# Lab 5: I²C

## Objective

Demonstrate knowledge of using I$^2$C to interface a microcontroller to peripheral devices.

## Background

### B1: I²C

#### The Interface

I$^2$C (Inter-Integrated Circuit, usually pronounced I-squared-C) is a two-wire bus used for low-speed (typ. 100kbps), short distance, bidirectional serial communication between a device and one or more peripherals. Devices that initiate requests from a peripheral are called masters and the peripherals are called slaves. There may be multiple masters on the bus and slaves can also be masters, but for the purposes of this lab, only the one master and multiple slaves case will be discussed.

The two wires used for I$^2$C are called Serial Data Line (SDA) and Serial Clock Line (SCL). Generally, SDA is used to send data and SCL is used to indicate when SDA should be read. These lines are also open-drain lines with pull-up resistors, as depicted in Figure 1. In other words, logic 0 is indicated as a high voltage by disconnecting the line from ground, allowing the pull-up resistor to increase the voltage of the line (the resistor "pulls up" the voltage), and logic 1 is indicated as low voltage by connecting the line to ground. Further specifics of how these lines are used in I$^2$C will not be discussed here.

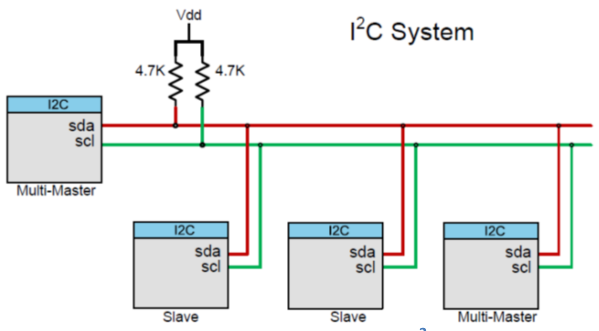


**Figure 1: Block diagram of an I$^2$C bus**

Slave devices each have their own 7-bit address. These addresses are often hard-coded at manufacture, but some devices allow their address to be modified to some extent to help avoid having duplicate addresses on the same bus, especially if the same device is used multiple times on the same bus. Slaves wait for a master to initiate communication with their address and then proceed to either accept or send data, depending on what the master commanded. A slave that is not being addressed simply ignores any communication until the master ceases the communication. A slave may also force the master to wait on it if it isn't ready to accept or send data yet by holding the clock low.

Master devices have no address, as they initiate communication with the slaves. The master also controls the generation of the clock for the bus. When it wants to communicate with a slave, it sends a start signal followed by the address of the slave and whether it wants to read or write. The slave then responds with an ACK signal and the transmission proceeds.

- When writing, the master proceeds to transmit 8-bit words. Between each word, the slave produces an ACK signal to acknowledge that it received the data.
- When reading, the slave proceeds to transmit 8-bit words. Between each word, the master produces an ACK signal when it wants another word, or a NAK signal for the last word it wants to receive.

The master then either ends the transmission and frees up the bus with a stop bit or sends another start (known as a restart) to initiate a new transmission without freeing up the bus. The states of an I$^2$C bus are depicted in Figure 2.

