University of Cincinnati

Date: 7/6/2017

I, Srikar Chintapalli, hereby submit this original work as part of the requirements for the degree of Master of Science in Computer Engineering.

It is entitled:

Communication Protocols on the PIC24EP and Arduino - A Tutorial for Undergraduate Students

Student's name: **Srikar Chintapalli**

This work and its defense approved by:

Committee chair: Carla Purdy, Ph.D.

Committee member: Rui Dai, Ph.D.

Committee member: Wen-Ben Jone, Ph.D.

UNIVERSITY OF Cincinnati

27853

Communication Protocols on the PIC24EP and Arduino – A Tutorial for Undergraduate Students

A thesis submitted to the graduate school of the University of Cincinnati in partial fulfillment of the requirements for the degree of

Master of Science

In the Department of Electrical Engineering and Computing Systems of the College of Engineering and Applied Science

By

Srikar Chintapalli

Bachelor of Technology: Electronics and Communications Engineering

NIT Warangal, May 2015

Committee Chair: Dr. Carla Purdy

Abstract

With embedded systems technology growing rapidly, communication between MCUs, SOCs, FPGAs, and their peripherals has become extremely crucial to the building of successful applications. The ability for designers to connect and network modules from different manufacturers is what allows the embedded computing world to continue to thrive and overcome roadblocks, driving us further and further towards pervasive and ubiquitous computing. This ability has long been afforded by standardized communication protocols developed and incorporated into the devices we use on a day-to-day basis.

This thesis aims to explain to an undergraduate audience and to implement the major communication protocols that are used to exchange data between microcontrollers and their peripheral modules. With a thorough understanding of these concepts, students should be able to interface and program their microcontroller units to successfully build projects, giving them hands on experience in embedded design. This is an important skill to have in a field in which configuring the electronics and hardware to work correctly is equally as integral as writing code for the desired application. The protocols that are discussed are the three main serial communication protocols: I2C (inter-integrated circuit), SPI (serial peripheral interface), and TTL UART (universal asynchronous receiver transmitter). BLE (Bluetooth low energy) is also explored to try and help students add cheap wireless functionality to their designs. In order to successfully put forth and apply the concepts, this thesis uses the Arduino Uno R3 (Atmel ATmega328 microcontroller) and the mikromedia for PIC24EP (PIC24EP512GU810 microcontroller) boards. On the Arduino, we use high level functions afforded by library support to go through the steps of implementing and using the UART, SPI, and I2C serial communication protocols. For the PIC, we write our own functions to write the protocols from

scratch using registers in the hardware communication modules provided on board. To test these out, we use the BME280 and MCP9808 sensors that can be interfaced with I2C or SPI. In addition, we also go through the steps of using the Bluefruit LE shield by Adafruit to integrate multiple modules adding Bluetooth connectivity to the embedded controller. Based on the tutorial nature of this thesis, students should be able to implement these protocols and interface their own controllers to sensors and modules regardless of manufacturer or library support (given the operating voltages are compatible). At the end of this thesis, there will be a set of exercises to test procedural knowledge and to enable students to assess how well they retained the information.

Table of Contents

List of Figures

List of Tables

Chapter 1 - Introduction

1.1 Why Communication?

Microcontroller based embedded systems exist and control so much of the world around us, from consumer electronics such as our televisions and toasters to high criticality systems such as GPS modules and automobile safety applications. Per a study by Zion Research titled "Embedded Systems Market", the global demand for embedded systems is expected to generate $225.34 billion in revenue by the end of 2021, making it one of the fastest growing fields not only in electrical and computer engineering, but in all fields [1]. It used to be that only the high criticality and safety related applications demanded excellence in terms of low latency and high performance, but as computing power and memory reliability has increased exponentially, all systems are showcasing a much smaller margin of error. As these systems become increasingly more complex, a single design can contain a multitude of microcontroller or SoC units and sensors/peripherals. As a result of applications containing so many devices, the communication protocols that facilitate high performance data transfer between these controllers and peripherals are of paramount importance.

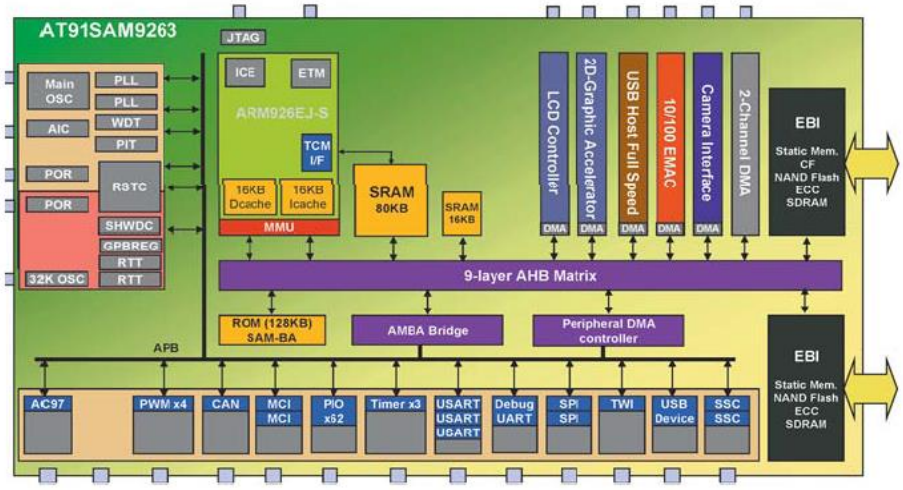

Fig 1.1: SoC Architecture (AtmelAT91) [2]

In Figure 1.1, a block diagram of a basic SoC is shown indicating several interfaces to connect the system to its peripherals.

1.2 Communication Protocols, Why Serial?

A communication protocol at the hardware level defines the way raw data is transmitted across a physical medium between different endpoints or nodes. What is done with the data once received in raw form is decided by higher level software; the communication protocols we discuss will not be concerned with that. To achieve effective data transfer between two devices, we can largely categorize physical communication into parallel and serial communication protocols.

Parallel communication involves sending multiple bits over different data lines at the same time, effectively allowing the speed of the data transmission to be dictated by the width of the data bus. In serial communication, data is shifted out serially on one line and shifted in at the destination. Parallel communication protocols such as PCI and Parallel ATA were much more common before the advent of high speed serial technologies. Despite the obvious advantage of parallel communication regarding the number of data bits that can be sent at the same time, it has some glaring disadvantages that eventually helped high speed serial communication take over. Ideally, even though it may seem that parallel communication is much faster than serial communication, as frequencies rise, clock skew becomes more and more of an issue that needs to be accounted for. Every parallel data bit will not reach its destination at exactly the same time due to a variety of conditions including but not limited to path resistance, temperature, and path length. This limits the speed of the entire bus to that of the slowest data line because the receiver needs to wait till all the data bits have arrived. The second disadvantage is the phenomenon of crosstalk; the parallel lines interfere with each other and this effect increases with distance

between the endpoints. This makes parallel communication difficult for long distances because erroneous receptions due to the combined effects of skew and crosstalk require retransmissions, defeating the advantage of speed. Even for shorter speed communications, newer technologies are taking over and are being used in most systems with PCI Express, a serial expansion bus.

In embedded systems, a huge advantage is that a very small number of pins are necessary to facilitate serial communication at very decent speeds; this is very important because microcontroller units and SoCs have a limited number of pins to work with. In addition, a very large number of sensors and peripheral modules use serial communication interfaces. There are two forms of serial transmission, synchronous and asynchronous. In asynchronous serial communication, there is no clock, and in synchronous serial communication, a clock line is used to synchronize data transfer; sampling occurs with respect to clock pulses. Simplex, half duplex, and full duplex communication are one-way data transfer, two-way data transfer but not at the same time, and two-way data transfer at the same time. Serial communication protocols include basic asynchronous serial communication using TTL logic and UARTs, I2C, SPI, FireWire, Serial ATA, 1-Wire serial bus, and many others.

1.3 Motivation and Aim

Most undergraduate embedded systems classes use a standard microcontroller with library support for communication and other modules because the sheer number of concepts to cover is too much. There is simply not enough time to go into detail and learn how each part of the controller or SoC works and how these functions are written. The aim of this thesis is to give a student the ability to pick up any major microcontroller and go about interfacing with and communicating to standard sensors and peripherals on the market, given that the electrical characteristics are compatible. This thesis assumes that the students using it as a guide have

reached their sophomore year in an electrical or computer engineering curriculum, giving them a background in elementary digital design and basic programming. Most microcontrollers and SoCs available today have on board UARTs, I2C, and SPI hardware modules that can be used to connect to the outside world. A vast number of peripheral modules on the embedded market also use TTL serial communication using UARTs, I2C, and SPI. Therefore, this thesis will go over these three serial communication protocols; we will discuss how they work on the lowest level and how we can make use of them in the most efficient manner possible. We will also go over BLE (Bluetooth low energy) for the Arduino to enable students to add wireless functionality to their projects and designs. From the information provided in this report, a student should be able to pick up any major MCU and use its onboard serial communication hardware to interface it with modules of different manufacturers without having to rely on library support. The following chapter will go over the conceptual details of the protocols before we eventually get to controller specific implementation.

Chapter 2 – Background

2.1 - Basic Serial Asynchronous Communication

In basic serial asynchronous communication, each device has its own clock that defines how fast data is transmitted. Baud rate is the rate at which data is transmitted in a communication channel in bits per second. Since there is no synchronization clock in this method of communication, the communicating devices need to be operating at the same baud rate; otherwise, there will be erratic transmissions. The baud rates can be set up according to what speed the devices on either side can handle; the slower device takes priority here.

Every time data is sent, it is sent in a packet or frame consisting of synchronization, parity, and data bits. Usually the number of data bits is eight, but it can be set to ten depending on the device in question. A key parameter that the communicating devices need to agree upon is whether data is transmitted MSB-first or LSB-first (endianness). The synchronization bits are the start and stop bits transmitted to signal the receiver of the beginning and end of every frame, respectively. An idle line is logic high, so a start bit will signal the bus by pulling an idle line low, and a stop bit is when the bus is pulled high again after the data bits [3]. Parity bits are forms of low-level error checking; a parity bit is sent after the data bits. In even parity, if there is an odd number of 1s in the data bits, the parity bit is set to 1, and if parity is odd, the parity bit is set to 0[6].

From a hardware perspective, to implement basic asynchronous serial communication, only two lines are needed, a transmit line and a receive line on each communicating device. From a signaling standpoint, two hardware protocols can be used, either TTL or RS-232. RS-232 is a signaling protocol that was introduced in 1962 and is still around today for legacy purposes

despite not being available on most devices anymore. In the RS-232 standard, signaling is done by voltages from +15V to -15V, with a logic 1 represented by -15V to -3V and a logic 0 represented by +3V to +15V. The voltages necessary for operation are generated by the RS-232 line driver circuit [8].

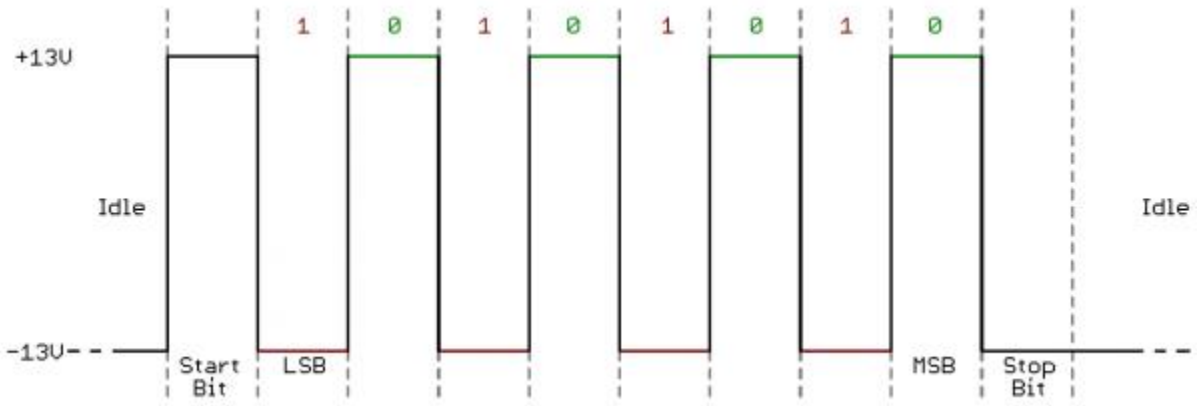

Fig 2.1 RS-232 Signaling [7]

Figure 2.1 shows the basic frame for basic asynchronous serial transmission, with the start, stop, and data bits.

Due to RS-232 having several drawbacks, it is rarely seen except in legacy settings. Because of the large voltage swings required, a lot of power consumption takes place during signaling and this also limits the maximum speed at which communication can take place. Another issue with the standard is that it defines two types of devices, a DTE device (data terminal equipment) and a DCE device (data connection equipment) but doesn't define how to connect two devices of the "same kind." The standard also has a lot of flow control and handshaking lines that most devices either don't need or can implement more minimally. Due to the size of the connectors among other drawbacks, most MCUs and SoCs do not provide RS-232 interfaces [8,9].

Most devices now use TTL serial signaling, where a logic 1 is defined by the MCU's V_{CC} which is usually 3.3V or 5V, and a logic 0 is represented by 0V. Peripherals respond to the MCUs in a similar manner.

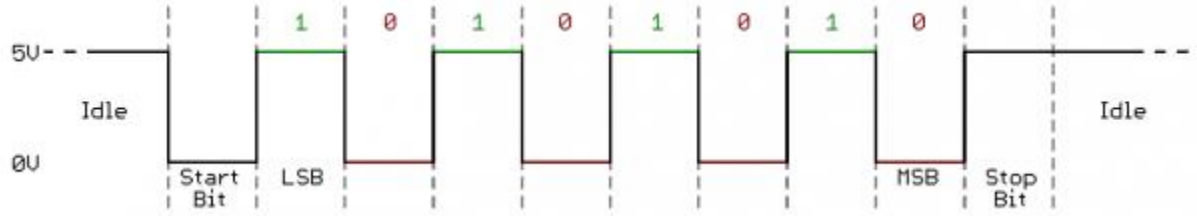

Fig 2.2 TTL Serial Signaling (V_{CC} = 5V) [7]

Figure 2.2 shows a basic frame transmitted with a TTL voltage level of 5V

A universal asynchronous receiver/transmitter or UART is a device in the form of an integrated circuit that assists the host controller or system with asynchronous serial communication. Most microcontrollers and SoCs now come with UARTs built in on board. The data format and baud rate can usually be set via registers belonging to the UART module. A UART module usually consists of a clock generator, transmit and receive shift registers and buffers, and control logic for correct operation [3].

The UART transmitter is responsible for serializing the data that needs to be sent, appending the start and stop bits, and calculating and appending the parity bit if necessary. The baud clock is on chip and it takes in its input clock from the processor. Data bits are shifted out of the transmit shift register, and a flag is set when it is full. Newer UARTs have FIFO buffers that bytes can be queued up in for transmission so that the host controller doesn't have to keep putting bytes into the transmit register one at a time [4,5]. The UART receiver is tasked with testing the incoming data line to look for the beginning of the start bit, and if the start bit lasts at least half the designated bit time, the receiver starts to sample for data bits. As each bit is

received, it is shifted into the receive register, and is available for the receiving device to access. Receive FIFO buffers allow for a cushion time for the receiving controller to read the data before it is erased by the arrival of a new byte [6].

To ensure that seamless communication occurs, it is essential that the receiving and transmitting UARTs are operating with the same setting for baud rate, parity, number of data bits, and number of stop bits. UARTs have flags that are set when the transmit buffer is full/empty, when the receive buffer is full/empty, and when there are errors. Detectable errors include overrun errors when characters are lost because the receive buffer wasn't emptied fast enough, parity errors when the parity of a received byte doesn't match up, and framing errors when a stop condition cannot be detected [3].

Hardware flow control signals are also a part of the UART modules. The RTS, or request to send, signal is used by a device to signal the device on the other end that it is ready to receive data. The CTS, or clear to send, signal is an input line that notifies the host device that the device on the other end is ready to receive data.

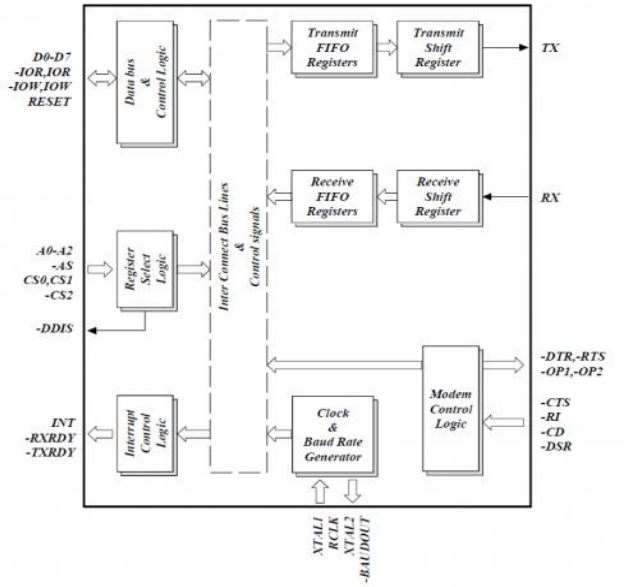

Fig 2.3 UART Block Diagram [7]

Figure 2.3 shows the block diagram of a basic UART including its internals, transmit and shift registers, and buffers.

2.2 – I2C Communication

I2C, or inter-integrated circuit, is a serial synchronous communication protocol that was developed by Phillips in 1982 to allow multiple masters and multiple slaves to operate on a single bus. The protocol is half-duplex and only calls for two open-drain lines, a clock line (SCL) and a data line (SDA). Because the lines are open-drain, by definition, there are pull-up resistors on each of them. The concept is that each device on the bus when operating as a slave only responds when it is addressed with its unique address. Each complete transaction, whether a read or a write, is different from the next in the sense that an address needs to be specified at the beginning of each transaction [10]. Initial I2C bus specifications allowed for 100 kHz operation, but later versions allowed for 400 kHz, 1 MHz, 3.4 MHz, and 5 MHz operation [13]. A device on the I2C bus can either be a master or a slave. Communication is initiated by the master and the slave responds, and the master also generates the clock that the slave uses as reference for the transaction. In any given transaction the master can either transmit to or receive from the slave device.

From a master device's perspective, the way to initiate communication is by sending a START condition on the I2C bus. A START condition can be initialized only when the bus is idle, and it is physically represented by the SDA line being pulled low when the SCL line is still high. A START condition on the bus puts all the slave devices on alert [13].

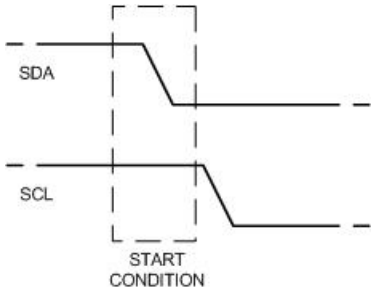

Fig 2.4 I2C Bus START Condition [13]

Figure 2.4 shows a start condition on the I2C bus (SDA pulled low when SCL is high).

The next step for the master to take after putting a START condition on the bus is to send the 7-bit device address on the bus appended with a read or write bit to indicate which device it wishes to communicate with and if it wants to write to it or read from it. Since the START condition alerted all the slave devices, they clock in the address byte that is put on the SDA line immediately after that. If this address matches a slave's own device address, it drives the SDA line low to acknowledge, and the other slave devices whose addresses didn't match ignore everything from that point onward until a STOP condition occurs on the bus [12,13,14].

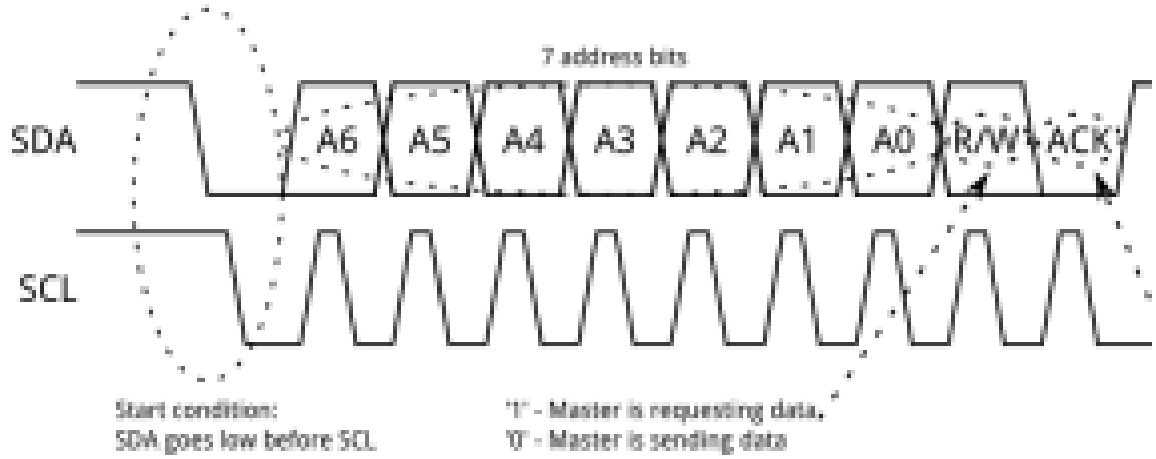

Fig 2.5 Address Byte [14]

Figure 2.5 shows the address byte being sent by the master on the I2C bus.

After the address byte is acknowledged by the slave, the master can now either send data or receive it from the slave based on what was specified in the read/write bit. When the clock line is high, data is stable and can be read by the receiving device. When the clock line is low, that is when the data line can be changed. For each clock pulse, one data bit is transferred. For every byte of data that is transferred, the bit that immediately follows is the ACK/NACK bit signifying whether or not the byte was received properly [13].

