mode,
	   By setting the SDAHOLD bit field as 0x02, SDA hold time is configured for 300-600ns,
	   By setting the RUNSTDBY bit as 0x01,Generic clock is enabled in all sleep modes,
	   any interrupt can wake up the device,
	   SERCOM2 is configured as an I2C Master by writing the MODE bitfield as 0x5 */
	SERCOM2->I2CM.CTRLA.reg = SERCOM_I2CM_CTRLA_SPEED (STANDARD_MODE_FAST_MODE)   |
	                          SERCOM_I2CM_CTRLA_SDAHOLD(0x2)                      |
	                          SERCOM_I2CM_CTRLA_RUNSTDBY                          |
	                          SERCOM_I2CM_CTRLA_MODE_I2C_MASTER;
	/* smart mode enabled by setting the bit SMEN as 1 */
	SERCOM2->I2CM.CTRLB.reg = SERCOM_I2CM_CTRLB_SMEN;
	/* synchronization busy */
	while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
	/* BAUDLOW is non-zero, and baud register is loaded */
	SERCOM2->I2CM.BAUD.reg = SERCOM_I2CM_BAUD_BAUD(34) | SERCOM_I2CM_BAUD_BAUDLOW(34);
	/* synchronization busy */
	while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
	/* SERCOM2 peripheral enabled by setting the ENABLE bit as 1*/
	SERCOM2->I2CM.CTRLA.reg |= SERCOM_I2CM_CTRLA_ENABLE;
	/* SERCOM Enable synchronization busy */
	while((SERCOM2->I2CM.SYNCBUSY.reg & SERCOM_I2CM_SYNCBUSY_ENABLE));
	/* bus state is forced into idle state */
	SERCOM2->I2CM.STATUS.bit.BUSSTATE = 0x1;
	/* Both master on bus and slave on bus interrupt is enabled */
	SERCOM2->I2CM.INTENSET.reg = SERCOM_I2CM_INTENSET_MB | SERCOM_I2CM_INTENSET_SB;
	/* SERCOM2 handler enabled */
	system_interrupt_enable(SERCOM2_IRQn);
}
```

- The CTRLA register is used to configure the I²C speed, SDA hold time, and I²C mode. In the above function the I²C speed is configured in standard mode (100kHz), SDA hold time is set for 300 - 600ns. I²C is configured as master and the I²C module is made to run even in standby sleep mode.
- CTRLB register is used to write the commands and to enable Smart Mode. In the above function SMEN – Smart Mode Enable bit is set.
- For BAUD.BAUDLOW non-zero, the following equation is used to determine the SCL frequency, FSCL = f_{GCLK} / (10 + BAUD +BAUDLOW + f_{GCLK} T_{RISE})

  F_{SCL} = I²C clock frequency

  f_{GCLK} = SERCOM generic clock frequency

  BAUD = BAUD register value

  BAUDLOW = BAUD LOW register value

  TRISE = Rise time for I²C in the defined mode

Rise time for the respective speed modes can be found in section Electrical Characteristics → Timing Characteristics → SERCOM I²C Mode Timing in SAM D21 device datasheet.

In this configuration, the I²C runs at the speed of 100kHz and the worst case rise time for standard and fast mode is 300ns.

From the equation:

BAUD + BAUDLOW = f_{GCLK} / F_{SCL} – (f_{GCLK} T_{RISE}) – 10

= 8M/100k – (8M x 300ns) – 10

= 67.6

The BAUD value is set to 34 and BAUDLOW value is set to 34.

- CTRLA, CTRLB, and BAUD registers can be written only when the I²C is disabled because these registers are enable protected. So once configuring these registers, the I²C is enabled.
- Due to the asynchronicity between CLK_SERCOMx_APB and GCLK_SERCOMx_CORE, some registers must be synchronized when accessed. The ENABLE bit in CTRLA register is Write-Synchronized so the application has to wait until the SYSOP bit in SYNCBUSY register is cleared after writing to the ENABLE bit.
- The I²C bus-state is unknown, when the master is disabled. During this time writing 0x1 to BUSSTATE forces the bus state into the idle state.
- 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 SERCOM2 interrupt request line (IRQ - 11) is enabled.
- The INTENSET register is used to enable the required interrupts. In the above function SB – Slave on bus interrupt and MB – Master on bus interrupts are enabled

5.3.8 I²C Master Transaction

The `i2c_master_transact` function is used to perform a transaction with the connected slave device.

