I^2C

Inter-Integrated Circuit Communication

Luca Marsilio, Giulia Alessandrelli

Electronic Technologies and Biosensors Laboratory

A.A. 2021/2022 – II Semester

Outline

1. **Introduction to I^2C**
2. Hardware Configuration
3. General Operation
4. I^2C projects with PSoC

What is I²C?

I²C (or I2C or IIC) – **Inter Integrated Circuit** – is a serial communication BUS

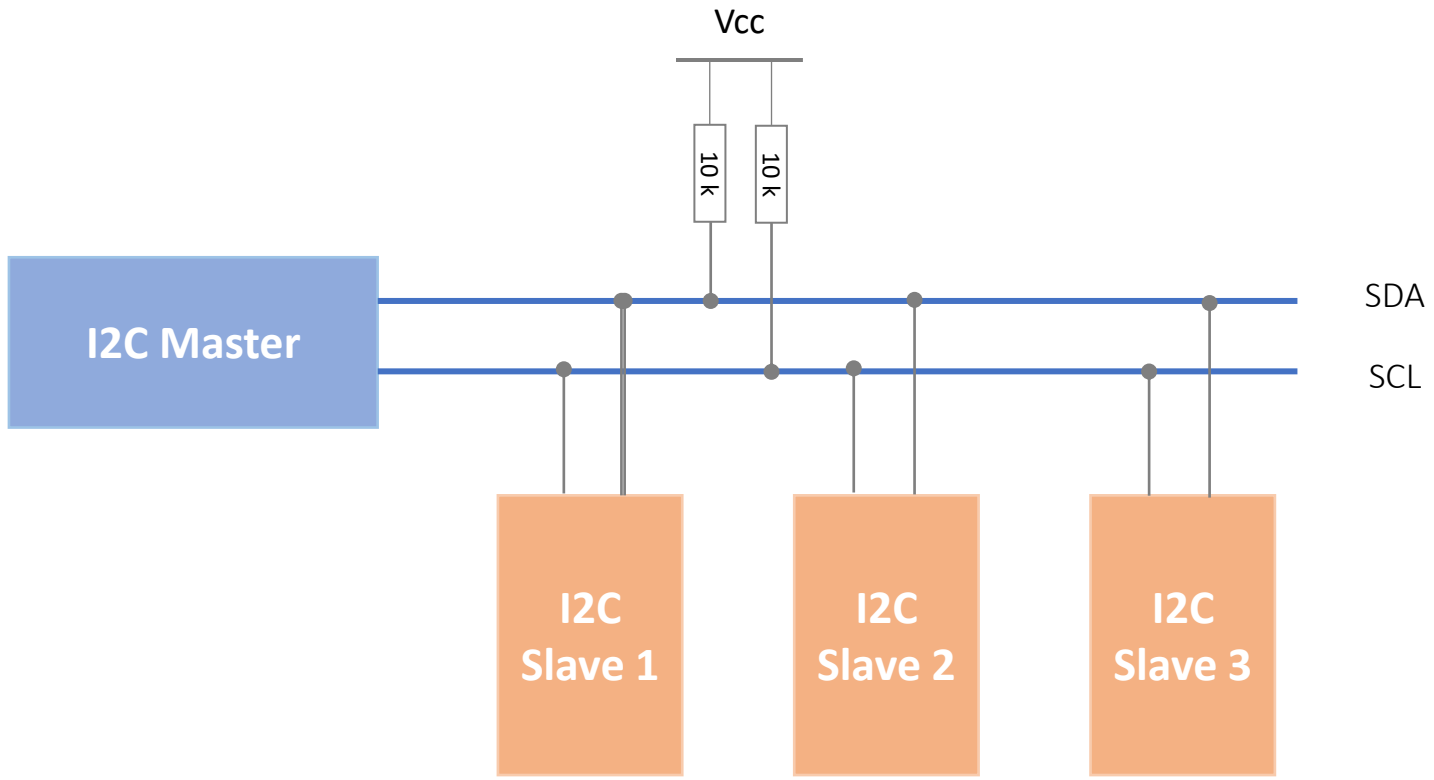
- Very popular and powerful 2-wire communication BUS for ICs
- Developed by Philips Semiconductors in 1982
- Synchronous, multi-master multi-slave, serial communication BUS

I2C is widely used to attach (low speed) peripherals to processors/microcontrollers in the case of short-distance (intra-board) communication

What is I²C?

I²C (or I2C or IIC) – **Inter Integrated Circuit** – is a serial communication interface with a bidirectional two-wire synchronous serial bus (SDA, SCL) and pull-up resistors

- 8 bit communication
- Speeds: 100kbps, 400kbps and 3.4Mbps
- Master and slave concept
- Two IOs – SDA and SCL
- 7-bit slave addressing



I^2C Basics

- Each signal line in I2C contains pull-up resistors to restore the signal to a high of the wire when no device is pulling it low.

- All transfers are initiated and terminated by the master(s).
- The master can write data to one (or more) slave or request data from the slave. Any device can be used as master or slave as long as it is configured with appropriate hardware or firmware.

- The data is transmitted in one byte, and each byte is followed by a 1-bit "handshake signal" as the ACK/NACK bit (acknowledgement/no response).

Why use I^2C?

- It is currently a common communication peripheral used by various circuits
- Multi-master bus
- Simple to implement: only two signal lines are needed (SCL, SDA) regardless the number of devices involved
- Flexible: it allows to communicate with slow devices while also having high speed modes to transmit large data

Serial Communication Protocols

Parameter	UART	I2C	SPI
Complexity	*	**	***
Max data rate	921.6 kbps	1000 kbps	**> 10 Mbps**
# Devices	2	127 (1)	/
# Wires	2	2	3 + N (2)
Duplex	**Full**	Half	**Full**
Master:Slave	1:1	**Multi:Multi**	1:Multi

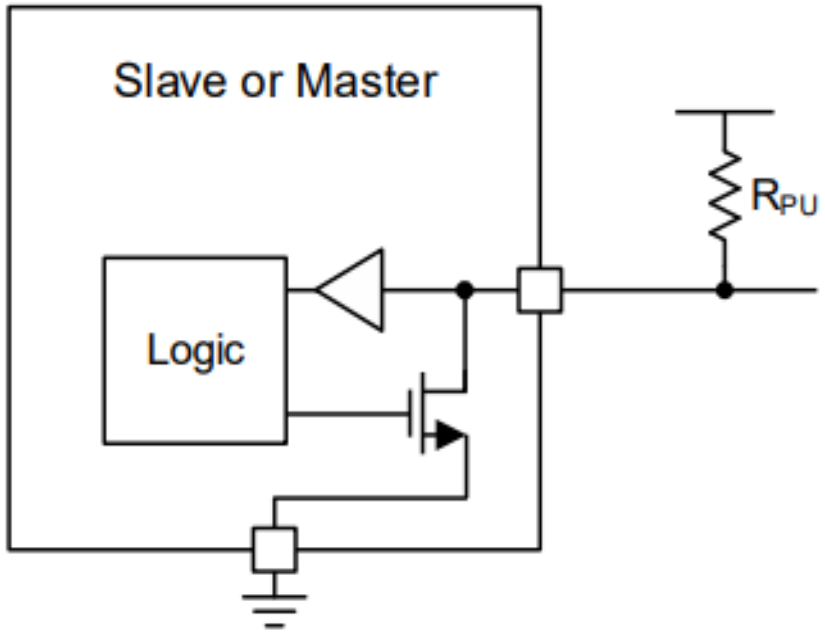
(1) Up to 127 *unique* devices
(2) N = number of connected devices (1 *Chip Select* line per each device)

Outline

1. Introduction to I^2C
2. **Hardware Configuration**
3. General Operation
4. I^2C projects with PSoC

Hardware Configuration

I²C uses an open-drain/open-collector with an input buffer on the same line, which allows a single data line to be used for half-duplex, **bidirectional data flow**.

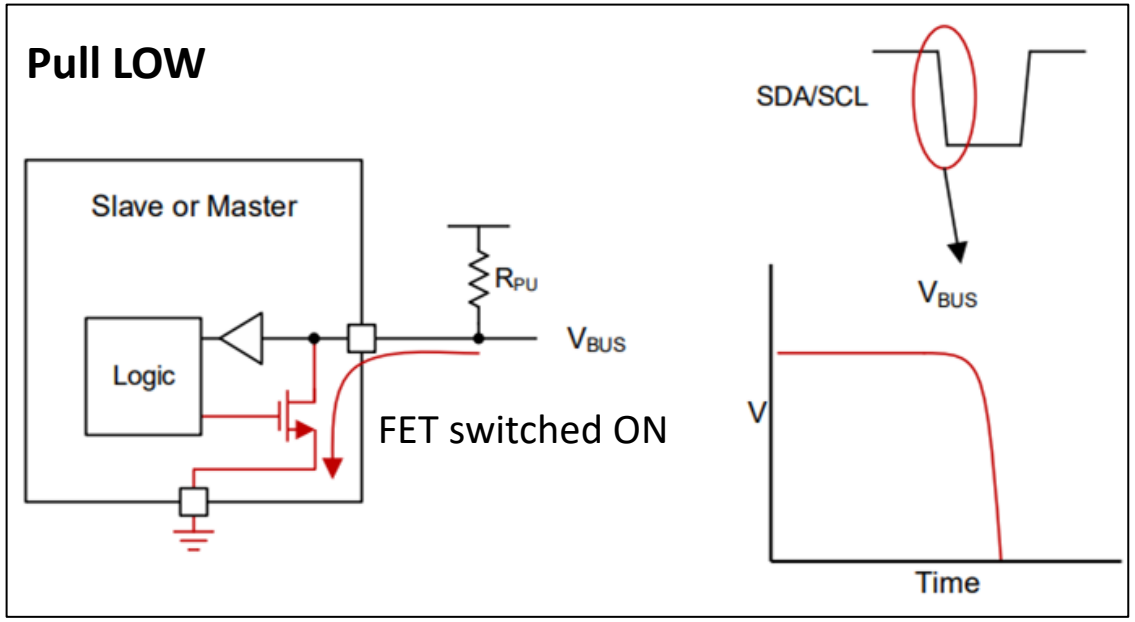


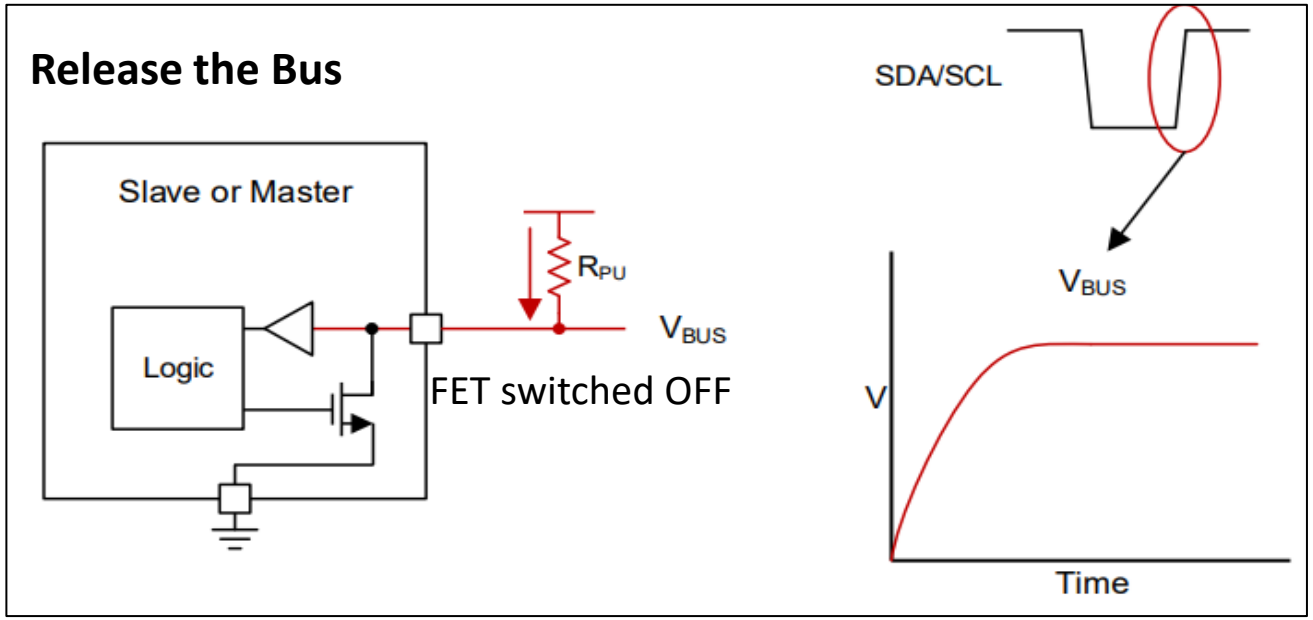
Open-drain: output which can either pull the BUS down to a voltage, or *release* the bus and let it be pulled high by a pull-up resistor.