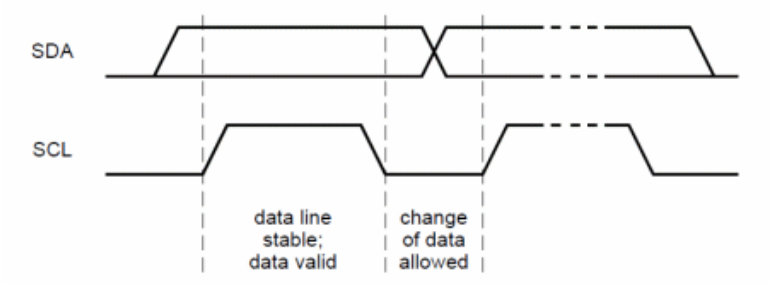

Fig 2.6 I2C Data Transfer [13]

Figure 2.6 shows the changing of data when SCL is low and its stability when SCL is high.

After the end of the transaction, a STOP byte is sent by the master to indicate the end of communication with the slave. This is designated by setting the data line high while the clock line is high.

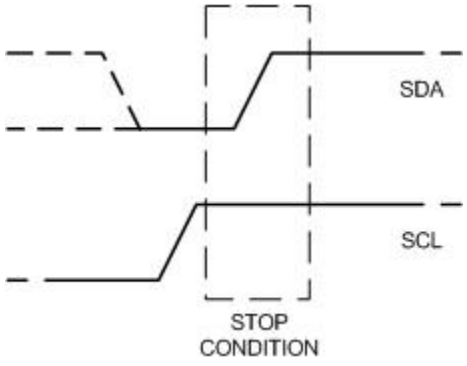

Fig 2.7 I2C Bus STOP Condition [13]

Figure 2.7 shows an I2C stop condition on the bus (SDA being pulled high while SCL is high).

If the transaction is a read operation initiated by the master, then the same procedure is followed, but it is the master sending the ACK/NACK bits after each data byte. When the master sends an NACK bit (SDA high) on the ninth clock pulse of a given byte, this means that it wishes to stop receiving data from the slave. One of the most important features of the I2C protocol is called clock stretching [9, 13]. The slave device can hold the clock line low in case it is not ready to receive or send more data. When the slave holds the SCL line low, the master has to wait until it goes high to either process the data that the slave sent to it or send more data to the slave. Clock stretching is a feature that isn't used all the time, but it is handy when the slave device needs extra time to service interrupts or interpret the last received byte from the master [12,13].

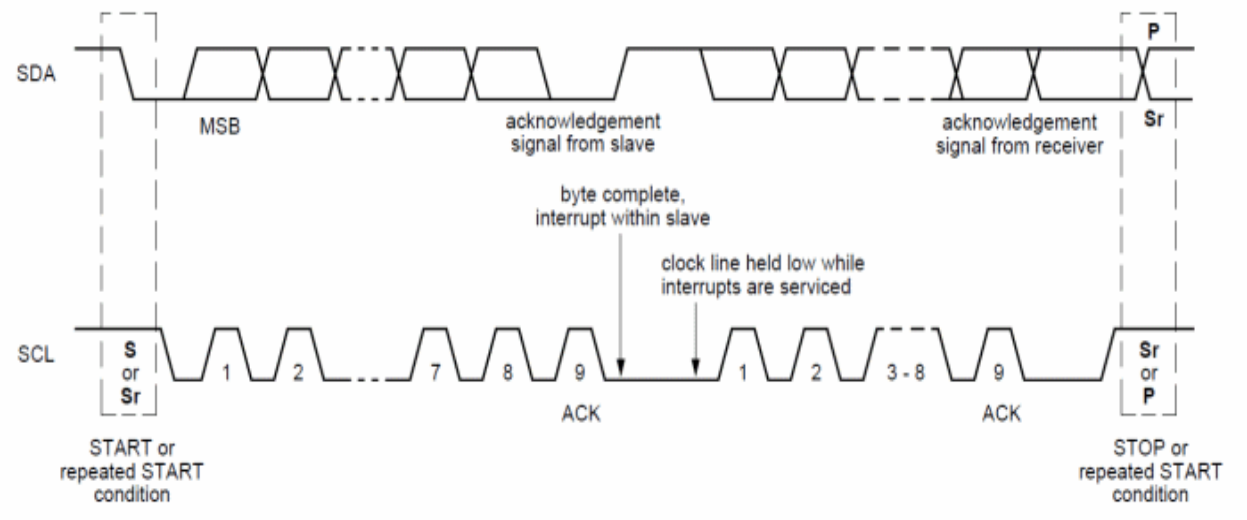

Fig 2.8 I2C Transaction with Clock Stretching [33]

Figure 2.8 shows the ability of the slave to stretch the clock when it isn't ready.

Bus arbitration occurs in a multi-master environment when two master devices issue a command at the same time on the I2C bus. The process of arbitration allows one of the two master devices to win and take control of the bus. When one of the devices tries to drive a line and sees that the signal on the SDA line is different from what it intended, it knows that it has lost arbitration and it backs off until it detects a STOP condition on the bus. If a master wants to write data and then read data from a slave device, it can issue a repeated start condition after the last write. This ensures that the master can retain control of the bus. It then must retransmit the address of the slave and indicate a read this time to move forward with the read transaction.

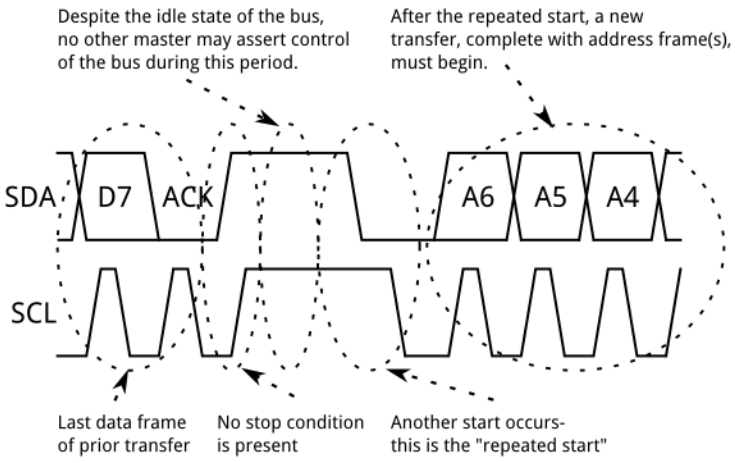

Fig 2.9 I2C Bus Repeated START Condition [14]

Figure 2.9 shows a repeated start condition on the bus which occurs when the master wants to talk to the same slave but change operation.

The I2C protocol also offers a 10-bit addressing mode in which two frames of data are required to address the device in question. The first frame contains the bits '11110ABR' where A and B are the two most significant bits of the 10-bit address and R is the read/write bit. The next eight bits of the address follow in the next byte, and the slave responds with an ACK if its address matches [13].

2.3 – SPI Communication

SPI, or serial peripheral interface, is a full-duplex serial synchronous communication protocol that was developed by Motorola to be able to connect devices without having addressing being the mode of device selection. The protocol calls for four signal lines: MOSI (Master Out Slave In), MISO (Master In Slave Out), SCK (Clock), and SS (Slave Select). In order for a master to communicate with a slave device, the slave select line connected to that particular slave needs to be pulled low. Generally, SPI is used as a single master, multiple slave bus where each slave needs its own slave select line. If using only one slave, one can have the slave select always pulled low and keep the slave toggled [9].

In order to communicate as a SPI master, the master must configure the clock to a frequency that can be supported by the particular slave device it wishes to communicate with. Once the clock is configured and the slave select line is pulled low, the slave device is toggled. The master puts out the clock on the SCK line and during each clock cycle, one bit is transferred from the master to the slave and from the slave to the master [16]. Communication is always full duplex, you cannot receive data without sending data and vice versa. The hardware is generally set up such that there is an 8-bit shift register in both the master and the slave. The output of the master (MOSI) is connected to the shift register in the slave and the slave's output (MISO) is connected to the master's shift register. By the end of eight clock cycles, the registers should have exchanged values, completing a full duplex byte transmission/reception [12].

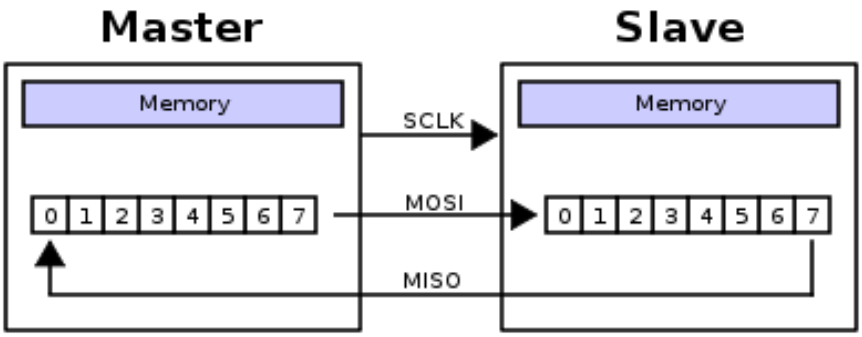

Fig 2.10 SPI Shift Registers [21]

Figure 2.10 shows the simultaneous shifting of data in and out of the master and slave registers.

Though one byte transmissions are common, depending on the device, word transmissions can also take place [18]. Clock polarity and phase are two configurable settings for the SPI master to set up; these can be referred to as CPOL and CPHA respectively. When CPOL is set to 0, the clock is active high and when it is set to 1, the clock is active low. When CPHA is set to 0, data is captured on the clock's idle to active edge and output on the active to idle edge. When CPHA is set to 1, data is captured on the clock's active to idle edge and output on the idle to active edge [17,20].

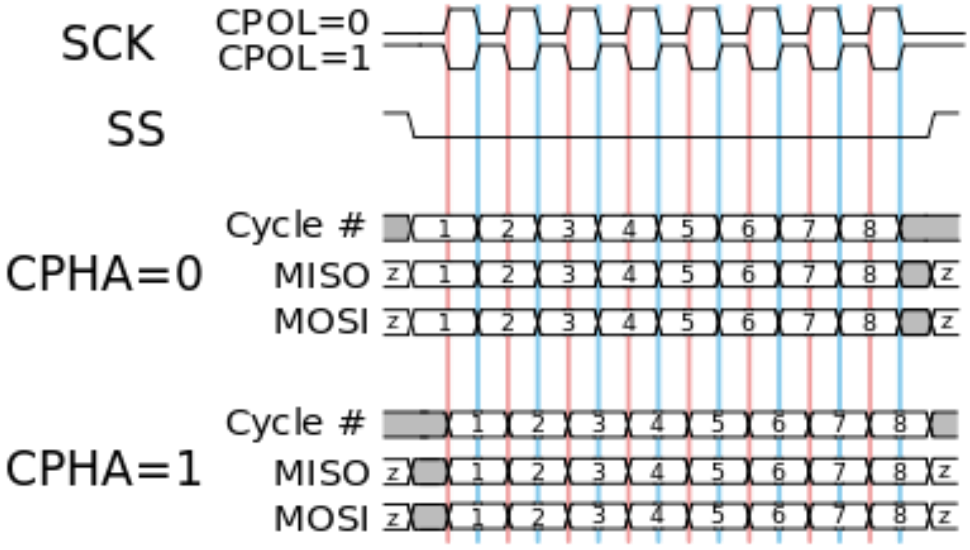

Fig 2.11 SPI Clock Polarity and Phase [21]

Figure 2.11 shows the different CPOL and CPHA combinations.

When the master initiates communication with the slave, the slave should have data preloaded in its transmit register in order to have it sent simultaneously. Usually, to support this, the master will send a command and the necessary number of clock cycles afterwards to allow the slave to send its response. Read-only and write-only operations are not possible; dummy bytes can be used to facilitate transfers where only half-duplex communication is desired. For example, if a master wants to receive data from the slave but has nothing to send, it will load its register with a dummy byte [19].

When multiple slaves are connected on the SPI bus in normal configuration, each one of them has a slave select line. It is recommended that when implementing SPI without a module on board, pull-up resistors should be used between the slave select lines and the device to reduce crosstalk [15].

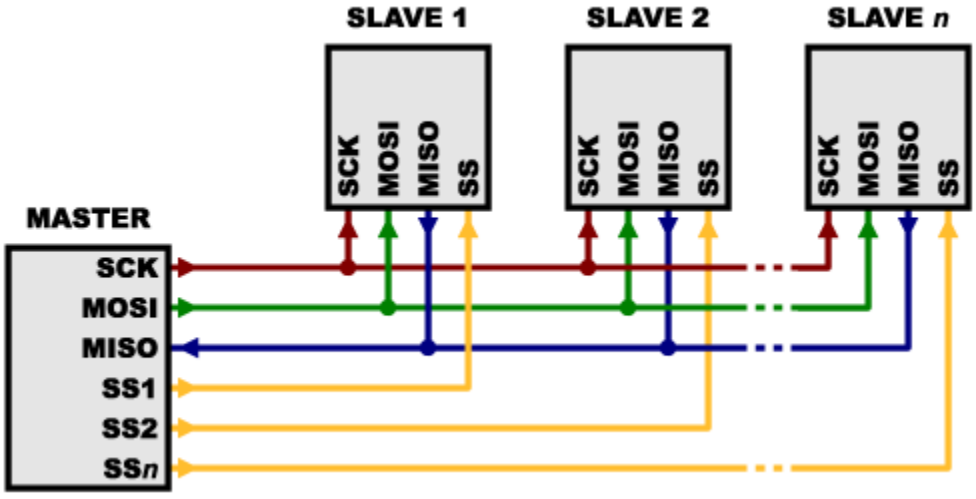

Fig 2.12 SPI with Multiple Slaves [15]

Figure 2.12 shows a SPI master connected to multiple slaves via multiple slave select lines.

A daisy-chain configuration, which is not as commonly used, is when the master's output is going to the first slave, and the slave's output is going to the next slave on the chain. The last slave's output finally connects to the master's input, so data from the master will reach all slaves.

2.4 – Comparison

Protocol	*Speeds*	*Pins*	*Advantages*	*Disadvantages*
Basic Asynchronous - UART	9.6 kHz, 19.2 kHz, 115.2 kHz (beyond this it is device specific)	2	- Low pin count - very easy to bit-bang - no need for a clock - full-duplex - parity for basic error checking - supported by almost all devices	- throughput wastes on synchronization bits - only two devices can be involved in communication - both devices must be configured to the same speed beforehand
SPI	No speed limit defined, depends on the device.	4	- highest throughput of the three - no addressing needed for multiple slaves - full-duplex - easy to implement - unidirectional signals - very flexible; no defined message size	- more pins - multiple slave select lines needed for multiple slaves - no flow control for the slave - multiple masters difficult to implement - no acknowledgement - good only for short distances
I2C	100 kHZ, 400 kHz, 1 MHz, 3.4 MHz, 5 MHz	2	- ability to have many slaves with just 2 pins - supports a multi-master environment - clock stretching ability for slave devices - offers advanced features to avoid collisions	- limited device address space - complex protocol - relatively slow protocol except when on high speed mode (not supported by all devices) - bus sharing can lead to starvation - too many error possibilities

Table 2.1 Protocol Comparison

UARTs are commonly included in microcontrollers for a variety of peripheral modules such as Bluetooth, GPS, and several sensors. SPI has carved out a solid role in the embedded computing world with applications in sensors, control devices, communications, memory access such as flash, EEPROM and SD cards. The reason it has grown so popular is because it has no speed limitations and works very well for short distance communications, allowing microcontrollers to interface with close proximity peripherals with high performance. SPI is a good choice for applications that don't call for several peripherals so we don't need to use many slave select lines. I2C has several applications as well, being popular for EEPROMs, ADCs, LCD displays, real-time clocks, and sensors. I2C is a great application to choose when several sensors need to be interfaced and very high throughput isn't needed. Its main drawback is its complexity, which is why students tend to stay away from it if library support isn't available [11,12,13].

2.5 Bluetooth Low Energy

Bluetooth low energy, or BLE, is a wireless network standard that was introduced by the Bluetooth Special Interest Group in 2010 to assist new applications in the embedded computing field that include fitness and home entertainment/automation among others. The goal of BLE is to provide an energy efficient wireless module with decent throughput. BLE is supported by several mobile operating systems such as Android, iOS, Windows Phone, and Blackberry as well general purpose operating systems such as macOS, Linux, and Windows 10. This allows for a design that uses BLE to span several different platforms without major roadblocks. BLE is used mainly for very short range communication applications, and generally provides a throughput in the order of kbps with most applications ranging between 10 and 100kbps [25,26].

BLE devices can be categorized either as central or peripheral devices, where central devices are usually CPUs or mobile phones with higher processing power. Advertising packets

are sent by all BLE devices in order to be seen by other devices and scan response data packets are sent in response to data requests by listening devices. Two types of communication can be established between BLE devices: broadcasting and connection-based communication. Broadcasting is where data is sent out to all listening devices, whereas connection-based communication is where a permanent data exchange path is established between two BLE devices. The central device acts as the master in a connection from initiating the connection itself to managing timing and data exchanges. Connection events dictate when data is to be exchanged between the peripheral and the central device; this allows for power saving as the devices can be dormant until the next connection event.

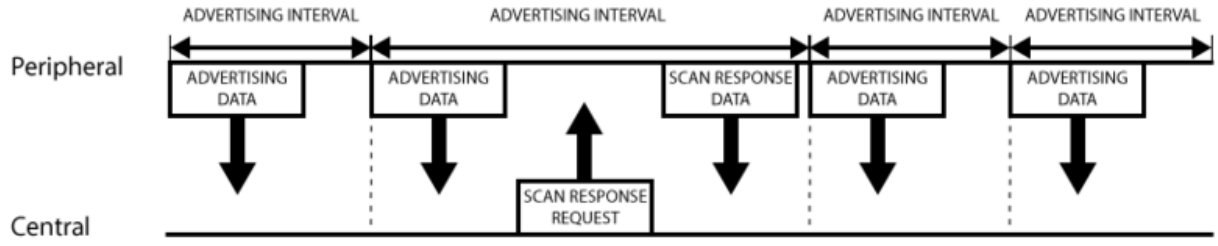

Fig 2.13 BLE Packets [27]

Figure 2.3 shows BLE packets being exchanged between a peripheral and central device.

To help facilitate BLE communication among different devices, different protocol layers are defined with different purposes that allow the devices to function properly. These layers can be grouped together into three different blocks, the application, the host, and the controller [22,23,24]. The application block contains the logic and user interface for the application itself. The host contains different layers that allow for the software handling of data coming through

the controller block, and the controller block is responsible for the physical handling of the data packets and interfacing to upper layers.

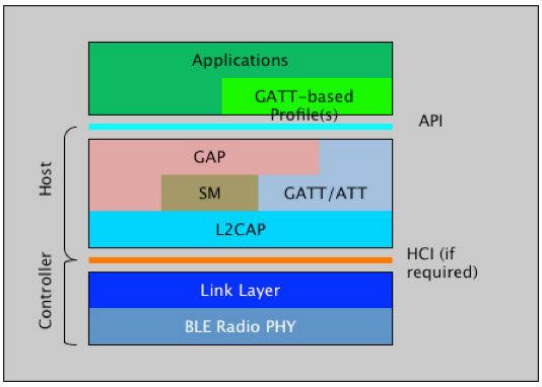

Fig 2.14 BLE Layers [26]

Figure 2.14 shows the different layers in the BLE protocol stack.

The controller block contains the physical layer which consists of circuitry responsible for modulating and demodulating analog signals and transforming them into digital symbols [26]. Modulation is the process of varying of specific properties of the message waveform. The controller block also in charge of allocating the frequency for a particular connection and maintaining connection channels. The link layer is directly above the physical layer and it is responsible for defining device roles, managing the device address, and filtering out packets to establish connections. It also handles timeouts between connection events and encryption. The host-controller interface allows the CPU to interface with the Bluetooth controller, usually via USB or UART [26, 27].

The lowest layer in the host block is the L2CAP (Logical Link Control and Adaptation Protocol) which handles the assembly and fragmentation of BLE packets to interface with higher layers. The attribute protocol, or ATT, layer is a client/server protocol that represents the attributes presented by the device in question [26,27,28]. Each server/device contains attributes

and in turn each attribute is identified by a 16-bit attribute handle, a unique UUID, permissions corresponding to it, and the value of the attribute itself. The handle is used to access the attribute value and the UUID specifies the type of data that the attribute holds. When a client requests data from the server/device, the value sent back is defined by the UUID of the particular attribute and when the client writes data to the server, data must be consistent with the format defined by the UUID. The next part in in the host layer is the generic attribute profile, or the GATT, which is responsible for the organization of data into profiles, services, and characteristics. A profile is a collection of services that is defined by the device designers or by the Bluetooth Special Interest Group. A service is a collection of characteristics and each service is distinguished by its UUID [26,27,28]. Each characteristic is a single data point that is directly interacted with when talking to a peripheral. For example, the Heart Rate Service contains the following characteristics: Heart Rate Measurement, Body Sensor Location, and Heart Rate Control Point [29]. The highest layer, the GAP, or generic access profile, is responsible for high level control involving device discovery, connection establishment, and security [28].

2.6 Zigbee

Zigbee is a communication standard and protocol that is a low cost and low power alternative to WiFi and BLE. Zigbee's wireless personal area networks operate at 868 MHz, 902-928MHz and 2.4 GHz frequencies with a data rate of 250 kbps [43]. There are three types of devices in a Zigbee network: an coordinating device, a router, and an end device. The coordinator handles the transmission and reception of data and is the device in control of the network. Routers are solely used as intermediate devices that route packets to and from the coordinator and end devices. End devices are usually sensors or peripherals to which the

coordinator sends commands and receives data. The arrangement of these devices depends on the topology of the network chosen for a particular application [43].