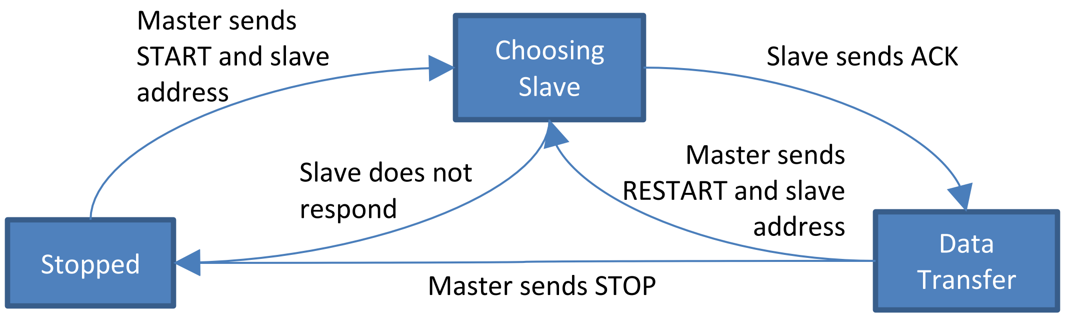


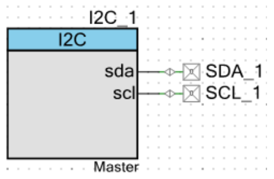
**Figure 2: States of an I$^2$C Bus**

Many I$^2$C peripherals, including the one used in this lab, use registers that hold data such as configuration and results. Each register is assigned an address. To read from a register (excluding write-only registers), the master writes the address of the register to the slave then reads from the slave. To write to a register (excluding read-only registers), the master writes the address of the register and the new value to store in the register to the slave.

## The Component

The I$^2$C implementation in the PSoC 5LP is able to work as both a master and a slave, along with the multi-master and multi-master-slave modes that won't be covered here. The PSoC can be a slave because some use cases of the device place it as a peripheral in a larger system. For the purposes of this lab, only the master mode of the I$^2$C implementation will be discussed. For further information, read the datasheet for the component.

This implementation also has both an FF and UDB implementation, as the 5LP includes two blocks that are dedicated to I$^2$C. There is no real advantage to use UDB, other than small power savings at slow speeds, and it also loses a wake from sleep feature that FF has. Therefore, only the FF implementation will be discussed here.



There are no inputs to the component in the FF implementation. SDA and SCL are bidirectional I/O that is automatically connected to bidirectional pins with the drive mode "Open Drain, Drives Low."

The only important configuration of a FF $I^2C$ Master component is the data rate, which is set to 100 kbps by default. The data rate should not exceed the slowest device on the bus.

Here are the most important functions in the programming interface. "I2C" is replaced with the name given to the component in the schematic view.

| | |
|---|---|
| void I2C_Start() | Initializes the $I^2C$ component. |
| uint8 I2C_MasterStatus() | Returns the current communication status. Used for non-blocking communication. Bitwise AND (&) the return value with the following constants to determine the status:<br>I2C_MSTAT_RD_CMPLT – Read transfer complete, error or not<br>I2C_MSTAT_WR_CMPLT – Write transfer complete, error or not<br>I2C_MSTAT_XFER_INP – Transfer in progress<br>Others: error conditions. See the datasheet for the complete list. |
| uint8 I2C_MasterClearStatus() | Returns the current communication status and clears it. **Call this before initiating a non-blocking transfer**. |
| uint8 I2C_MasterWriteBuf (uint8 slaveAddress, uint8* wrData, uint8 cnt, uint8 mode) | Initiates a non-blocking write to a slave. Use MasterStatus to determine when it is complete. See MasterSendStart for an explanation of the return value.<br>slaveAddress is the address of the slave to write to<br>wrData is an array of data to write to the slave<br>cnt is the number of bytes from the array to write to the slave<br>mode determines whether the start of the transfer should be a start or restart and if a stop condition should be generated at the end of the transfer. Use I2C_MODE_COMPLETE_XFER to perform a complete transfer from start to stop |
| uint8 I2C_MasterReadBuf (uint8 slaveAddress, uint8* rdData, uint8 cnt, uint8 mode) | Initiates a non-blocking read from a slave. Use MasterStatus to determine when it is complete. See MasterSendStart for an explanation of the return value.<br>The parameters are the same as MasterWriteBuf, except rdData is where the data being read is stored to. |
| uint8 I2C_MasterSendStart (uint8 slaveAddress, uint8 R_nW) | Generates a start condition for the specified slave and read/write mode. This function is blocking.<br>The return value indicates if there were any bus errors when generating the start condition. If the return value is I2C_MSTR_NO_ERROR, the start was a success.<br><br>R_nW: I2C_WRITE_XFER_MODE for initiating a write, I2C_READ_XFER_MODE for initiating a read. |
| uint8 I2C_MasterSendRestart (uint8 slaveAddress, uint8 R_nW) | Same as MasterSendStart, except it generates a restart condition. Only valid when the previous transfer wasn't stopped. |
| uint8 I2C_MasterSendStop() | Generates a stop condition on the bus. Blocking; does nothing if a previous start or restart failed. See MasterSendStart for the return. |
| uint8 I2C_MasterWriteByte (uint8 theByte) | Writes the given byte to the slave. Blocking; does nothing if the bus hasn't started or restarted. Return value is I2C_MSTR_NO_ERROR if there were no errors. |

