COMMUNICATION PROTOCOLS FOR EMBEDDED SYSTEMS

EE491: Independent Study with Dr. Fourney

Nathan Genetzky

Spring 2016

Table of Contents

[Introduction 1](#_Toc468793981)

[Theory 2](#_Toc468793982)

[Serial Communication 3](#_Toc468793983)

[I2C Protocol 3](#_Toc468793984)

[CAN Protocol 4](#_Toc468793985)

[SPI Protocol 5](#_Toc468793986)

[Cloud Communication 6](#_Toc468793987)

[Implementation 6](#_Toc468793988)

[I2C – Inter-Integrated Circuit 6](#_Toc468793989)

[I2C Protocol on Particle Device 6](#_Toc468793990)

[I2C Protocol on PIC18 7](#_Toc468793991)

[I2C Protocol on Raspberry Pi 7](#_Toc468793992)

[Monitoring I2C Communication with Oscilloscope 8](#_Toc468793993)

[Using I2C to interface with Peripheral Devices 9](#_Toc468793994)

[CAN – Controller Area Network 11](#_Toc468793995)

[CAN Protocol on Particle Device 11](#_Toc468793996)

[Particle Cloud Communication 12](#_Toc468793997)

[Conclusion 13](#_Toc468793998)

# Introduction

Electrical Engineers use various electronic devices to measure signals and control systems. These individual devices can accomplish many things, but with a communication system in place they can work together to accomplish even greater tasks. A microcontroller can communicate with various devices to expand its functionality. An embedded system can be interfaced with a device with greater processing power to allow more complex algorithms to be run or to coordinate with systems at other locations. Alternatively, a microcontroller could also utilize a sensor at a remote location. This paper will examine the theory and procedure to implement a few of these communication protocols.

## Literature Review

I2C was developed in 1980s by Philips Electronics [1]; the patent has since expired and the protocol is openly implemented. A consumer level explanation for I2C is provided by Texas Instruments [2]. An introductory tutorial for using I2C with Particle devices was provided by a github user “rickkas7” [3].

SPI was developed in the last 1980s by Motorola. It is one of the most commonly used protocols; for instance, all SD and MicroSD cards used SPI communication for data transfer. There are very few resources available for SPI with Particle, but Microchip released a great guide for using SPI with PIC microcontrollers [4]. An article by John Patrick provides an excellent comparison of Serial Protocols [5]; the comparisons are much more in-depth than the comparison provided by this paper.

The CAN bus was developed in 1986 and released for use in automotive systems [6]. Since then it has expanded to be used in various other systems due to its many features. There are no examples of using CAN for particle because the library was only just recently released [7].

There have been many projects demonstrated for specific applications with the Particle microcontroller on websites such as Hackster.io [8] and Instructables [9]. Typically, these projects are simply using GPIO on the Particle device and using prebuilt services to access the data from the Particle Cloud. The projects listed by these sites showcases some of the amazing web services available for IOT devices.

## References

[1] "The I2C Specification," Version 2.1, Philips Semiconductors.

[2] Texas Instruments. "I2C Guide." (n.d.): n. pag. I2C Guide. Texas Instruments, 2013. Web. 24 Apr. 2017. <<http://www.ti.com/lit/sg/sszc003e/sszc003e.pdf>>.

[3] Rickkas7. "Rickkas7/particle-i2c-tutorial." GitHub. N.p., 08 Apr. 2017. Web. 24 Apr. 2017. <<https://github.com/rickkas7/particle-i2c-tutorial>>.

[4] Microchip. ”Overview and Use of the PICmicro Serial Periperal Interface”. (n.d.) MicroChip, n.d. Web. 24 Apr. 2017. <<http://ww1.microchip.com/downloads/en/devicedoc/spi.pdf>>.

[5] Patrick, John. "Serial Protocols Compared." Embedded. N.p., 31 May 2002. Web. 24 Apr. 2017. <<http://www.embedded.com/design/connectivity/4023975/Serial-Protocols-Compared>>.