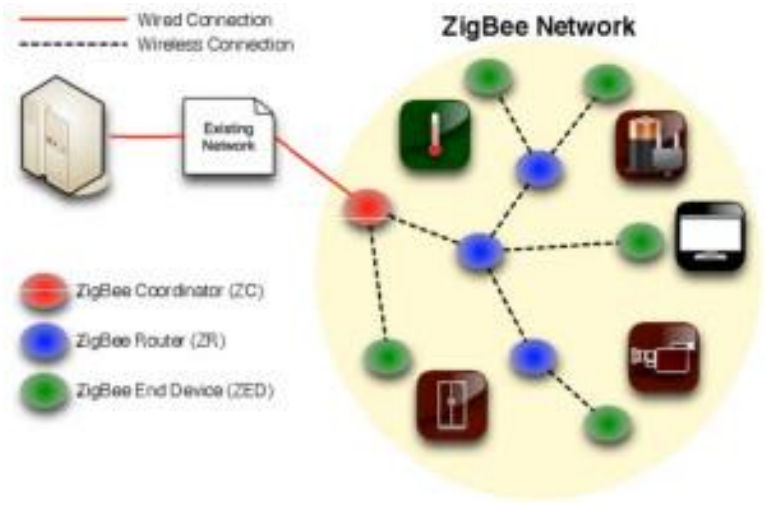

Fig 2.15 Zigbee Network [43]

Figure 2.15 shows an example of a Zigbee network with the three types of devices.

The Zigbee protocol stack is built on top of the stack defined by the IEEE 802.15.4 standard. The physical layer is responsible for the modulation and demodulation of signals that are being transmitted and received. The MAC layer is responsible for reliable data transmission, collision detection, and the transmission of beacon frames. The network layer is responsible for the connection between a coordinating device and end devices as well the routing of data. The application layers take care of matching devices based on their services and needs as well as facilitating the retrieval of attributes within application objects [43]. One of Zigbee's operating modes is called non-beacon modes in which the coordinator and routers constantly monitor incoming data from the end devices. In beacon mode, the coordinator periodically sends beacon packets to the end devices via the routers [43]. The three main Zigbee network topologies include star, mesh, and cluster.

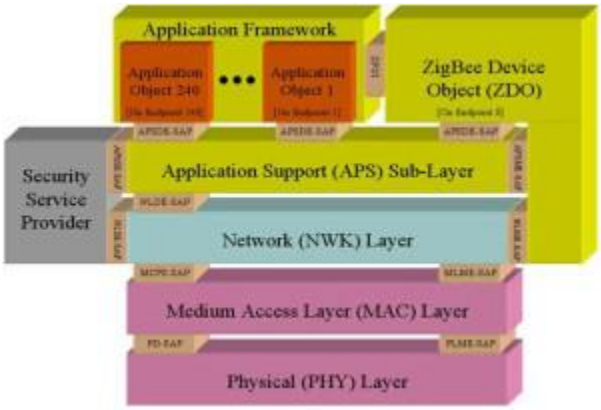

Fig 2.16 Zigbee Protocol Stack [43]

Figure 2.16 shows the Zigbee protocol stack, which is built on top of the IEEE 802.15.4 stack.

Chapter 3 – Design

3.1 – Hardware Platforms

To properly illustrate the communication protocols discussed earlier, this thesis utilizes two development boards that use popular microcontroller families. The first board chosen is an Arduino Uno (ATmega328P microcontroller) because of the open source support it has with module-specific libraries which enables easier testing for students starting out in embedded systems. The second board is a Mikromedia for PIC24EP which runs on a PIC microcontroller (PIC24EP512GU810). Implementing the protocols on this board will allow students to get a more in-depth perspective on how registers are manipulated in order to correctly interface peripherals.

3.1.1 – Arduino Uno (ATmega328P)

The Arduino Uno is an 8-bit microcontroller that runs at 16MHz frequency with an operating voltage of 5V. It has 14 digital I/O pins, 6 analog input pins, and 32kb of flash memory [31]. On top of this flexibility with a good number of I/O pins, the main reason this controller was chosen was because of the communication modules provided on board. UART TTL serial communication is provided on dedicated hardware pins as well as being available via a SoftwareSerial library that bit bangs using any two I/O pins available. SPI and I2C hardware peripheral modules are also available on board with library support to facilitate seamless communication. This board doesn't need a separate programmer, so the code can be dumped on using just the USB cable that also powers it. It is a great board for new embedded systems developers and tinkerers because of the relative ease it takes to start a new project and interface with a number of peripherals.

Fig 3.1 Arduino Uno [35]

3.1.2 Mikromedia for PIC24EP (PIC24EP512GU810)

The PIC24EP512GU810 microcontroller is a 16-bit microcontroller with a modified Harvard architecture that has a maximum operating speed of 70 MIPS and an operating voltage of 3.3V. The microcontroller has several other features, including nine 16-bit timers, two ADC modules, and 15-channel DMA that can be integrated into a variety of applications. The main reason this board was chosen was because of the abundance of serial communication peripheral modules that it possesses. The microcontroller has 4 hardware UARTs, 4 SPI modules, and 2 I2C modules that can be programmed and operated via a series of registers on board [30]. From a student's perspective, this microcontroller is more difficult to program, but it allows for a deeper understanding of the lower level details of the communication protocols we will discuss. If a student can grasp the procedures discussed for enabling communication on this microcontroller, using any chip from the PIC family will be very easy because of the similar configuration and operation mechanisms. This development board has an LCD screen that allows for easy debugging and displaying of received data on the screen.

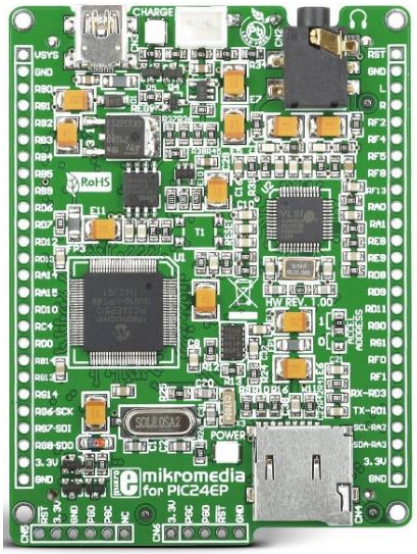

Fig 3.2 Mikromedia for PIC24EP [36]

3.2 - UART TTL Serial Communication

3.2.1 – Arduino Uno UART

Pins 0 and 1 represent RX and TX respectively. (Locate the correct pins on your Arduino model; usually they are pins 0 and 1.) Arduino's serial library can be used to initialize asynchronous serial communication settings as well as send and receive characters via pins 0 and 1 [37].

Arduino Serial Library [37]

- The Serial.begin() function sets up the UART with a baud rate of your choosing.
- Serial.write() writes data to the chosen serial port in the form of bytes. (Parameter: data to be written)
- Serial.print() writes data to the chosen serial port in the form of ASCII characters represented by the bytes. (Parameter: data to be written)
- Serial.available() returns the number of bytes available for reading (if available)

- Serial.read() returns the first byte available in the receive buffer
- Serial.readBytes() reads characters into a buffer specified as the first parameter with the second parameter being the number of bytes to read
- Serial.readBytesUntil() does the same thing as Serial.readBytes() with the additional constraint of a terminating character that is specified as the first parameter
- serialEvent() is a function that is called whenever data is available (RX interrupt); the body can be specified according to the application
- Serial.end() disables the UART

Below is a code snippet that echoes received characters: *serialArduinoecho.txt*

```
//basic serial communication echo program using Arduino Serial Library
//Srikar Chintapalli, 03/10/2017


void setup() {

Serial.begin(9600);                    //initialize serial
communication with a baud rate of 9600
Serial.println('X');                   // send the letter X
}

void loop() {
  if (Serial.available()) {            //check if data available on the
serial port
    Serial.println("Data:");
    Serial.write(Serial.read());       //echo the data byte received
  }
}
```

Below is a snippet that processes received data on interrupt using serialEvent():

serialEventArduino.txt

```
// echo program using serialEvent() from Arduino Serial Library
//Srikar Chintapalli, 03/10/2017

void setup() {
```

```
Serial.begin(9600);                    //initialize serial
communication with a baud rate of 9600
}

void loop() {

}

void serialEvent() {
    int rcv;
    if (Serial.available()) {          //check if data available on the
serial port
    rcv = Serial.read();               //store the received data in
'rcv' variable
    if (rcv == 5) {                    //process data here, prepare
response if you wish
      Serial.print('A');
    }
  }
}
```

The SoftwareSerial library has many of the same functions and it allows any two pins to be configured as RX and TX (limitations should be checked in the specific Arduino's datasheet). However, only one software serial port can be receiving data at a time, so the listen() function must be enabled each time if there are two or more software serial ports configured. Setting up software serial is as follows:

```
SoftwareSerial softSer1(8,9); // pin 8 is set as Rx and pin 9 as Tx
softSer1.begin(9600);         // enable the software serial port with
a baud rate of 9600
```

3.2.2 – PIC UART

This 16-bit microcontroller is not supported by Microchip's peripheral library, but it is easy to establish communication if the UART is set up properly. There are 4 hardware UARTs on this microcontroller, so the register definitions ahead will start with 'Ux' where the letter 'x' is to be substituted with the UART module number that is to be configured i.e. U1. UxMODE and

UxBRG registers are used to set up the parameters of the serial communication and the baud rate, respectively. The UxSTA is the UART status and control registers. The UxTXREG and UxRXREG are the transmit and receive registers used for communication [30,32].

UxMODE key bits:

- UARTEN: UART enable bit
- USIDL: UART stop in idle mode bit
- RTSMD: flow control or simplex mode for UxRTS
- ALTIO: alternate IO bits for UART
- UEN[1:0]: UART enable bits with 4 modes of operation
- WAKE: enable wake up on start bit during sleep mode
- LPBACK: internally connect TX to RX and enable loopback mode
- ABAUD: auto-baud enable bit
- URXINV: receive polarity inversion bit
- BRGH: high baud rate enable bit
- PDSEL: parity and data selection bits
- STSEL: stop selection bits

UxSTA key bits:

- UTXISEL[1:0]: UARTx transmission interrupt mode selection bits
- URXEN: receive enable bit
- UTXBRK: transmit break bit
- UTXEN: transmit enable bit
- UTXBF: transmit buffer full status

- TRMT: transmit shift register is empty bit
- ADDEN: address character detect bit
- RIDLE: receiver is idle bit
- PERR: parity error status bit
- FERR: framing error status bit
- OERR: receive buffer overrun error
- URXDA: receive buffer data available

To calculate the value to put into the UxBRG register, we need to decide on a specific baud rate. If we set up the BRGH bit to be 0, the maximum baud rate possible is $F_P/16$ and if the bit is 1 then the maximum possible baud rate is $F_P/4$ [32].

The value needed to load the UxBRG register can be calculated using the following formulas for BRGH = 0 and BRGH =1, respectively:

$$\text{UxBRG} = \frac{F_P}{16 * Desired\ Baud\ Rate} - 1$$

$$\text{UxBRG} = \frac{F_P}{4 * Desired\ Baud\ Rate} - 1$$

In these equations, F_P is the instruction cycle clock frequency, or $F_{OSC}/2$ where F_{OSC} is the oscillator frequency.

To initiate UART communication for UART1, the necessary steps are as follows [32]:

Assign the UART1 pins using peripheral pin select.

1) Set up the necessary U1MODE bits for the desired parity and data, and stop bits (BRGH, PDSEL[1:0], STSEL).

2) Initialize the U1BRG register to the necessary value for the desired baud rate.
3) If you want to use transmit and receive interrupts, enable them using the U1TXIE and U1RXIE bits in the IEC0 register.
4) Using the U1TXIP and U1RXIP control bits in the IPC3 and IPC2 registers respectively, set the interrupt priority levels for the transmit and receive interrupts (if enabled in the previous step) Select the desired transmit and receive interrupt modes using the UTXISEL[1:0] and URXISEL[1:0] bits in the USTA register.
5) Enable the UART1 module by setting the UARTEN bit in U1MODE.
6) Enable transmission and reception by setting the UTXEN and URXEN bits in the U1STA register.
7) Clear the U1TXIF bit in the IFS0 register and load the U1TXREG register to start transmission of data.
8) If interrupts are not enabled, the URXDA bit in the U1STA register can be polled to check for availability of data in the receive buffer.
9) If transmit and receive interrupts are enabled, the U1TXIF and U1RXIF bits in the IFS0 register need to be cleared by software in the interrupt service routines.
10) If there are parity, framing, or receive buffer overrun errors, the PERR, FERR, and OERR bits will be set in the U1STA register. To enable interrupts upon these errors, the U1EIE bit needs to be set in the IEC4 register, and the corresponding interrupt flag bit is the U1EIF bit in the IFS4 register.

Operation with UxRTS and UxCTS control pins [32]:

Assume the PIC is the DTE and the device on the other side is the DCE.

- The UxCTS (clear to send) pin, if used, is a DTE input that can be used to control UART transmission. It is a pin that is controlled by the DCE, and it is sampled by the DTE. When CTS goes low, the DTE starts transmitting from the transmit shift register.
- To activate simplex mode, the RTSMD bit of the UxMODE register is set to 1. In simplex mode, the DTE's UxRTS pin is connected to the UxRTS pin of the DCE. Similarly, the DTE's UxCTS pin is connected to the DCE's UxCTS. When the RTS of the DTE is driven high, that means that the DTE is ready to transmit data; the RTS of the DCE takes in this high data bit as an input. Then, when the DCE is ready to receive data, it drives its CTS pin low, which is connected to the DTE's CTS pin. The transition to low on the DTE's CTS pin tells it to start transmitting data.
- In simplex mode, the RTS is an output on the DTE side and an input on the DCE side. The CTS signal is an input to the DTE side and an output on the DCE side.
- To activate flow control mode, the RTSMD bit is set to 1. In the flow control mode, the RTS and CTS pins have the same direction on both the DTE and the DCE. When the DTE is ready to receive data, it drives the RTS line low; this low line is connected to the CTS input of the DCE and hence signals the DCE to start transmitting. The RTS pin on the DCE drives the CTS on the DTE in the same fashion.

Depending on the mode of operation chosen (UxMODE.UEN[1:0]), the RTS and CTS pins can either be controlled by the UART module itself using peripheral pin select, or they can be controlled using their assigned port registers [30,32].

A basic library to be used to set up and use the UART1 module : *uart1.h uart1.c*

```
/*
 uart1.h
```

```
 Srikar Chintapalli 02/01/2017
 These functions make use of the registers involved in the hardware
UART module on the PIC to facilitate transmission and reception of
data.
*/


#define RTS      _RE7   // Clear To Send, in, HW handshake
#define CTS      _RC1   // Request To Send, out, HW handshake


// initialize UART module
void InitUART1(void);

// send a character to UART1
int putUART1(int c);

// wait for and receive new character from UART1
char getUART1(void);

// send a string to UART1
void putsUART1(char *s);

// receive a null terminated string of length 'l' in a buffer
char * getsUART1(char *s, int l);
```

```
/*
 uart1.c

 Srikar Chintapalli 02/10/2017

 library functions to send and receive bytes of data using the UART1
Module on the:
 PIC24EP512GU810
*/

#include <p24EP512GU810.h>
#include "uart1.h"

#define RTSD    TRISEbits.TRISE7    // direction control for RTS pin
#define CTSD    TRISCbits.TRISC1    // direction control for CTS pin

// initialization settings
#define BRATE       26          // 9600 baud, BRGH=0, FOSC=8MHz,
FP=4000000
#define U1_ENABLE   0x8008      // load the U1MODE register and enable
the UART1 module: 8-bit data, no parity, 1 stop bit, CTS/RTS
controlled by port latches, flow control mode
```

```
#define U1_TX       0x0400     // load the U1STA register and enable
only transmission
#define U1_TX_RX    0x1400     // load the U1STA register and enable
transmission and reception

// initialize the UART1 module
void InitUART1(void)
{
    _ANSE5 = 0;                 //disable analog functionality of pins
    _ANSE6 = 0;


    _U1RXR  = 0x56;             //UART1 RX is mapped to RPI86 using
PPS
    _RP85R  = 0x01;             //UART1 TX is mapped to RP85 using PPS

    /* use only when the proper mode bits are set to hand over flow
control to hardware
    _U1CTSR = 0x31;             //UART1 CTS is mapped to RPI49 using
PPS
    _RP87R  = 0x02;             //UART1 RTS is mapped to RP87 using
PPS
    */

    _ANSE7 = 0;
    _ANSC1 = 0;
    RTSD    = 0;                // make RTS output
    CTSD    = 1;                // make CTS input
    RTS     = 1;                // set RTS default status

    U1BRG   = BRATE;
    U1MODE  = U1_ENABLE;        //enable the UART1 with settings
specified
    U1STA   = U1_TX_RX;


}

// send a character to the UART1 serial port
int putUART1(int c)
{
    while (CTS);                // wait for CTS to go low
    while (U2STAbits.UTXBF);    // wait while Transmit buffer full
    U2TXREG = c;
    return c;
}

// send a null terminated string to the UART2 serial port
void putsUART1(char *s)
{
    while( *s)                  // loop until *s == '\0' the  end of
the string
```

```
        putUART1( *s++);        // send the character and point to the
next one
}

// wait for and receive new character from the UART1 serial port
char getUART1(void)
{
    RTS = 0;                    // drive RTS low to indicate ready to
receive (in simplex mode this drives the CTS input of the other
device)
    while (!U2STAbits.URXDA);   // wait for a new character to arrive
    RTS = 1;                    // drive RTS high again until next
time of desired reception
    return U2RXREG;             // read the character from the receive
buffer
}



char *getsUART1(char *s, int l)
{
    char *p = s;                // copy the buffer pointer
    while (l > 0) {
        *s = getUART1();        // wait for a new character
        if(*s == '#'){break;}   //'#' is the delimiter

        s++;                    // increment buffer pointer
        l--;
    }                           // until buffer full
    *s = '\0';                  // null terminate the string

    return p;                   // return buffer pointer
}
```

3.3 - SPI Communication

3.3.1 – Arduino Uno SPI

Pins 10, 11, 12, and 13 are SS, MOSI, MISO, and SCLK respectively (locate the correct pins on your Arduino model) [38]. Arduino's SPI library can be used to initialize and facilitate SPI communication. The first step to take when using the Arduino to communicate with a peripheral using SPI is to check all of the peripheral's settings. The clock speed at which the

communication takes place must be supported by the peripheral. The clock polarity, edge at which data is output, and also the manner in which the data is shifted in and out of the registers all need to be verified in order to configure the communication interface properly.

Arduino SPI Library [38]:

- SPISettings() is the function used to set up the communication parameters of the SPI bus. The maximum speed at which transmission/reception will take place, the data order (MSB first or LSB first) and the data mode are all parameters. There are four data modes that are different combinations of clock polarity and clock phase (SPI_MODEx where x = 0,1,2,3). SPISettings() can be called inside SPI.beginTransaction() or an object of the SPISettings type can be passed.
- SPI.begin() initializes the SPI bus pins.
- SPI.beginTransaction() is used to begin the transaction using the settings specified by a SPISettings object parameter
- SPI.transfer() is the function that takes a byte to send on the bus and also returns the received byte.
- SPI.endTransaction() is used to relinquish control of the SPI bus
- SPI.end() disables the SPI bus

Below is the code snippet to operate an Arduino as a SPI bus master: *SPI_Arduino.txt*

```
//SPI read and write Arduino Uno
// Using the SPI library to operate the Arduino as a SPI bus master
//Srikar Chintapalli 03/01/2017

#include <SPI.h>

int slaveSelect = 10;
byte dummy = 0x0;
```

```
void setup() {

  SPI.begin();
  Serial.begin(9600);

}

void loop() {
  SPI.beginTransaction(SPISettings(1000000,MSBFIRST,SPI_MODE0));
  digitalWrite(10,LOW);
  SPI.transfer(0x10);
  char rcv = SPI.transfer(0);
  Serial.print(rcv);
  digitalWrite(10,HIGH);
}
```

The Arduino SPI library doesn't support slave operation, so we have to work with certain registers in the ATmega328P microcontroller to enable and use the Arduino as a slave [31].

SPCR – SPI Control Register

SPDR – SPI Data Register

SPSR – SPI Status Register

To operate the Arduino as a SPI bus slave, we need to set the MSTR bit of the SPCR register to 0 to set slave operation. We must also set the MISO line to an output and all others (MOSI, SS, SCK) to input. Once this is taken care of, we then enable the SPI module by setting the SPE (SPI enable) bit of the SPCR register to 1. We can then loop to check if the SPIF (SPI interrupt flag) bit in the SPSR register is set, and if it is, we can read the SPDR and return it to check what data has been received. These two actions will clear the SPIF flag bit. bit If interrupts are desired, the SPIE (SPI interrupt enable) bit in the SPCR should be set to 1, and this will set the SPIF flag bit whenever there is a completed serial transfer. An ISR (interrupt service routine) needs to be set up for this purpose, and hardware will clear the SPIF flag bit in that routine [31].

3.3.2 – PIC SPI

The main registers involved in setting up the SPI module are the SPIxSTAT, SPIxCON1, and SPIxCON2. The SPIxBUF register is the transmit/receive buffer register that the user reads from and writes to [30,34].