```
void i2c_master_transact(void)
{
	i = 0;
	/* ACK signal */
	SERCOM2->I2CM.CTRLB.reg &= ~SERCOM_I2CM_CTRLB_ACKACT;
	/* synchronization busy */
```

```
        while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
        /* slave address 0 with receive */
        SERCOM2->I2CM.ADDR.reg = (SLAVE_ADDR_0 << 1) | 1;
        while(!rx_done);
        rx_done = false;
        SERCOM2->I2CM.CTRLB.reg &= ~SERCOM_I2CM_CTRLB_ACKACT;
        /* synchronization busy */
        while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
        /* slave address 1 with receive */
        SERCOM2->I2CM.ADDR.reg = (SLAVE_ADDR_1 << 1) | 1;
        while(!rx_done);
        rx_done = false;
        SERCOM2->I2CM.CTRLB.reg &= ~SERCOM_I2CM_CTRLB_ACKACT;
        /* synchronization busy */
        while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
        /* slave address 2 with receive */
        SERCOM2->I2CM.ADDR.reg = (SLAVE_ADDR_2 << 1) | 1;
        while(!rx_done);
        rx_done = false;
        SERCOM2->I2CM.CTRLB.reg &= ~SERCOM_I2CM_CTRLB_ACKACT;
        /* synchronization busy */
        while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
        /* slave address 3 with receive */
        SERCOM2->I2CM.ADDR.reg = (SLAVE_ADDR_3 << 1) | 1;
        while(!rx_done);
        rx_done = false;
        /*Interrupts are cleared */
        SERCOM2->I2CM.INTENCLR.reg = SERCOM_I2CM_INTENCLR_MB | SERCOM_I2CM_INTENCLR_SB;
}
```

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

```
uint8_t i
volatile bool tx_done = false, rx_done = false
```

- In CTRLB register, ACKACT field is used to define the I²C master's acknowledge behavior after a data byte is received from the I²C slave. The acknowledge action will be executed when a command is written to CTRLB.CMD bits or after a transfer with Smart Mode is enabled.
- In the above function, ACKACT is set to 0, so ACK will be sent by master after a data byte is received
- Since the application uses range of addresses configuration, the first slave address 'SLAVE_ADDR_0' is written in the address register by shifting the address left by 1 bit
- The I²C operation is read activity, so LSB of ADDR register is set into 1
- The Boolean flag variable tx_done is initialized as false, so it remains in the while loop until SERCOM2 handler sets it to true indicating the completion of transfer

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

- Once transmitting the address of 'SLAVE_ADDR_0', the Master on bus interrupt will be set and no action is taken in `SERCOM2_Handler`
- Once the master receives the data from I²C bus the Slave on bus interrupt (SB) is set. In SERCOM2 handler, application will check for slave on bus interrupt set condition.
- If it is set, then code enters inside the loop and NACK is sent to the slave. This because once reading the data byte the I²C master will stop the transaction. The last byte read in I²C transaction should have NACK as acknowledge signal.

- For last byte read first stop command should be issued then only the data byte should be read from the DATA register
- After reading the data byte, Boolean variable 'rx_done' is set as true and control reaches the `i2c_master_transact` function and writes the address of second slave and reaches the SERCOM2 handler as above
- Once completing all the four slave address transactions, interrupts are cleared in the INTCLR register