Open Drain

SDA/SCL lines are pulled HIGH by the pull up resistor when not in use (*released*). The device controlling the line (master/slave) can pull down the voltage when needed.

- To transmit a logic 1 (HIGH), I2C devices do not need to drive HIGH the BUS thanks to the Open Drain configuration
- To transmit a logic 0 (LOW), I2C devices can pull down the voltage on the BUS





Pull Up Resistors

Pull Up resistors (RP) are a fundamental part of the I2C BUS: their value is **critical** to assure the **correct operation** of the BUS, in terms of signal integrity, power and speed.

$$RP_{min} < RP < RP_{max}$$

RP_{min}: assure I2C devices can pull down the BUS voltage (signal integrity, power limit)

RP_{max}: maximum value imposed by the BUS capacitance (speed/rise-time)

Pull Up Resistors | RP_{min}

$$RP_{min} = \frac{V_{CC} - V_{OL,max}}{I_{OL}}$$

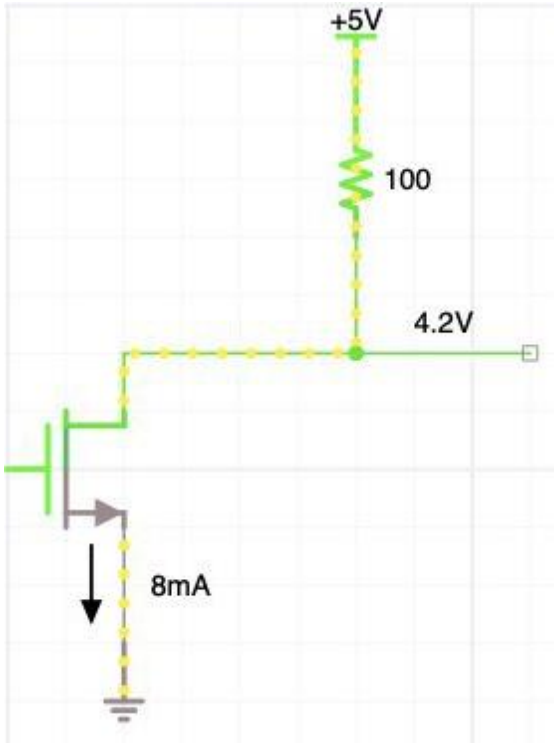
V_{CC} : BUS voltage

$V_{OL,max}$: upper value for a signal to be read as logic low

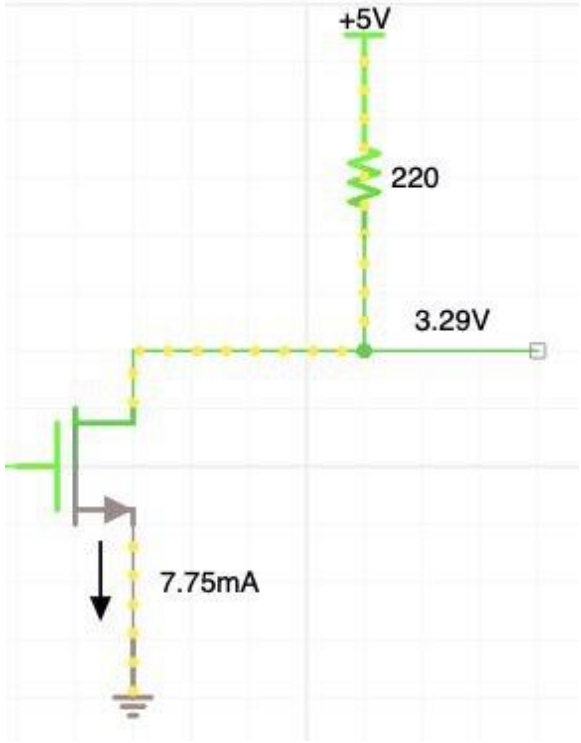
I_{OL} : current limit of the driving FET

Strong pull up (small RP): very high current (high power consumption). If $I > I_{OL}$, the driving FET may not be able to pull down the voltage on the BUS, resulting in a signal loss.

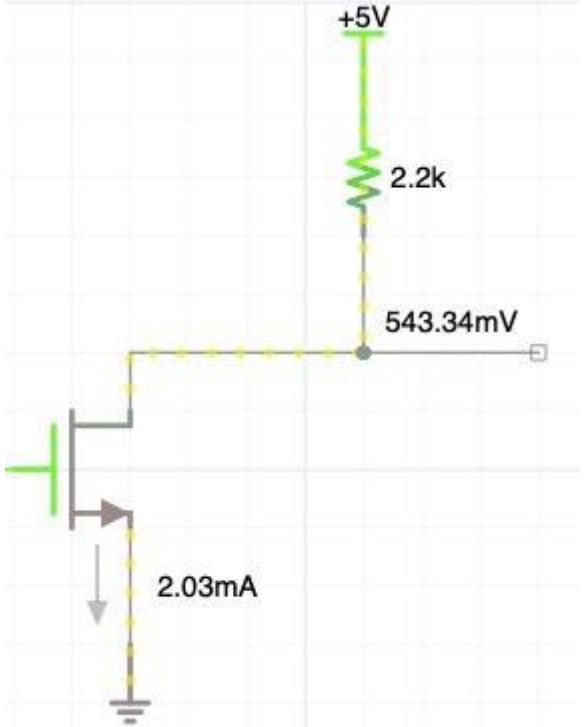
Pull Up Resistors | Example



$RP = 100\ \Omega$
Current $\gg$ current limit
Very small voltage drop across RP



$RP = 220\ \Omega$
Current $<$ current limit
Voltage drop across RP yet too small



$RP = 2.2\ \mathrm{k}\Omega$
Current $<$ current limit
The line is now pulled LOW

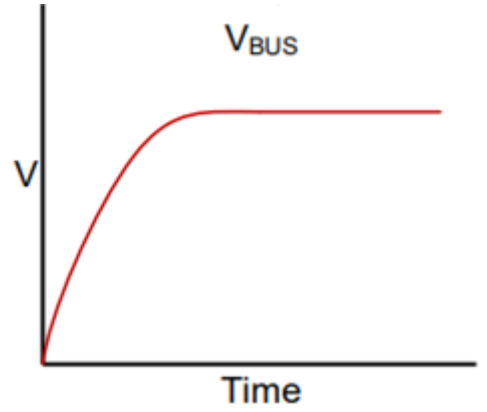
Pull Up Resistors | RP_{max}

$$RP_{max} = \frac{t_r}{0.8473 \cdot C_b}$$

t_r: required rise time (defined by the I2C standard)

C_b: capacitance of the BUS

Increasing the value of RP will improve signal integrity and reduce power consumption, at the cost of an increased rise time. It must be $RP < RP_{max}$ to make sure $t_r < t_{r,max}$



Outline

1. Introduction to I^2C
2. Hardware Configuration
3. **General Operation**
4. I^2C projects with PSoC

General Operation

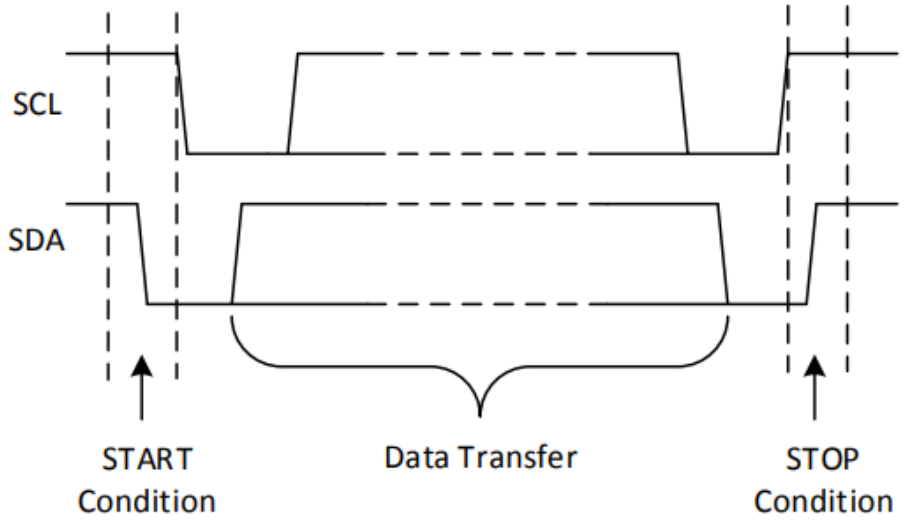
I2C is a multi-master multi-slave BUS. The master(s) controls the BUS to communicate with the slave devices.

- I2C provides an *addressable BUS*: the slave devices may transmit data only when addressed by the master (no Chip Select required)
- **7-bit address map** (0x00 – 0x7F): up to 127 slaves on a single I2C BUS.
- Only 2 wires to interface with peripherals: clock (**SCL**) and data (**SDA**)

General Operation | Start Condition

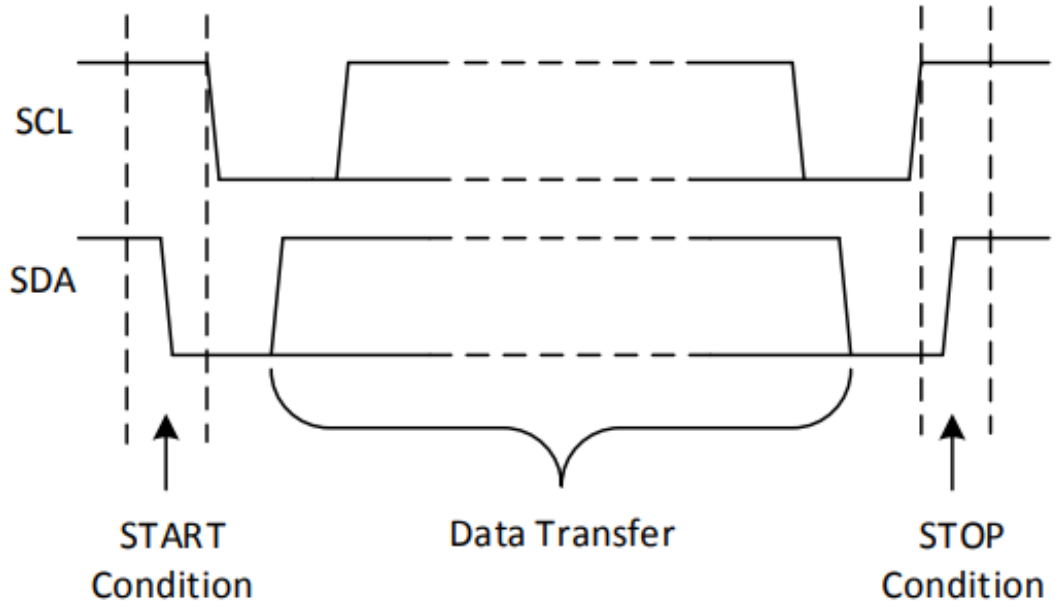
The data transfer on the BUS may be initiated only when the BUS is ***idle***, i.e., when both SDA and SCL are HIGH (pulled up by the resistors).

