Aerospace & Defence | Lighting | Power Management | Internet of Things | Transportation

Components – EMEA

Engineering DACH

# Lab 4: I2C

**Quang Hai Nguyen**

Revision 0.99, 07.01.2020

# SAMD21 workshop

Five Years Out

## Revision History

| Revision, Date | Editor | Subject (major changes) |
|---|---|---|
| Revision 0.99, 07.01.2020 | Quang Hai Nguyen | Preliminary |

# Table of Contents

## List of Figures

**No table of figures entries found.**

## List of Icon Identifiers

*Table 1: Icon Identifiers List*

| | |
|---|---|
| ⓘ | Extra information about a topic |
| ✎ | Task needs to be done |
| ⚠ | Important information or a warning |
| ✓ | Result expected to see |

## Prerequisite

### Hardware

- Laptop or PC with Windows 7 or Windows 10
- A SAMD21-Xplained-Pro evaluation board (provided during the workshop)
- Extension boards, LEDs, Potential meter, buttons and jumper wires (provided during the workshop)
- A micro USB cable (provided during the workshop)
- Breadboard (provided during the workshop)
- **Internet access without proxy or firewall**

### Software

- Terminal program (e.g. TeraTerm)
- Atmel Studio 7 (link)

## Labs description

### Description

This lab will show you how to configure the I2C bus by using Atmel Start and how to interface with an ATECC508 secure element via the I2C bus.

### Assignment

1. Setup I2C bus with Atmel Start
2. Writing the code to wake up the ATECC508 secure element

⚠ Some of the basic configuration steps are skipped in this lab so make sure you have already started lab 1 to learn the basic steps before continuing with this lab.

## Lab procedure

### Setup I2C bus with Atmel Start

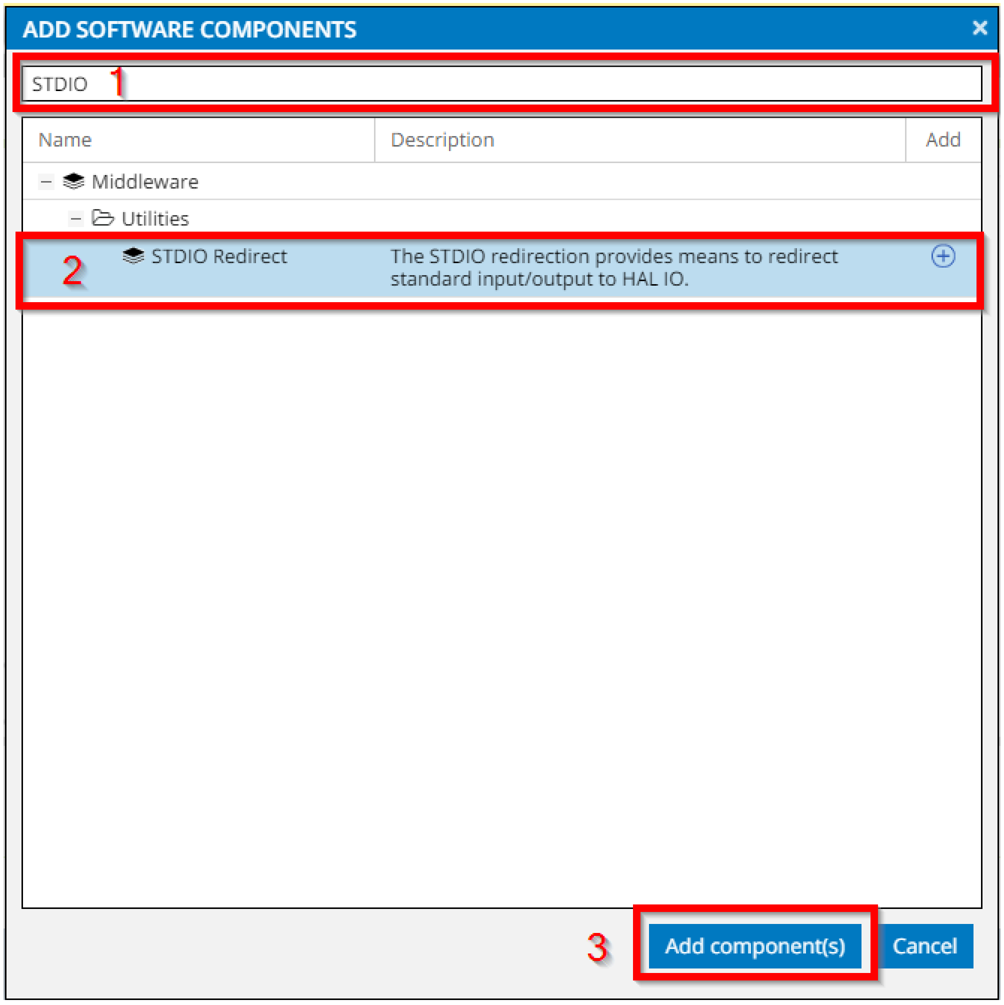
In Atmel Studio, create a new Atmel Start project.