[6] "CAN in Automation (CiA)." History of the CAN Technology. CAN in Automation (CiA), n.d. Web. 24 Apr. 2017. <<https://www.can-cia.org/can-knowledge/can/can-history/>>.

[7] Infinity, Mdma, Bspranger, Peekay123, Rowifi, and Flyingfedora. "Photon CAN Bus [Completed]." Particle. Particle Community, 17 June 2015. Web. 24 Apr. 2017. <<https://community.particle.io/t/photon-can-bus-completed/12634>>.

[8] "Particle Projects." Particle Projects. Hackster, 24 Apr. 2017. Web. 24 Apr. 2017. <<https://particle.hackster.io/?sort=respected>>.

[9] "Particle Instructables." How To. Instructables, n.d. Web. 24 Apr. 2017. <<http://www.instructables.com/howto/Particle/>>.

# Theory

### Serial Communication

RS232 serial is a simple asynchronous communication between two devices. The rate of data transfer must be preconfigured and is usually limited to 230 KHz. Every frame of data requires a start and stop bit. Since the devices do not sharing the clock signal, it becomes very intensive to monitor incoming data because the serial port must be sampled much faster than the clock to properly read the bytes.

### I2C Protocol

Inter-integrated Circuit (I2C) Protocol is an efficient communication protocol that requires only two signal wires between devices, Fig. 1. The Serial Clock (SCL) and Serial Data (SDA) are pulled up to a logic one signal through a pull-up resistor; this allows any device on the network to pull either signal down to logic zero.

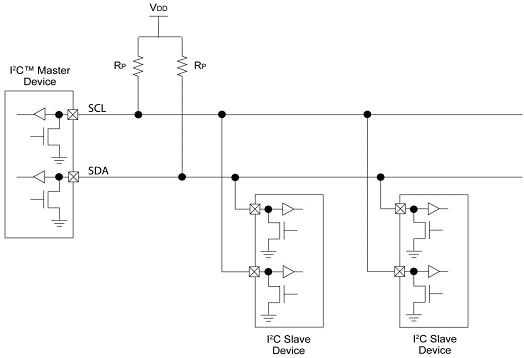


Figure 1. Circuit diagram of generic I2C communication[[1]](#endnote-2).

There are many advantages to I2C being a synchronous protocol. The incoming data does not need to be sampled at a rate faster than transmission because the clock indicates when the data should be sampled. The clock rate can vary and is not restricted to certain discrete values. The I2C master controls the clock speed but the slave can hold the clock low if it needs to pause the transmission to catch up – this is referred to as clock stretching.

Each device on the network has an address which it responds to. This address is included at the beginning of any new communication sequence; after sending the seven-bit address, starting with the MSB, (MSB first) then the bit indicating the direction of data transfer will be sent. This type of address allows the master to either send data to a slave or to request data from the slave.

The I2C communication between devices utilize special blocks. There are start, restart and stop conditions; these indicate the boundaries around messages. In-between a start and either a stop or restart condition there will exist one or more data blocks. Each 8-bit data block should be followed by a ACK or NACK condition. The slave device will pull SDA to a logic zero during the 9th clock pulse to indicate that it successfully received the data; however, if the slave does not do this then the SDA line will float to logic one and the master will read this as a NACK condition.

### CAN Protocol

The Controller Area Network (CAN) protocol is the most complicated of the protocols discussed in this paper. CAN uses differential voltages, cyclic redundant checks, and complex message identifiers. The automotive industry and many others make extensive use of CAN communication; CAN handles many devices and many different priority levels better than the other communication protocols discussed in this document. I2C would be a close second, but is limited to a single slave receiving a message (with the exception to a message sent to the general call address).

There are many advantages that span the various abstraction layers. At the physical level, the differential input provides noise isolation and ensures that the bus is only in either logical on or off state. The CAN protocol performs many functions in the transfer layer: fault confinement, error detection, message validation, acknowledgement, arbitration, message framing, transfer rate and timing, and information routing. At the object layer, CAN provides message filtering, message handling and status handling. Similar to how each I2C message is composed of blocks, a CAN message consists of various frames as shown in Fig. 2. Unlike I2C communication, much of the error detection is handled in hardware and is built into the format of a CAN Message, Fig. 2. This introduced more hardware and complexity.