SPIxSTAT key bits [34]:

- SPIEN: SPIx enable bit
- SPISIDL: SPIx stop in idle mode bit
- SPIBEC[2:0]: SPIx buffer element count bits (valid in enhanced buffer mode)
- SRMPT: SPIx shift register empty bit (valid in enhanced buffer mode)
- SPIROV: SPIx receive overflow bit
- SRXMPT: SPIx receive FIFO buffer empty bit (valid in enhanced buffer mode)
- SISEL: SPIx buffer interrupt mode bits (valid in enhanced buffer mode)
- SPITBF: SPIx transmit buffer full status bit
- SPIRBF: SPIx receive buffer full status bit

SPIxCON1 key bits:

- DISSCK: disable SCKx pin bit
- DISSDO: disable SDOx pin bit
- MODE16: 8 or 16 bit wide communication selection
- SMP: SPIx data input sample phase
- CKE: SPIx clock edge select bit
- SSEN: slave select enable bit
- CKP: clock polarity select bit

- MSTEN: master mode enable bit
- SPRE[2:0]: secondary prescale bits (master mode)
- PPRE[2:0]: primary prescale bits (master mode)

SPIxCON2 key bits:

- FRMEN: framed SPIx support (SSx pin used as a frame sync pulse)
- SPIFSD: frame sync pulse direction control bit
- FRMPOL: frame sync pulse polarity bit
- FRMDLY: frame sync pulse edge select bit
- SPIBEN: enhanced buffer enable bit

The SPIxTXB and SPIxRXB are unidirectional 16-bit registers that share the SFR address of the SPIxBUF register. If the user writes to the SPIxBUF register (data to be transmitted), the data is written to the SPIxTXB register, and when the user reads data from the SPIxBUF register (data to be received), data is read from the SPIxRXB register. When enhanced buffer is enabled, the SPIxBUF becomes a gateway to two FIFOs that are 8 levels deep, one each for transmission and reception. The SPIx peripheral will transfer data from this register and queue transfers in the transmit FIFO; after each transfer, it populates the reception FIFO buffer with the received data [30,34].

Master Mode Using the SPI1 Module [34]:

1) Assign SPI1 pins using peripheral pin select. Remember in master mode to configure SCK1 as both an input and an output (use one of the pins that can be remapped as either). Clear analog functionality of the pins using the corresponding ANSELx registers.

2) Slave Select (SS1) is operated using the PORT and LAT registers
3) If you decide to use interrupts, enable the SPI1 interrupt flag using the SPI1IE bit in the IEC0 register, and set its priority using the SPI1IP[2:0] bits in the IPC2 register. The SPI1IF bit in the IFS0 register represents the flag.
4) If you want error interrupts every time the SPIROV bit is set in the SPI1STAT register, enable the SPI1EIF interrupt and set its priority as well.
5) Set desired settings using the SPI1CON1 and SPI1CON2 registers and then enable the master using the MSTEN bit in the SPI1CON1 register.
6) Clear the SPIROV bit in the SPI1STAT register.
7) Enable the SPI1 operation by setting the SPIEN bit in the SPI1STAT register.
8) Write to the SPI1BUF register; transmission and reception will start as soon as you do so.
9) Standalone reads from a slave can only be done by sending a dummy byte to the slave to initiate a read (a write from the slave side).

Slave Mode [34]:

1) Use the same steps as when setting up the module for master mode operations, except for a few key differences.
2) SCK1 is configured only as an input during peripheral pin selection.
3) MSTEN needs to be cleared and SSEN needs to be toggled to turn on slave select.
4) Once the SPIROV bit is cleared, the module can be enabled with SPIEN.
5) The key to slave mode is that the device is not at all in control of communication; it can only send data when the master initiates communication. Therefore, SPI1BUF needs to be preloaded with data to send to the master before the master initiates a transfer. If the slave need only read data, it should stay idle until the master transfers data.

In the enhanced buffer modes, both master and slave, a prescaler is used on the system clock and this prescaled clock is used as the serial clock. The SPIBEC[2:0] bits are a count of how many pending transfers/unread receptions remain, since up to eight elements of data can be loaded into the buffer. Slave and master modes in the enhanced buffer mode are set up and operate the same way from a user standpoint; the only difference is that the SPIBEN bit in the SPI1CON2 register needs to be set before the module is enabled [34].

Framed SPI modes are supported in the master and slave modes; in framed SPI, the SSx pin is used as a frame synchronization pulse input or output.

Frame synchronization works as follows [34]:

- Whichever module is in the framed mode, that module will initiate the frame synchronization pulse after the transmit data is written to the SPIxBUF register; at the end of the sync pulse is when data is transferred.
- SPI Master/Framed Master mode: SPIx module generates the clock and frame sync signals; the serial clock is continuously output at the SCK pin. When the frame sync pulse is driven active (SS pin), it stays active for one frame, and depending on the FRMDLY bit, the sync pulse either coincides with or comes before the data transmission.
- SPI Master/Framed Slave mode: SPIx module generates the clock, but the sync signal is generated by the slave; master has to load its SPIxBUF register before the sync signal arrives
- SPI Slave/Framed Master mode: the host module is the slave but it generates the sync signal, so it still takes the clock signal from the other SPI module
- SPI Slave/Framed Slave mode: the host module takes both the clock and the sync signal in as inputs from the other module.

SPI Master Operation Functions: *spi.h spi.c*

```
/****
spi.h

This library allows for the use of the onboard SPI hardware modules
and modifies registers to allow transmission and reception of data.
Srikar Chintapalli 03/20/2017
*****/

#define SS1     _RF8     //in master mode, slave select can be driven
using ports
#define DUMMY   0x00     // dummy byte

//initialize SPI with desired register settings
void SPI1_Init(void);


void SPIPut(int data);


char SPIGet(void);
```

```
/***********
spi.c
Srikar Chintapalli

library functions to send and receive bytes using the SPI1 module on
the:
PIC24EP512GU810

MASTER OPERATION
************/
#include <xc.h>
#include "spi.h"

#define SS1Dir              TRISFbits.TRISF8
#define SPI1Enable          0x8000              // enable SPI1 module
(set the SPIEN bit and clear the SPIROV bit)
#define MasterEnable        0x0022              // enable master mode,
disable the SSEN bit, set the SMP, CKE, and CKP bits as desired;
prescalers set 8:1 and 4:1
#define FramedModeDisable   0x0000              // disable framed mode
and enhanced buffer mode


void SPI1_Init(void)
{
```

```
    _ANSG6 = 0;

    _ANSG7 = 0;
    _ANSG8 = 0;

    _SCK1R =  0x76;  //SPI1 SCK is mapped to RP118 using PPS, RG6
(SCK input mapping)
    _SDI1R =  0x77;  //SPI1 SDI is mapped to RP119 using PPS, RG7

    _RP118R =  0x06;  //SCK output mapping
    _RP120R = 0x05;  //SPI1 SDO is mapped to RP120 using PPS , RG8

    SS1Dir = 0;  // make slave select an output
    SS1 = 1;     //disable slave (default)

    SPI1CON1 = MasterEnable;
    SPI1CON2 = FramedModeDisable;
    SPI1STAT = SPI1Enable;

}

//commence communication by sending (and receiving a byte)
void SPIPut(unsigned char data)
{
    unsigned char dummy;
    if (SPI1STATbits.SPIRBF = 1) {      //if you don't care what was
received last exchange, clear the SPIRBF bit by reading to a dummy
byte
    dummy = SPI1BUF;
    }

    SS1 = 0;               // pull slave select 1 low, enable device
on the other side
    // Wait for free transmit buffer
    while(SPI1STATbits.SPITBF);
    SPI1BUF = data;

    // Wait for data exchange to complete
    while(!SPI1STATbits.SPIRBF);
    SS1 = 1;              // pull slave select 1 high, disable device
on the other side
    return;

}

//retrieve the byte that was received when the data exchange
occurred(if you care about the received data)
unsigned char SPIGet (void)
{
    char recData = 0;
    recData = SPI1BUF;
```

```
    return recData;
}
```

3.4 I2C Communication

3.4.1 Arduino Uno I2C

Pins A4 and A5 are SDA and SCL respectively (locate the correct pins on your Arduino model). Arduino's Wire library can be used to initialize and facilitate I2C communication [39].

Arduino Wire Library [39]:

- The Wire.begin() function initiates the library and the device joins the bus. An address is needed as a parameter if joining the bus as a slave.
- The Wire.requestFrom() function allows the master to request bytes from a specific address. Parameters are the slave device address and the number of bytes to request. This function abstracts sending the start condition and address with a read request.
- After a call to the requestFrom() function, Wire.available() is used to check if any bytes are available to read, and Wire.read() returns these.
- The Wire.beginTransmission() function takes an address as a parameter and begins a write transaction. This function abstracts sending the start condition and address with a write request.
- Wire.write() is used after the beginTransmission() function to write bytes to the slave.
- Wire.endTransmission() is used to end the transmission that was initiated with beginTransmission().

- The onReceive and onRequest functions allow the slave device to set up routines to react to either requests or data sent by a master device. These routines will be used like interrupt service routines when operating as a slave.

Below is a code snippet to operate the Arduino as a I2C bus master: *I2C_Arduino_Master.txt*

```
#include <Wire.h>
//this snippet makes use of the Arduino Wire Library to operate the
Arduino as a I2C bus master
// Srikar Chintapalli 04/01/2017

void setup() {
    Wire.begin();        //join I2C bus as a master
    Serial.begin(9600); //use serial monitor to debug and print
received characters
}

byte address = 0x05;
byte dummy = 0x00;
char c;

void loop() {
    Wire.beginTransmission(address);         //this effectively starts
communication with a slave by addressing it (parameter is slave
address)
    Wire.write(dummy);                       //write to the slave
device; parameter is byte written
    Wire.write(0x15);
    Wire.endTransmission();                  //ends transmission

    Wire.requestFrom(address, 4);            //request 4 bytes from
slave

    while(Wire.available()) {
        c = Wire.read();
        Serial.print(c);
    }

}
```

To use the Arduino as a I2C bus slave, we need to set up handlers for receptions and data requests. Below is a code snippet: *I2C_Arduino_Slave.txt*

```
//This snippet uses the Arduino Wire Library to operate the Arduino as
a I2C bus slave
//Srikar Chintapalli 04/01/2017
void setup() {
  Wire.begin(5);                    // join i2c bus as a slave with
address 5
  Wire.onReceive(rcvData);          // handler that receives data
  //Wire.onRequest(transmitData);   //handler that writes data upon a
data request
  Serial.begin(9600);               // start serial for output
}

void loop() {
  delay(100);
}

// function that acts as data reception handler

void rcvData() {
  while (Wire.available()) {        // check for available data
    char c = Wire.read();           // receive byte
    Serial.print(c);                // print the character to serial
monitor
  }
}

//function that acts as a data transmit handler upon request from
master
//void transmitData() {
//  Wire.write("abcdef");
//}
```

3.4.2 PIC I2C

The main registers involved in setting up the I2C module are I2CxCON, I2CxSTAT, I2CxMSK, I2CxADD and I2CxBRG. The registers involved in transmission and reception are I2CxRCV and I2CxTRN. There are two I2C modules on board and the pins are pre-defined; they cannot be changed except to alternate I2C pins also specified by the datasheet [33].

I2CxCON key bits:

- I2CEN: I2Cx enable bit

- I2CSIDL: I2Cx stop in idle mode bit
- SCLREL: in slave mode, this is the SCL release control bit
- A10M: 10-bit slave address bit (7-bit address if bit is cleared)
- DISSLW: disable slew rate control bit
- GCEN: in slave mode, general call enable bit
- STREN: in slave mode, clock stretch enable bit
- ACKDT: acknowledge data bit in master mode
- ACKEN: acknowledge sequence enable bit in master mode
- RCEN: receive enable bit in master mode
- PEN: stop condition enable bit in master mode
- RSEN: repeated start condition enable bit in master mode
- SEN: start condition enable bit in master mode

I2CxSTAT key bits:

- ACKSTAT: acknowledge status bit in master transmit operation
- TRSTAT: transmit status bit in master operation that is cleared upon transmission of a byte and and reception of the acknowledge corresponding to that byte
- BCL: bus collision detected bit (during master operation)
- GCSTAT: general call status bit
- ADD10: 10-bit address status bit
- IWCOL: write collision status bit that is set if a write is attempted and failed when the module is busy
- I2COV: I2Cx receive overflow bit
- D_A: indicates whether the last received byte was data or an address (in slave mode)

- P: stop bit detected last
- S: start bit detected last
- R_W: read/write information bit based on address byte received (in slave mode)
- RBF: receive buffer full status bit
- TBF: transmit buffer full status bit

The I2CxRCV and I2CxTRN registers are used in both the master and slave modes of operation to receive and transmit data respectively. Operation of the I2Cx module can be carried out using either polling or interrupts. If interrupts are used, there are three flags that are set upon the occurrence of certain events. The MI2CxF is set upon ***completion*** of some master related events: start condition, stop condition, byte transmitted/received, acknowledge transmitted, repeated start, and detection of a bus collision event. The SI2CxIF is set upon ***detection*** of some events in the slave mode: start condition, stop condition, repeated start condition, valid device address detection, ACK/NACK reception from the master to start/stop transmitting data, and reception of data [30,33].

These interrupts can be enabled, given priorities and flag-checked using the proper registers.

In the case of the I2C1 module:

- Flags: MI2C1IF and SI2C1IF bits in the IFS1 register
- Interrupt enabling: MI2C1IE and SI2C1IE bits in the IEC1 register
- Interrupt priority setting: MI2C1IP[2:0] and SI2C1IE[2:0] in the IPC4 register

In order to set up I2C communication, whether communicating as a master or a slave, the I2CxBRG register needs to be set with the correct reload value that will derive the I2C system SCLx clock. The formula is as follows [33]:

$$\text{I2CxBRG} = \left(\left(\frac{1}{F_{SCL}} - Delay\right) * F_{CY}\right) - 2$$

The 'Delay' part of the equation is known as the pulse gobbler delay, and it is usually between 110ns and 130ns (check data sheet for exact value; if you can't find one, use 120 ns).

F_{SCL} is the I2C SCL clock frequency: this is usually set at 100 kHz, 400 kHz, or 1 MHz; this is what we will be trying to achieve by changing the values we set to I2CxBRG.

F_{CY} is the processor instruction cycle clock frequency, which is $F_{OSC}/2$ where F_{OSC} is the primary configured oscillator frequency [33].

Example:

If we want to achieve a F_{SCL} of 100 kHz using an estimated pulse gobbler delay of 120 ns and a F_{CY} of 4 MHz,

$$\text{I2CxBRG} = \left(\left(\frac{1}{100000} - (1.2 * 10^{-7})\right) * 4000000\right) - 2$$

$$\text{I2CxBRG} = 37.52$$

We would then load I2CxBRG with either 37 or 38 to get as close to a F_{SCL} of 100 kHz as possible.

To successfully initiate and carry out communication as a single master on the I2C bus using the I2C1 module, these are the steps that need to be followed [33]:

The I2C module completes individual parts of the protocol, but the responsibilities of the correct ordering and checking of the completion of these parts fall on the user software.

1) Enable master driven interrupts (MI2C1F) if desired using the bits and registers detailed earlier.
2) Disable I2C1 module by clearing the I2CEN bit.
3) Load the I2C1BRG register with the desired value acquired from the formula detailed earlier.
4) Set I2C1CON to a known state by disabling everything; use 0x1000.
5) Enable the I2C1 module by setting the I2CEN bit. The module will override and take control of the I/O pins (SDA, SCL) necessary for communication; there is no need to map these pins using peripheral pin select.
6) Before ANY operation, check to see if the I2C bus is idle; poll the I2C1CON register to see if PEN, SEN, RSEN, RCEN are low and see if the TRSTAT bit in the I2C1STAT register is low.
7) Set the SEN bit in the I2C1CON register to assert a start condition on the bus and wait for the operation to conclude (this can be done by polling the SEN bit or using interrupts in which case MI2C1IF will be set upon the finish of the start condition).
8) Check to see if the transmit buffer is full by polling the TBF bit in the I2C1STAT register; if it is set, wait until it is cleared.
9) Send the address byte of the slave device and either a read or write bit, depending on what the master operation will be. Write the address to the I2C1TRN register.

10) Wait until the transaction is complete by polling the TRSTAT bit in the I2C1STAT register. Then check the ACKSTAT bit in the I2C1STAT register to see if the slave responded with an acknowledge.
11) If the operation bit specified along with the address byte in step 9 was a write bit, poll to see if the transmit buffer is full, send the byte that you wish to send, and check for the completed transaction and slave acknowledge as in steps 8-10.
12) If the operation bit specified along with the address byte in step 9 was a read bit, set the RCEN bit in the I2C1CON register to enable master reception. Set the ACKDT bit in the I2C1CON register to set up the transmission of an ACK.
13) Poll the RBF bit in the I2C1CON register to see if it is set; if it is set, that means that data is available in the receive register.
14) Read the I2C1RCV register; this will clear the RBF bit. If the I2COV bit is set in the I2C1STAT register, this means there was an overflow; a byte was received without us reading the previously received byte.
15) Once you read the I2C1RCV register, set the ACKEN bit in the I2C1CON register to enable the acknowledge sequence. Wait until the sequence is completed by polling the same bit until it goes low.
16) Continue reception with steps 12-15 until necessary, and on receiving the last desired data byte, clear ACKDT to make it low before setting the ACKEN bit; this will send a NACK sequence and let the slave know to stop sending data.
17) To indicate end of communication, initiate the stop condition by setting the PEN bit in the I2C1CON register. Poll the same bit until it is cleared to make sure the operation has concluded.

When switching from a read sequence to a write sequence or vice versa, after the first sequence has concluded, the user must do the following: A) initiate a restart sequence by setting the RSEN bit in the I2C1CON register. B) Poll the same bit until it is cleared to make sure the operation has concluded. C) Once it has concluded, write the address of the slave with the new operation bit (read or write) to the I2C1TRN register to start the next sequence [33].

Below is a set of functions written in order to make operation as an I2C bus master easy: *i2c1.h*

i2c1.c

```
/*
 i2c1.h
 The following library allows the user to transmit and receive data
with the PIC on the I2C bus by making use of the registers associated
with the I2C1 hardware module.

 Srikar Chintapalli 04/10/2017

 Master operation
 */



void I2C1Init(void);
void Delay100uS(int num);
int I2C1Idle(void);
int  I2C1Start(void);
int  I2C1Stop(void);
int  I2C1Restart(void);
int  I2C1Send(char);
int  I2C1Receive(int);
```

```
/*
i2c1.c

Srikar Chintapalli

PIC24EP512GU810

I2C master operations on I2C1 module
*/
#include <xc.h>
#include "i2c1.h"


void I2C1Init(void);
```

```
void Delay100uS(int num);
int I2C1Idle(void);
int  I2C1Start(void);
int  I2C1Stop(void);
int  I2C1Restart(void);
int  I2C1Send(char);
int  I2C1Receive(int);



//initializes the I2C1 module with a FCL of 100 kHz (master operation)

void I2C1Init(void)
{
    I2C1CONbits.I2CEN = 0;                  //Disable until everything
set up is complete
    I2C1CON = 0x1000;                       //continue operation in
idle mode, release SCL clock, disable IPMI mode,
                                            //disable 10 bit address,
enable slew rate control, set all conditions to known states (0)
    I2C1BRG = 38;                           //To achieve a FSCL of
100kHz where FCY = 4000000 (formula is in documentation)
    I2C1CONbits.I2CEN = 1;                  //Enable I2C1 module, pins
will be taken care of by module; no need for peripheral pin select
    //IEC1bits.MI2C1IE = 1;                 //Enable I2C master
interrupt if you want to use interrupts
}


void Delay100uS(int num)
{
    T1CON = 0x8000;                         //enable TMR1 with 1:1
prescaler, clock used is FCY = 4000000
    while (num > 0)                         //wait for specified
number of multiples of 100 microseconds
    {
        TMR1 = 0;
        while ( TMR1 < (4000000/10000));    //wait 100 microseconds
        num = num-1;
    }
}

//checks if master is busy
int I2C1_Idle(void) {
    if ((I2C1CONbits.SEN == 1) || (I2C1CONbits.PEN == 1) ||
(I2C1CONbits.RSEN == 1) || (I2C1CONbits.ACKEN == 1) ||
(I2C1CONbits.RCEN == 1) || (I2C1STATbits.TRSTAT == 1)) {
       return 1;
    } else {
        return 0;
    }
}
```

```
//Initiates start sequence on I2C bus
//Returns: 0 for success, 1 for busy bus, 2 for timeout, and 3 for bus
collision
int I2C1Start()
{

    if (I2C1_Idle() == 1) {
      Delay100uS(1);
      if (I2C1_Idle() == 1) {
        return 1;                           //bus is busy even after
100 microsecond delay
      }
    }

    I2C1CONbits.SEN = 1;                    //Initiate the start
condition on the bus

    int t = 10;                             // timeout number = 10;
10*100 microseconds = 1 millisecond

    while(I2C1CONbits.SEN)                  //hardware clears when
complete
    {
        Delay100uS(1);                      //wait 100 microseconds
each time; specify number of iterations above
        t--;
        if(t == 0)
        {
            return 2;
        }
    }

    if(I2C1STATbits.BCL)                    //test for a bus collision
and report back
    {
        I2C1STATbits.BCL = 0;               //Clear bus collision
error
        return 3;
    }

    return 0;                               //if we made it to this
stage, the start condition was successfully executed
}

//Initiates stop sequence on I2C bus
//Returns: 0 for success, 1 for busy bus, 2 for timeout, and 3 for bus
collision

int I2C1Stop()
{
```

```
    if (I2C1_Idle() == 1) {
      Delay100uS(1);
      if (I2C1_Idle() == 1) {
        return 1;                               //bus is busy even after
100 microsecond delay
      }
    }

    I2C1CONbits.PEN = 1;                        //Initiate the stop
condition on the bus

    int t = 10;                                 // timeout number = 10;
10*100 microseconds = 1 millisecond

    while(I2C1CONbits.PEN)                      //hardware clears when
complete
    {
        Delay100uS(1);                          //wait 100 microseconds
each time; specify number of iterations above
        t--;
        if(t == 0)
        {
            return 2;
        }
    }
    return 0;                                   //if we made it to this
stage, the start condition was successfully executed
}

//Initiates repeated start sequence on I2C bus
//Returns: 0 for success, 1 for busy bus, 2 for timeout, and 3 for bus
collision
int I2C1Restart()
{
    if (I2C1_Idle() == 1) {
      Delay100uS(1);
      if (I2C1_Idle() == 1) {
        return 1;                               //bus is busy even after
100 microsecond delay
      }
    }

    I2C1CONbits.RSEN = 1;                       //Initiate the start
condition on the bus

    int t = 10;                                 // timeout number = 10;
10*100 microseconds = 1 millisecond

    while(I2C1CONbits.RSEN)                     //hardware clears when
complete
    {
```

```
        Delay100uS(1);                          //wait 100 microseconds
each time; specify number of iterations above
        t--;
        if(t == 0)
        {
            return 2;
        }
    }

    if(I2C1STATbits.BCL)                        //test for a bus collision
and report back
    {
        I2C1STATbits.BCL = 0;                   //Clear bus collision
error
        return 3;
    }

    return 0;
}

//Sends a data byte to a slave
//Returns: 0 for ACK, 1 for NACK, 2 for busy bus, and 3 for timeout, 4
for bus collision
int I2C1Send(char toSend)
{
    if (I2C1_Idle() == 1) {
      Delay100uS(1);
      if (I2C1_Idle() == 1) {
        return 2;                               //bus is busy even after
100 microsecond delay
      }
    }

    int t = 10;
    while(I2C1STATbits.TBF) {                   //wait while the transmit
buffer is full
        Delay100uS(1);                          //wait 100 microseconds
each time; specify number of iterations above
        t--;
        if(t == 0)
        {
            return 3;
        }
    }

    I2C1TRN = toSend;                           //Send data byte

    t = 10;                                     //timeout number = 10;
10*100 microseconds = 1 millisecond

    while(I2C1STATbits.TRSTAT)              //hardware clears when
complete
```

```
    {
        Delay100uS(1);                          //wait 100 microseconds
each time; specify number of iterations above
        t--;
        if(t == 0)
        {
            return 3;
        }
    }

    if(I2C1STATbits.BCL)                        //test for a bus collision
and report back
    {
        I2C1STATbits.BCL = 0;                   //Clear bus collision
error
        return 4;
    }


    if(I2C1STATbits.ACKSTAT)                    //check slave's response
after transmission complete
        return 1;                               //NACK
    else
        return 0;                               //ACK
}

//Reads a data byte from the currently addressed slave (read should
have been indicated in address byte)
//Returns:
int I2C1Receive(int done)
{
    int rcdata;

    if (I2C1_Idle() == 1) {
      Delay100uS(1);
      if (I2C1_Idle() == 1) {
        return 2;                               //bus is busy even after
100 microsecond delay
      }
    }

    if(done == 1)                               //if this is the last byte
you want to receive, call function with done = 1 so we can send NACK
to slave
    {
        I2C1CONbits.ACKDT = 1;//NACK
    }
    else
    {
        I2C1CONbits.ACKDT = 0;//ACK
    }
```

```
    I2C1CONbits.RCEN = 1;                   //Enable receive

    int t = 10;

    while(!I2C1STATbits.RBF)                //hardware clears when
receive is complete
    {
        Delay100uS(1);                      //wait 100 microseconds
each time; specify number of iterations above
        t--;
        if(t == 0)
        {
            return 99998;
        }
    }

    I2C1CONbits.ACKEN = 1;                  //Send ACK bit which must
be done after reception of every byte; send NACK on last byte

    t = 10;

    while(I2C1CONbits.ACKEN)                //hardware clears when
complete
    {
        Delay100uS(1);                      //wait 100 microseconds
each time; specify number of iterations above
        t--;
        if(t == 0)
        {
            return 99998;
        }
    }
    if(I2C1STATbits.BCL)                    //test for a bus collision
and report back
    {
        I2C1STATbits.BCL = 0;               //Clear bus collision
error
        return 99999;
    }

    if(I2C1STATbits.I2COV)                  //overflow error
(shouldn't occur if receive register is read each time)
    {
        I2C1STATbits.I2COV = 0;
    }
    rcdata = I2C1RCV;                       //clears the RBF bit;

    return rcdata;
}
```