When it is prompted, choose the correct device for your project. In this case, it is SAM D21 Xplained Pro.

Give your project a name. For my case, I name it SAMD21_Lab4.

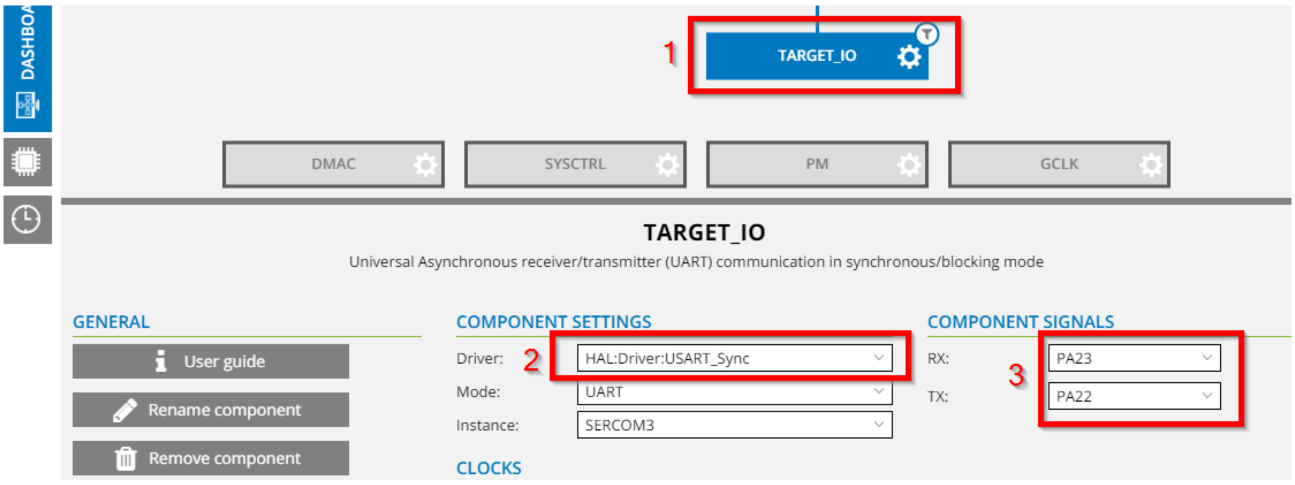
For this project, we need 2 components: I2C to interface to the secure element and STDIO to send the debug messages to the terminal. We will start setting up the STDIO.

Click “Add software component”. In the prompted window, enter “STDIO” into the filter text box and select “STDIO Redirect”



| Name | Description | Add |
|---|---|---|
| STDIO Redirect | The STDIO redirection provides means to redirect standard input/output to HAL IO. | ⊕ |

Click “TARGET_IO” and set the correct pin for the UART bus. From the “Board user guide”, we know that pin PA22 and PA23 are routed to output pins of USB Virtual Com. Therefore, they are configured as follows:



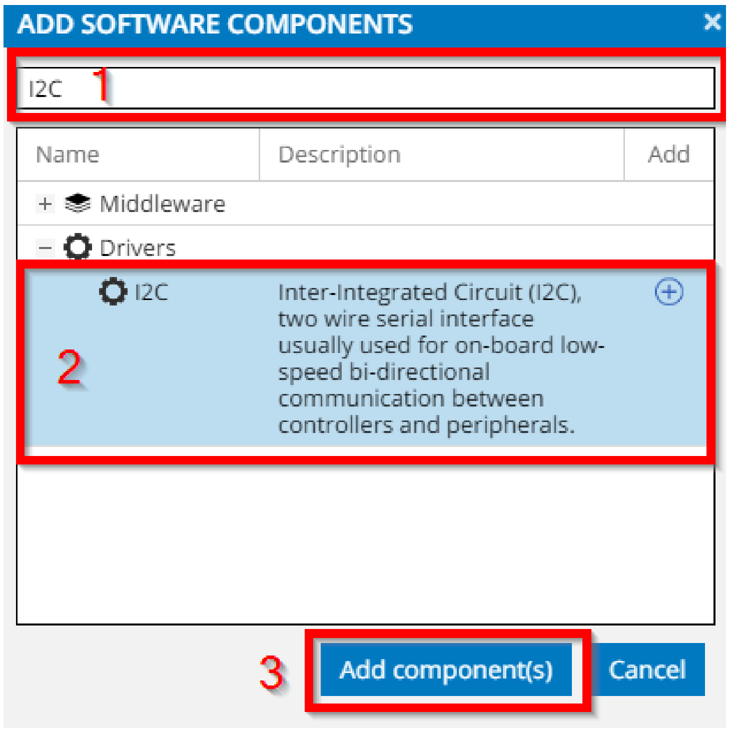
TARGET_IO

Universal Asynchronous receiver/transmitter (UART) communication in synchronous/blocking mode

| GENERAL | COMPONENT SETTINGS | | COMPONENT SIGNALS | |
|---|---|---|---|---|
| User guide | Driver: | HAL:Driver:USART_Sync | RX: | PA23 |
| Rename component | Mode: | UART | TX: | PA22 |
| Remove component | Instance: | SERCOM3 | | |

We will keep other configurations as they are.

Next step, we will set up our I2C bus. Click “Add software component”. In the prompted window, enter I2C into the filter text box and choose “I2C” under “Drivers category”.



| Name | Description | Add |
|---|---|---|
| + Middleware | | |
| − Drivers | | |
| I2C | Inter-Integrated Circuit (I2C), two wire serial interface usually used for on-board low-speed bi-directional communication between controllers and peripherals. | ⊕ |

According to the board user guide, SDA and SCK pins are located at pins PA08 and PA09.