```
void SERCOM2_Handler(void)
{
	/* Slave on bus interrupt checking */
	if (SERCOM2->I2CM.INTFLAG.bit.SB)
	{
		/* sending NACK signal*/
		SERCOM2->I2CM.CTRLB.reg |= SERCOM_I2CM_CTRLB_ACKACT;
		while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
		/* execute acknowledge action followed by stop */
		SERCOM2->I2CM.CTRLB.bit.CMD = 0x3;
		while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
		/*reading the data byte into receive buffer */
		rx_buf[i++] = SERCOM2->I2CM.DATA.reg;
		while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
		rx_done = true;
	}
}
```

The final application "Address mode configuration" in main.c file will be as below for **MASTER**.

```
#include <asf.h>
#define STANDARD_MODE_FAST_MODE  0x0
#define FAST_MODE_PLUS           0X01
#define HIGHSPEED_MODE           0X02
#define SLAVE_ADDR_0            0x50
#define SLAVE_ADDR_1            0x51
#define SLAVE_ADDR_2            0x52
#define SLAVE_ADDR_3            0x53

/* Function Prototype */
void i2c_clock_init(void);
void i2c_pin_init(void);
void i2c_master_init(void);
void i2c_master_transact(void);


uint8_t rx_buf[4];
uint8_t i;
volatile bool tx_done = false, rx_done = false;

/* SERCOM2 I2C handler */
void SERCOM2_Handler(void)
{
	/* Slave on bus interrupt checking */
	if (SERCOM2->I2CM.INTFLAG.bit.SB)
	{
		/* sending NACK signal*/
		SERCOM2->I2CM.CTRLB.reg |= SERCOM_I2CM_CTRLB_ACKACT;
```

```
                        while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
                        /* execute acknowledge action followed by stop */
                        SERCOM2->I2CM.CTRLB.bit.CMD = 0x3;
                        while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
                        /*reading the data byte into receive buffer */
                        rx_buf[i++] = SERCOM2->I2CM.DATA.reg;
                        while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
                        rx_done = true;
        }
}

/*Assigning pin to the alternate peripheral function*/
static inline void pin_set_peripheral_function(uint32_t pinmux)
{
        uint8_t port = (uint8_t)((pinmux >> 16)/32);
        PORT->Group[port].PINCFG[((pinmux >> 16) - (port*32))].bit.PMUXEN = 1;
        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)));
}

/* SERCOM clock and peripheral bus clock initialization */
void i2c_clock_init()
{
        struct system_gclk_chan_config gclk_chan_conf;
        uint32_t gclk_index = SERCOM2_GCLK_ID_CORE;
        /* Turn on module in PM */
        system_apb_clock_set_mask(SYSTEM_CLOCK_APB_APBC, PM_APBCMASK_SERCOM2);
        /* Turn on Generic clock for I2C */
        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);
}
/* I2C pin initialization */
void i2c_pin_init()
{
        /* PA08 and PA09 set into peripheral function D*/
        pin_set_peripheral_function(PINMUX_PA08D_SERCOM2_PAD0);
        pin_set_peripheral_function(PINMUX_PA09D_SERCOM2_PAD1);
}
/* I2C master initialization */
void i2c_master_init()
{
        /* By setting the SPEED bit field as 0x00, I2C Master runs at standard mode,
           By setting the SDAHOLD bit field as 0x02, SDA hold time is configured for 300-600ns,
           By setting the RUNSTDBY bit as 0x01,Generic clock is enabled in all sleep modes,
           any interrupt can wake up the device,
           SERCOM2 is configured as an I2C Master by writing the MODE bitfield as 0x5 */
        SERCOM2->I2CM.CTRLA.reg = SERCOM_I2CM_CTRLA_SPEED (STANDARD_MODE_FAST_MODE)   |
                                  SERCOM_I2CM_CTRLA_SDAHOLD(0x2)                      |
                                  SERCOM_I2CM_CTRLA_RUNSTDBY                          |
                                  SERCOM_I2CM_CTRLA_MODE_I2C_MASTER;
        /* smart mode enabled by setting the bit SMEN as 1 */
        SERCOM2->I2CM.CTRLB.reg = SERCOM_I2CM_CTRLB_SMEN;
        /* synchronization busy */
        while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
```