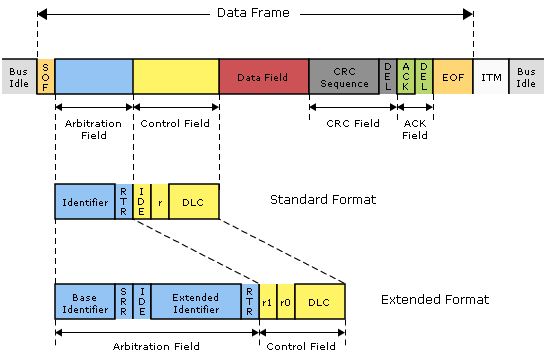


Figure . Blocks within CAN message frame .

The CAN Bus Transceiver is external to the microcontroller and handles the interpreting the differential voltage between CAN High and CAN Low. The CAN Protocol Machine handles the transfer of data from the transmit buffer to the transceiver and from the transceiver to the Acceptance Filter. The Acceptance Filter of the CAN core, Fig. 3, will only pass the message through if the identifier of messages matches conditions set by the receiving micro controller. When a message matches the conditions, then the Host Controller Interface will signal an interrupt that will alert the microcontroller program. The micro controller program also uses the Host Controller Interface to queue outgoing messages or to read incoming messages.

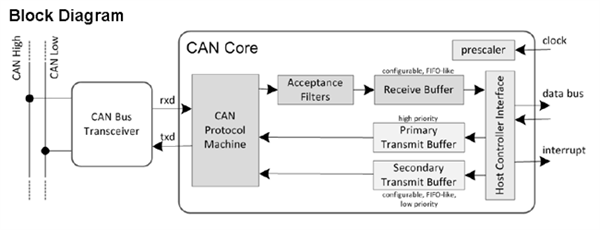


Figure . Block Diagram of components used in CAN communication.

### SPI Protocol

The Serial Peripheral Interface (SPI) typically requires more signal wires than the other interfaces discussed in this document but achieves very fast full-duplex communication up to 10 MHz. It supports full-duplex communication by allowing each device to simultaneously send and receive data; for instance, the master will send out data of MOSI but receive data simultaneously from MISO. It is possible to omit the MISO wire and only have half duplex communication. The largest disadvantage is that a system with multiple slave devices requires that each slave has a dedicated Slave Select / Chip Select. The other three signals (MISO, MOSI and SCK) can be shared between all the devices on the SPI bus.

Every SPI transaction requires that both devices transmit and receive a byte of data. The master controls SCK and the slave devices are not able to modify the clock as in I2C. There are four modes of SPI communication which are variations of the clock edge and clock polarity. An in-depth discussion of these modes of operation is covered sufficiently on web resources and will be excluded from this paper.

### Cloud Communication

Cloud communication encompass many cutting-edge communication techniques that are crucial for Internet of Things (IOT). There are four major components to Internet of Things: Back-End services, internet, local network, and the “thing”. The “thing” typically incorporates a microcontroller or a microprocessor. The local network could connect to the device in various ways (such as Ethernet, Wi-Fi, or Bluetooth). The Back-End services could involve data logging, data analysis, data routing, user access or user control.

# Implementation

This section will discuss the implementation of communication protocols. The master or initiator of communication is one of the following devices: Particle Photon, Raspberry Pi or PIC18. The master will communicate with another device of its kind or with peripheral devices such as an LCD character display or an IO Expander.

## I2C – Inter-Integrated Circuit

The circuit for most I2C communication is very simple because only two wires are required for the communication. pull-up resistors were used because this value is recommended for transmission speeds between 100-400 kbps. How I2C was utilized on different micro controllers and with different peripheral devices will be discussed below.