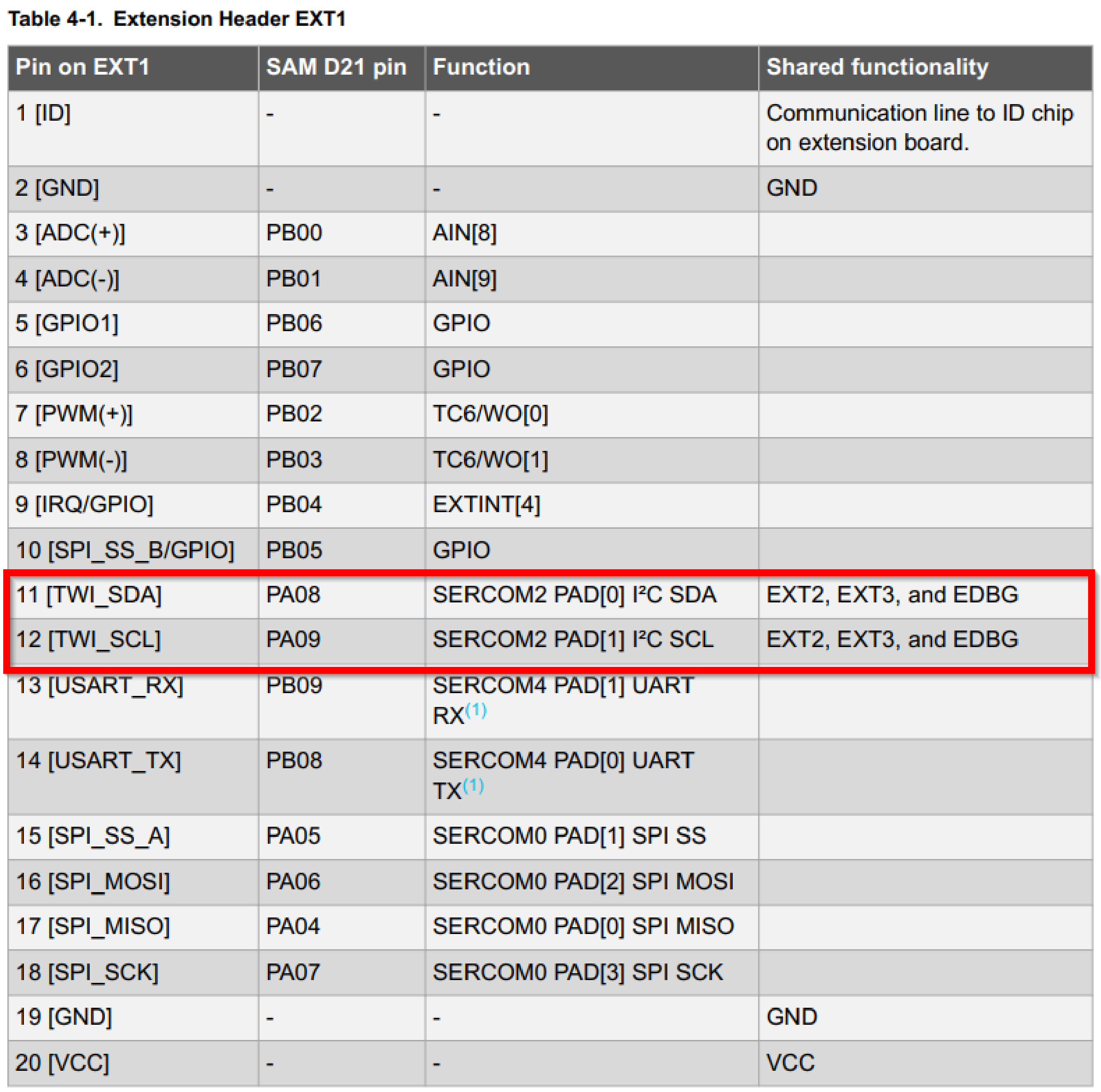
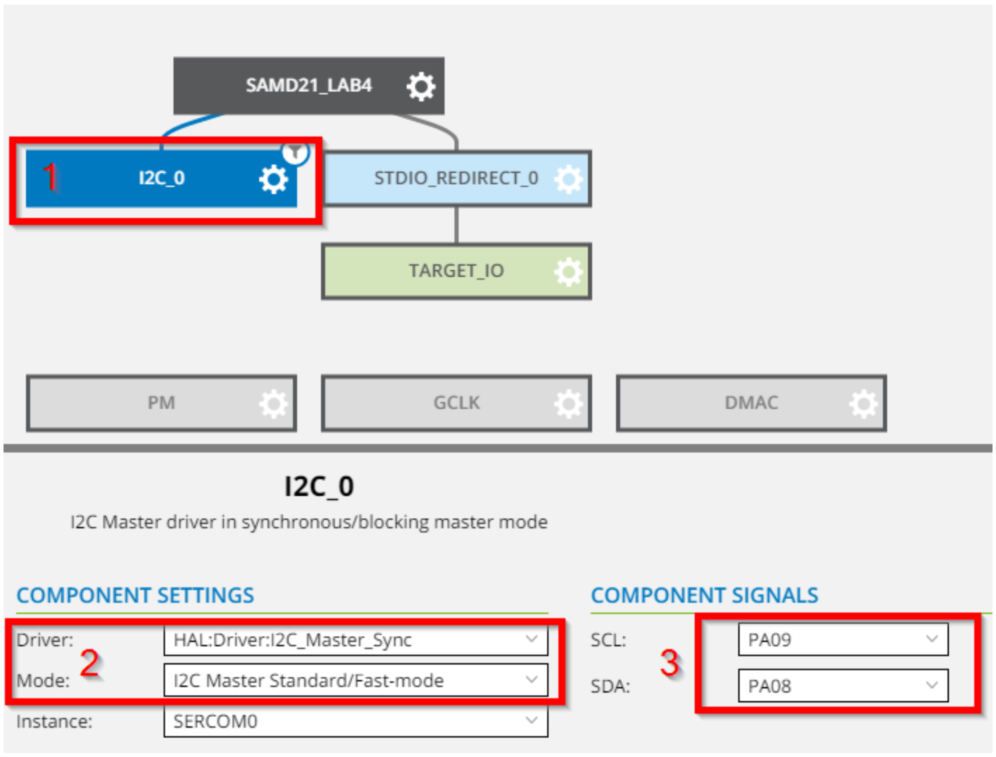