```
	/* BAUDLOW is non-zero, and baud register is loaded */
	SERCOM2->I2CM.BAUD.reg = SERCOM_I2CM_BAUD_BAUD(34) | SERCOM_I2CM_BAUD_BAUDLOW(34);
	/* synchronization busy */
	while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
	/* SERCOM2 peripheral enabled by setting the ENABLE bit as 1*/
	SERCOM2->I2CM.CTRLA.reg |= SERCOM_I2CM_CTRLA_ENABLE;
	/* SERCOM Enable synchronization busy */
	while((SERCOM2->I2CM.SYNCBUSY.reg & SERCOM_I2CM_SYNCBUSY_ENABLE));
	/* bus state is forced into idle state */
	SERCOM2->I2CM.STATUS.bit.BUSSTATE = 0x1;
	/* Both master on bus and slave on bus interrupt is enabled */
	SERCOM2->I2CM.INTENSET.reg = SERCOM_I2CM_INTENSET_MB | SERCOM_I2CM_INTENSET_SB;
	/* SERCOM2 handler enabled */
	system_interrupt_enable(SERCOM2_IRQn);
}
/* I2C master Transaction */
void i2c_master_transact(void)
{
	i = 0;
	/* ACK signal */
	SERCOM2->I2CM.CTRLB.reg &= ~SERCOM_I2CM_CTRLB_ACKACT;
	/* synchronization busy */
	while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
	/* slave address 0 with receive */
	SERCOM2->I2CM.ADDR.reg = (SLAVE_ADDR_0 << 1) | 1;
	while(!rx_done);
	rx_done = false;
	SERCOM2->I2CM.CTRLB.reg &= ~SERCOM_I2CM_CTRLB_ACKACT;
	/* synchronization busy */
	while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
	/* slave address 1 with receive */
	SERCOM2->I2CM.ADDR.reg = (SLAVE_ADDR_1 << 1) | 1;
	while(!rx_done);
	rx_done = false;
	SERCOM2->I2CM.CTRLB.reg &= ~SERCOM_I2CM_CTRLB_ACKACT;
	/* synchronization busy */
	while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
	/* slave address 2 with receive */
	SERCOM2->I2CM.ADDR.reg = (SLAVE_ADDR_2 << 1) | 1;
	while(!rx_done);
	rx_done = false;
	SERCOM2->I2CM.CTRLB.reg &= ~SERCOM_I2CM_CTRLB_ACKACT;
	/* synchronization busy */
	while(SERCOM2->I2CM.SYNCBUSY.bit.SYSOP);
	/* slave address 3 with receive */
	SERCOM2->I2CM.ADDR.reg = (SLAVE_ADDR_3 << 1) | 1;
	while(!rx_done);
	rx_done = false;
	/*Interrupts are cleared */
	SERCOM2->I2CM.INTENCLR.reg = SERCOM_I2CM_INTENCLR_MB | SERCOM_I2CM_INTENCLR_SB;
}

int main (void)
{
	system_init();

	i2c_clock_init();

	i2c_pin_init();
```

```
        i2c_master_init();

        i2c_master_transact();

        while(1);

}
```

5.3.9 I²C Slave Initialization

The i2c_slave_init function will initialize the I²C slave function by configuring the control registers, address register with range of addresses, and setting the respective interrupt enable bits.