The **START condition** requires to pull down the SDA line (high-to-low transition) while SCL is HIGH.



General Operation | Stop Condition

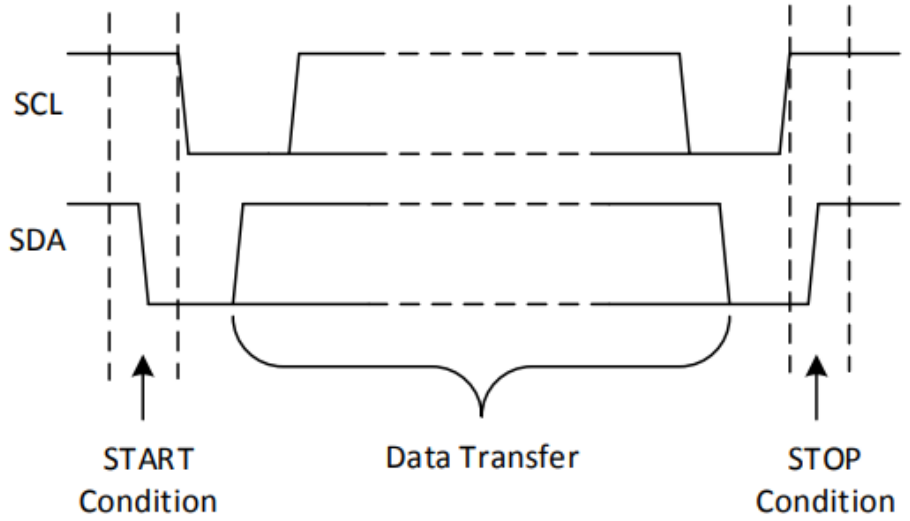
The **STOP condition** occurs when the SDA line is released (low-to-high transition, thanks to the pull-up resistor) while the clock line (SCL) is in the high state.



Data Transfer | Master Write

The I2C Master can send data to one slave *writing* on the BUS, following these steps:

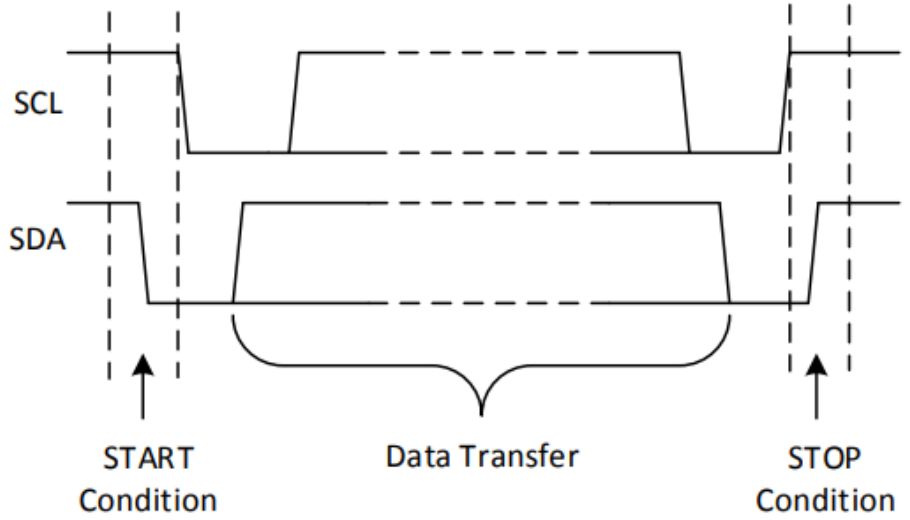
1. The master sends a START condition and addresses the slave with the **write** instruction
2. The master sends the data to the slave
3. The master ends the data transfer with the STOP condition



Data Transfer | Master Read

The I2C Master can request data from one slave *reading* on the BUS, following these steps:

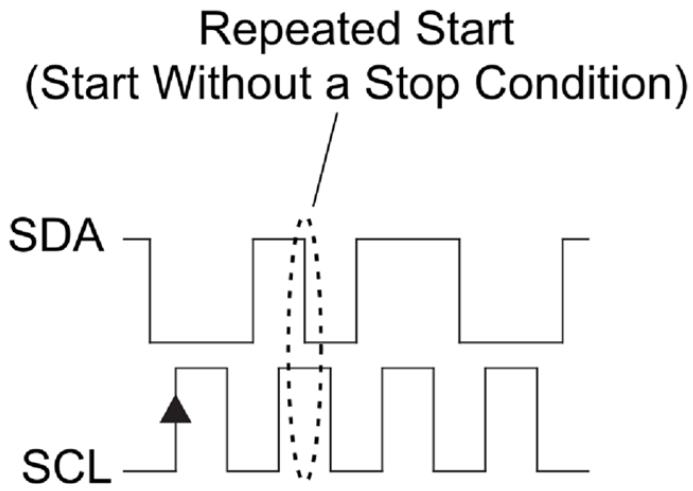
1. The master sends a START condition and addresses the slave with the **read** instruction
2. The master sends the requested register address to the slave device
3. The master reads from the BUS the data transmitted by the slave
4. The master ends the data transfer with the STOP condition



General Operation | Repeated Start

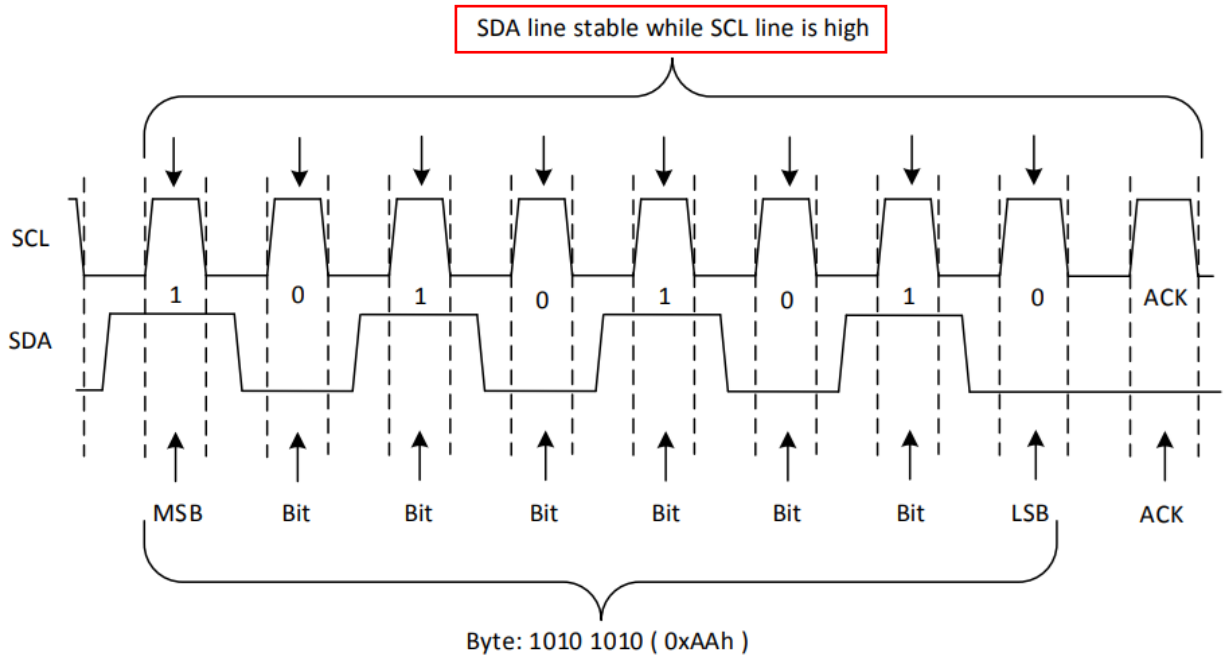
The **REPEATED START condition** may be used in place of a back-to-back STOP and START condition. It is identical to the START condition, although it happens without any STOP condition before it.

A repeated start may be useful for a new communication, avoiding the BUS to go idle in the meantime



General Operation | Data Format

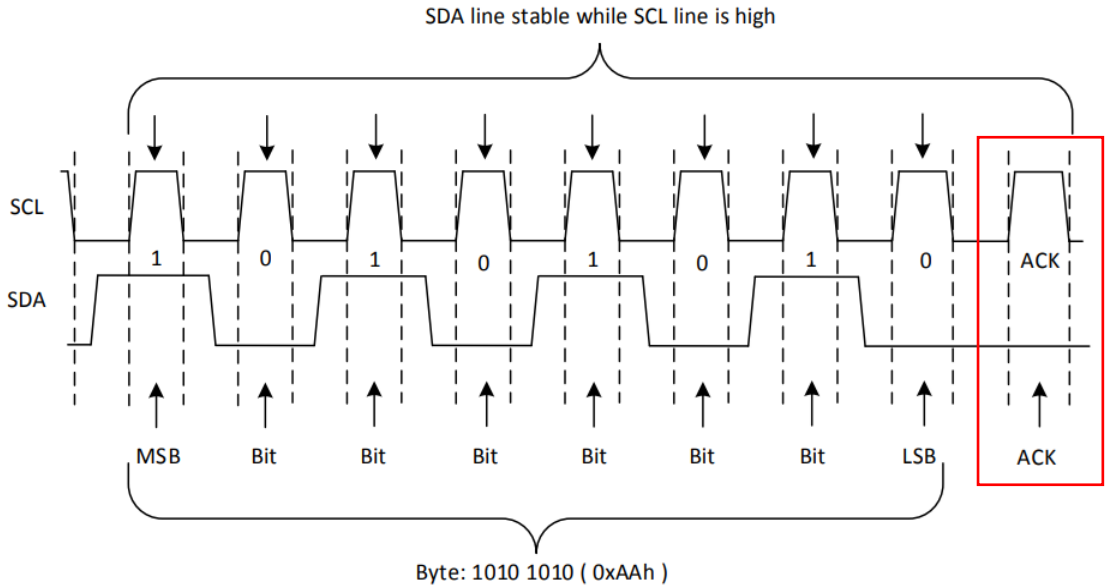
Each data bit on the SDA line is transferred during one clock pulse of the SCL line following the **MSB-first** scheme (MSB: Most Significant Byte)



Acknowledgement

Each data byte is followed by one **acknowledgement bit** sent from the receiver: this communicates to the transmitter that the byte was successfully received, and another byte may be transmitted.