**Table 4-1. Extension Header EXT1**

| Pin on EXT1 | SAM D21 pin | Function | Shared functionality |
|---|---|---|---|
| 1 [ID] | - | - | Communication line to ID chip on extension board. |
| 2 [GND] | - | - | GND |
| 3 [ADC(+)] | PB00 | AIN[8] | |
| 4 [ADC(-)] | PB01 | AIN[9] | |
| 5 [GPIO1] | PB06 | GPIO | |
| 6 [GPIO2] | PB07 | GPIO | |
| 7 [PWM(+)] | PB02 | TC6/WO[0] | |
| 8 [PWM(-)] | PB03 | TC6/WO[1] | |
| 9 [IRQ/GPIO] | PB04 | EXTINT[4] | |
| 10 [SPI_SS_B/GPIO] | PB05 | GPIO | |
| 11 [TWI_SDA] | PA08 | SERCOM2 PAD[0] I$^2$C SDA | EXT2, EXT3, and EDBG |
| 12 [TWI_SCL] | PA09 | SERCOM2 PAD[1] I$^2$C SCL | EXT2, EXT3, and EDBG |
| 13 [USART_RX] | PB09 | SERCOM4 PAD[1] UART RX(1) | |
| 14 [USART_TX] | PB08 | SERCOM4 PAD[0] UART TX(1) | |
| 15 [SPI_SS_A] | PA05 | SERCOM0 PAD[1] SPI SS | |
| 16 [SPI_MOSI] | PA06 | SERCOM0 PAD[2] SPI MOSI | |
| 17 [SPI_MISO] | PA04 | SERCOM0 PAD[0] SPI MISO | |
| 18 [SPI_SCK] | PA07 | SERCOM0 PAD[3] SPI SCK | |
| 19 [GND] | - | - | GND |
| 20 [VCC] | - | - | VCC |

Therefore, we configure I2C as following:



I2C_0

I2C Master driver in synchronous/blocking master mode

COMPONENT SETTINGS

Driver: HAL:Driver:I2C_Master_Sync

Mode: I2C Master Standard/Fast-mode

Instance: SERCOM0

COMPONENT SIGNALS

SCL: PA09

SDA: PA08

For other configurations, we will keep them as they are.

Generate the project.

## Writing the code to wake up the secure element

Compile the project to check if the generation process causes no errors.

If errors happen, please start Atmel Studio and re-generate the project again.

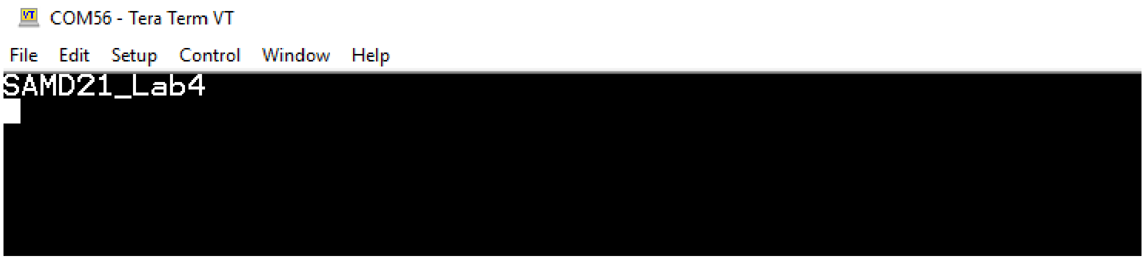
In the main.c file, we will add a `hex_dump(args)` function, which prints hex values to the terminal. In main.c, before the `main()` function, add the following code snippet:

```
void hex_dump(uint8_t * buff, uint32_t size)
{
        uint16_t i = 0;
        uint8_t line_count = 0;
        for(;i < size; i++) {
                printf("0x%02x, ",buff[i]);
                line_count++;
                if(line_count == 8) {
                        printf("\r\n");
                        line_count = 0;
                }
        }
        printf("\r\n");
}
```