When operating in a multi-master environment, bus collisions can occur during start conditions, repeated start conditions, stop conditions, or address/data/acknowledge bits. We can either poll for a bus collision by looking at the BCL bit in I2C1STAT register or we can enable interrupts, in which case one will be generated upon the collision happening. When using the polling method, the BCL bit should be polled after checking for the start/stop/repeated start/acknowledge condition bit goes low in the I2C1CON register because the bus collision will cause the condition to abort and its corresponding bit to go low. So if we check beforehand, the collision may not have yet occurred and the bit will still go low, giving the user software the false impression that the condition successfully completed [33].

Operating as a slave on the I2C1 bus:

1) If slave interrupts are desired, enable them by setting the registers specified above for the SI2C1IF interrupt.
2) Set up the I2C1CON register to your preference; either enabling or disabling clock stretch, stop in idle mode, 10-bit address (A10M), general call enable, and the IPMI bit. For this example, we will set the I2C1CON register to 0x1040 (enabling clock stretch and disabling general call enable, 10-bit address and IPMI).
3) Set the 7-bit address in the I2C1ADD register and if you want to use any mask bits, properly set the I2C1MSK register as well (IPMI must be disabled for masking).
4) Enable the I2C1 module by setting the I2CEN bit; the SDA and SCL pins will be controlled by the module.
5) As soon as the module is enabled, the slave device waits for a start condition to occur on the bus. When the start condition occurs, the S bit in the I2C1STAT register will be set.

6) The next 8 bits after the start condition is detected will be shifted into the I2C1RSR register and the address bits are compared using the I2C1ADD and I2C1MSK registers.
7) If there is an address match, the device sends an ACK on the bus automatically and the D/A bit is cleared, noting that the last byte received was an address that matched.
8) Based on what was received from the master, the R/W bit is set to a 1 if the slave is to output data to the master, and to a 0 if it is to accept data from the master.
9) If the master is writing to our slave device (R/W = 0), we can wait and see when the RBF bit will be set in the I2C1STAT register, indicating that the receive buffer is full; if interrupts are enabled, an interrupt will occur on reception of a byte. We can then read from the I2C1RCV register, which will clear the RBF flag. The D_A bit in the I2C1STAT register will also be high, indicating that the last byte received was a data byte.
10) We then set the SCLREL bit in the I2C1CON register to 1 to release the clock, and the master will continue to transmit until it has no more data to send.
11) If in step 8, the R/W bit was set to a 1, the master will be receiving data from the slave. Upon reception of the address byte with the R/W bit as 1, the module pulls SCLREL low, initiating a clock stretch.
12) At this point, we must load the I2C1TRN register (checking if the TBF bit is set first) and then set the SCLREL bit to release the clock, which initiates data transmission to the master. ACKs from the master will trigger a slave interrupt (if enabled).
13) Upon reception of a NACK from the master, the slave knows to cease communication, and it doesn't pull the SCLREL line low.

14) At the end of either a slave read or slave write sequence, a stop condition from the master can be detected; the P bit in the I2C1STAT register is set. This signals the end of the transaction.

If the GCEN (General Call Enable) bit in the I2C1CON register is set before slave operation, the GCSTAT bit in the I2C1STAT will be set any time the slave receives the general call address. This address consists of all 0's with the R/W bit also set to 0. In this scenario, even though the address doesn't match its own, the slave readies itself for the master to write data to it.

Below is code that can be used to operate in I2C slave mode [30,33]: *i2c1slave.h* *i2c1slave.c*

```
/*
 i2c1slave.h
 Srikar Chintapalli 04/20/2017
 This library allows for the PIC to be operated as a slave on the I2C
bus by making use of the registers associated with the I2C1 hardware
module.
 */
void I2C1SlaveInit(void);
void _ISR _SI2C1Interrupt(void);
```

```
/*
i2c1slave.c

Srikar Chintapalli

PIC24EP512GU810

I2C slave operation on I2C1 module
*/
#include <xc.h>
#include "i2c1slave.h"
#include "PICconfig_I2C.h"

#define __ISR  __attribute__((interrupt, shadow, no_auto_psv))


void I2C1SlaveInit(void);
void _ISR _SI2C1Interrupt(void);
```

```
int rcvbuffer[100];
int * bufptr = rcvbuffer;
unsigned char dummy = 0xFF;

//initializes the I2C1 module as a slave

void I2C1Init(void)
{
    I2C1CONbits.I2CEN = 0;                  //Disable until everything
set up is complete
    I2C1CON = 0x1040;                       //continue operation in
idle mode, release SCL clock, disable IPMI mode, enable clock
stretching
                                            //disable 10 bit address,
enable slew rate control, set all conditions to known states (0)
    I2C1ADD = 0x05;                         //set slave address
    I2C1CONbits.I2CEN = 1;                  //Enable I2C1 module, pins
will be taken care of by module; no need for peripheral pin select

    IEC1bits.SI2C1IE = 1;                   //Enable I2C slave
interrupt
    IFS1bits.SI2C1IF = 0;                   //clear flag


    while(1);
}


void _ISR _SI2C1Interrupt(void) {

    unsigned char tempstore;

    IFS1bits.SI2C1IF = 0;                   //clear the interrupt flag

    if (!I2C1STATbits.P) {                  //if the start condition
was the last to be detected
        if (!I2C1STATbits.D_A) {            //if the last byte
received was an address byte
            if (!I2C1STATbits.R_W) {        //if the master is writing
to the slave based on R_W = 0
                if (I2C1STATbits.RBF) {     //check to see if a byte
was received
                tempstore = I2C1RCV;        //since last received byte
was an address byte and it was acknowledged, clear the buffer
                I2C1CONbits.SCLREL = 1;     //release the clock so
that the master can continue to send
                }
            } else {                        //if the master is reading
from the slave based on R_W = 1
                I2C1TRN = dummy;            //here I am just feeding
dummy bytes to master, but you can choose what to send based on your
application
```

```
                I2C1CONbits.SCLREL = 1;     //release the clock to
allow master to clock in the data you just set up for transmission
            }
        } else {                            //if the last byte
received was a data byte (slave must be in receive mode because if it
was sending, D_A would be 0)
            if (!I2C1STATbits.R_W) {
                if (I2C1STATbits.RBF) {
                *bufptr = I2C1RCV;          //here I am just storing
received bytes in a buffer, but you can choose what to do based on
your application
                bufptr++;
                I2C1CONbits.SCLREL = 1;     //release the clock to
allow master to keep sending
                }
            }
        }
    }
}
```

3.5 Bluetooth Low Energy

Arduino Uno

The Bluefruit LE Shield by Adafruit is a great way to get started with Bluetooth Low Energy, as Adafruit has several modules that allow you to test out the different features that the shield offers. Once you have downloaded the Adafruit libraries and soldered the pins onto the shield as directed by the tutorial on Adafruit's website, you can begin to tinker with some of the main capabilities of the board [40].

We will go over some of the more important features and steps that will enable you to incorporate Bluetooth low energy into your embedded applications. The first thing you need to do is enable the serial communication bus of your choice by uncommenting the respective lines in the BluefruitConfig.h header file. The options include hardware and software SPI and UART, with the default option being hardware SPI. Once you've selected a mode of serial communication, you can open a series of examples that the library provides and test out their

functionality. There is also an application available for download called Bluefruit for both iOS and Android that allows interaction with the BLE shield via a central Bluetooth device.

AT Commands

The Bluefruit LE module can be run in one of two modes, either command mode or data mode. When in command mode, the module can be interacted with using a set of commands called AT commands. AT Commands are used to do everything from setting BLE transmit power levels and manipulating hardware pins to setting up new GATT profiles and characteristics for a specific application. An in-depth explanation of a list of AT commands can be found on the tutorial page for the Bluefruit LE shield on the Adafruit website [40].

The ATCommand example/sketch can be used to get a grasp of the different commands in the set. The commands you type and the responses from the BLE module are displayed on the serial monitor for easy access.

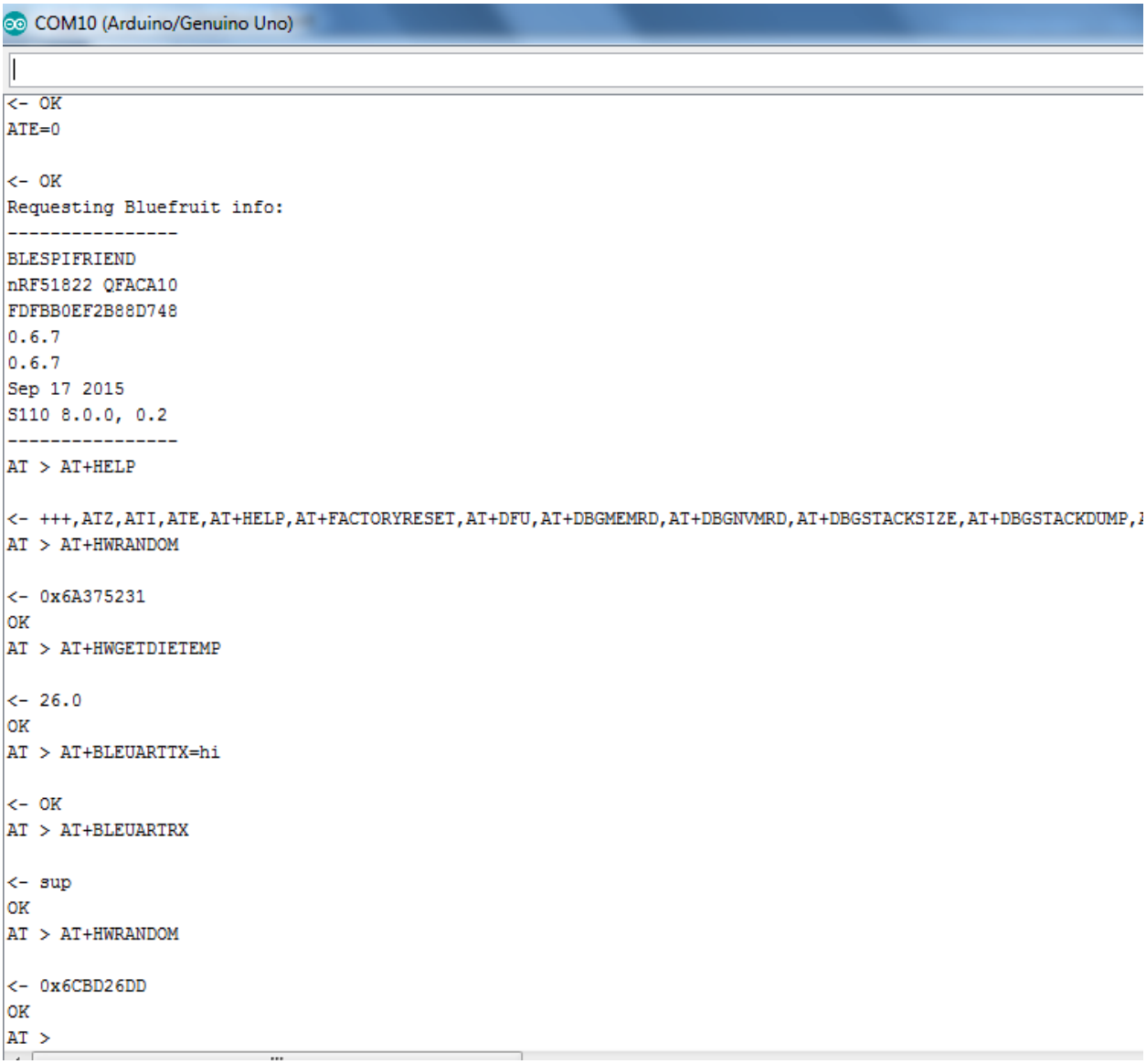

Fig 3.3 ATCommand Sketch on Serial Monitor

Figure 3.3 shows a series of AT Commands that each do different things. The AT+HELP command lists all the commands in the set, the 'AT+HWGETDIETEMP' command returns the temperature of the die on the BLE module, and 'AT+HWRANDOM' returns a random number generated by the module. When the app is downloaded, you can connect to the Bluefruit LE in UART mode and use 'AT+BLUEARTTX=' to send strings to your mobile device, and use

‘AT+BLEUARTRX’ to receive strings from the mobile device and display them on the serial monitor.

The BLEUart capability of the BLE shield is the one that we are most interested in; this allows us to send strings to and receive strings from a central BLE enabled device (mobile phone) via the Bluefruit application. This exchange of data can be facilitated in one of two modes, either command mode or data mode. Generally, command mode is used for occasional transfer of data over BLE to the central device, as it leaves flexibility for other commands. In command mode, we use ‘AT+BLUEARTTX=’ and ‘AT+BLEUARTRX’ within the ble.print() function to send data to the central device. When we can expect large amounts of data to be sent back and forth, we enable the BLE module in data mode. In this mode, the AT command doesn’t need to be used and we can directly use the strings we want to send as parameters to the ble.print() function [40].

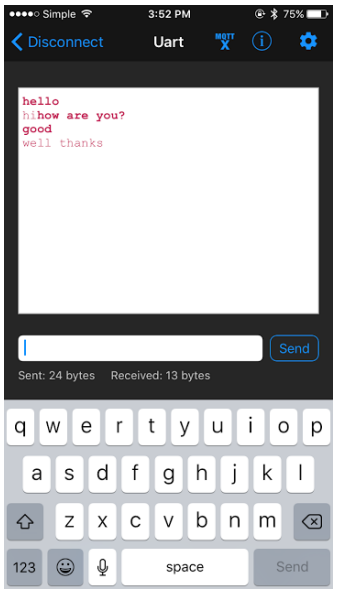

Fig 3.4 BLEUART on the Bluefruit Application Side

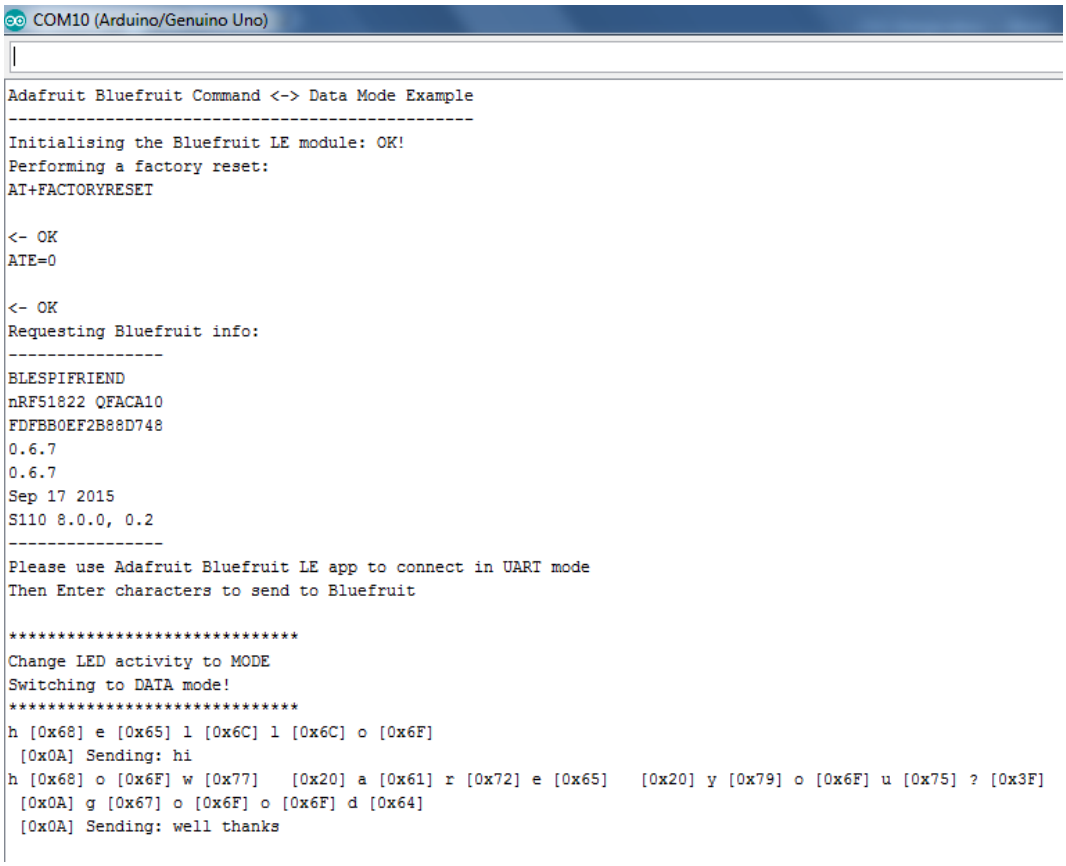

Fig 3.5 BLEUART on the Arduino Serial Monitor

Figures 3.4 and 3.5 show the BLEUART operation on the application and the Arduino sides.

The controller sketch available in the downloadable libraries allows you to stream sensor data from your central device via the data mode on the BLE module. The application Bluefruit allows you to select which data to send to the BLE module and gives you the ability to toggle it on and off. Available data include accelerometer and location data among others. On the side of the BLE module, we enable data mode to facilitate the quick transfer of data from the central device to the BLE module [40].