The transmitter must release the SDA line, which is then pulled down by the receiver during the low phase of the ACK/NACK clock period.



ACK and NACK

Acknowledge (ACK): the SDA line is pulled down by the receiver, meaning that the communication was successful.

Not Acknowledge (NACK): the SDA line remains high during the acknowledgement clock pulse, meaning either that:

- the receiver is not able to communicate with the master
- during the transfer, the receiver gets data/commands that it does not understand
- during the transfer, the receiver cannot receive any more data bytes
- the master-receiver is done reading data and indicates the slave to stop sending.

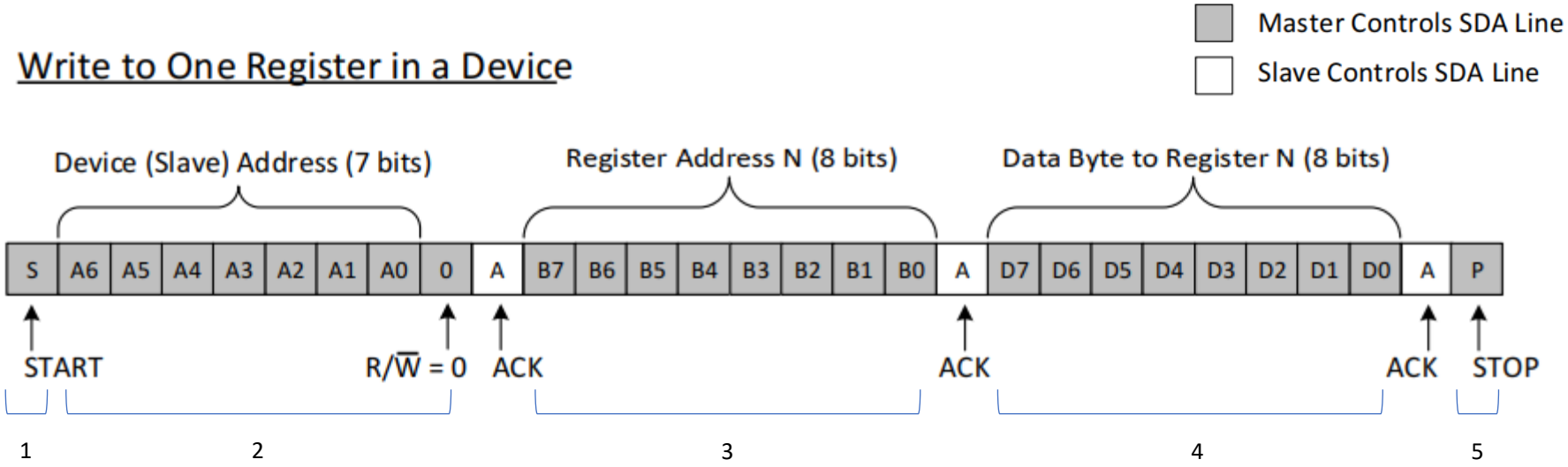
I^2C Data Transfer

The master can send or receive data from one slave writing/reading to/from one register of the slave, respectively.

A **register** is typically an 8-bit memory cell which contains information, either configuration parameters or data.

Each register in the memory of the slave is addressable via its unique address (8 or 16 bits, depending on the device). The master must transmit to the slave the address of the register it wants to read/write.

Write a Single Byte



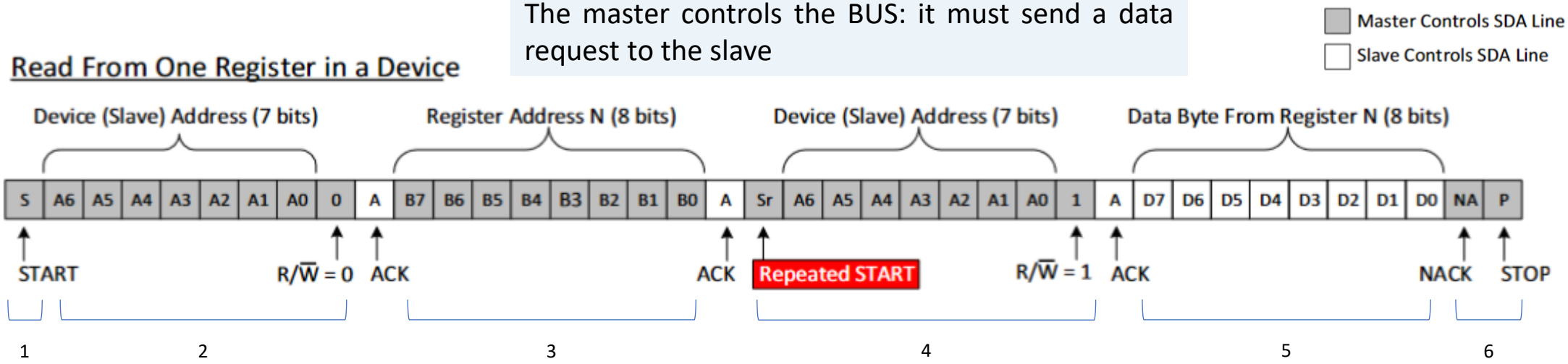
1. Start condition issued by the master
2. The master writes the 7-bit address of the slave + write instruction, i.e., R/W bit = 0

3. The master writes on the SDA line the memory address of the slave's register

4. The master clocks out the bits on the SDA line and the byte is written to the register of the slave, which then sends the ACK
5. The master issues the stop conditions

Read Single/Multiple Data Bytes

The master controls the BUS: it must send a data request to the slave



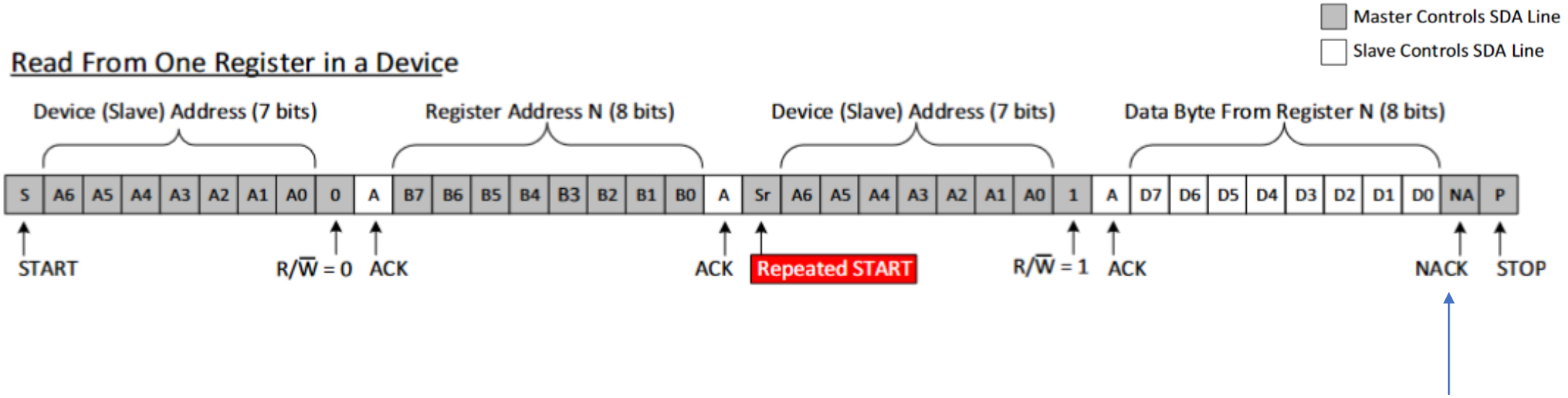
1. Start condition issued by the master
2. The master writes the 7-bit address of the slave + write instruction, i.e., R/W bit = 0

3. The master transmits on the SDA line the memory address of the slave's register to read from

4. The master sends a repeated start condition, and writes the 7-bit address of the slave + read instruction, i.e., R/W bit = 1

5. The slave clocks out the bits on the SDA line and the byte is read from the master, which then sends the NACK to stop the data transfer
6. The master issues the stop conditions to release the BUS

Read Single/Multiple Data Bytes



For continuous data transfer, the master here can send an ACK to request more data bytes from the slave. The slave register address is automatically increased by the device, so that the next register in memory is read.
Hence, to read N bytes we have N-1 ACKs and 1 final NACK.

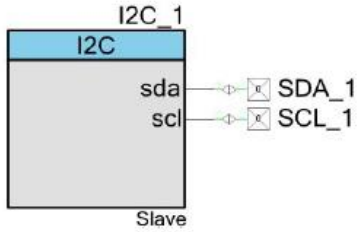
Outline

1. Introduction to I^2C
2. Hardware Configuration
3. General Operation
4. **I^2C projects with PSoC**

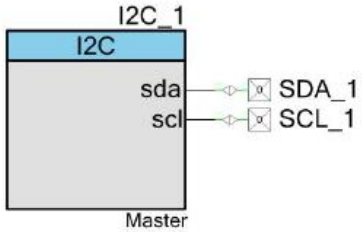
I²C on the PSoC

The I2C component supports master, slave, and multi-master configurations. The standard I2C speed of 100 kbps can be increased up to 400 kbps (Fast Mode) or 1000 kbps (Fast Mode Plus)

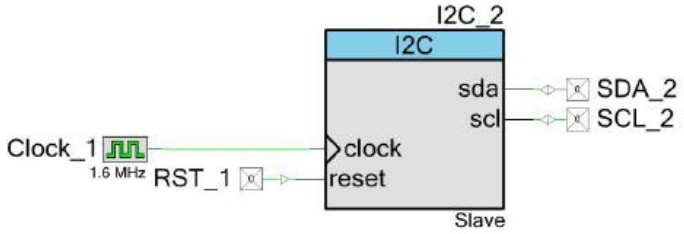
Fixed-Function I²C Slave with Pins



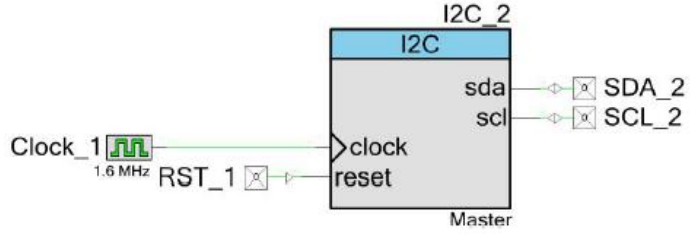
Fixed-Function I²C Master Pins



UDB I²C Slave with Clock and Pins



UDB I²C Master with Clock and Pins



Generic API for I²C

Function	Description
I2C_Start()	Initializes and enables the I²C component. The I²C interrupt is enabled, and the component can respond to I²C traffic.
I2C_Stop()	Stops responding to I²C traffic (disables the I²C interrupt).
I2C_EnableInt()	Enables interrupt, which is required for most I²C operations.
I2C_DisableInt()	Disables interrupt. The I2C_Stop() API does this automatically.
I2C_Sleep()	Stops I²C operation and saves I²C nonretention configuration registers (disables the interrupt). Prepares wake on address match operation if Wakeup from Sleep Mode is enabled (disables the I²C interrupt).
I2C_Wakeup()	Restores I²C nonretention configuration registers and enables I²C operation (enables the I²C interrupt).
I2C_Init()	Initializes I²C registers with initial values provided from the customizer.
I2C_Enable()	Activates I²C hardware and begins component operation.
I2C_SaveConfig()	Saves I²C nonretention configuration registers (disables the I²C interrupt).
I2C_RestoreConfig()	Restores I²C nonretention configuration registers saved by I2C_SaveConfig() or I2C_Sleep() (enables the I²C interrupt).