In the main() function, below atmel_start_init() line, add the following line:

```
printf("SAMD21_Lab4");
```

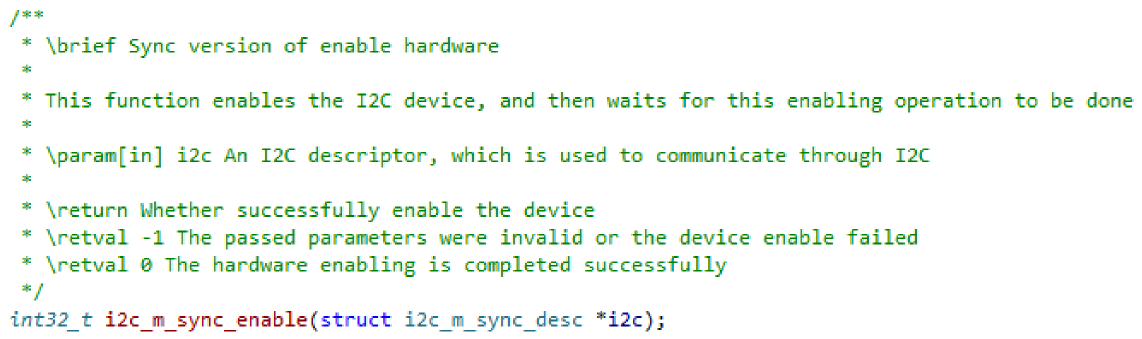
Compile and run the application, you should get the result on the terminal console as following:



```
SAMD21_Lab4
```

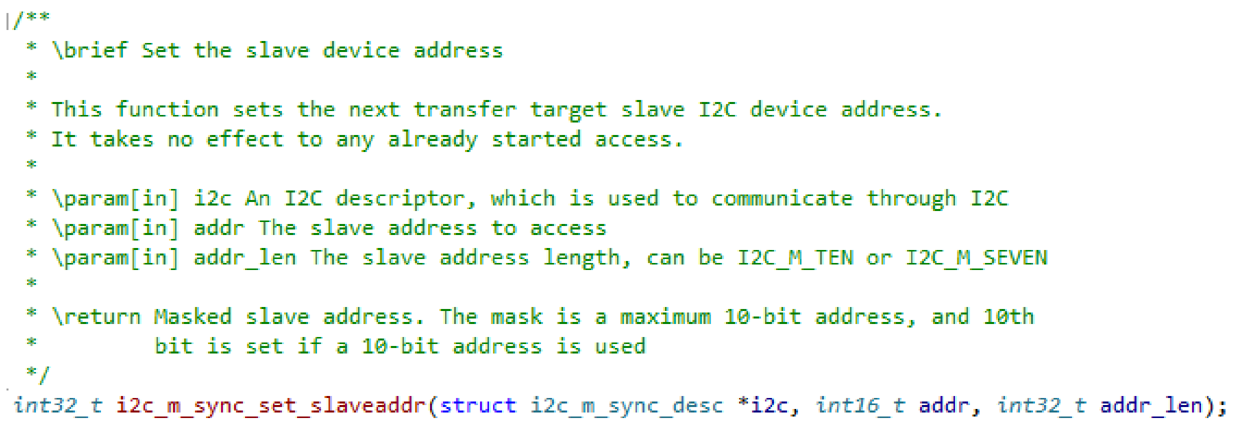
Next step, we will wake the secure element up by sending 0x00 byte to the device and read back the response values which are 0x04, 0x11, 0x33, 0x43. To understand which APIs we must implement, let take a look at "hal/include/hal_i2c_m_sync.h" file. In hal_i2c_m_sync.h file, the three functions, which are interesting for us are: i2c_m_sync_set_slaveaddr(), i2c_m_sync_enable() and i2c_m_sync_transfer().

i2c_m_sync_enable() will enable the selected I2C bus.