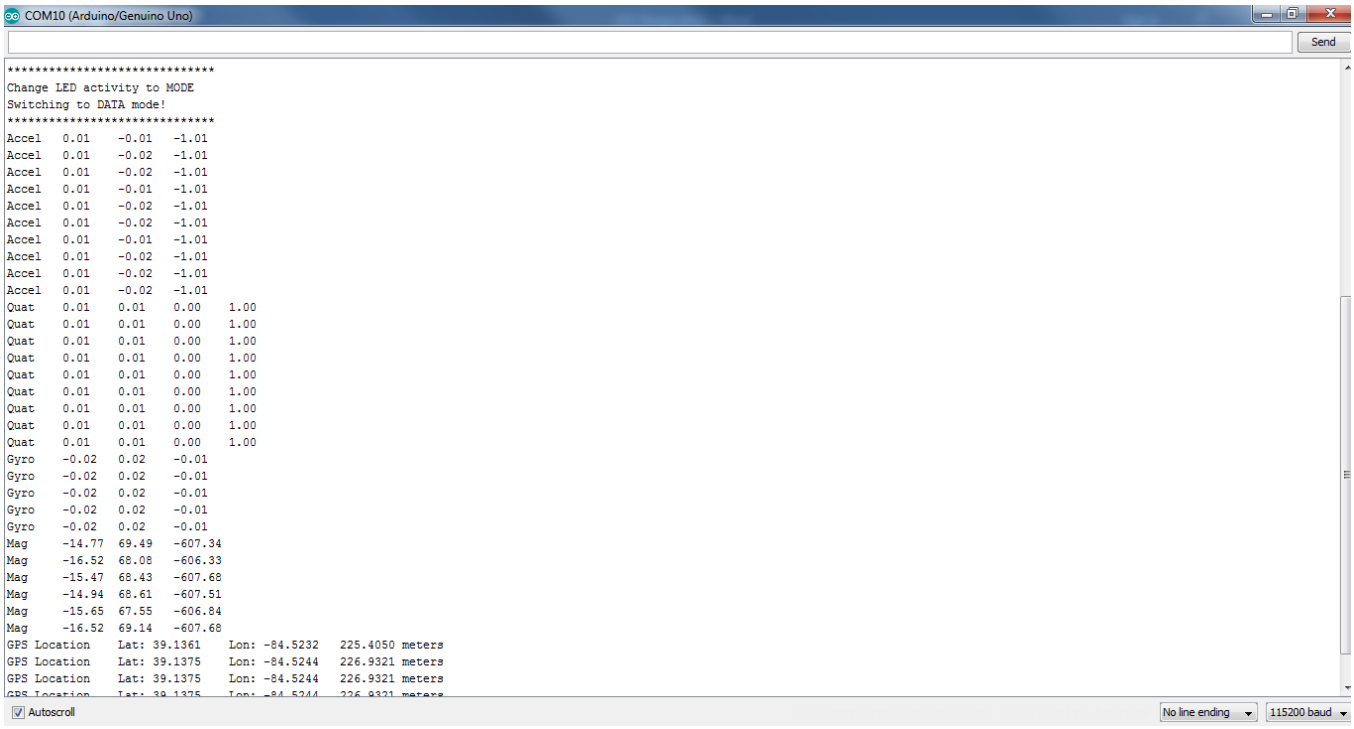

Fig 3.6 Accelerometer Data on Serial Monitor

Figure 3.6 shows accelerometer data being streamed onto the serial monitor

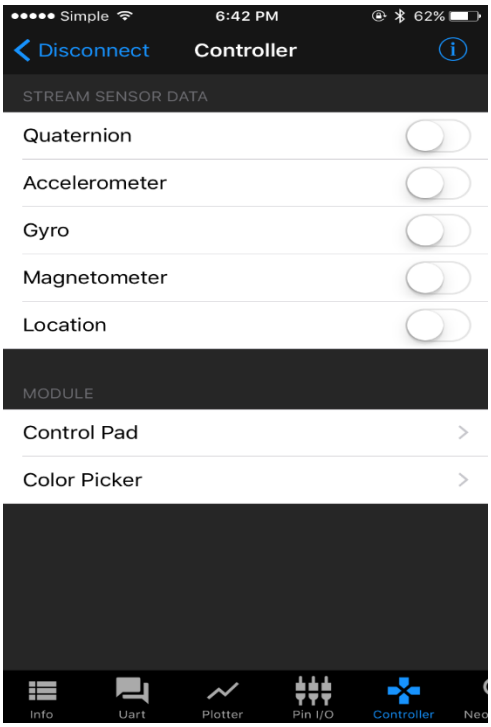

Fig 3.7 Application View of Available Data to be Toggled to Stream

Figure 3.7 shows the application side of the controller sketch that can stream data from the phone to the Arduino.

There are several GATT services that are defined by the Bluetooth Special Interest Group. One such service is the Heart Rate service that is assigned the service ID 0x180D. In the heart rate monitor sketch provided by the BLE tutorial page, we can see how to set up a service and characteristics to stream data to a central device [40].

```
ble.sendCommandWithIntReply( F("AT+GATTADDSERVICE=UUID=0x180D"),
&hrmServiceId);
```

This line of code adds the heart rate service to the BLE module using the group defined service ID of 0x180D.

```
ble.sendCommandWithIntReply( F("AT+GATTADDCHAR=UUID=0x2A37,
PROPERTIES=0x10, MIN_LEN=2, MAX_LEN=3, VALUE=00-40"),
&hrmMeasureCharId);
```

This line of code adds the heart rate measurement characteristic to the heart rate service with a characteristic ID of 0x2A37.

```
ble.sendCommandCheckOK( F("AT+GAPSETADVDATA=02-01-06-05-02-0d-18-0a-
18") );
```

This line of code adds the service UUID to the advertising payload so that it can be detected by central device applications.

If we run this sketch, the nRF Toolbox application for iOS shows heart rate monitor data streaming in from the BLE module.

In order to send data from the Arduino BLE peripheral device to a web server, you would need to purchase an Ethernet shield for the Arduino. Once the Ethernet shield is purchased, it can be stacked on the Bluefuit LE shield and data can be streamed to a website. Figure 3.8 below shows an Ethernet shield for the Arduino.

Fig 3.8 Arduino Ethernet Shield [44]

A tutorial is linked in the reference number 44.

Chapter 4 – Implementation

4.1 – TTL UART

On the Arduino Uno, to model serial communication, we'll first use the small snippet of code to echo characters with the PC using the serial monitor in the Arduino IDE.

Same echo program as before

```
//initialize serial communication with a baud rate of 9600
void setup() {
 (9600);                    //initialize serial communication
Serial.begin with a baud rate of 9600
Serial.println("Begin Echo Program");
}


void loop() {
  if (Serial.available()) {           //check if data available on the
serial port
    Serial.println("Data:");
    Serial.write(Serial.read());      //echo the data byte received
    Serial.println(" ");
  }
}
```

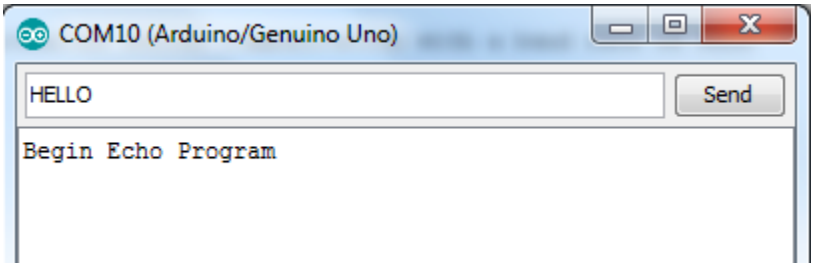

Fig 4.1 Before Transmission of Data via Serial UART

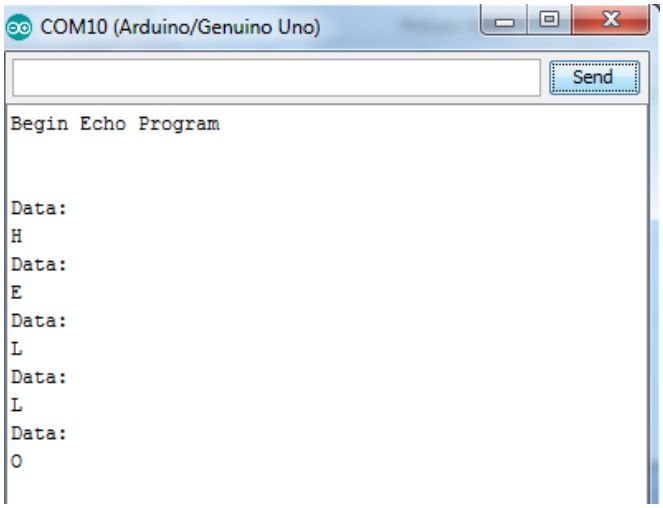

Fig 4.2 After Reception and Echo of Characters

Figures 4.1 and 4.2 show the serial monitor before and after transmission and reception of characters.

To facilitate communication between the Arduino and the PIC24EP, we first have to overcome an electrical barrier. The Arduino operates at 5V TTL and the PIC at 3.3V TTL, so if we hook up them up without any way to circumvent this difference, the PIC will sustain damage. There are two ways to overcome this voltage difference. The first way is to find the 5V tolerant pins on the PIC24EP and program two of those pins to be the RX and TX for serial communication.

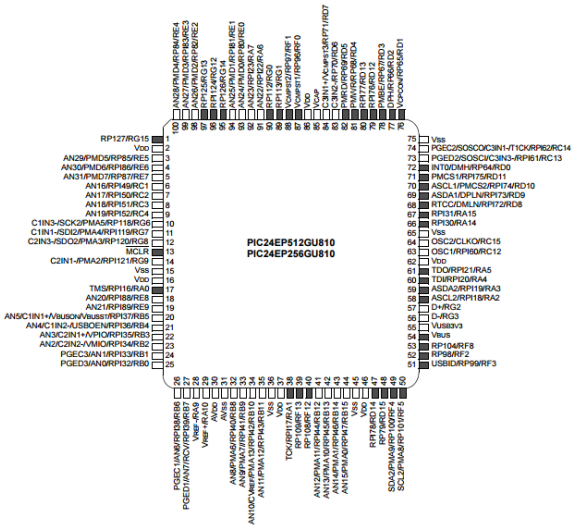

Fig 4.3 Pin Diagram PIC24EP512GU810 [30]

In Figure 4.3 above, all the pins shaded dark grey are 5V tolerant. You can find an enlarged version of this in the datasheet.

The second way to overcome the difference in voltages is to use a level shifter IC, which is a cheap and easy solution to this problem. The way a level shifter works is that it takes two reference voltage inputs from each of the two non-compatible devices; in our case we would wire a 5V line from the Arduino and a 3.3V line from the PIC. There are signals going in either direction, and they are level shifted using the reference voltages that we provide. This is the safer option and it is recommended because it will work regardless of whether your controller of choice has pins that are tolerant of higher operating voltages.

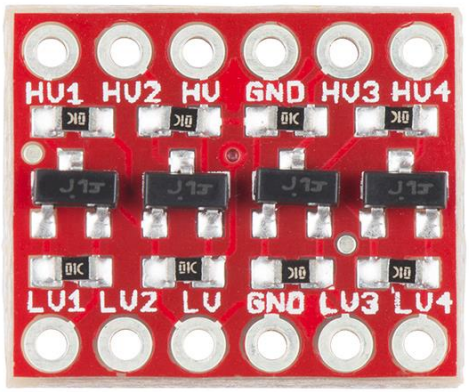

Fig 4.4 Level Shifter bySparkfun

Figure 4.4 shows a basic level shifter available on the market.

Regardless of the method you use, the code that you flash onto the controllers will remain the same. To make things simpler, we can set up a software serial port on the Arduino to communicate with the PIC and the native hardware serial port to display things on the serial monitor.

Arduino_to_PIC_UART.txt

```
//This program uses the software serial port functions from the
Arduino Serial Library to communicate with the PIC
// Srikar Chintapalli 04/15/2017
void setup() {

Serial.begin(9600);                    //initialize serial
communication with a baud rate of 9600 (for serial monitor display)
SoftwareSerial softSer1(8,9); // pin 8 is set as Rx and pin 9 as Tx
softSer1.begin(9600);          // enable the software serial port with
baud 9600
softSer1.println("Hello PIC!#");
}

void loop() {
  if (softSer1.available()) {            //check if data available on
the serial port
                                        //print onto serial monitor
    Serial.write(softSer1.read());    // the data byte received
  }
}
```

This program on the Arduino side initiates communication with the PIC, and when the PIC receives a string, it puts it on the LCD screen and responds to the Arduino. Below is the code snippet to do so. The code to interface with the LCD screen is taken from the MLA libraries. I have used the higher-level functions that the MLA libraries provide to print strings to the screen and to detect touches on the screen. Some of this code will be reused when experimenting with the I2C and SPI protocols on the PIC, so it will not be repeated then.

PIC_to_Arduino_UART.c

```
/*/
```

```
 * File:    main.c
 *
 * Requires: MLA 1306
   This program facilitates communication between the Arduino and the
PIC from the PIC side using the higher level library functions I wrote
in the design chapter (3). It uses functions from the MLA 1306
peripheral library to interact with the touch LCD display.
Srikar Chintapalli 04/15/2017
 */


#include "PICconfig.h"
#include "LCDTerminal.h"
#include "TouchScreen.h"
#include "TimeDelay.h"
#include "uart1.h"
#include "uMedia.h"
#define __ISR  __attribute__((interrupt, shadow, no_auto_psv))

char s1[64];
void conditiondisplay()
{
    LCDCenterString( +2, s1);
    while( TouchGetX() < 0);    // wait for tap
    while( TouchGetX() > 0);    // wait for release
    s1[0] = '\0';
    LCDClear();
}
void __ISR _T3Interrupt( void)
{
    _T3IF = 0;
    TouchDetectPosition();
}
#define TICK_PERIOD( ms)  (GetPeripheralClock() * (ms)) / 8000
```

```
void TickInit( unsigned period_ms)
{
    // Initialize Timer3
    TMR3 = 0;
    PR3 = TICK_PERIOD( period_ms);
    T3CONbits.TCKPS = 1;        // Set prescale to 1:8
    IFS0bits.T3IF = 0;          // Clear flag
    IEC0bits.T3IE = 1;          // Enable interrupt
    T3CONbits.TON = 1;          // Run timer
}
int TouchGet( void)
{   // returns 10..1F if screen pressed, 0 = none
    int  x, y, r;
    // 1. get the latest reading
    x = TouchGetX();  y = TouchGetY();
    // 2. if one of the two is null the other is too
    if (( x < 0) || ( y < 0))
        return 0;
    // 3. identify point on grid (4x4 = 0001yyxx)
    r  = ((y / ( GetMaxY()/4)) <<2) + ( x / ( GetMaxX()/4) );
    return r + 0x10;
} // TouchGe
int TouchGrid( void)
{   // wait for a key pressed and debounce
    int released = 0;           // released counter
    int pressed = 0;            // pressed counter
    int code;                   // grid code
    int r = 0;                  // return value
    // 1. wait for a key pressed for at least 10 loops
    while ( pressed < 10)
    {
        code = TouchGet();
```

```
        if ( code > 0) pressed++;
        else         pressed = 0;

        DelayMs( 1);
    }

    // 2. wait for key released for at least 10 loops
    while ( released < 10)
    {
        code = TouchGet();
        if ( code > 0)
        {
            r = code;
            released = 0;       // not released yet
            pressed++;          // still pressed, keep counting
        }
        else released++;
        DelayMs( 1);
    }
    // 3. check if a button was pushed longer than 500ms
    //if ( pressed > 500)
        //r += 0x80;              // add a flag in bit 7 of the code
    // 4. return code
    return r;
} // TouchGrid
int main( void )
{
    LCDInit();
    DisplayBacklightOn();
    TickInit( 1);
    TouchInit( NULL, NULL, NULL, NULL);
    LCDClear();
```

```
    InitUART1();                    //UART1 module is initialized
    while (1) {
       getsUART1(s1, 64);           //character buffer is filled till
the delimiter is received
       conditiondisplay();          //put received string on LCD screen
and detect touch press
       putsUART1("Hello Arduino!"); //send string back to Arduino
    }
}
```

4.2 – SPI

We will use the BME280 sensor and go through the code that uses SPI to transfer data to and from the Arduino. The first step to take is to wire the Arduino and the BME280 chip using a bread board. Make the connections as follows:

VIN → Arduino 5V

GND → Arduino GND

SCK → Arduino Pin 13

SDO → Arduino Pin 12

SDI → Arduino Pin 11

CS → Arduino Pin 10

Once you make these connections, open the BME280 test sketch on the Arduino IDE.

```
Adafruit_BME280 bme(BME_CS); // hardware SPI
```

This line creates an object that allows us to use the hardware SPI method of communication. You can comment in/out I2C if you wish to communicate with that method. The readTemperature(),

readPressure(), readAltitude(), and readHumidity() functions are called from the support library Adafruit_BME280.cpp [41].

We'll now go over some of the functions in that library.

In the begin() function, SPI.begin() is used to initialize the SPI module. Then, a register is read and a couple registers are written for the purpose of checking chip ID and setting up sampling parameters.

The primary functions used to read and write are the read8(), read16(), read24(), and write8() functions, which in turn call the spixfer() function. The spixfer() function essentially calls SPI.transfer() from the Arduino SPI library.

Adafruit_BME280.cpp

```
if (_sck == -1)
      SPI.beginTransaction(SPISettings(500000, MSBFIRST, SPI_MODE0));
    digitalWrite(_cs, LOW);
    spixfer(reg | 0x80); // read, bit 7 high
    value = spixfer(0);
    digitalWrite(_cs, HIGH);
    if (_sck == -1)
      SPI.endTransaction();              // release the SPI bus
  }
  return value;
```

The above code is the read8() function, which takes a register address as a parameter and reads a byte from that register. It begins by using SPI.beginTransaction() to set up communication parameters and proceeds to drive the slave select line low. Once this is done, a byte is sent to the

BME280 specifying the register address and making sure bit 7 is high. For a write operation, bit 7 should be low. Once the register address is sent, depending on whether it is a read or write, either a NULL byte (a byte must be sent to receive one) or the byte to be transferred is sent. The read16() and read24() functions are very similar; the only difference is that larger variables are declared and byte shifted upon the arrival of a new byte. The slave select is pulled high and the transaction is ended [41].

```
value = spixfer(0);
value <<= 8;
value |= spixfer(0);
value <<= 8;
value |= spixfer(0);
```

Above is a snippet from the read24() function in which a 32-bit variable is declared, and the bytes are shifted in as they arrive.

These functions are used to read both the sensor registers on board as well as registers that contain calibration data. Finally, using the manufacturer's datasheet, decimal readable values for the temperature, pressure, humidity, and altitude are calculated.

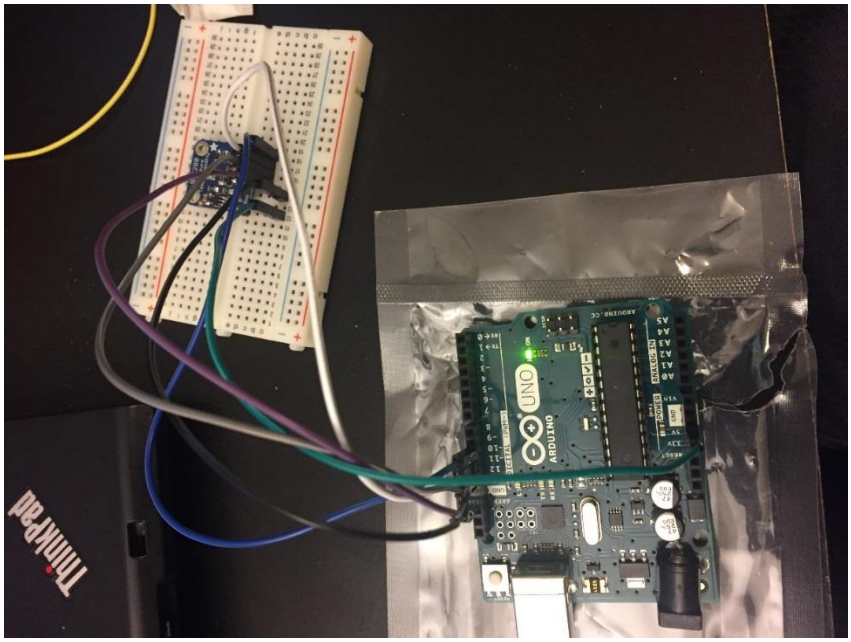

Fig 4.5 Arduino to BME280 Setup

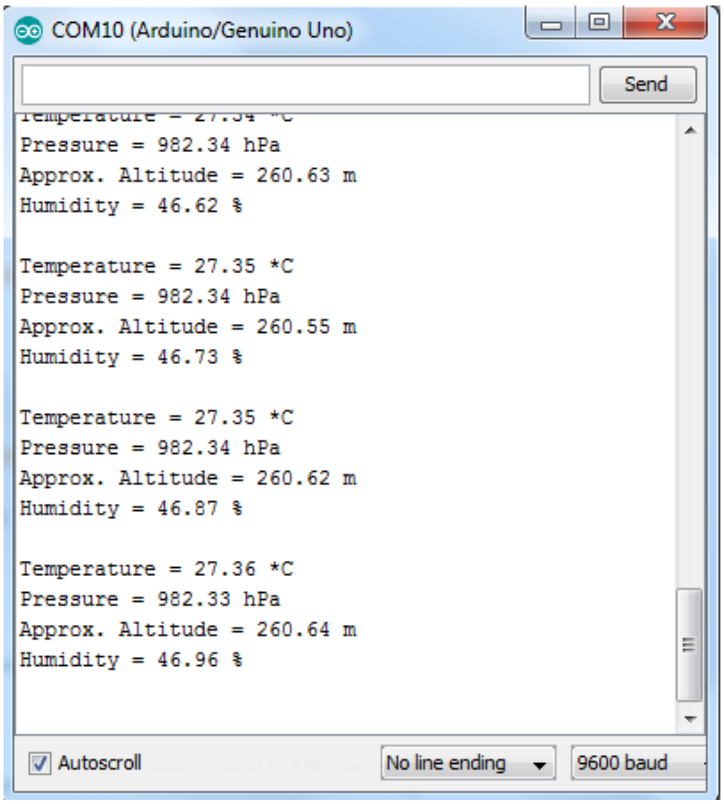

Fig 4.6 BME280 Readings on Serial Monitor

Figures 4.5 and 4.6 show the Arduino to BME280 setup and streaming of sensor data to the serial monitor.

For the PIC24EP, to test out SPI, we will connect wires to the BME280 sensor the same way we did for the Arduino.