```
void i2c_slave_init()
{
         /* By setting the SPEED bit field as 0x00,I2C communication runs at standard mode-
100KHz,
          By setting the SDAHOLD bit field as 0x02, SDA hold time is configured for 300-600ns,
          By setting the RUNSTDBY bit as 0x01,Generic clock is enabled in all sleep modes,
          any interrupt can wake up the device,
          SERCOM2 is configured as an I2C Slave by writing the MODE bitfield as 0x04 */
        SERCOM2->I2CS.CTRLA.reg = SERCOM_I2CS_CTRLA_SPEED (STANDARD_MODE_FAST_MODE)   |
                                  SERCOM_I2CS_CTRLA_SDAHOLD(0x2)                      |
                                  SERCOM_I2CM_CTRLA_RUNSTDBY                          |
                                  SERCOM_I2CS_CTRLA_MODE_I2C_SLAVE;
        /* smart mode enabled by setting the bit SMEN as 1,
         Address match and mask mode is selected by setting the AMODE bitfield as 0x00 */
        SERCOM2->I2CS.CTRLB.reg = SERCOM_I2CS_CTRLB_SMEN | SERCOM_I2CS_CTRLB_AMODE(0);
        /*ADDR_ADDR is the main address and
         setting 0x3 in ADDRMASK will not include the bits 0 and 1 for address compare */
        SERCOM2->I2CS.ADDR.reg = SERCOM_I2CS_ADDR_ADDR(SLAVE_ADDR_0) | SERCOM_I2CS_ADDR_AD-
        DRMASK(0x3);
        /* Address match interrupt, Data ready interrupt,stop received
        interrupts are enabled */
        SERCOM2->I2CS.INTENSET.reg = SERCOM_I2CS_INTENSET_PREC | SERCOM_I2CS_INTENSET_AMATCH |
        SERCOM_I2CS_INTENSET_DRDY;
        /* SERCOM2 peripheral enabled */
        SERCOM2->I2CS.CTRLA.reg |= SERCOM_I2CS_CTRLA_ENABLE;
        /* SERCOM enable synchronization busy */
        while((SERCOM2->I2CS.SYNCBUSY.reg & SERCOM_I2CS_SYNCBUSY_ENABLE));
        /* SERCOM2 handler enabled */
        system_interrupt_enable(SERCOM2_IRQn);
}
```

- I²C slave CTRLA register is used to configure the I²C speed, SDA hold time, and the I²C mode of the device. In the above function, I²C runs at standard speed – 100kHz, SDA hold time as 300 – 600ns, and the device runs as I²C slave mode. Generic clocks will be enabled in all the sleep modes.
- CTRLB register is used to configure the acknowledge action, command to configure, smart mode enable, and address mode. In the above function the Smart Mode is enabled and range of addresses is selected in address mode.
- ADDR bits hold the upper limit of the range and ADDRMASK holds the lower limit of the range. The slave will respond for an address within this range.

- The INTENSET register is used to enable the required interrupts. In the above function, address match interrupt, stop received interrupt, and Data ready interrupt are enabled.
- CTRLA, CTRLB, and BAUD registers can be written only when the I²C is disabled because these registers are enable protected. So once configuring these registers, the I²C is enabled.
- Due to the asynchronicity between CLK_SERCOMx_APB and GCLK_SERCOMx_CORE, some registers must be synchronized when accessed. CTRLA register is Write-Synchronized so needs to check the synchronization busy.
- 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 SERCOM2 interrupt request line (IRQ - 11) is enabled

5.3.10 I²C Slave Transaction

SERCOM2 Handler will serve the I²C slave transaction function.

- In SERCOM2 handler, address match interrupt set will be checked and once it is set, the address is stored in a variable from the DATA register
- The address from the DATA register is right shifted by 1 bit and stored in a variable, because LSB of DATA register value is R/W value of I²C transaction.
- Now DRDY interrupt set condition is checked and if it sets it will enter inside the loop
- Inside the loop the Boolean flag ‘tx_done’ is checked for true condition, as its initialized value is false. This flag will be true once loading the value “i” in DATA register so now code enters into the checking for direction loop.
- Since it is master read – slave write operation so DIR flag will be set and enters inside the loop
- Code will check for the first slave address with the received address in the variable ’rxd_addr’
- Now writes the variable “i” value into the DATA register and writes the command 0x3 in CTRLB register, which is execute a byte read operation by master (slave write) followed by ACK/NACK reception by slave.
- Now Boolean flag ‘tx_done ’is set to true
- On receiving the ACK/NACK DRDY interrupt sets and flag ‘tx_done’ is checked for true and enters inside the loop. Now sets the flag ‘tx_done’ as false and command 0x2 is written into CTRLB register.
- Command 0x2 is waiting for stop condition
- The above steps repeats for the remaining range of addresses