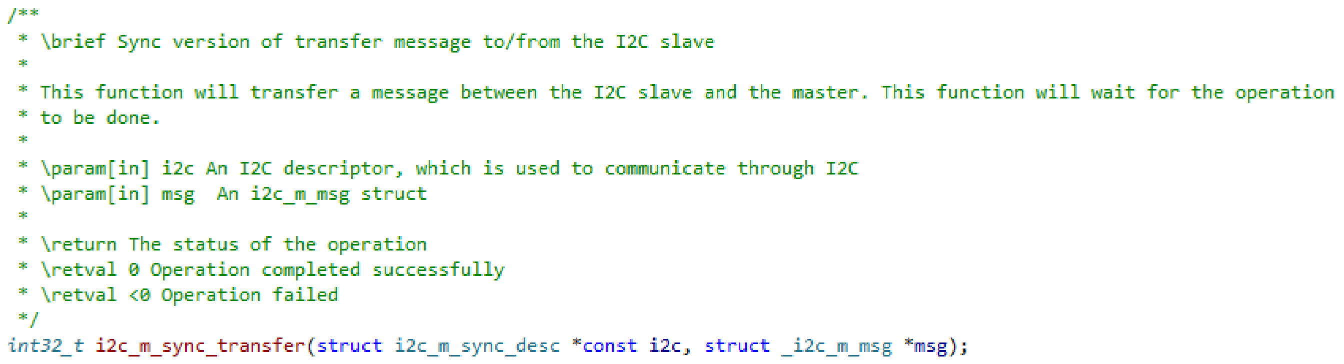
```c
/**
 * \brief Sync version of enable hardware
 *
 * This function enables the I2C device, and then waits for this enabling operation to be done
 *
 * \param[in] i2c An I2C descriptor, which is used to communicate through I2C
 *
 * \return Whether successfully enable the device
 * \retval -1 The passed parameters were invalid or the device enable failed
 * \retval 0 The hardware enabling is completed successfully
 */
int32_t i2c_m_sync_enable(struct i2c_m_sync_desc *i2c);
```

i2c_m_sync_set_slaveaddr() will set the address, whose device we want to communicate.



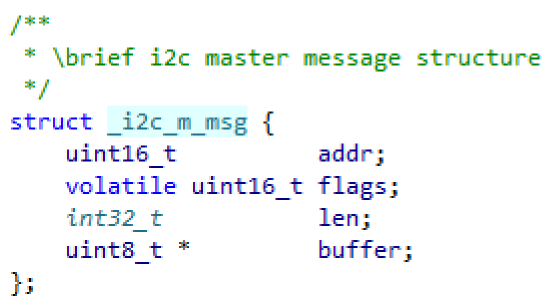
```c
/**
 * \brief Set the slave device address
 *
 * This function sets the next transfer target slave I2C device address.
 * It takes no effect to any already started access.
 *
 * \param[in] i2c An I2C descriptor, which is used to communicate through I2C
 * \param[in] addr The slave address to access
 * \param[in] addr_len The slave address length, can be I2C_M_TEN or I2C_M_SEVEN
 *
 * \return Masked slave address. The mask is a maximum 10-bit address, and 10th
 *         bit is set if a 10-bit address is used
 */
int32_t i2c_m_sync_set_slaveaddr(struct i2c_m_sync_desc *i2c, int16_t addr, int32_t addr_len);
```

i2c_m_sync_enable() will transfer or receive the message on the i2c bus



```c
/**
 * \brief Sync version of transfer message to/from the I2C slave
 *
 * This function will transfer a message between the I2C slave and the master. This function will wait for the operation
 * to be done.
 *
 * \param[in] i2c An I2C descriptor, which is used to communicate through I2C
 * \param[in] msg  An i2c_m_msg struct
 *
 * \return The status of the operation
 * \retval 0 Operation completed successfully
 * \retval <0 Operation failed
 */
int32_t i2c_m_sync_transfer(struct i2c_m_sync_desc *const i2c, struct _i2c_m_msg *msg);
```