VIN → PIC 3.3V

GND → PIC GND

SCK → PIC Pin RG6

SDO → PIC Pin RG7

SDI → PIC Pin RG8

CS → PIC Pin RF8

After the connections are set up, I used code that I ported from the adafruit BME280 library (from the adafruit website) to work with the SPI functions I wrote [41].

BME280_PIC.h *BME280_PIC.c*

```
/**************************************************************************
******
  This is a library for the BME280 humidity, temperature & pressure
sensor

  Designed specifically to work with the Adafruit BME280 Breakout
  ----> http://www.adafruit.com/products/2650
```

```
  These sensors use I2C or SPI to communicate, 2 or 4 pins are
required
  to interface.

  Adafruit invests time and resources providing this open source code,
  please support Adafruit andopen-source hardware by purchasing
products
  from Adafruit!

  Written by Limor Fried & Kevin Townsend for Adafruit Industries.
  BSD license, all text above must be included in any redistribution

*************************************************************************
*****/

    #define BME280_ADDRESS                (0x77)

/*=========================================================================
=====
    REGISTERS
    -----------------------------------------------------------------------
-----*/
    #define  BME280_REGISTER_DIG_T1              0x88
    #define  BME280_REGISTER_DIG_T2              0x8A
    #define  BME280_REGISTER_DIG_T3              0x8C

    #define  BME280_REGISTER_DIG_P1              0x8E
    #define  BME280_REGISTER_DIG_P2              0x90
    #define  BME280_REGISTER_DIG_P3              0x92
    #define  BME280_REGISTER_DIG_P4              0x94
    #define  BME280_REGISTER_DIG_P5              0x96
    #define  BME280_REGISTER_DIG_P6              0x98
```

```
    #define  BME280_REGISTER_DIG_P7              0x9A
    #define  BME280_REGISTER_DIG_P8              0x9C
    #define  BME280_REGISTER_DIG_P9              0x9E

    #define  BME280_REGISTER_DIG_H1              0xA1
    #define  BME280_REGISTER_DIG_H2              0xE1
    #define  BME280_REGISTER_DIG_H3              0xE3
    #define  BME280_REGISTER_DIG_H4              0xE4
    #define  BME280_REGISTER_DIG_H5              0xE5
    #define  BME280_REGISTER_DIG_H6              0xE7

    #define  BME280_REGISTER_CHIPID             0xD0
    #define  BME280_REGISTER_VERSION            0xD1
    #define  BME280_REGISTER_SOFTRESET          0xE0

    #define  BME280_REGISTER_CAL26              0xE1  // R
calibration stored in 0xE1-0xF0

    #define  BME280_REGISTER_CONTROLHUMID       0xF2
    #define  BME280_REGISTER_CONTROL            0xF4
    #define  BME280_REGISTER_CONFIG             0xF5
    #define  BME280_REGISTER_PRESSUREDATA       0xF7
    #define  BME280_REGISTER_TEMPDATA           0xFA
    #define  BME280_REGISTER_HUMIDDATA          0xFD



/*=========================================================================
=====*/

/*=========================================================================
=====
```

```
    CALIBRATION DATA
    ---------------------------------------------------------------
-----*/
      unsigned short dig_T1;
      short  dig_T2;
      short  dig_T3;
      unsigned short dig_P1;
      short  dig_P2;
      short  dig_P3;
      short  dig_P4;
      short  dig_P5;
      short  dig_P6;
      short  dig_P7;
      short  dig_P8;
      short  dig_P9;
      unsigned char  dig_H1;
      short  dig_H2;
      unsigned char  dig_H3;
      short  dig_H4;
      short  dig_H5;
      char   dig_H6;

      int t_fine;
/*===============================================================
=====*/

    int  SPIbegin(void);
    float readTemperature(void);
    float readPressure(void);
    float readHumidity(void)
```

```
    void readCoefficients(void);

    void SPIwrite8(unsigned char reg, unsigned char value);
    unsigned char SPIread8(unsigned char reg);
    unsigned short SPIread16(unsigned char reg);
    unsigned int SPIread24(unsigned char reg);
    short SPIreadS16(unsigned char reg);
    unsigned short SPIread16_LE(unsigned char reg); // little endian
    short SPIreadS16_LE(unsigned char reg); // little endian
/**************************************************************************
******
  This is a library for the BME280 humidity, temperature & pressure
sensor

  Designed specifically to work with the Adafruit BME280 Breakout
  ----> http://www.adafruit.com/products/2650

  These sensors use I2C or SPI to communicate, 2 or 4 pins are
required
  to interface.

  Adafruit invests time and resources providing this open source code,
  please support Adafruit andopen-source hardware by purchasing
products
  from Adafruit!

  Written by Limor Fried & Kevin Townsend for Adafruit Industries.
  BSD license, all text above must be included in any redistribution

**************************************************************************
*****/
```

```
#include "BME280_PIC.h"
#include "spi.h"
//these support functions for the BME280 to work with the PIC were
ported from the adafruit BME280 library and integrated with the
functions I wrote for SPI operation in the design chapter
// Srikar Chintapalli 04/20/2017

int SPIbegin(void) {

   SPI1_Init();



  if (SPIread8(BME280_REGISTER_CHIPID) != 0x60)
    return 0;

  readCoefficients();

  //Set before CONTROL_meas (DS 5.4.3)
  SPIwrite8(BME280_REGISTER_CONTROLHUMID, 0x05); //16x oversampling

  SPIwrite8(BME280_REGISTER_CONTROL, 0xB7); // 16x ovesampling, normal
mode
  return 1;
}

/**************************************************************************
*****/
/*!
    @brief  Writes an 8 bit value over SPI
*/
/**************************************************************************
*****/
```

```
void SPIwrite8(unsigned char reg, unsigned char value)
{
    SS1 = 0;
    SPIPut(reg & ~0x80); // write, bit 7 low
    SPIPut(value);
    SS1 = 1;
  }

/*************************************************************************
*****/
/*!
    @brief  Reads an 8 bit value over SPI
*/
/*************************************************************************
*****/
unsigned char SPIread8(unsigned char reg)
{
  unsigned char value;
  SS1 = 0;
  SPIPut(reg | 0x80); // read, bit 7 high
  SPIPut(0);
  value = SPIGet();
  SS1 = 1;
  return value;
}

/*************************************************************************
*****/
/*!
    @brief  Reads a 16 bit value over SPI
*/
```

```
/*************************************************************************
*****/
unsigned short SPIread16(unsigned char reg)
{
  unsigned short value;
    SS1 = 0;
    SPIPut(reg | 0x80); // read, bit 7 high
    SPIPut(0);
    value = SPIGet();
    value = value << 8;
    SPIPut(0);
    value = value | SPIGet();
    SS1 = 1;

  return value;
}

unsigned short SPIread16_LE(unsigned char reg) {
  unsigned short temp = SPIread16(reg);
  return (temp >> 8) | (temp << 8);
}

/*************************************************************************
*****/
/*!
    @brief  Reads a signed 16 bit value over I2C
*/
/*************************************************************************
*****/
short SPIreadS16(unsigned char reg)
{
```

```
  return (short)SPIread16(reg);

}

short SPIreadS16_LE(unsigned char reg)
{
  return (short)SPIread16_LE(reg);

}

/*************************************************************************
*****/
/*!
    @brief  Reads a 24 bit value over I2C
*/
/*************************************************************************
*****/

unsigned int SPIread24(unsigned char reg)
{
  unsigned int value;
    SS1 = 0;
    SPIPut(reg | 0x80); // read, bit 7 high
    SPIPut(0);
    value = SPIGet();
    value <<= 8;
    SPIPut(0);
    value |= SPIGet();
    value <<= 8;
    SPIPut(0);
    value |= SPIGet();
```

```
    SS1 = 1;
  return value;
}

/*************************************************************************
*****/
/*!
    @brief  Reads the factory-set coefficients
*/
/*************************************************************************
*****/
void readCoefficients(void)
{
    dig_T1 = SPIread16_LE(BME280_REGISTER_DIG_T1);
    dig_T2 = SPIreadS16_LE(BME280_REGISTER_DIG_T2);
    dig_T3 = SPIreadS16_LE(BME280_REGISTER_DIG_T3);

    dig_P1 = SPIread16_LE(BME280_REGISTER_DIG_P1);
    dig_P2 = SPIreadS16_LE(BME280_REGISTER_DIG_P2);
    dig_P3 = SPIreadS16_LE(BME280_REGISTER_DIG_P3);
    dig_P4 = SPIreadS16_LE(BME280_REGISTER_DIG_P4);
    dig_P5 = SPIreadS16_LE(BME280_REGISTER_DIG_P5);
    dig_P6 = SPIreadS16_LE(BME280_REGISTER_DIG_P6);
    dig_P7 = SPIreadS16_LE(BME280_REGISTER_DIG_P7);
    dig_P8 = SPIreadS16_LE(BME280_REGISTER_DIG_P8);
    dig_P9 = SPIreadS16_LE(BME280_REGISTER_DIG_P9);

    dig_H1 = SPIread8(BME280_REGISTER_DIG_H1);
    dig_H2 = SPIreadS16_LE(BME280_REGISTER_DIG_H2);
    dig_H3 = SPIread8(BME280_REGISTER_DIG_H3);
```

```
    dig_H4 = (SPIread8(BME280_REGISTER_DIG_H4) << 4) |
(SPIread8(BME280_REGISTER_DIG_H4+1) & 0xF);

    dig_H5 = (SPIread8(BME280_REGISTER_DIG_H5+1) << 4) |
(SPIread8(BME280_REGISTER_DIG_H5) >> 4);

    dig_H6 = (char)SPIread8(BME280_REGISTER_DIG_H6);

}

*/

/*************************************************************************
*****/

float readTemperature(void)

{

  int var1, var2;


  int adc_T = SPIread24(BME280_REGISTER_TEMPDATA);

  adc_T >>= 4;


  var1  = ((((adc_T>>3) - ((int)dig_T1 <<1))) *

       ((int)dig_T2)) >> 11;


  var2  = (((((adc_T>>4) - ((int)dig_T1)) *

         ((adc_T>>4) - ((int)dig_T1))) >> 12) *

       ((int)dig_T3)) >> 14;


  t_fine = var1 + var2;


  float T  = (t_fine * 5 + 128) >> 8;

  return T/100;

}

/*!


*/
```

```
/**************************************************************************
*****/
float readPressure(void) {
  long long var1, var2, p;

  readTemperature(); // must be done first to get t_fine

  int adc_P = SPIread24(BME280_REGISTER_PRESSUREDATA);
  adc_P >>= 4;

  var1 = ((long long)t_fine) - 128000;
  var2 = var1 * var1 * (long long)dig_P6;
  var2 = var2 + ((var1*(long long)dig_P5)<<17);
  var2 = var2 + (((long long)dig_P4)<<35);
  var1 = ((var1 * var1 * (long long)dig_P3)>>8) +
    ((var1 * (long long)dig_P2)<<12);
  var1 = (((((long long)1)<<47)+var1))*((long long)dig_P1)>>33;

  if (var1 == 0) {
    return 0;  // avoid exception caused by division by zero
  }
  p = 1048576 - adc_P;
  p = (((p<<31) - var2)*3125) / var1;
  var1 = (((long long)dig_P9) * (p>>13) * (p>>13)) >> 25;
  var2 = (((long long)dig_P8) * p) >> 19;

  p = ((p + var1 + var2) >> 8) + (((long long)dig_P7)<<4);
  return (float)p/256;
}
```

```
/**************************************************************************
*****/
/*!


*/
/**************************************************************************
*****/
float readHumidity(void) {

  readTemperature(); // must be done first to get t_fine

  int adc_H = SPIread16(BME280_REGISTER_HUMIDDATA);

  int v_x1_u32r;

  v_x1_u32r = (t_fine - ((int)76800));

  v_x1_u32r = (((((adc_H << 14) - (((int)dig_H4) << 20) -
          (((int)dig_H5) * v_x1_u32r)) + ((int)16384)) >> 15) *
           (((((((v_x1_u32r * ((int)dig_H6)) >> 10) *
            (((v_x1_u32r * ((int)dig_H3)) >> 11) + ((int)32768))) >>
10) +
          ((int)2097152)) * ((int)dig_H2) + 8192) >> 14));

  v_x1_u32r = (v_x1_u32r - (((((v_x1_u32r >> 15) * (v_x1_u32r >> 15))
>> 7) *
                  ((int)dig_H1)) >> 4));

  v_x1_u32r = (v_x1_u32r < 0) ? 0 : v_x1_u32r;
  v_x1_u32r = (v_x1_u32r > 419430400) ? 419430400 : v_x1_u32r;
  float h = (v_x1_u32r>>12);
  return  h / 1024.0;
```

```
}
```

4.3 – I2C

We will use the MCP9808 sensor and go through the code that uses I2C to transfer data to and fro. The first step to take is to wire the Arduino and the MCP9808 chip using a bread board. Make the connections as follows [42]:

VDD → Arduino 5V

GND → Arduino GND

SCL → Arduino Pin A5

SDA → Arduino Pin A4

Once you make these connections, open the MCP9808 test sketch on the Arduino IDE.

```
Adafruit_MCP9808 tempsensor = Adafruit_MCP9808();
```

This line creates an object that allows us to use the hardware SPI method of communication. You can comment in/out I2C if you wish to communicate with that method. The readTemperature(),functions are called from the support library Adafruit_MCP9808.cpp.

We'll now go over some of the functions in that library.

In the begin() function, Wire.begin() is used to initialize the I2C module. Then, a couple of registers are read and a couple registers are written for the purpose of checking chip ID and manufacturer ID.

The primary functions used to read and write are the read16() and write16() functions, which use functions from the Arduino Wire library to work.

Adafruit_MCP9808.cpp

```
Wire.beginTransmission(_i2caddr);
    Wire.write((uint8_t)reg);
    Wire.write(value >> 8);
    Wire.write(value & 0xFF);
    Wire.endTransmission();
```

This is the body of the write16() function which takes in an address and the value to be written as parameters. It uses the Wire.write() function to first write the address and then the value byte by byte.

```
uint16_t val;

  Wire.beginTransmission(_i2caddr);
  Wire.write((uint8_t)reg);
  Wire.endTransmission();

  Wire.requestFrom((uint8_t)_i2caddr, (uint8_t)2);
  val = Wire.read();
  val <<= 8;
  val |= Wire.read();
  return val;
```

The read16() function first sends the address on the bus and then requests data from the sensor using the requestFrom function.

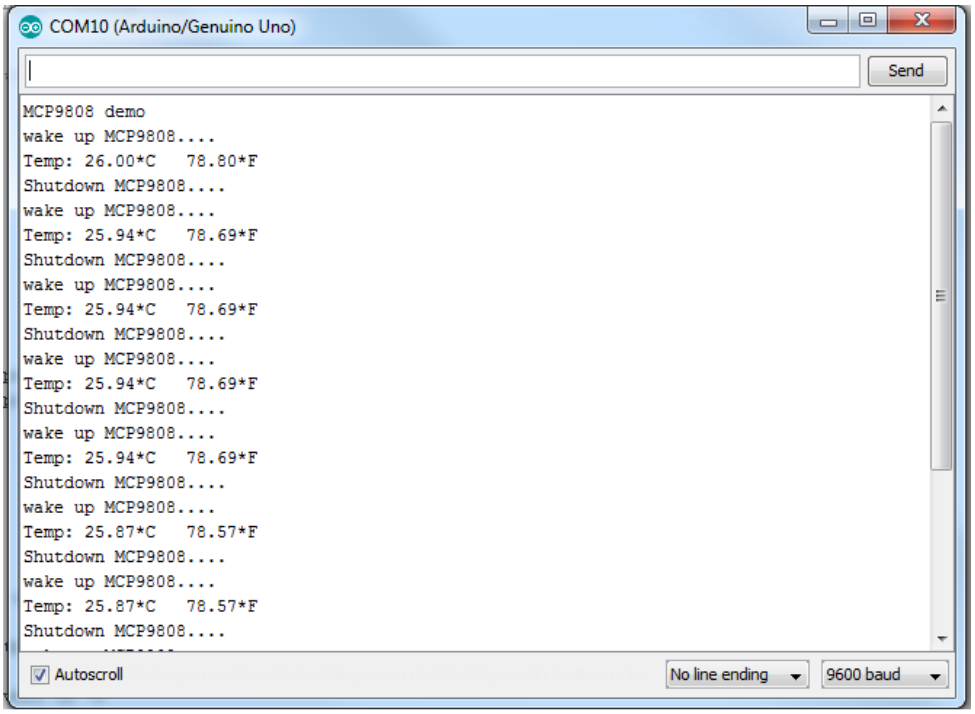

Fig 4.7 MCP9808 sensor readings on Serial Monitor

Figure 4.7 shows the Arduino streaming data onto the serial monitor from the MCP9808.

For the PIC24EP, to test out I2C, we will connect wires to the MCP9808 sensor the same way we did for the Arduino.

VDD → PIC 3.3V

GND → PIC GND

SCL → PIC Pin RD10

SDA → PIC Pin RD9

After the connections are set up, I used code that I ported from the adafruit MCP9808 library (from theadafruit website) to work with the I2C functions I wrote [42].

MCP9808_PIC.h *MCP9808_PIC.c*

```
//MCP9808_PIC.h
//these support functions for the MCP9808 to work with the PIC were
ported from the adafruit MCP9808 library and integrated with the
functions I wrote for I2C operation in the design chapter
// Srikar Chintapalli 04/20/2017
#define MCP9808_I2CADDR_DEFAULT        0x18
#define MCP9808_REG_CONFIG             0x01

#define MCP9808_REG_CONFIG_SHUTDOWN    0x0100
#define MCP9808_REG_CONFIG_CRITLOCKED  0x0080
#define MCP9808_REG_CONFIG_WINLOCKED   0x0040
#define MCP9808_REG_CONFIG_INTCLR      0x0020
#define MCP9808_REG_CONFIG_ALERTSTAT   0x0010
#define MCP9808_REG_CONFIG_ALERTCTRL   0x0008
#define MCP9808_REG_CONFIG_ALERTSEL    0x0004
#define MCP9808_REG_CONFIG_ALERTPOL    0x0002
#define MCP9808_REG_CONFIG_ALERTMODE   0x0001

#define MCP9808_REG_UPPER_TEMP         0x02
#define MCP9808_REG_LOWER_TEMP         0x03
#define MCP9808_REG_CRIT_TEMP          0x04
#define MCP9808_REG_AMBIENT_TEMP       0x05
#define MCP9808_REG_MANUF_ID           0x06
#define MCP9808_REG_DEVICE_ID          0x07

 int begin(void);
 float readTempF( void );
 float readTempC( void );
 int shutdown_wake( unsigned char sw_ID );
```

```
 void write16(unsigned char reg, unsigned short val);
 unsigned short read16(unsigned char reg);


/**************************************************************************
*****/
/*!
    @file     Adafruit_MCP9808.cpp
    @author   K.Townsend (Adafruit Industries)
    @license  BSD (see license.txt)

    I2C Driver for Microchip's MCP9808 I2C Temp sensor

    This is a library for the Adafruit MCP9808 breakout
    ----> http://www.adafruit.com/products/1782

    Adafruit invests time and resources providing this open source
code,
    please support Adafruit and open-source hardware by purchasing
    products from Adafruit!

    @section  HISTORY

    v1.0 - First release
*/
/**************************************************************************
*****/


/*
tempsensor code ported to PIC
```

```
*/


#include "i2c1.h"
#include "MCP9808_PIC.h"


unsigned char slaveaddW = MCP9808_I2CADDR_DEFAULT << 1 | 0;
unsigned char slaveaddR = MCP9808_I2CADDR_DEFAULT << 1 | 0x01;

 int begin(void);
 float readTempF( void );
 float readTempC( void );
 int shutdown_wake( unsigned char sw_ID );
 void write16(unsigned char reg, unsigned short val);
 unsigned short read16(unsigned char reg);

 int TSbegin(void) {

  if (read16(MCP9808_REG_MANUF_ID) != 0x0054) return 0;
  if (read16(MCP9808_REG_DEVICE_ID) != 0x0400) return 0;

  return 1;
}

void write16(unsigned char reg, unsigned short val) {

    I2C1Start();
    I2C1Send(slaveaddW);
```

```
    I2C1Send(reg);

    I2C1Send(val >> 8);
    I2C1Send(val & 0xFF);
    I2C1Stop();
  }

 unsigned short read16(unsigned char reg) {
  unsigned short val;
 I2C1Start();
 I2C1Send(slaveaddW);
 I2C1Send(reg);
 I2C1Restart();
 I2C1Send(slaveaddR);
 val = I2C1Receive(0);
 val <<= 8;
 val |= I2C1Receive(1);
 I2C1Stop();
 return val;
}

float readTempC( void )
{
  unsigned short t = read16(MCP9808_REG_AMBIENT_TEMP);

  float temp = t & 0x0FFF;
  temp /=  16.0;
  if (t & 0x1000) temp -= 256;

  return temp;
```

```
}

int shutdown_wake( unsigned char sw_ID )
{
    unsigned short conf_shutdown ;
    unsigned short conf_register = read16(MCP9808_REG_CONFIG);
    if (sw_ID == 1)
    {
       conf_shutdown = conf_register | MCP9808_REG_CONFIG_SHUTDOWN ;
       write16(MCP9808_REG_CONFIG, conf_shutdown);
    }
    if (sw_ID == 0)
    {
       conf_shutdown = conf_register ^ MCP9808_REG_CONFIG_SHUTDOWN ;
       write16(MCP9808_REG_CONFIG, conf_shutdown);
    }

    return 0;
}
```