### I2C Protocol on Particle Device

The Particle framework supports similar commands as Arduino. These functions make it very easy to communication using I2C without having to become overly familiar with the device specific details. There is an Interrupt Service Routine (ISR), the function provided to OnReceive(), called whenever data is received from the master, or requested from the slave. Begin() will configure the slave address and a beginTransmisison() represents the start condition. Any subsequent values used with send() will be sent to that slave until the endTransmission(); note that for this library the data is merely queued up (length can’t exceed buffer size) and all the communication occurs when endTransmission() is called.

Table : I2C (Wire) library functions provided by Particle.

|  |  |
| --- | --- |
| Setup and receiving messages   * Wire.begin() * Wire.OnReceive() * Wire.available() | Transmission of message   * Wire.beginTransmission() * Wire.endTransmission() * Wire.send() |

### I2C Protocol on PIC18

Using the I2C protocol on the PIC18 requires knowledge of the internal registers on the microcontroller. There is a single interrupt service routine (ISR) that indicates that something occurred involving I2C; however, the status bits must be read to determine what exactly happened. When the code was created, our goal was to create functions similar to those provided by the Particle device so that the C++ code utilizing the I2C code could be the same (or very similar).

The PIC18F8722 implemented the I2C protocol as part of the Master Synchronous Serial Port (MSSP) Module. In order to configure, control, and interface with this module the following registers were utilized: SSP1CON1, SSP1CON2, SSP1STAT, SSP1ADD, SSP1BUF. In addition to these registers, there were specific bits (Register.Bit) utilized for the interrupt (PIR1.SSP1IF, PIE1.SSP1IE, IPR1.SSP1IP) and for controlling the direction of the SDA and SCL ports (TRIS).

### I2C Protocol on Raspberry Pi

The Linux kernel provides easy access to the I2C functions on the Raspberry Pi (RPI). Messages can be sent to slaves right from the command line. There are device files for each I2C bus connected to the device, such as “/dev/i2c-0. “ioctl” commands can be used to control the I2C bus, and then the device file can be used with file “read” and “write” commands. The kernel also provides a C library, “i2c-dev.h”, for I2C communication.

There are are also libraries available for C++ and Python for I2C communication; such as “pigpio”, “WiringPi” and “Gnublin”. The python code to read a sensor value is very short. There are specific libraries that allow even more abstract I2C derived buses (Such as PMBUS, SMBUS).

Kernal support for I2C can be added manually by adding the following to modules to your “/etc/modules” file:

i2c\_bcm2708

i2c-dev

Then modify the “/boot/config.txt” file to include the following two statements:

dtparam=i2c1=on

dtparam=i2c\_arm=on

To determine what devices are on the bus you can simply run: “sudo i2cdetect -y 1”, where “1” is the i2c bus to use as the master.

### Monitoring I2C Communication with Oscilloscope

The MSO8014A Mixed Signal Oscilloscope by Agilent Technologies was used to observe the communication at the physical layer. We desire to observe digital signals; more specifically, some of these signals are part of I2C communication. First the oscilloscope was changed to digital mode. Since we are using a 3.3V microcontroller the threshold was set to TTL, which is 1.4 volts. This oscilloscope has a special trigger mode for I2C and many other communication protocols, as shown in Fig. 4. When analyzing I2C communication there are many conditions that can be used to trigger the start of measurement; these conditions are shown in Fig. 5. The trigger settings were modified to trigger on the start condition.

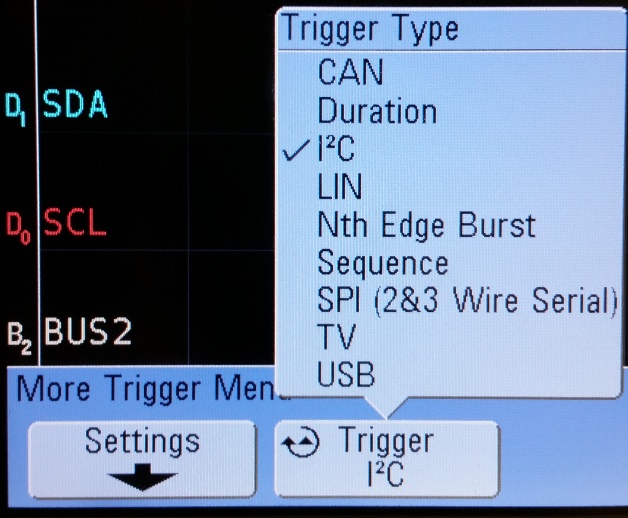


Figure 4. Trigger types available on MSO8014A Oscilloscope.