```
void SERCOM2_Handler(void)
{
	/* Check for Address match interrupt */
	if(SERCOM2->I2CS.INTFLAG.bit.AMATCH)
	{
		/* getting the address from DATA register
		by shifting 1 bit right */
		rxd_addr = (SERCOM2->I2CS.DATA.reg >> 1);
		/* clearing the Address match interrupt */
		SERCOM2->I2CS.INTFLAG.bit.AMATCH = 1;
	}
	/* Data Ready interrupt check */
	if(SERCOM2->I2CS.INTFLAG.bit.DRDY)
	{
		if (tx_done)
		{
			tx_done = false;
			/* wait for stop condition */
```

```
                        SERCOM2->I2CS.CTRLB.bit.CMD = 0x2;
                }
                /* Checking for direction,
                DIR - 0 for slave read,
                DIR - 1 for slave write */
                else if (SERCOM2->I2CS.STATUS.bit.DIR)
                {
                        /*checking the received address with range of slave addresses */
                        if (rxd_addr == SLAVE_ADDR_0) {
                                i = 5;
                        }
                        else if (rxd_addr == SLAVE_ADDR_1) {
                                i = 10;
                        }
                        else if (rxd_addr == SLAVE_ADDR_2) {
                                i = 15;
                        }
                        else if (rxd_addr == SLAVE_ADDR_3) {
                                i = 20;
                        }
                        /* writing into the DATA register */
                        SERCOM2->I2CS.DATA.reg = i;
                        /* Execute a byte read operation followed by ACK/NACK reception */
                        SERCOM2->I2CS.CTRLB.bit.CMD = 0x3;
                        tx_done = true;
                }
        }
}
```

The final application “Address mode configuration” in main.c file will be as below for **SLAVE**.

```
#include <asf.h>
#define STANDARD_MODE_FAST_MODE   0x0
#define FAST_MODE_PLUS            0X01
#define HIGHSPEED_MODE            0X02
#define SLAVE_ADDR_0             0x50
#define SLAVE_ADDR_1             0x51
#define SLAVE_ADDR_2             0x52
#define SLAVE_ADDR_3             0x53

/* Function Prototype */
void i2c_clock_init(void);
void i2c_pin_init(void);
void i2c_slave_init(void);

uint8_t i = 0;
uint8_t rxd_addr;
volatile bool tx_done = false;

/* SERCOM2 I2C handler */
void SERCOM2_Handler(void)
{
        /* Check for Address match interrupt */
        if(SERCOM2->I2CS.INTFLAG.bit.AMATCH)
        {
                /* getting the address from DATA register
                by shifting 1 bit right */
```