The i2c_m_msg has the following format:



```
/**
 * \brief i2c master message structure
 */
struct _i2c_m_msg {
    uint16_t          addr;
    volatile uint16_t flags;
    int32_t           len;
    uint8_t *         buffer;
};
```

Go back to main.c, in the `main()` function, add the following code to initialize buffer for transceiving data, enable I2C bus, and set the address of the secure element, which is 0x60

```
uint8_t i2c_data[4] = {0};

i2c_m_sync_enable(&I2C_0);
i2c_m_sync_set_slaveaddr(&I2C_0, 0x60, I2C_M_SEVEN);
```

According to the datasheet of the ATECC508, to wake it up, we must pull the SDA line low for a certain time, which equals approximately 1 byte. After that, we wait for 800 microseconds before reading the data on the SDA line. If we receive the four following bytes 0x04 0x11 0x33 0x43, the secure element is woken up successfully and it is ready to work.

Firstly, we will send 0x00 byte. Add the following snippet:

```
// send the wake by writing to an address of 0x00
struct _i2c_m_msg packet = {
        .addr   = 0x00,
        .len    = 0,
        .buffer = NULL,
        .flags  = I2C_M_SEVEN | I2C_M_STOP,
};

// Send the 00 address as the wake pulse; part will NACK, so don't check for status
i2c_m_sync_transfer(&I2C_0, &packet);

delay_us(800);
```

After that, we read back the data on SDA line with the following snippet

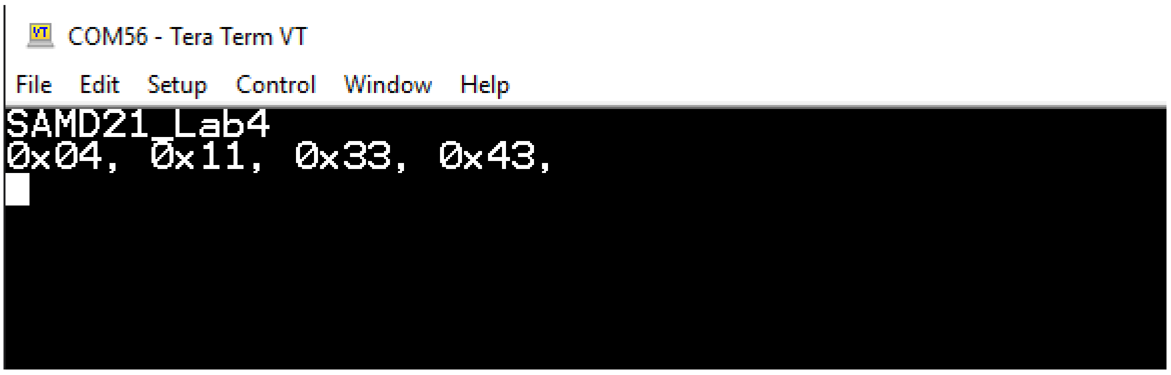
```
// receive the wake up response
packet.addr = 0x60;
packet.len = 4;
packet.buffer = i2c_data;
packet.flags  = I2C_M_SEVEN | I2C_M_RD | I2C_M_STOP;

i2c_m_sync_transfer(&I2C_0, &packet);

hex_dump(i2c_data, 4);

/* Replace with your application code */
while (1) {
}
```

Compile and Run the application. This is the result on the terminal:



```
SAMD21_Lab4
0x04, 0x11, 0x33, 0x43,
```

Congratulation, you have finished lab 5

## Contact information

Quang Hai Nguyen, B. Eng.
Field Application Engineer

P  +49 6102 50308228
M  +49 1511 6242003
quanghai.nguyen@arrow.com

Arrow Central Europe GmbH
Frankfurter Straße, 211
63263 Neu-Isenburg

**THE END**