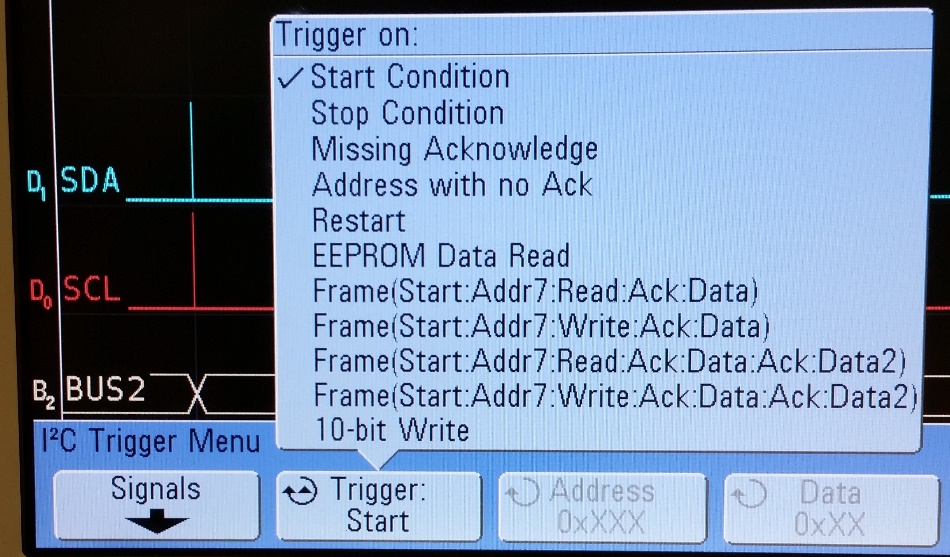


Figure 5. Trigger conditions available on MSO8014A Oscilloscope.