API – I²C Slave

Function	Description
I2C_SlaveStatus()	Returns the slave status flags.
I2C_SlaveClearReadStatus()	Returns the read status flags and clears the slave read status flags.
I2C_SlaveClearWriteStatus()	Returns the write status and clears the slave write status flags.
I2C_SlaveSetAddress()	Sets the slave address, a value between 0 and 127 (0x00 to 0x7F).
I2C_SlaveInitReadBuf()	Sets up the slave receive data buffer. (master <- slave)
I2C_SlaveInitWriteBuf()	Sets up the slave write buffer. (master -> slave)
I2C_SlaveGetReadBufSize()	Returns the number of bytes read by the master since the buffer was reset.
I2C_SlaveGetWriteBufSize()	Returns the number of bytes written by the master since the buffer was reset.
I2C_SlaveClearReadBuf()	Resets the read buffer counter to zero.
I2C_SlaveClearWriteBuf()	Resets the write buffer counter to zero.

API – I²C Master

Function	Description
I2C_MasterStatus()	Returns the master status.
I2C_MasterClearStatus()	Returns the master status and clears the status flags.
I2C_MasterWriteBuf()	Writes the referenced data buffer to a specified slave address.
I2C_MasterReadBuf()	Reads data from the specified slave address and places the data in the referenced buffer.
I2C_MasterSendStart()	Sends only a Start to the specific address.
I2C_MasterSendRestart()	Sends only a Restart to the specified address.
I2C_MasterSendStop()	Generates a Stop condition.
I2C_MasterWriteByte()	Writes a single byte. This is a manual command that should only be used with the I2C_MasterSendStart() or I2C_MasterSendRestart() functions.
I2C_MasterReadByte()	Reads a single byte. This is a manual command that should only be used with the I2C_MasterSendStart() or I2C_MasterSendRestart() functions.
I2C_MasterGetReadBufSize()	Returns the byte count of data read since the I2C_MasterClearReadBuf() function was called.
I2C_MasterGetWriteBufSize()	Returns the byte count of the data written since the I2C_MasterClearWriteBuf() function was called.
I2C_MasterClearReadBuf()	Resets the read buffer pointer back to the beginning of the buffer.
I2C_MasterClearWriteBuf()	Resets the write buffer pointer back to the beginning of the buffer.

I²C Master | Automatic Mode

Automatic Write

A buffer is created to hold the entire transfer. In the case of a write operation, the buffer is prefilled with the data to be sent. To write an array of bytes to a slave in automatic mode, use the following function:

```
uint8_t I2C_MasterWriteBuf(uint8_t slaveAddr, uint8_t* data, uint8_t count, uint8_t mode)
```

`slaveAddr`: right-justified, 7-bit address of the slave address
`data`: pointer to the data buffer to transmit
`count`: number of bytes to transfer
`mode`: I2C transfer mode (complete transfer, repeated start, or transfer without stop)

Automatic Read

A buffer of the proper size is allocated to hold the data to receive from the slave. In the automatic mode, the read operation is almost identical to the write operation, using the analogous function:

```
uint8_t I2C_MasterReadBuf(uint8_t slaveAddr, uint8_t* data, uint8_t count, uint8_t mode)
```

`slaveAddr`: right-justified, 7-bit address of the slave address
`data`: pointer to the data buffer to store the received data
`count`: number of bytes to read
`mode`: I2C transfer mode (complete transfer, repeated start, or transfer without stop)

Manual Write

In the manual mode, each operation of the I2C data transfer workflow has to be executed by the user with lower-level functions of the API.

```
status = I2C_MasterSendStart(slaveAddr, I2C_WRITE_XFER_MODE);
if( status == I2C_MSTR_NO_ERROR ) {
    /* Write 5 bytes */
    for( uint8_t i = 0; i < 5; i++ ) {
        status = I2C_MasterWriteByte(data[i]);
        /* Stop on error */
        if( status != I2C_MSTR_NO_ERROR ) break;
    }
}
```

Manual Read

In the manual mode, each operation of the I2C data transfer workflow has to be executed by the user with lower-level functions of the API. In this case, the user also has to control the acknowledgement.

```
status = I2C_MasterSendStart(slaveAddr, I2C_READ_XFER_MODE);
if( status == I2C_MSTR_NO_ERROR ) {
    /* Read 5 bytes */
    for( uint8_t ; i < 5; i++ ) {
        if( i < 4 ) data[i] = I2C_MasterReadByte(I2C_ACK_DATA);
        else        data[i] = I2C_MasterReadByte(I2C_NAK_DATA);
    }
}
```

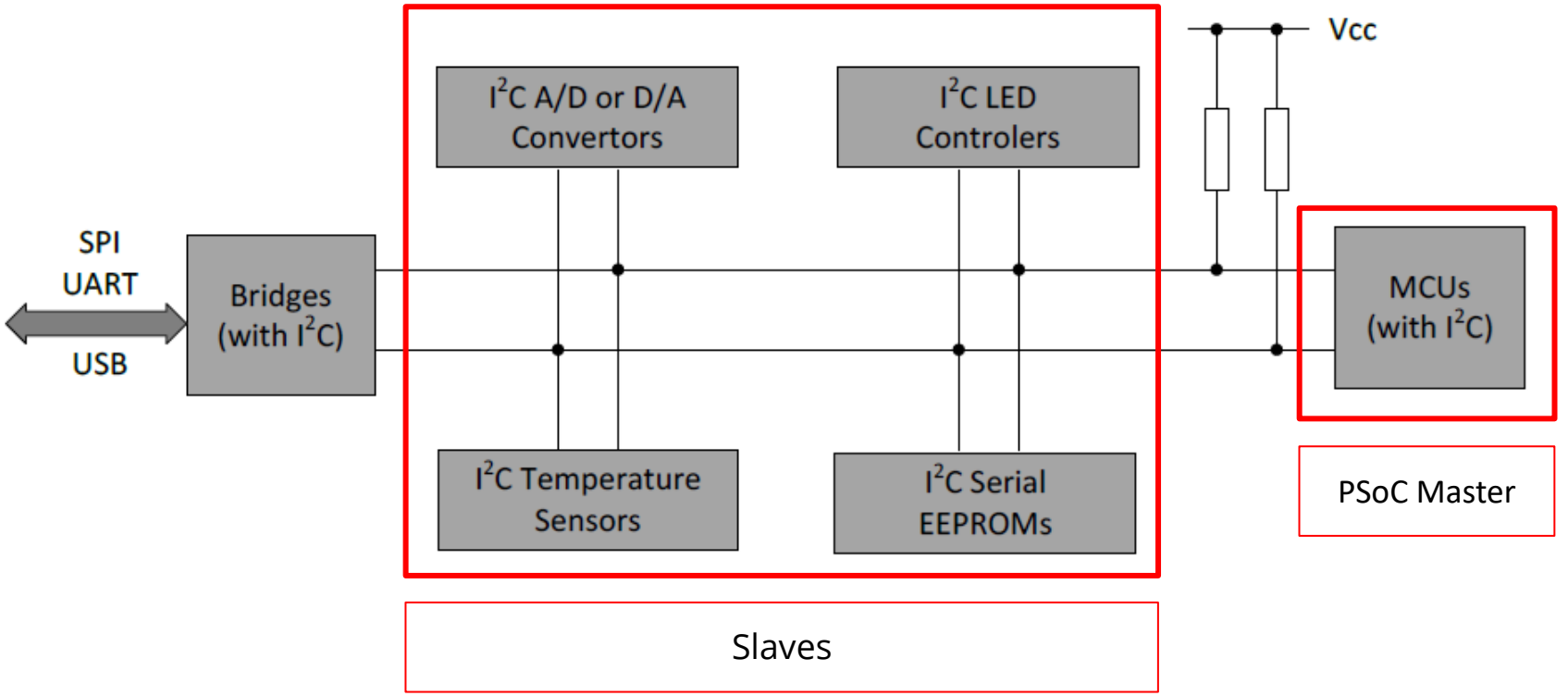
I^2C
Projects with PSoC Creator

Luca Marsilio

Electronic Technologies and Biosensors Laboratory

A.Y. 2021/2022 – II Semester

I²C Projects with the PSoC



Instructions

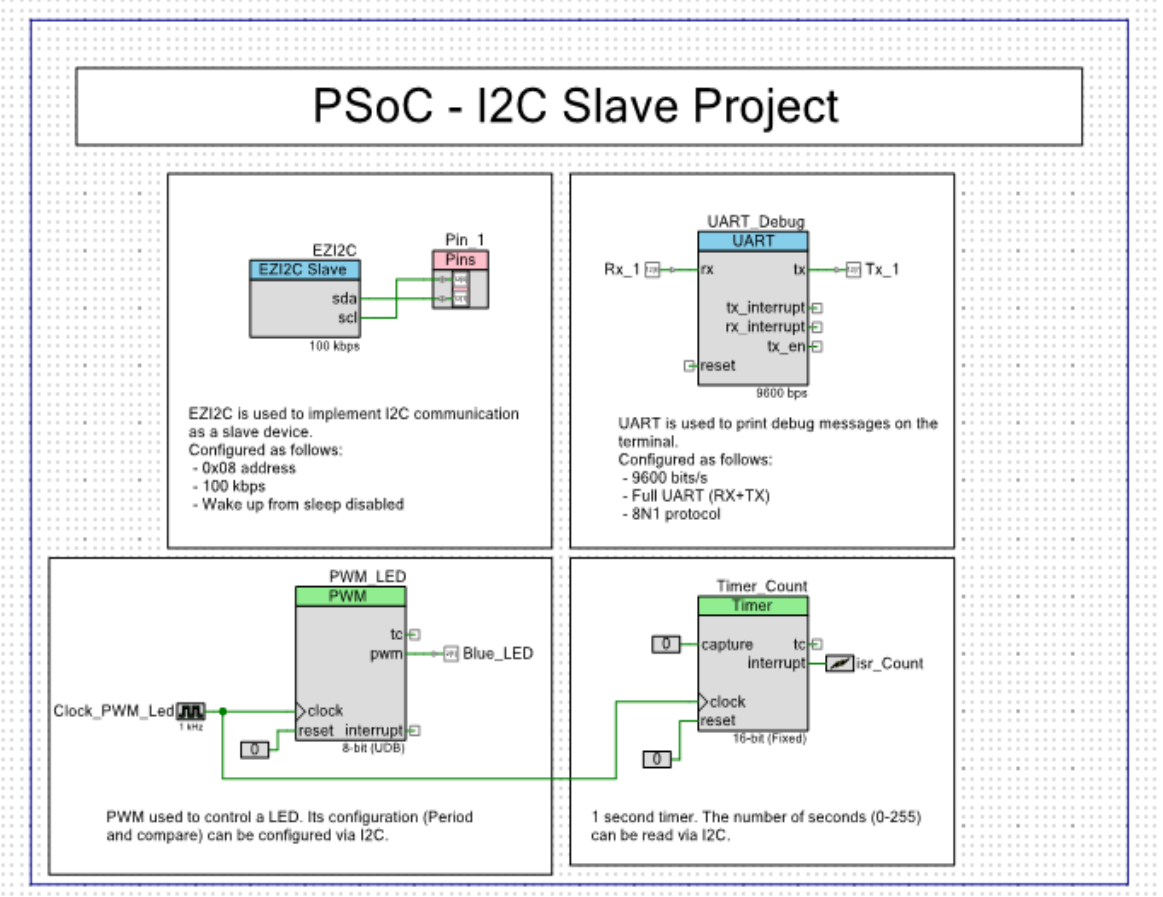
The workspace for today's projects is already available on GitHub

1. Clone this Repository from GitHub
2. You will have two branches: the `master` branch and the `solution` branch
3. You can work in the master branch or create a development branch
    - At the end of each project, the `solution` branch will be updated with the code for that task; you can either compare your work with the solution (using `git diff <your_branch>..solution`) or merge the solution into your active branch (master/dev)
    - Go back to your master/dev branch and work on the next task

Project 1: I²C Slave

In this project, we will configure the PSoC as an I2C slave device and interact with it.