```
                rxd_addr = (SERCOM2->I2CS.DATA.reg >> 1);
                /* clearing the Address match interrupt */
                SERCOM2->I2CS.INTFLAG.bit.AMATCH = 1;
        }
        /* Data Ready interrupt check */
        if(SERCOM2->I2CS.INTFLAG.bit.DRDY)
        {
                if (tx_done)
                {
                        tx_done = false;
                        /* wait for stop condition */
                        SERCOM2->I2CS.CTRLB.bit.CMD = 0x2;
                }
                /* Checking for direction,
                DIR - 0 for slave read,
                DIR - 1 for slave write */
                else if (SERCOM2->I2CS.STATUS.bit.DIR)
                {
                        /*checking the received address with range of slave addresses */
                        if (rxd_addr == SLAVE_ADDR_0) {
                                i = 5;
                        }
                        else if (rxd_addr == SLAVE_ADDR_1) {
                                i = 10;
                        }
                        else if (rxd_addr == SLAVE_ADDR_2) {
                                i = 15;
                        }
                        else if (rxd_addr == SLAVE_ADDR_3) {
                                i = 20;
                        }
                        /* writing into the DATA register */
                        SERCOM2->I2CS.DATA.reg = i;
                        /* Execute a byte read operation followed by ACK/NACK reception */
                        SERCOM2->I2CS.CTRLB.bit.CMD = 0x3;
                        tx_done = true;
                }
        }
}

/*Assigning pin to the alternate peripheral function*/
static inline void pin_set_peripheral_function(uint32_t pinmux)
{
        uint8_t port = (uint8_t)((pinmux >> 16)/32);
        PORT->Group[port].PINCFG[((pinmux >> 16) - (port*32))].bit.PMUXEN = 1;
        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)));
}
/* SERCOM clock and peripheral bus clock initialization */
void i2c_clock_init()
{
        struct system_gclk_chan_config gclk_chan_conf;
        uint32_t gclk_index = SERCOM2_GCLK_ID_CORE;
        /* Turn on module in PM */
        system_apb_clock_set_mask(SYSTEM_CLOCK_APB_APBC, PM_APBCMASK_SERCOM2);
        /* Turn on Generic clock for I2C */
        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);
}
/* I2C pin initialization */
void i2c_pin_init()
{
	/* PA08 and PA09 set into peripheral function D*/
	pin_set_peripheral_function(PINMUX_PA08D_SERCOM2_PAD0);
	pin_set_peripheral_function(PINMUX_PA09D_SERCOM2_PAD1);
}

/* I2C Slave initialization */
void i2c_slave_init()
{
	/* By setting the SPEED bit field as 0x00, I2C communication runs at standard mode -
	100KHz,
	   By setting the SDAHOLD bit field as 0x02, SDA hold time is configured for 300-600ns,
	   By setting the RUNSTDBY bit as 0x01,Generic clock is enabled in all sleep modes,
	   any interrupt can wake up the device,
	   SERCOM2 is configured as an I2C Slave by writing the MODE bitfield as 0x04 */
	SERCOM2->I2CS.CTRLA.reg = SERCOM_I2CS_CTRLA_SPEED (STANDARD_MODE_FAST_MODE)   |
	                          SERCOM_I2CS_CTRLA_SDAHOLD(0x2)                      |
	                          SERCOM_I2CM_CTRLA_RUNSTDBY                          |
	                          SERCOM_I2CS_CTRLA_MODE_I2C_SLAVE;
	/* smart mode enabled by setting the bit SMEN as 1,
	 Address match and mask mode is selected by setting the AMODE bitfield as 0x00 */
	SERCOM2->I2CS.CTRLB.reg = SERCOM_I2CS_CTRLB_SMEN | SERCOM_I2CS_CTRLB_AMODE(0);
	/*ADDR_ADDR is the main address and
	 setting 0x3 in ADDRMASK will not include the bits 0 and 1 for address compare */
	SERCOM2->I2CS.ADDR.reg = SERCOM_I2CS_ADDR_ADDR(SLAVE_ADDR_0) | SERCOM_I2CS_ADDR_AD-
	DRMASK(0x3);
	/* Address match interrupt, Data ready interrupt,stop received
	interrupts are enabled */
	SERCOM2->I2CS.INTENSET.reg = SERCOM_I2CS_INTENSET_PREC | SERCOM_I2CS_INTENSET_AMATCH |
	SERCOM_I2CS_INTENSET_DRDY;
	/* SERCOM2 peripheral enabled */
	SERCOM2->I2CS.CTRLA.reg |= SERCOM_I2CS_CTRLA_ENABLE;
	/* SERCOM enable synchronization busy */
	while((SERCOM2->I2CS.SYNCBUSY.reg & SERCOM_I2CS_SYNCBUSY_ENABLE));
	/* SERCOM2 handler enabled */
	system_interrupt_enable(SERCOM2_IRQn);
}