| | |
|---|---|
| uint8 I2C_MasterReadByte (uint8 acknNak) | Reads and returns one byte from the slave and ACKs or NAKs it. Blocking; does nothing if the bus hasn't started or restarted. Last byte read should be NAKed.<br>ACK: I2C_ACK_DATA NAK: I2C_NAK_DATA |

There are two ways to read or write data in this implementation: blocking or non-blocking. The blocking method means the CPU will wait for each transaction to finish before moving on while the non-blocking method uses an interrupt that comes with the component to handle transactions in the background while the CPU does other tasks. If there is nothing for the CPU to do while waiting for data transactions, the blocking method works fine. The non-blocking method, while a bit more complicated to set up, is ultimately better for complex projects.

### *Process for a Blocking Write*

- Start or restart the bus, addressing the slave that will receive data (restart when a stop wasn't sent after a previous transfer). Make sure the return value indicates no error.
    - I2C_MasterSendStart(slaveAddress, I2C_WRITE_XFER_MODE)
    - I2C_MasterSendRestart(slaveAddress, I2C_WRITE_XFER_MODE)
- Write data to the slave one byte at a time
    - I2C_MasterWriteByte(theByte)
- If there won't be another transmission to the bus immediately after writing is complete, send a stop signal
    - I2C_MasterSendStop()

### *Process for a Non-Blocking Write*

- Create and fill a byte array with the data to send
- Check to make sure there is no transfer in progress
    - if( !(I2C_MasterStatus() & I2C_MSTAT_XFER_INP) )
- Clear the master status
    - I2C_MasterClearStatus()
- Initiate the write. **Do not modify the array until the write is complete.** It may be a good idea to use a double buffer (two arrays that alternate between buffering outgoing data and being used to write to the bus) so the next write can be buffered while a write is taking place.
    - I2C_MasterWriteBuf(slaveAddress, wrData, count, I2C_MODE_COMPLETE_XFER)

### *Process for a Blocking Read*

- Start or restart the bus, addressing the slave that will send data. Make sure there is no error.
    - I2C_MasterSendStart(slaveAddress, I2C_READ_XFER_MODE)
    - I2C_MasterSendRestart(slaveAddress, I2C_READ_XFER_MODE)
- Read data from the slave one byte at a time, sending an ACK for each byte except the final one
    - uint8 theByte = I2C_MasterReadByte(I2C_ACK_DATA)
- When reading the final (or only) byte, NAK it
    - uint8 theByte = I2C_MasterReadByte(I2C_NAK_DATA)
- If there won't be another transmission to the bus immediately after reading is complete, send a stop signal
    - I2C_MasterSendStop()

### *Process for a Non-Blocking Read*

- Create a byte array to hold the bytes to be read.
- Check to make sure there is no transfer in progress
    - if( !(I2C_MasterStatus() & I2C_MSTAT_XFER_INP) )
- Clear the master status
    - I2C_MasterClearStatus()
- Initiate the read
    - I2C_MasterReadBuf(slaveAddress, rdData, count, I2C_MODE_COMPLETE_XFER)
- Wait until the transfer is complete before reading the array
    - If(I2C_MasterStatus() & I2C_MSTAT_RD_CMPLT)

## B2: MCP9808 I$^2$C Temperature Sensor

This lab will make use of the MCP9808 temperature sensor breakout board from Adafruit. The MCP9808 chip is an I$^2$C peripheral that measures temperature with a maximum resolution of 0.625 °C, a range of -40 °C to 125 °C, and a typical accuracy of ±0.25 °C. In addition to reading temperature, there is also a feature to send an alert signal on a third pin when the temperature exceeds a boundary. There are also three more pins that can alter the I$^2$C address of the device, so eight of these devices may be used on the same bus. Finally, the breakout board includes the pullup resistors required for the I$^2$C bus, so additional pullup resistors are not required.