1. Clone this Repository from GitHub
2. Build Project 1 and Program the PSoC
3. Verify that the on-board LED is blinking



https://github.com/LucaMarsilio/I2C_Class_06_04_22/tree/main

Project 1

The slave device is configured as follows:

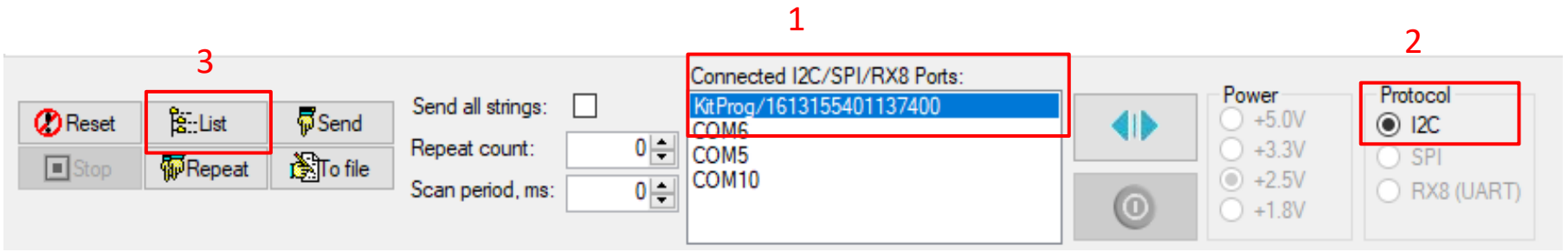
- I2C 7-bit address: 0x08
- Number of registers: 4

Register Address	Default Value	Read/Write	Description
0x00	0x7F	R/W	PWM Compare Value
0x01	0xFF	R/W	PWM Period Value
0x02	0x00	R	1-Second Timer Counter
0x03	0xBC	R	Who Am I

Project 1 | Bridge Control Panel

We can interact with our I2C slave from our PC using the KitProg (I2C master) and the Bridge Control Panel

1. Select the KitProg port of the slave device in the device list and connect to it
2. Select I2C as Protocol
3. Click on “List”



Project 1 | Bridge Control Panel

The List command will check the I2C devices that are present on the bus. You should see the following message on the output window:

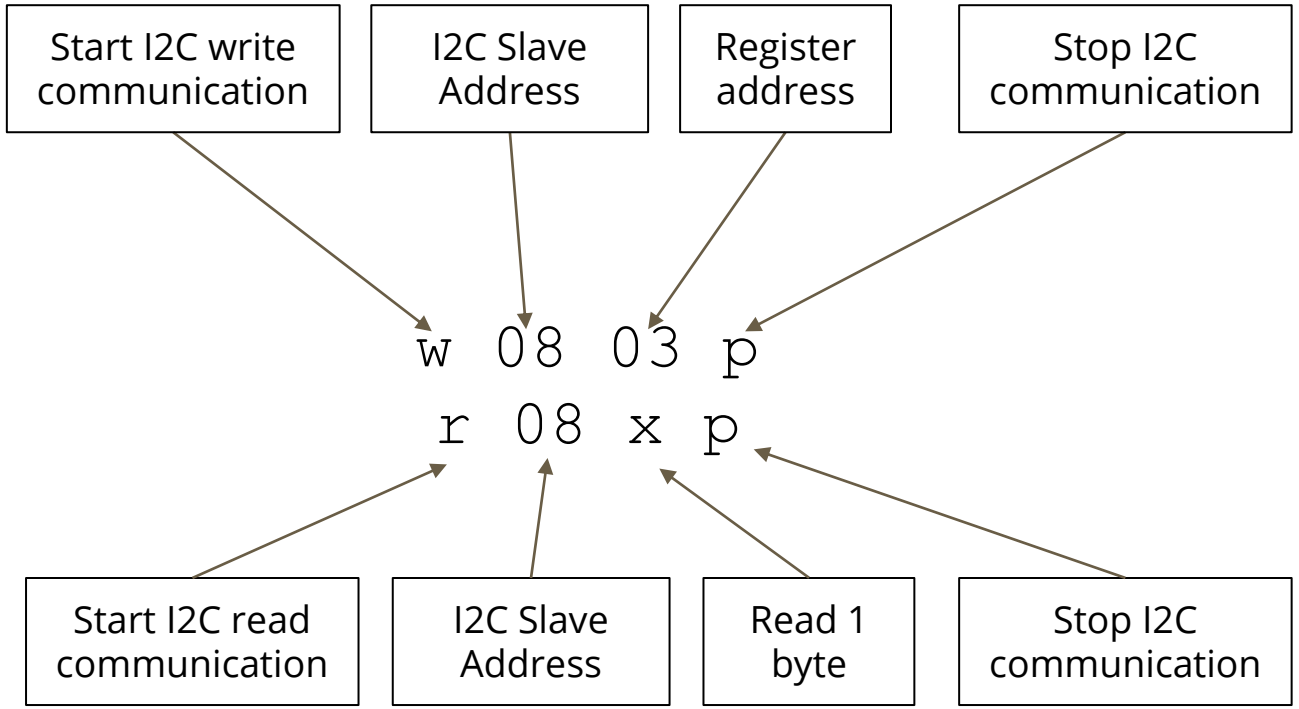
```
Opening Port
Successfully Connected to KitProg/161315401137400
KitProg Version 2.21
Devices list:   8bit   7bit
     address:   10     08
```

Address of the slave devices on the I2C bus

Project 1

Enter the following command in the Editor window (right click to insert a new row)

```
w 08 03 p
r 08 x p
```



Project 1

Select and run (pressing Enter) the two lines, you should receive the following:

```
w 08+ 03+ p
r 08+ BC+ p
```

Content of the register 0x03 of the slave @ 0x08

The + sign means that the slave acknowledged our requests. If we attempt to read a register that cannot be accessed (for example 0x05), we get a NACK from the slave:

```
w 08+ 05- p
```

Project 1

Select and run the following line

```
w 08 00 00 p
```

Register Address	Description
0x00	PWM Compare Value

You should receive :

```
w 08+ 00+ 00+ p
```

Now the on-board LED should be turned off.

Changing this register changes the compare value of our PWM, effectively changing the blinking frequency

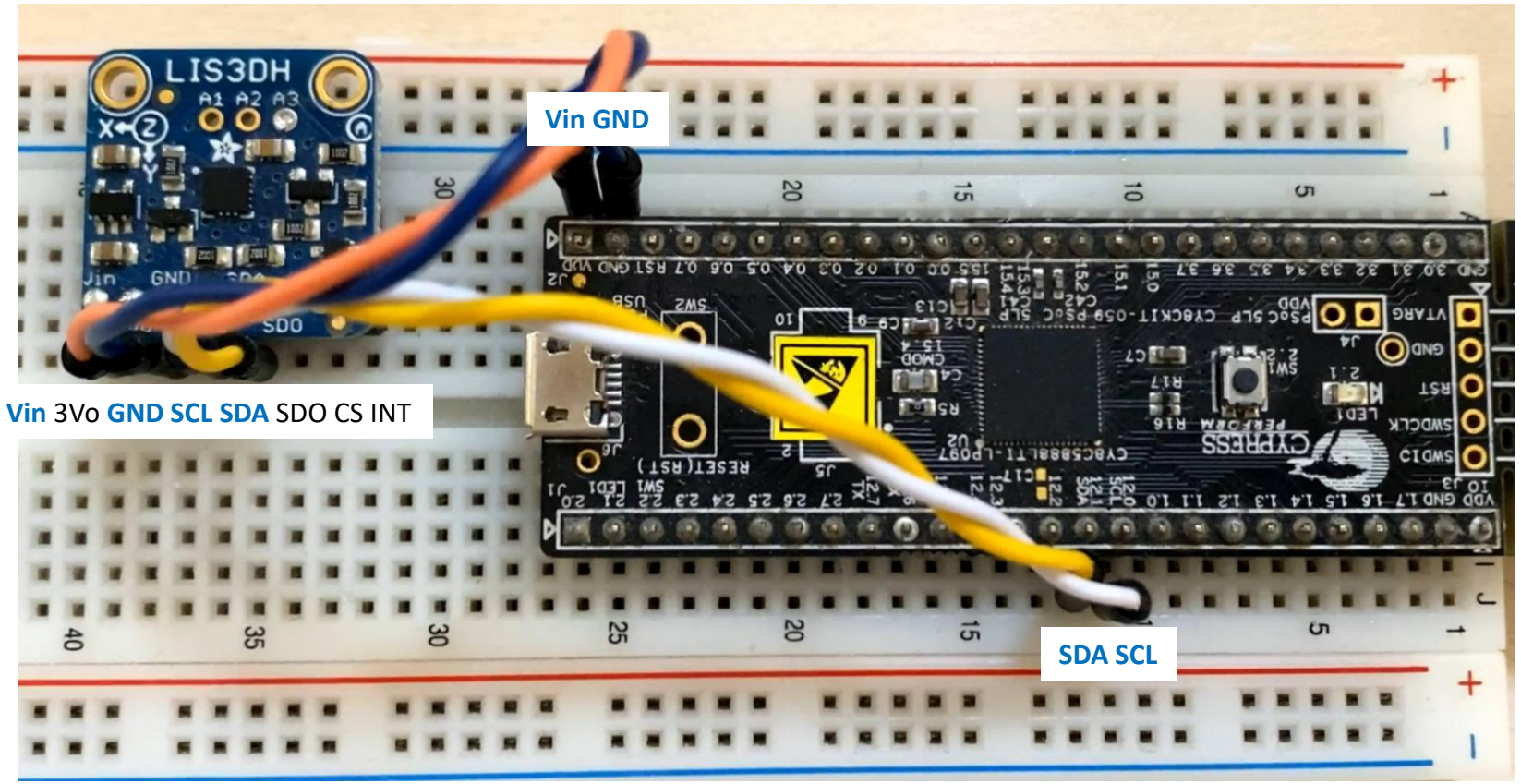
Project 2: I²C Master | Setup

Now, we need to connect the device to the I2C BUS of our PSoC.

1. Place the LIS3DH on the breadboard
2. Connect GND and Vin input pins to ground and power supply line
3. Connect the pins 12[0] and pin 12[1] to LIS3DH board's SCL and SDA respectively
4. ~~Pull-up these pins at 5V with 10k resistors~~ this is already done on the breakout board (thank you Adafruit!)

Project 2 | The LIS3DH Accelerometer

Let’s take a look at the **Adafruit LIS3DH** Triple Axis Accelerometer breakout board.