### Using I2C to interface with Peripheral Devices

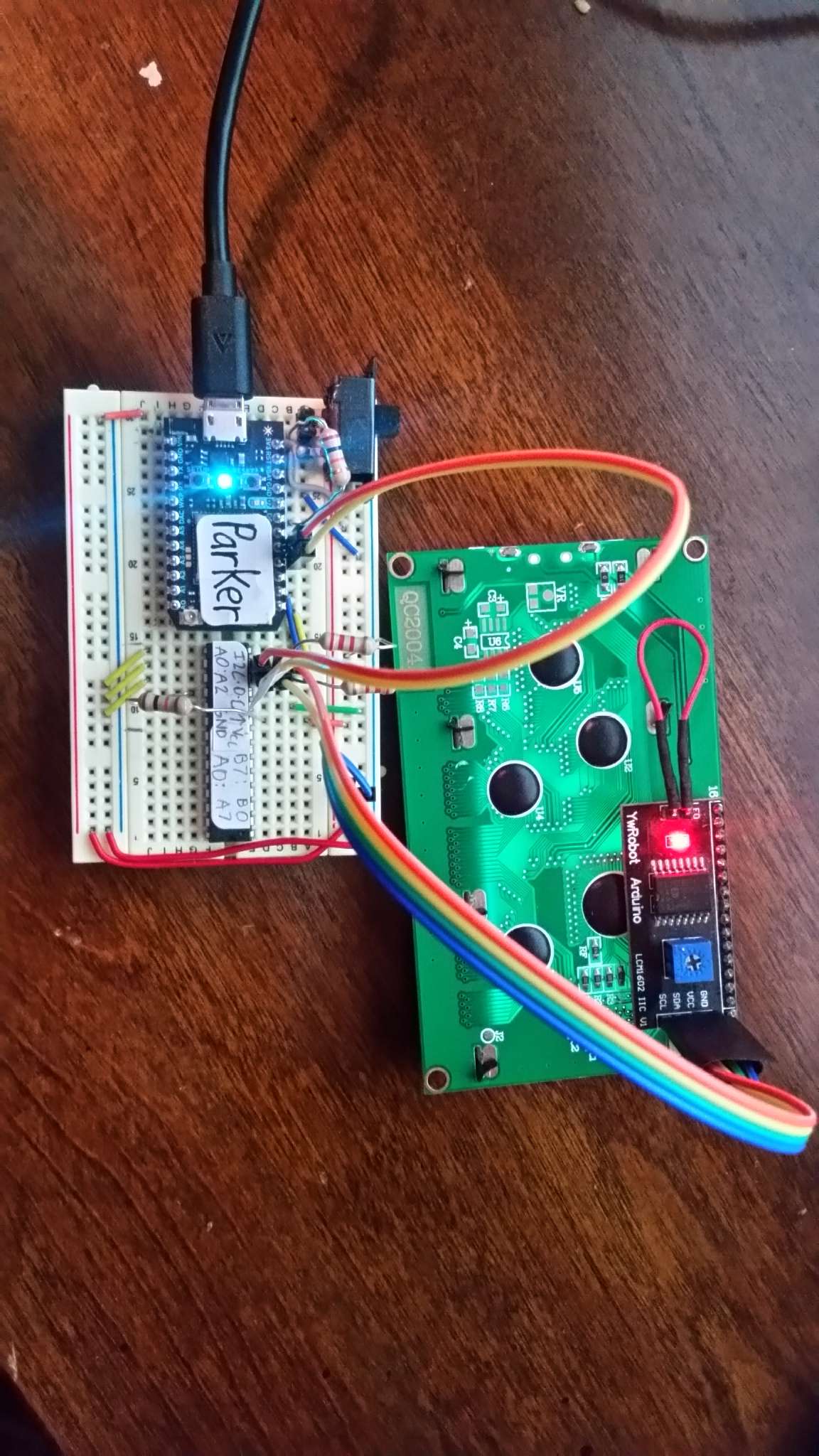
**Multiple devices and peripherals were connected to the I2C network to demonstrate how to interface with multiple slave devices. A IO expander and LCD display were connected to a Particle device, Fig. 6. The next two sections will describe the procedure in greater detail.

Figure 6. Circuit diagram of two I2C slave devices communicating with Particle device.

#### I2C IO Expander – MCP23017

The address of the IO expander was set to 32 (0x20) by grounding all the address pins (A2, A1, A0). The device address (referred to as an Opcode in the datasheet) must be provided after any Start or Restart block to indicate the type of operation that should be performed. Messages sent to the IO Expander utilize the I2C blocks mentioned above but the data block can represent commands or data; it can be either a register address or data. A register address can be specified after a write is specified; this allows the device’s registers to be set with the data provided after the register address.

After performing a few simple operations with the device, the code was adapted to utilize a library written for the MCP23017, and the code’s functionality was expanded. This allowed multiple applications to be quickly developed with an abstract interface for the IO Expander. The idea behind this abstraction is that a user should be able to treat the IO Expander’s GPIO as though it was GPIO directly on the micro controller.

The ports on the IO expander were connected to multiple devices that required many GPIO ports. Interfacing with these devices was viewed as outside the scope of this project and was merely mentioned to demonstrate the usefulness of such peripherals. The following devices were connected to the GPIO: Seven Segment display, Button Matrix and LED Matrix.

The I2C communication with the IO Expander was observed with the help of the previously mentioned oscilloscope. The various components of the communication have been identified; this includes the slave address, and the command for the slave, Fig. 7. After the command was sent, then the I2C data observed after the command were arguments or values for the command as seen in Fig. 8.