When powered on, the device will start continuously reading the temperature at the maximum resolution. Each reading takes approximately 250 ms to complete. The master should request the temperature from the device periodically, but it should not poll any more frequently than every 250 ms.

The product page is here: http://www.adafruit.com/product/1782
The MCP9808 datasheet is here: http://www.adafruit.com/datasheets/MCP9808.pdf

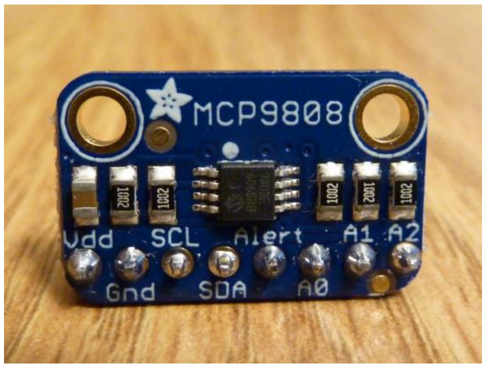


Figure 3: The MCP9808 Breakout by Adafruit

The pins on the device depicted in Figure 3 are as follows:

- Vdd – power input (2.7v – 5.5v)
- Gnd – ground input
- SCL, SDA – I$^2$C bus connections. These have a pullup resistor
- Alert – signals when the temperature is outside the configured threshold

- A0-A2 – Address modifier. The device address is, in binary, 0011xxx, where xxx corresponds to A2-A0 in order. Connecting to ground or leaving disconnected makes the pin's bit 0 and connecting to Vdd makes the bit 1.

The **device address** is **0x18** when all three address pins are disconnected or connected to ground. Attaching an address pin to Vdd changes the corresponding address bit (one of the lower three bits) to 1.

This device has a handful of registers to read and write to; however, this lab will only read from registers. See section 5.1 of the datasheet (linked in the blue box above) for a full list of registers.

All registers contain **16-bits** with an **8-bit register address**. In order to read from a register, **write** the **register** address to the device and **then read** two bytes from the device. The first byte returned is the most significant byte (bits 15-8).

## Manufacturer ID Register

Address: **0x06**

This register identifies the manufacturer of the device. This should be checked along with the device ID to make sure the device at the address is the expected device.

Value: **0x0054**

## Device ID Register

Address: **0x07**

This register identifies the device's ID. This should be checked along with the manufacturer ID to make sure the device at the address is the expected device.

Value: **0x04XX**, where XX is the revision number. Only the upper eight bits need to be checked, since a device revision shouldn't change the register mapping.

## Ambient Temperature Register

Address: **0x05**

This register holds the last temperature reading produced by the sensor. Note that this register is double buffered, so the temperature sensor can be taking a reading while the master reads this.

The lower 13 bits of the register is the temperature in sixteenths of degrees Celsius, in two's complement form. The upper three bits contain data related to the alert functionality.

To convert the temperature reading to degrees Celsius, first assign the value to an **int16** (not uint16), then extend the sign bit (bit 13; doing the following also discards the upper three bits) by left-shifting three times and right shifting three times (the right shift extends the sign since the type is signed). Then, convert the value (whose unit is sixteenths of degrees Celsius) to a floating point value in degrees Celsius by multiplying by 0.0625. Here is code that demonstrates this process:

```
int16 read = <16-bit reading>;

//Reading is a signed, two's complement 13-bit number. Extend it to 16-bit
read <<= 3;
read >>= 3;

//Convert to degrees Celsius. Convert from scale 1 = 1/16
float temperature = read * 0.0625;
```

## B3: Prototyping Area

On the lower half of the PSoC 5LP Development Kit, as seen in Figure 4, there is a section that includes a breadboard and headers with pins from the PSoC and a few other things. This is the prototyping area. It is used for connecting external hardware and peripherals to the PSoC without having to solder anything.