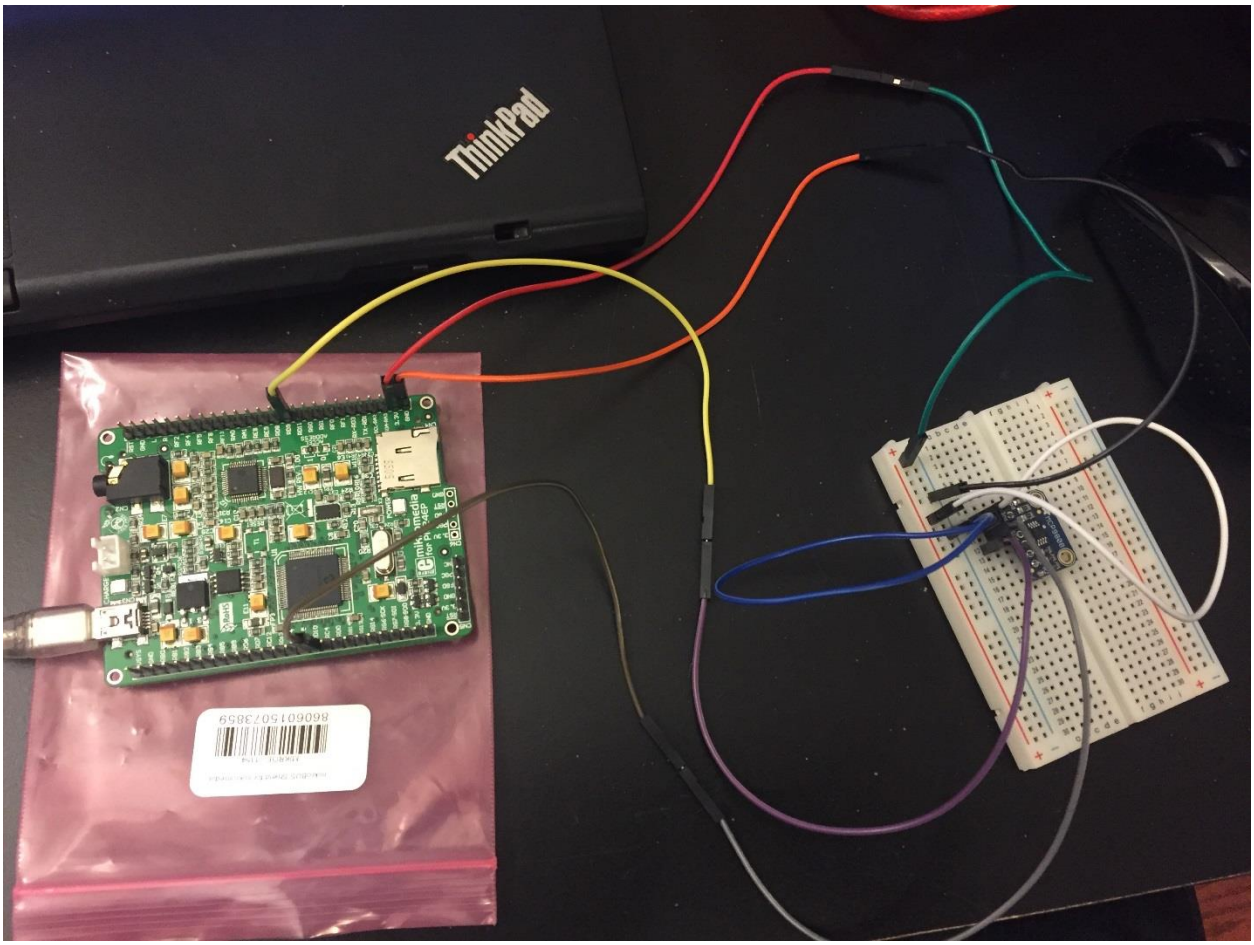

Fig 4.8 PIC24EP to MCP9808 Setup

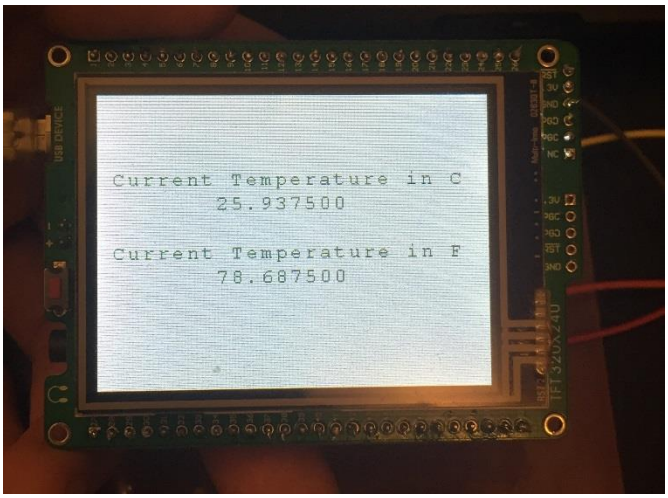

Fig 4.9 PIC24EP MCP9808 Sensor Readings

Figures 4.8 and 4.9 show the PIC setup to stream data from the MCP9808.

Chapter 5 – Conclusion

In conclusion, this thesis has first reviewed the details of the three major communication protocols: TTL UART, I2C, and SPI. It has also discussed BLE and how to use the Adafruit Bluefruit LE Shield for beginner level exercises in wireless data transmission using the Arduino. After thoroughly explaining how the communication protocols work at a register and bit level, the thesis proceeds to show the student audience how to implement these protocols on a beginner level board, the Arduino Uno. The students gain hands-on experience with how data is transmitted and received using higher order library functions that the Arduino community provides. After being exposed to the Arduino Uno, the students are introduced to an intermediate level microcontroller called the PIC24EP512GU810. This controller lacks the library support that the Arduino provided, so the thesis details how to go into the datasheet and user manual of a new controller to build up the protocols from scratch. It provides the students with libraries for TTL UART, I2C, and SPI along with detailed instructions on how each line of code was written, how each register was toggled, and why certain steps were taken.

Once the students go through the extensive process of understanding how the functions were written for the PIC family, the thesis aims to test out the protocols for both the Arduino and the PIC. For UART TTL, directions are given as to how to connect the Arduino and PIC to make sure they are electrically stable and code is given to facilitate an exchange of data between the two. For SPI and I2C, sensors are taken from the vendor Adafruit and students first gather data from these sensors using a basic step by step tutorial on how to interface them with the Arduino. These steps were taken from the sensor vendor and written in the thesis in a concise manner. Once the students use the libraries provided by Adafruit to gather data from the Arduino, they know what to expect. Then, the thesis provides details as to how to connect these same sensors to

the PIC24EP512GU810. It uses the library functions written in the design chapter to facilitate communication between the PIC and the sensors.

The aim of this thesis was to first provide a foundation of conceptual knowledge as to how the protocols work. The next steps aimed to first bring the students hands-on experience by first allowing them to work with a familiar and easy microcontroller with open source support and then making them step out of their comfort zone to work with an unfamiliar microcontroller unit where they are responsible for all the details. The goal is that in the future, a student can pick up any microcontroller unit and understand the steps to be followed in order to interface his or her controller to a peripheral using UART, SPI, or I2C.

Chapter 6 – Exercises

[1] What are the main drawbacks of parallel communication that caused serial communication to become so prominent?

[2] How many bits can be sent in one frame of data with UART communication and what is the parity bit?

[3] How many lines are needed for I2C communication and how does the slave device know when to send data or when it is receiving data? How many data lines are needed to connect multiple devices on the I2C bus versus on the SPI bus?

[4] Can you receive something on the SPI bus without sending anything? Why or why not?

[5] Explain when you would use each of the three serial communication protocols.

[6] How are services and their characteristics distinguished in the BLE protocol?

[7] When first setting up a new microcontroller, how do you know if it is compatible with a sensor that you already have?

[8] When using UART and communication doesn't seem to be working properly, what things should be checked and in what order?

[9] When using I2C and communication doesn't seem to be working properly, what things should be checked and in what order? How does the slave indicate that it is not ready for more communication?

[10] When using SPI and communication doesn't seem to be working properly, what things should be checked and in what order?

[11] If you had six leftover i/o pins on your microcontroller and the need to connect three more temperature sensors how would you go about this?

[12] When using BLE, what needs to be done when adding an attribute to a service?

References

[1] Embedded Systems Market (Embedded Hardware and Embedded Software) Market for Healthcare, Industrial, Automotive, Telecommunication, Consumer Electronics, Defense, Aerospace and Others Applications: Global Industry Perspective, Comprehensive Analysis and Forecast, 2015 - 2021. Rep. Market Research Store, 2 June 2016. Web. 01 Feb. 2017. <http://www.marketresearchstore.com/news/global-embedded-systems-market-249>.

[2] "SoC Bus Architecture." SoC Bus Architecture - MechatronicsUSP. N.p., n.d. Web. 02 May 2017. http://www.mecatronica.eesc.usp.br/wiki/index.php/SoC_Bus_Architecture

[3] U. Nanda and S. K. Pattnaik, "Universal Asynchronous Receiver and Transmitter (UART)," 2016 3rd International Conference on Advanced Computing and Communication Systems (ICACCS), Coimbatore, 2016, pp. 1-5.

[4] S. Muppalla and K. R. Vaddempudi, "A novel VHDL implementation of UART with single error correction and double error detection capability," 2015 International Conference on Signal Processing and Communication Engineering Systems, Guntur, 2015, pp. 152-156.

[5] R. K. Agrawal and V. R. Mishra, "The design of high speed UART," 2013 IEEE Conference on Information & Communication Technologies, JeJu Island, 2013, pp. 388-390.

[6] Laddha, Neha R., and A. P. Thakare. "A Review on Serial Communication by UART." International Journal of Advanced Research in Computer Science and Software Engineering 3.1 (January 2013): 366-69. Web. 13 Feb. 2017.

[7] JIMB0. "Serial Communication." N.p., n.d. Web. 11 Mar. 2017. <https://learn.sparkfun.com/tutorials/serial-communication>.

[8] ARC Electronics. RS232 Data Interface: a Tutorial on Data Interface and cables N.p., n.d. Web. 11 Mar. 2017. <http://www.arcelect.com/rs232.htm>.

[9] Sven Knutsson. Embedded Systems Communication Interfaces with the Emphasis on Serial Protocols. Rep. Gothenburg: Chalmers - Department of Computer Science and Engineering, 2006. Print.

[10] P. Corcoran, "Two Wires and 30 Years : A Tribute and Introductory Tutorial to the I2C Two-Wire Bus," in IEEE Consumer Electronics Magazine, vol. 2, no. 3, pp. 30-36, July 2013.

[11] A. K. Oudjida, M. L. Berrandjia, R. Tiar, A. Liacha and K. Tahraoui, "FPGA implementation of I2C & SPI protocols: A comparative study," 2009 16th IEEE International Conference on Electronics, Circuits and Systems - (ICECS 2009), Yasmine Hammamet, 2009, pp. 507-510.

[12] F. Leens, "An introduction to I2C and SPI protocols," in IEEE Instrumentation & Measurement Magazine, vol. 12, no. 1, February 2009, pp. 8-13

[13] I2C. UM10204. Vol. Rev. 6. N.p.: NXP, 4 April 2014.

[14] SFUPTOWNMAKER. "I2C." N.p., n.d. Web. 16 Mar. 2017. <https://learn.sparkfun.com/tutorials/i2c>.

[15] MIKEGRUSIN. "Serial Peripheral Interface (SPI)." N.p., n.d. Web. 20 Mar. 2017. <https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi>.

[16] Z. Xin, H. Lu, L. Hu and J. Li, "Implementation of SPI and driver for CC2430 and C8051F120," 2012 2nd International Conference on Consumer Electronics, Communications and Networks (CECNet), Yichang, 2012, pp. 2638-2641.

[17] A. Szekacs, T. Szakaill and Z. Hegykozi, "Realising the SPI communication in a multiprocessor system," 2007 5th International Symposium on Intelligent Systems and Informatics, Subotica, 2007, pp. 213-216.

[18] T. Praveen Blessington, B. Bhanu Murthy, G. V. Ganesh and T. S. R. Prasad, "Optimal implementation of UART-SPI Interface in SoC," 2012 International Conference on Devices, Circuits and Systems (ICDCS), Coimbatore, 2012, pp. 673-677.

[19] D. N. Oruganti and S. S. Yellampalli, "Design of a power efficient SPI interface," 2014 International Conference on Advances in Electronics Computers and Communications, Bangalore, 2014, pp. 1-5.

[20] SPI Block Guide. Rep. no. V03.06. N.p.: Mototola Inc., 04 February 2003. Print.

[21] Wikipedia contributors. "Serial Peripheral Interface Bus." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 13 Jun. 2017. Web. 2 Jul. 2017

[22] L. F. Del Carpio, P. Di Marco, P. Skillermark, R. Chirikov, K. Lagergren and P. Amin, "Comparison of 802.11ah and BLE for a home automation use case," 2016 IEEE 27th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC), Valencia, Spain, 2016, pp. 1-6.

[23] S. Gowrishankar, N. Madhu and T. G. Basavaraju, "Role of BLE in proximity based automation of IoT: A practical approach," 2015 IEEE Recent Advances in Intelligent Computational Systems (RAICS), Trivandrum, 2015, pp. 400-405.

[24] A. E. Boualouache, O. Nouali, S. Moussaoui and A. Derder, "A BLE-based data collection system for IoT," 2015 First International Conference on New Technologies of Information and Communication (NTIC), Mila, 2015, pp. 1-5.

[25] K. Townsend, C. Cufi, Akiba & Robert Davidson *Getting Started with Bluetooth Low Energy* O'Reilly 2014

[26] Milovanovic , Viktor . "Bluetooth Low Energy - Part 1: Introduction To BLE." MikroElektronika Learn. N.p., 25 Mar. 2016. Web. 02 May 2017.

[27] Kristin. "Introduction to Bluetooth low energy and Bluetooth low energy development (video tutorial)." Nordic Semiconductor Developer Zone. N.p., n.d. Web. 02 June 2017. <https://devzone.nordicsemi.com/tutorials/37/>.

[28] "Bluetooth Core Specification | Bluetooth Technology Website." Bluetooth. N.p., n.d. Web. 03 June 2017. <https://www.bluetooth.com/specifications/bluetooth-core-specification>.

[29] "GATT Services | Bluetooth Technology Website." Bluetooth. N.p., n.d. Web. 10 June 2017. <https://www.bluetooth.com/specifications/gatt/services>.

[30] Microchip Technology, "16-Bit Microcontrollers and Digital Signal Controllers with High-Speed PWM, USB and Advanced Analog" PIC24EPXXX(GP/GU)810/814 datasheet, 2009 [Revised 2012].

[31] Atmel, "8-bit AVR Microcontrollers" ATmega328/P, 2016.

[32] Microchip Technical Staff, Universal Asynchronous Receiver Transmitter (UART), Family Reference Manual, PIC24EPXXX(GP/GU)810/814, Microchip Technology, 2009 [Revised 2013].

[33] Microchip Technical Staff, Inter-Integrated Circuit™ (I2C™), Family Reference Manual, PIC24EPXXX(GP/GU)810/814, Microchip Technology, 2007 [Revised 2014].

[34] Microchip Technical Staff, Serial Peripheral Interface (SPI) , Family Reference Manual, PIC24EPXXX(GP/GU)810/814, Microchip Technology, 2014.

[35] Arduino Uno Rev3. N.p., n.d. Web. 15 Apr. 2017. <https://store.arduino.cc/usa/arduino-uno-rev3>.

[36] "Mikromedia for PIC24EP." Development Board for Multimedia. N.p., n.d. Web. 15 Apr. 2017. <https://shop.mikroe.com/mikromedia-3-pic24ep>.

[37] Arduino - Serial. N.p., n.d. Web. 25 Apr. 2017. <https://www.arduino.cc/en/Reference/Serial>.

[38] Arduino - SPI. N.p., n.d. Web. 25 Apr. 2017. < https://www.arduino.cc/en/Reference/SPI>.

[39] Arduino - Wire. N.p., n.d. Web. 25 Apr. 2017. <https://www.arduino.cc/en/Reference/Wire>.

[40] Townsend, Kevin. Adafruit Bluefruit LE Shield Tutorial. Rep. N.p.: Adafruit, 09 October 2015.

[41] Ada, Lady. Adafruit BME280 Humidity + Barometric Pressure + Temperature Sensor Breakout Tutorial. Rep. N.p.: Adafruit, 24 July 2015. <https://learn.adafruit.com/adafruit-bme280-humidity-barometric-pressure-temperature-sensor-breakout/wiring-and-test> Web. 10 Apr. 2017.

[42] Ada, Lady. Adafruit MCP9808 Precision I2C Temperature Sensor Guide. Rep. N.p.: Adafruit, 03 April 2014. < https://learn.adafruit.com/adafruit-mcp9808-precision-i2c-temperature-sensor-guide/wiring> Web. 10 Apr. 2017.

[43] Agarwal, Tarun. "What is ZigBee Technology, Architecture and its Applications?" ElProCus - Electronic Projects for Engineering Students. N.p., 02 Oct. 2014. Web. 06 July 2017. <https://www.elprocus.com/what-is-zigbee-technology-architecture-and-its-applications/>.

[44] Seeedstudio Team. "W5500 Ethernet Shield v1.0." Seeed Wiki. N.p., n.d. Web. 07 July 2017. <http://wiki.seeed.cc/W5500_Ethernet_Shield_v1.0/>.

Appendix A – Answers to Exercises

[1] Every parallel data bit will not reach its destination at the same time due to a variety of conditions This limits the speed of the entire bus to that of the slowest data line because the receiver needs to wait till all the data bits have arrived. The second disadvantage is the phenomenon of crosstalk; the parallel lines interfere with each other and this effect increases with distance between the endpoints.

[2] The number of bits that can be sent is usually 8 but it is *device dependent* as some devices allow for the transmission of 10 bit frames. The parity bit is a way of low level error checking which involves counting the number of '1' bits in the data frame. If only one bit in the frame flips, the parity will change and the error flag will be toggled.

[3] Two lines are needed and the slave device knows whether it is a read or a write depending on the read or write bit sent with the slave address. Up to 127 devices can be connected on an I2C bus using just 2 lines, but for SPI, we need a slave select for every additional slave.

[4] No, in order to receive data on the SPI bus, you must send data first. This protocol is bidirectional by nature. If you need to receive something, most devices are set up so that reception of a dummy byte triggers sending data in the transmit register. Data to be sent must be pre-loaded into the transmit register.

[5] UART is used when high speed communication isn't a priority and it provides the advantage of being able to be implemented on any device using just 2 GPIO lines. SPI is used when high speed transmission is necessary; it has the highest throughput of the three protocols and it is easy to implement. However, a lack of pins is one of its biggest drawbacks; multiple slave select lines are needed for multiple slaves. The slave also has no data flow control; it is purely at the mercy

of the master. I2C is a nice common ground between the two protocols, it offers speeds up to 3.4 MHz, it is the king of flexibility, and it gives the slaves the ability to slow down transmissions.

[6] Services and characteristics are distinguished using their unique service IDs and the UUID (attribute ID) so that when data is sent, it is sent with the handle associated with the correct attribute.

[7] The first thing to do is to check the sensor datasheet and check what types of communication it supports. The next thing to do is to check what voltage level the sensor operates on; this is a critical step that will help avoid damage from occurring to either component. The third step to take is to check what speeds of transmission the sensor can support. Based on these steps and what kind of application you are trying to build, you can either connect the sensor using one the three protocols or purchase a new sensor that better fits your needs.

[8] The first thing you check is if you correctly wired the controller and the peripheral together. The RX of the controller goes to the TX of the peripheral and the TX of the controller goes to the RX of the peripheral. Reference voltage and ground also need to be hooked up correctly. If the hardware module doesn't take care of this, you need to check if TX is configured as an output and RX as an input. The next thing to check is if the baud rates are set up correctly for both components; if they don't match, transmission will not occur correctly. The final thing that needs to be checked is if any registers on the sensor need to be toggled; dig into the sensor data sheet to see if any registers need to be written first and in what order they need to be written. If none of these things work, use an oscilloscope to see if data is going out of the TX pin of your controller and if it is arriving at the RX pin of the sensor.

[9] The first thing to check is if the controller is wired properly to the sensor. The data line needs to be connected to the sensor data line and the clock line needs to be connected to the sensor's clock in line. Check if the voltage reference and ground lines are properly connected. Next, check if the speed of transmission set in the controller register is one that the sensor can support. Check if you are sending to the correct slave address; this address will be available in the datasheet of the sensor. Use an oscilloscope or logic analyzer to check for acknowledges coming from the slave as well as data flow in and out. The slave can indicate to the master that it is not ready for communication by pulling the clock line low; this is called clock stretching.

[10] The first thing to check is if the controller is wired properly to the sensor. The very first step in SPI communication is to pull the slave select line low; if you don't do this, the sensor will not respond to any commands. Using the datasheet, check what commands you must clock out of your controller in order to receive the data you are looking for. Since the SPI has no speed limits, you must make sure that the oscillator going into the SPI module on your controller is divided down to a speed that can be handled by the sensor. Make sure you pre-load data into your transmit register on the controller in order to be able to send data as soon as it is received from the sensor. A couple of back and forth transmissions may be necessary to start the streaming of data; this is sensor-dependent.

[11] With six pins left on your microcontroller and the need to connect three sensors, you can use any of the three major serial communication protocols. Basic asynchronous serial communication will use three RX lines and three TX lines, so you have just enough i/o lines. Using SPI will call for MOSI, MISO, SCK and three slave select lines, so you have just enough i/o lines. Using I2C will only need SCL and SDA, so you will have four i/o lines left. However, there are questions that need to be asked before you choose sensors that have any of these

modules on board. Does the data throughput need to be very high? Do I need to leave room for flexibility? If you don't need high speed, the best bet is to go with I2C because it will leave you four i/o lines for future use and you will get a decent data transfer rate. However, if the application is of higher criticality, you would be best served using SPI. This will come at the cost of the additional four lines. Then you could go about looking for sensors that have SPI or I2C controllers to help facilitate your interface.

[12] The attribute's data type and range need to be specified along with the assignment of specific ID to the attribute.