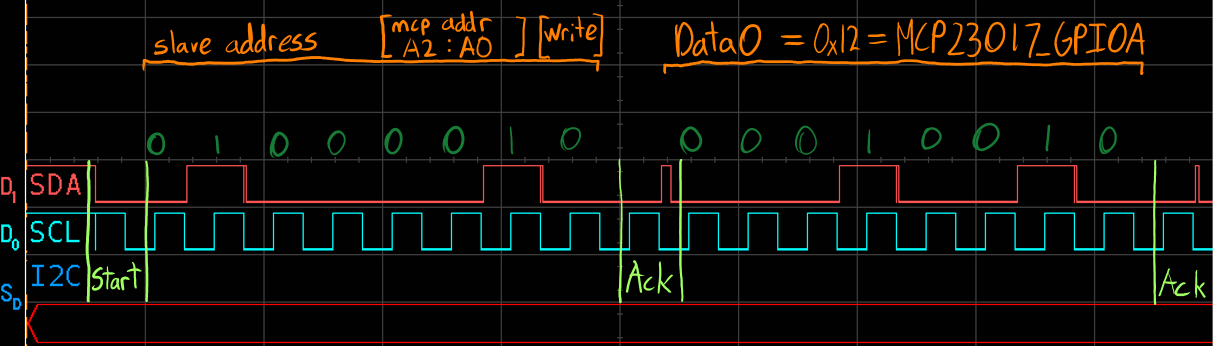


Figure 7: Screenshot showing the first part of the communication with the slave.

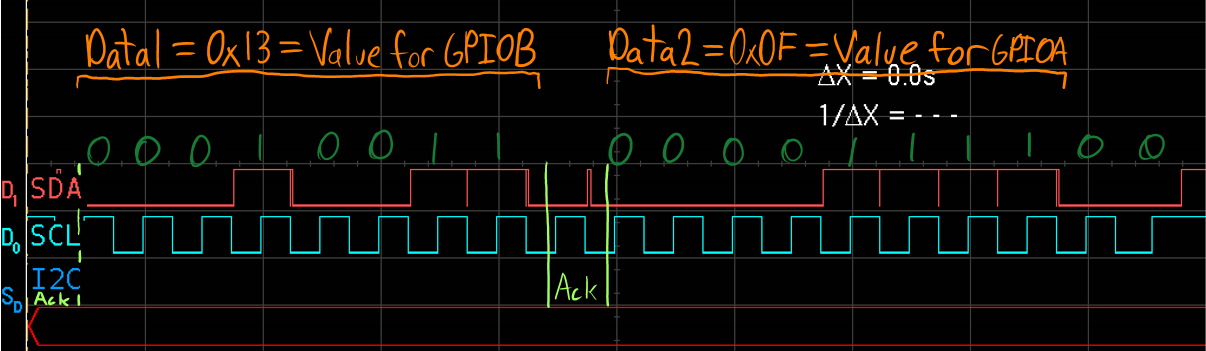


Figure 8: Screenshot showing the latter portion of the communication with the slave.

#### I2C Character LCD – HD44780

It was important to verify that the contrast knob of the LCD was in a position that ensures text is readable on the display. The LCD interface supports data communication via a set of parallel data pins. There is a IC provided as a module to add the LCD to the I2C bus; it is required to convert I2C commands into LCD commands put on the parallel data pins. In order to save time, I choose to use the LCD library built for the HD44780 model LCD. I could then supply commands directly to the LCD or print values to the screen.

## CAN – Controller Area Network

To communicate with the CAN network, a High-Speed CAN Transceiver (MCP2551) was used; the MCP2551 converts the transistor-transistor logic (TTL) logic applied at the input into the higher voltage differential signal required for the CAN protocol at the output. The chip also provides protection by isolating the micro controller from the CAN network. The circuit for CAN communication is slightly more complicated than the previously mentioned circuits. It does not require a pull-up resistor, but rather it requires a resistor between CANH and CANL.

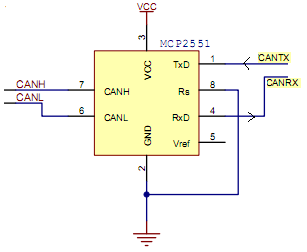


Figure 9. Typical wiring diagram involving MCP2551[[2]](#endnote-3).