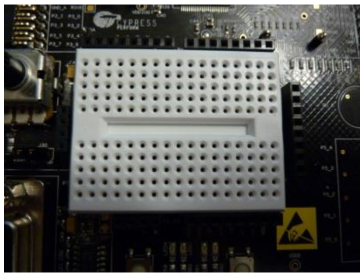


Figure 4: Prototyping Area

### Breadboards

Breadboards are a base for prototyping electronics without needing to solder anything. The breadboard on the development kit consists of two rows of 17 clips, where each clip has 5 holes. When a wire or component is inserted into a hole on a clip, it is electrically connected to anything else in the clip. See Figure 5 below for a depiction of what holes are connected together.

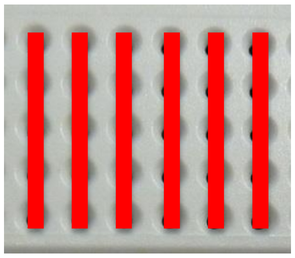
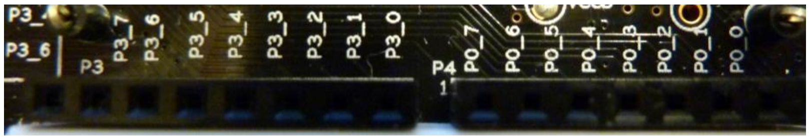


Figure 5: Electrical connections on a breadboard

### Headers

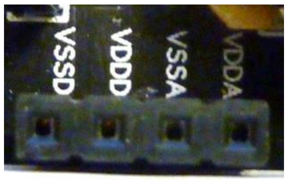
The headers, or rows of pins (female pins, in this case), surrounding the breadboard lead to pins on the PSoC and to a few other things, including power and ground connections. Here are lists of what the pins on each side of the prototyping board are.

### *Top*



- P3[7:0] – all eight pins of Port 3. Some of these pins have special purposes in the analog hardware, but they can still be used for anything.
- P0[7:0] – all eight pins of Port 0. Some of these pins have special purposes in the analog hardware, but they can still be used for anything.

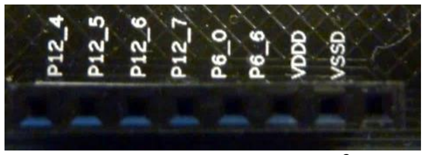
### *Left*



- VSSD – Digital ground
- VDDD – Digital power (3.3v default)
- VSSA – Analog ground
- VDDA – Analog power (3.3v default)

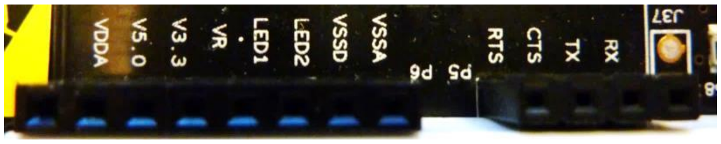
Note: digital devices should use VDDD and VSSD and analog devices should use VDDA and VSSA. Analog devices are often sensitive to noise, and using a ground isolated from digital circuits helps reduce noise.

### *Right*



- P12[4] – Pin 4 of port 12. When using a fixed function I$^2$C slave, this is the SCL pin that can be used for waking up the PSoC when it is addressed on the bus.
- P12[4] – Pin 5 of port 12. When using a fixed function I$^2$C slave, this is the SDA pin that can be used for waking up the PSoC when it is addressed on the bus.
- P12[7:6] – Other pins on port 12. General purpose.
- P6[0] – Pin 0 of port 6. General purpose.
- P6[6] – Pin 6 of port 6. General purpose.
- VDDD – Digital power
- VSSD – Digital ground

### *Bottom*



- VDDA – Analog power
- V5.0 – 5 volt power source. Do not use when power is provided over USB, as it'll be less than 5v.
- V3.3 – 3.3 volt power source.
- VR – Output of the potentiometer that is also connected to P6[5].