SA0 Pin

SDO	SPI serial data output (SDO)
SA0	I^2C less significant bit of the device address (SA0)

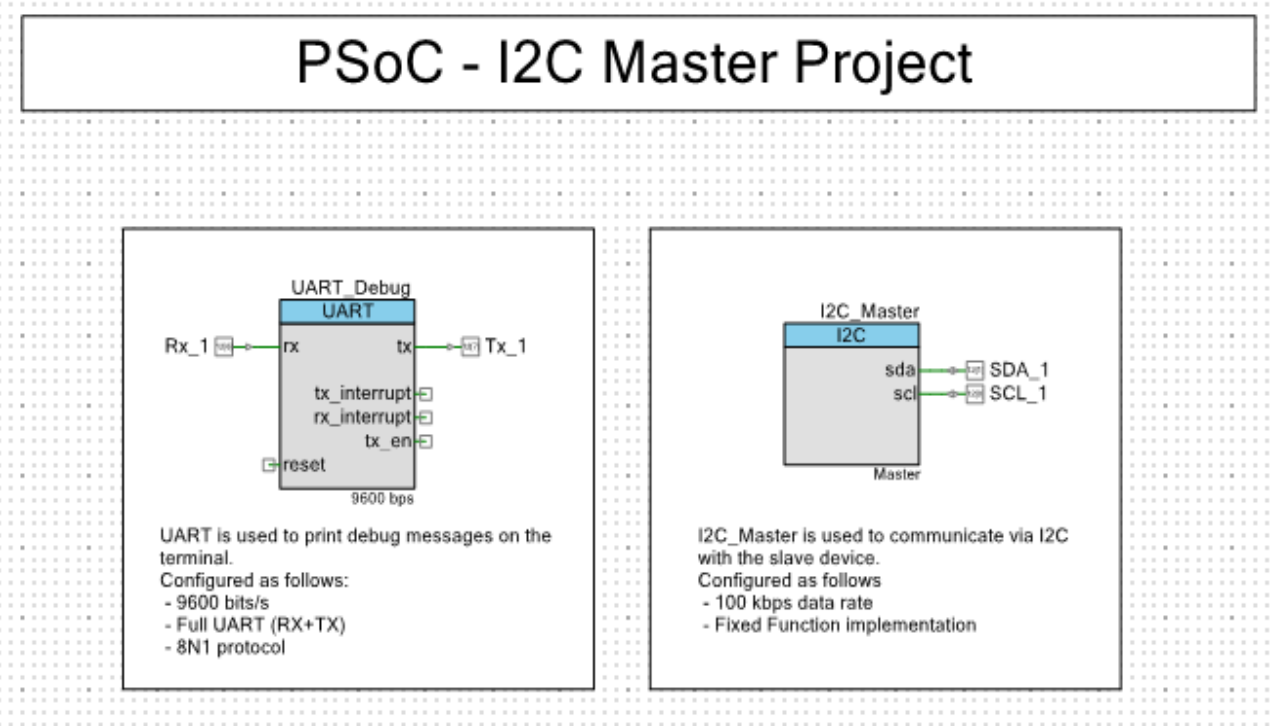
Device 0

x	**x**	**x**	**x**	**x**	**x**	**0**

Device 1

x	**x**	**x**	**x**	**x**	**x**	**1**

Set the project "*02-I2C_Master_Read*" as Active Project in the workspace



Project 2

Project goal: Communicate via I2C with the PSoC Slave device and perform the following tasks:

- Read the WHO AM I register of the LIS3DH and send its value over UART
- Read the STATUS_REG1 and CTRL_REG1 and send their values over UART
- Write into Control reg1 and check for non-volatile changes
- Read data from I2C and send their values over UART

Project 2

I2C_Interface.h - Function declarations:

Function	Description
I2C_Interface_Start()	Start the I2C interface.
I2C_Interface_Stop()	Stop the I2C interface
I2C_Interface_ReadRegister()	Read the value of a single register
I2C_Interface_ReadRegisterMulti()	Read the values of multiple registers
I2C_Interface_WriteRegister()	Write a new value to a single register
I2C_Interface_WriteRegisterMulti()	Write new values to multiple registers
I2C_Interface_IsDeviceConnected()	Check if a device is present on the I2C bus

Implement I2C Register Reading

***I2C_Interface.c* - Macros and function definitions.**

Start to implement the function `I2C_Interface_ReadRegister()`

This function takes the following parameters:

- `device_address`: address of the I2C device
- `register_address`: register that needs to be updated
- `*data`: address of the variable where the read data will be stored

And should return an ErrorCode (defined in *ErrorCodes.h*) as follows:

- `NO_ERROR` if no error occurred during I2C communication
- `ERROR` if some error occurred during I2C communication

Implement I2C Register Reading

***I2C_Interface.c* - Macros and function definitions.**

From the I2C component datasheet:

```
status = I2C_MasterSendStart(slaveAddr, I2C_READ_XFER_MODE);
if( I2C_MSTR_NO_ERROR == status ) {
    /* Read array of 5 bytes */
    for( int i = 0; i < 5; i++ ) {
        if ( i < 4) data[i] = I2C_MasterReadByte(I2C_ACK_DATA);
        else        data[i] = I2C_MasterReadByte(I2C_NAK_DATA);
    }
}
```

Read the WHO_AM_I register value

main.c - Main firmware for the application.
First task**:** read the register WHO AM I and send its value over UART
Include all the required header files in *main.c*

```
// Include required header files
#include "I2C_Interface.h"
#include "project.h"
#include "stdio.h"
#include "LIS3DH.h" // Contains the defines with slave register addresses
```

Read the WHO_AM_I register value

main.c - Main firmware for the application.
First task**:** read the register WHO AM I and send its value over UART
Init the system and scan the I2C BUS for devices

```
CyGlobalIntEnable; /* Enable global interrupts. */
/* Place your initialization/startup code here (e.g. MyInst_Start()) */
I2C_Peripheral_Start();
UART_1_Start();
CyDelay(5); // The boot procedure is complete about 5 ms after device power-up
// String to print out messages on the UART
char message[50];
// Check which devices are present on the I2C bus
for( int i = 0 ; i < 128; i++ ) {
    if (I2C_Peripheral_IsDeviceConnected(i)){
        // print out the address is hex format
        sprintf(message, "Device 0x%02X is connected\r\n", i);
        UART_1_PutString(message);
    }
}
```

Read the WHO_AM_I register value

main.c - Main firmware for the application.
First task**:** read the register WHO AM I and send its value over UART
Read WHO_AM_I and send the value over UART

```
/* Read WHO AM I REGISTER register */
uint8_t who_am_i_reg;
ErrorCode error = I2C_Peripheral_ReadRegister(LIS3DH_DEVICE_ADDRESS,
                                              LIS3DH_WHO_AM_I_REG_ADDR,
                                              &who_am_i_reg);
if( error == NO_ERROR ){
    sprintf(message, "WHO AM I REG: 0x%02X [Expected: 0x33]\r\n", who_am_i_reg);
    UART_1_PutString(message);
}
else
{
    UART_1_PutString("Error occurred during I2C comm\r\n");
}
```

Read the STATUS register value

main.c - Main firmware for the application.
Second task**:** read the register `LIS3DH_STATUS_REG` (0x27) and send its value over UART
Read STATUS_REG1 and send the value over UART

```
/* Read WHO AM I REGISTER register */
uint8_t status_reg;
ErrorCode error = I2C_Peripheral_ReadRegister(LIS3DH_DEVICE_ADDRESS,
                                               LIS3DH_WHO_AM_I_REG_ADDR,
                                               &status_reg);
if( error == NO_ERROR ){
    sprintf(message, "WHO AM I REG: 0x%02X [Expected: 0x33]\r\n", status_reg);
    UART_1_PutString(message);
}
else
{
    UART_1_PutString("Error occurred during I2C comm\r\n");
}
```

Read the CONTROL register value

main.c - Main firmware for the application.
Second task**:** read the register `LIS3DH_CTRL_REG1` (0x20) and send its value over UART
Read CONTROL_REG1 and send the value over UART

```
/* Read WHO AM I REGISTER register */
uint8_t control_reg;
ErrorCode error = I2C_Peripheral_ReadRegister(LIS3DH_DEVICE_ADDRESS,
                                              LIS3DH_WHO_AM_I_REG_ADDR,
                                              &control_reg);
if( error == NO_ERROR ){
    sprintf(message, "WHO AM I REG: 0x%02X [Expected: 0x33]\r\n", control_reg);
    UART_1_PutString(message);
}
else
{
    UART_1_PutString("Error occurred during I2C comm\r\n");
}
```

Implement I2C Register Writing

SWITCH TO PROJECT *"02-I2C_Master_Write"*

***I2C_Interface.c* - Macros and function definitions.**

Start to implement the function `I2C_Interface_WriteRegister()`

This function takes the following parameters:

- `device_address`: address of the I2C device
- `register_address`: register that needs to be updated
- `data`: the data byte to write into the register

And should return an ErrorCode (defined in *ErrorCodes.h*) as follows:

- `NO_ERROR` if no error occurred during I2C communication
- `ERROR` if some error occurred during I2C communication

Implement I2C Register Writing

***I2C_Interface.c* - Macros and function definitions.**

From the I2C component datasheet:

```
status = I2C_MasterSendStart(slaveAddr, I2C_WRITE_XFER_MODE);
if( I2C_MSTR_NO_ERROR == status ) {
    /* Read array of 5 bytes */
    for( int i = 0; i < 5; i++ ) {
        status = I2C_MasterWriteByte(data[i]);
        if( status != I2C_MSTR_NO_ERROR) break;
    }
}
```

Write into CTRL_REG1

main.c - Main firmware for the application.
Third task**:** write a new value into the register `LIS3DH_CTRL_REG1` (0x20)
Write the value `LIS3DH_NORMAL_MODE_CTRL_REG1` (0x47) **into CONTROL_REG1**

```
if (ctrl_reg1 != LIS3DH_NORMAL_MODE_CTRL_REG1) {
    ctrl_reg1 |= LIS3DH_NORMAL_MODE_CTRL_REG1;
    error = I2C_Peripheral_WriteRegister(LIS3DH_DEVICE_ADDRESS,
                                         LIS3DH_CTRL_REG1,
                                         ctrl_reg1);

    if (error == NO_ERROR) {
        sprintf(message, "CTRL_REG1 successfully written as: 0x%02X\r\n", ctrl_reg1);
        UART_1_PutString(message);
    }
    else {
        UART_1_PutString("Error occurred during I2C comm to set CTRL_REG1\r\n");
    }
}
```

Read the new value of CTRL_REG1

main.c - Main firmware for the application.
Third task: write a new value into the register `LIS3DH_CTRL_REG1` (0x20)
Read the value of CTRL_REG1 and compare it to `LIS3DH_NORMAL_MODE_CTRL_REG1` (0x47)