int main (void)
{
	system_init();

	i2c_clock_init();

	i2c_pin_init();

	i2c_slave_init();

	while(1);

}
```

Note: In this Address mode application, there is no scope plot attached. In this mode the slave responds to the master with specific data byte for each slave address.

Table 5-2. Data Byte for Different Slave Address

Slave address	Data byte respond to master by slave
0x50	5
0x51	10
0x52	15
0x53	20

6 References

SAM D21 Device Datasheet - http://www.atmel.com/Images/Atmel-42181-SAM-D21_Datasheet.pdf.

SAM D21 Xplained Pro user guide and schematics link - http://www.atmel.com/tools/atsamd21-xpro.aspx?tab=documents.

7 Revision History

Doc Rev.	Date	Comments
42631A	12/2015	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**

© 2015 Atmel Corporation. / Rev.: Atmel-42631A-SAM-D21-SERCOM-I2C-Configuration_ApplicationNote_AT11628_122015.

Atmel®, Atmel logo and combinations thereof, Enabling Unlimited Possibilities®, and others are registered trademarks or trademarks of Atmel Corporation in U.S. and other countries. ARM®, ARM Connected® logo, Cortex®, and others are the registered trademarks or trademarks of ARM Ltd. Windows® is a registered trademark of Microsoft Corporation in U.S. and or other countries. Other terms and product names may be trademarks of others.

DISCLAIMER: The information in this document is provided in connection with Atmel products. No license, express or implied, by estoppel or otherwise, to any intellectual property right is granted by this document or in connection with the sale of Atmel products. EXCEPT AS SET FORTH IN THE ATMEL TERMS AND CONDITIONS OF SALES LOCATED ON THE ATMEL WEBSITE, ATMEL ASSUMES NO LIABILITY WHATSOEVER AND DISCLAIMS ANY EXPRESS, IMPLIED OR STATUTORY WARRANTY RELATING TO ITS PRODUCTS INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. IN NO EVENT SHALL ATMEL BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, PUNITIVE, SPECIAL OR INCIDENTAL DAMAGES (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS AND PROFITS, BUSINESS INTERRUPTION, OR LOSS OF INFORMATION) ARISING OUT OF THE USE OR INABILITY TO USE THIS DOCUMENT, EVEN IF ATMEL HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Atmel makes no representations or warranties with respect to the accuracy or completeness of the contents of this document and reserves the right to make changes to specifications and products descriptions at any time without notice. Atmel does not make any commitment to update the information contained herein. Unless specifically provided otherwise, Atmel products are not suitable for, and shall not be used in, automotive applications. Atmel products are not intended, authorized, or warranted for use as components in applications intended to support or sustain life.

SAFETY-CRITICAL, MILITARY, AND AUTOMOTIVE APPLICATIONS DISCLAIMER: Atmel products are not designed for and will not be used in connection with any applications where the failure of such products would reasonably be expected to result in significant personal injury or death (“Safety-Critical Applications”) without an Atmel officer's specific written consent. Safety-Critical Applications include, without limitation, life support devices and systems, equipment or systems for the operation of nuclear facilities and weapons systems. Atmel products are not designed nor intended for use in military or aerospace applications or environments unless specifically designated by Atmel as military-grade. Atmel products are not designed nor intended for use in automotive applications unless specifically designated by Atmel as automotive-grade.