- LED1 – Connection to the LED that is directly below it. This LED is not connected to the PSoC and requires an input on this pin if using it is desired.
- LED2 – Connection to the LED that is directly below it. This LED is not connected to the PSoC and requires an input on this pin if using it is desired.
- VSSD – Digital ground
- VSSA – Analog ground
- RX/TX/CTS/RTS – Connections to the RS-232 interface on the development kit.

## Procedure

### Outcome

In this lab, the PSoC will continually read the temperature from the MCP9808 $I^2C$ device and print it to the display.

### Part 1: Designing the System

1. Create a new project in your workspace named Lab5.
2. Add a Communications>I2C>I2C Master (Fixed Function) to the schematic and rename it to I2C. Note that this configuration of the component comes with SDA and SCL pins pre-configured to be bidirectional and the drive mode “Open Drain, Drives Low.”
3. Add a Character LCD and name it Display.
4. Configure Display to use P2[6:0], SCL to use P12[4], and SDA to P12[5].
5. Generate the application.

### Part 2: Programming the Firmware

#### Overview

The firmware will first confirm that the device is on the $I^2C$ bus, and then continually request the temperature from the device.

#### Process

All of the information necessary to program the firmware and interface to the device is provided in the Background section. After initializing the display and $I^2C$ components:

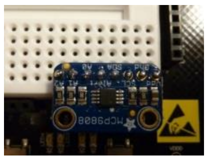
- Read the manufacturer ID register and confirm that it has the expected value.
- Read the device ID register and confirm its upper byte has the expected value.
- If either of those two checks failed, print an error message
- In the infinite for loop:
  - If either of the two earlier checks failed, do nothing; just don’t allow the main to exit
  - Otherwise, read the ambient temperature register
  - Convert the temperature reading as specified in the background section
  - If you want, convert to degrees Fahrenheit by multiplying by 9/5 and adding 32.
  - Print the reading to the display
    - Use the method for printing floating point numbers from past labs
  - Sleep for 250ms using CyDelay(250), because readings take that long to update

## Tips

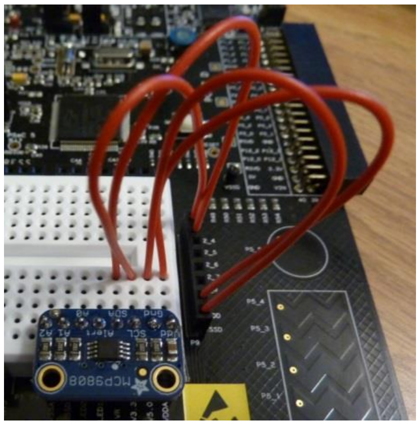
- Because this application is only getting and printing temperatures, there is no need to use the non-blocking method of reading and writing to the $I^2C$ bus.
- Because there are several occurrences of reading 16-bit registers, it would be a good idea to write a function that handles a register read.
- Use #defines for constants such as the registers and expected values

## Part 3: Running the Project

1. Before running the project, some setup on the development kit is required. Make sure the development kit is unplugged before continuing.
2. Take a MCP9808 breakout and plug it in horizontally across the top or bottom row of pins. Ensure there is at least one exposed row of pins.



3. In the development kit box, there should be a small bag of wires in the USB cable compartment. Use these wires to connect the following header pins to the same column of rows as the stated device pins:
    - VDDD – Vdd
    - VSSD – Gnd
    - P12_4 – SCL
    - P12_5 – SDA



**STOP: DO NOT POWER ON YET**

**Before continuing, let the lab instructor double check your wiring.**

4. Build and load the project onto the PSoC development kit.
5. Test the program to make sure it works:
    - The display should be continuously updated with the room temperature, approximately 4 times a second (when there are any changes).
    - **GENTLY** touch the chip in the center of the MCP9808 breakout with a **DRY** finger. The temperature should shoot up a few degrees.
    - Remove your finger. The temperature should eventually return to room temperature.