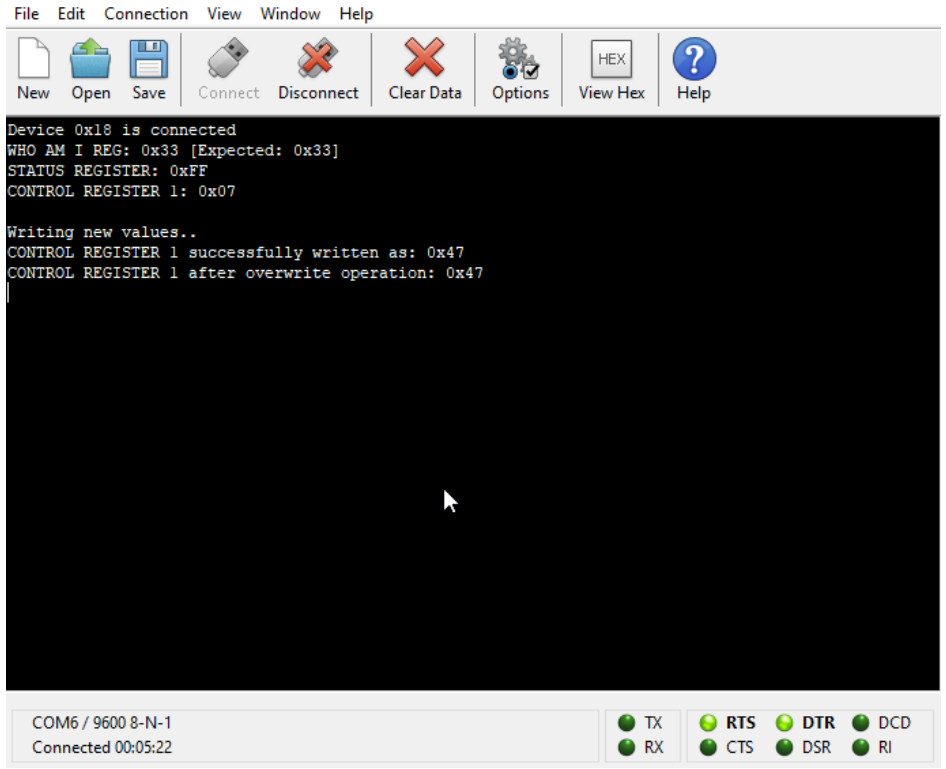
```
error = I2C_Peripheral_ReadRegister(LIS3DH_DEVICE_ADDRESS,
                                    LIS3DH_CTRL_REG1,
                                    &ctrl_reg1);

if( error == NO_ERROR ) {
    sprintf(message, "CTRL_REG1 after write operation: 0x%02X\r\n", ctrl_reg1);
    UART_1_PutString(message);
}
else {
    UART_1_PutString("Error occurred during I2C comm to read control register 1\r\n");
}
```

Read over UART

Test the project.

1. Open CoolTerm and set it up as described in the previous projects
2. Open the correct **COM** port
3. Check that the output is equal to the one shown below



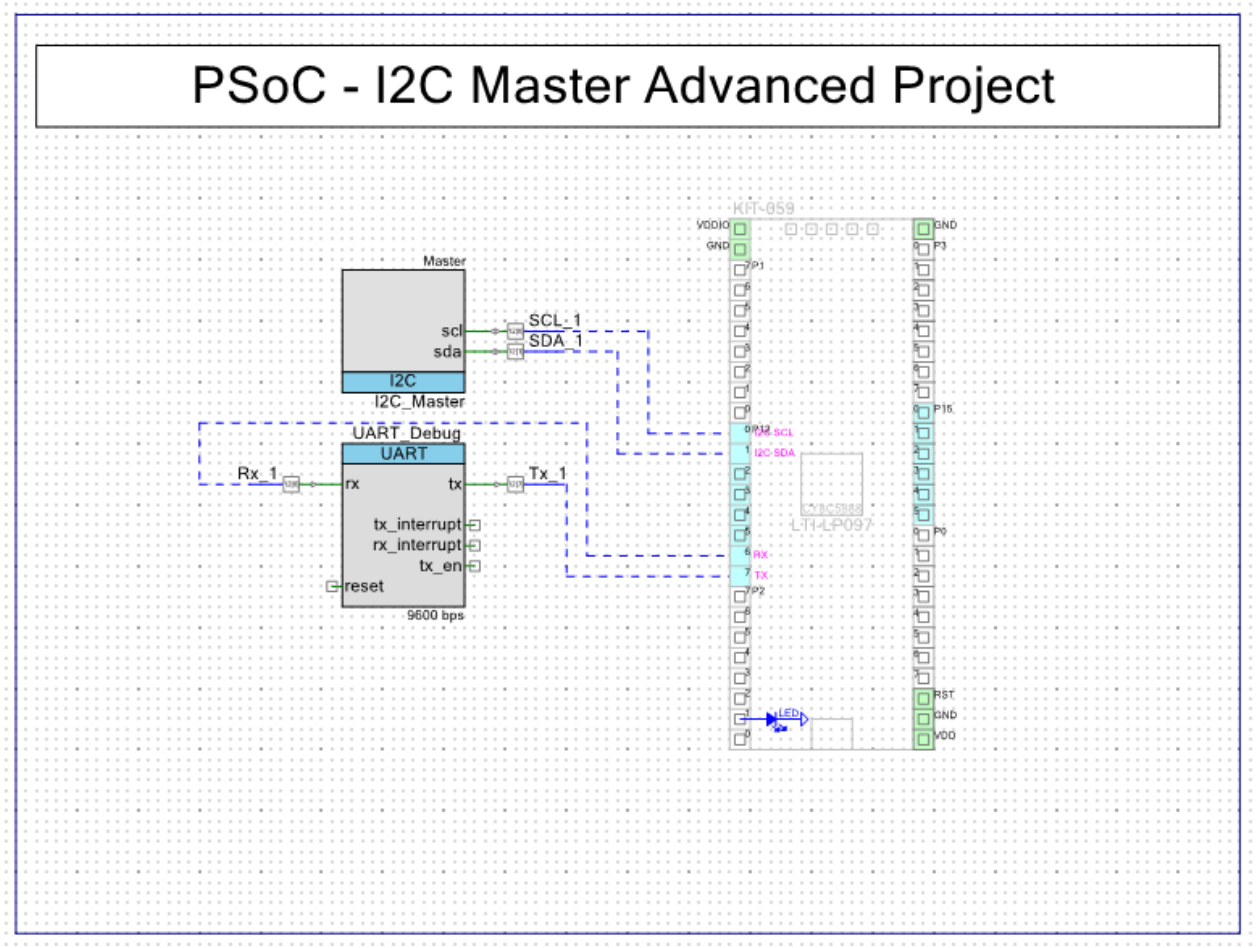
Project 3

Project goal: Communicate via I2C with the slave device and perform the following tasks:

- Read the Temperature Configuration register
- Write onto the appropriate registers in order to activate the Temperature output
- Read the Temperature values and send them over the UART after conversion

Project 3

Switch to the project "*03-I2C_Master_Advanced*"



Project 3

main.c - Main firmware for the application.
First task: read the temperature configuration register

```
uint8_t tmp_cfg_reg;

error = I2C_Peripheral_ReadRegister(LIS3DH_DEVICE_ADDRESS,
                                    LIS3DH_TEMP_CFG_REG,
                                    &tmp_cfg_reg);
if( error == NO_ERROR ) {
    sprintf(message, "LIS3DH_TEMP_CFG_REG: 0x%02X\r\n", tmp_cfg_reg);
    UART_1_PutString(message);
}
else {
    UART_1_PutString("Error occurred during I2C comm to read LIS3DH_TEMP_CFG_REG\r\n");
}
```

Project 3

main.c - Main firmware for the application.
Second task: write into the appropriate configuration registers to active the temperature sensor

```
tmp_cfg_reg |= 0x00; // must be changed to the appropriate value

error = I2C_Peripheral_WriteRegister(LIS3DH_DEVICE_ADDRESS, LIS3DH_TEMP_CFG_REG, tmp_cfg_reg);
error = I2C_Peripheral_ReadRegister(LIS3DH_DEVICE_ADDRESS, LIS3DH_TEMP_CFG_REG, &tmp_cfg_reg);

if (error == NO_ERROR) {
    sprintf(message, "LIS3DH_TEMP_CFG_REG updated: 0x%02X\r\n", tmp_cfg_reg);
    UART_1_PutString(message);
}
else {
    UART_1_PutString("Error occurred during I2C read of LIS3DH_TEMP_CFG_REG\r\n");
}
```

Project 3

main.c - Main firmware for the application.
Second task: write into the appropriate configuration registers to active the temperature sensor

```
...
#define LIS3DH_??_REG 0x00 // fill this in LIS3DH.h
...

uint8_t reg;

error = I2C_Peripheral_ReadRegister(LIS3DH_DEVICE_ADDRESS,
                                    LIS3DH_??_REG,
                                    &reg);

if( error == NO_ERROR ) {
    sprintf(message, "LIS3DH_??_REG: 0x%02X\r\n", reg);
    UART_1_PutString(message);
}
else {
    UART_1_PutString("Error occurred during I2C comm to read LIS3DH_??_REG\r\n");
}
```

Project 3

main.c - Main firmware for the application.
Second task: write into the appropriate configuration registers to active the temperature sensor

```
reg |= 0x00; // must be changed to the appropriate value

error = I2C_Peripheral_WriteRegister(LIS3DH_DEVICE_ADDRESS, LIS3DH_??_REG, reg);
error = I2C_Peripheral_ReadRegister(LIS3DH_DEVICE_ADDRESS, LIS3DH_??_REG, &reg);

if (error == NO_ERROR) {
    sprintf(message, "LIS3DH_??_REG updated: 0x%02X\r\n", reg);
    UART_1_PutString(message);
}
else {
    UART_1_PutString("Error occurred during I2C read of LIS3DH_??_REG\r\n");
}
```

Project 3

main.c - Main firmware for the application.
Third task: continuously read the temperature sensor and send the temperature value over UART

```
for(;;) {
    CyDelay(100);
    error = I2C_Peripheral_ReadRegisterMulti(LIS3DH_DEVICE_ADDRESS, LIS3DH_OUT_ADC_3L,
                                             2, TemperatureData);
    if(error == NO_ERROR) {
        OutTemp = (int16_t)((TemperatureData[0] | (TemperatureData[1]<<8)))>>6;
        OutTemp = (int16_t) (outtempconv * dirtytrick);

        OutArray[1] = (uint8_t)(OutTemp & 0xFF);
        OutArray[2] = (uint8_t)(OutTemp >> 8);
        UART_1_PutArray(OutArray, 4);
    }
    else {
        UART_1_PutString("Error occurred during I2C comm to read LIS3DH_OUT_ADC_3L\r\n");
        }
    }
}
```

Project 3

0 x 0D

x	**x**	**x**	**x**	**x**	**x**	**x**	**x**

OUT_ADC_3_H

0 x 0C

x	**x**	**0**	**0**	**0**	**0**	**0**	**0**

OUT_ADC_3_L

The input range is 1200 mv ±400 mV and the data output is expressed in 2's complement left-aligned.

The ADC resolution is 10 bits if the LPen (bit 3) in *CTRL_REG1 (20h)* is cleared (high-resolution / normal mode), otherwise, in low-power mode, the ADC resolution is 8-bit.

Project 3

```
OutTemp = (int16)(TemperatureData[0] | (TemperatureData[1] << 8)) >> 6;
```

TemperatureData[0]									x	x	0	0	0	0	0	0
TemperatureData[1] << 8	x	x	x	x	x	x	x	x	0	0	0	0	0	0	0	0
(TData[0] | Tdata[1] << 8)	x	x	x	x	x	x	x	x	x	x	0	0	0	0	0	0
>> 6	0	0	0	0	0	0	x	x	x	x	x	x	x	x	x	x

Temperature Output