### CAN Protocol on Particle Device

When connecting the MCP2551 chip to the Photon device it was important to distinguish between (+5V) and (+3.3V) lines. The CAN transceiver should be powered with (+5V); therefore, this must come from the “VIN” pin and the device must be power via USB.

The Circuit is not overly complex, but will be discussed as two halves. The circuit connected to Matt will be described, and Parker uses the same circuit.

* The two share the same CANH and CANL nodes.
* They could also share the same 5V and GND.
* Termination resistors are between CANH and CANL on both sides, R1=R2=120 Ohms.
* Vin supplies 5V to MCP2551.
* The pins (D0, D1) on the Photons output 3.3V but are 5V tolerant.

Using the CAN protocol on the Particle Device required more programming work than expected. Once this portion was programmed, it was very easy to provide functions to communicate between two devices. For this project ASCI characters/text were sent between the two Photon devices, Fig. 10. The next step with using CAN communication would be finding peripheral devices that could be communicated with; this would ensure that it was a practical application of CAN.

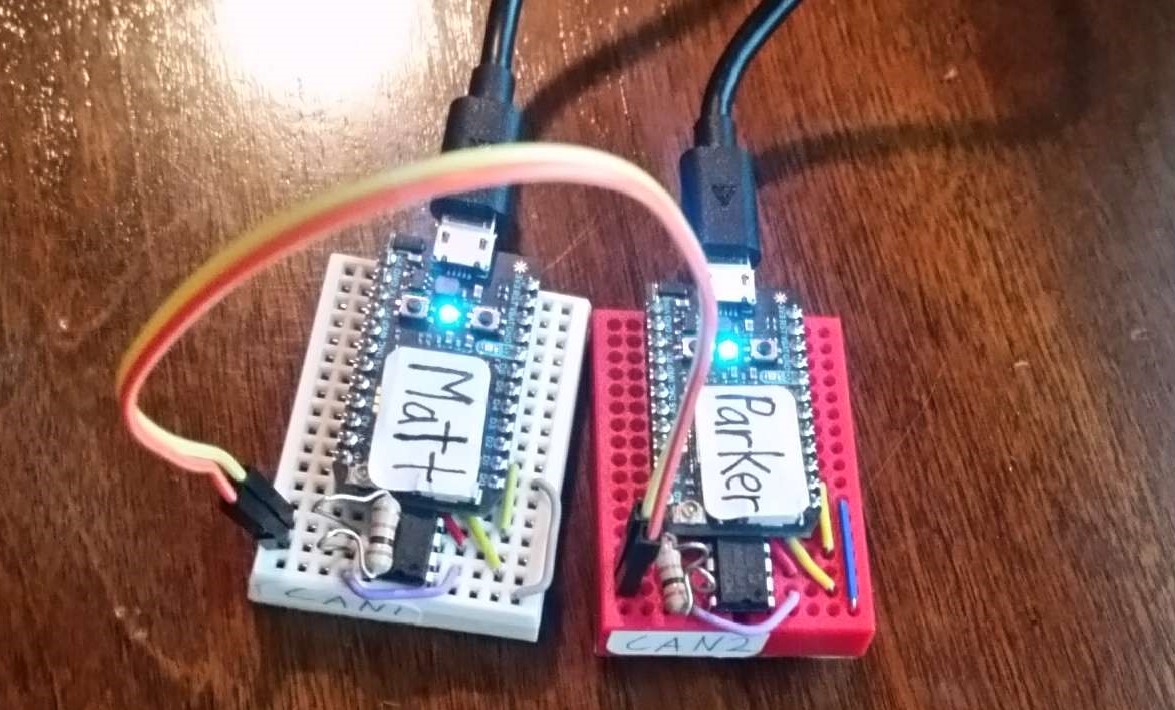


Figure 10. Photograph of the Particle ready for CAN communication.

## Particle Cloud Communication

When using the Particle device there is a specific set of communication methods specific to the Particle Cloud. The next two paragraphs are an explanation of the Firmware exposed through Cloud API.

**Variables**: A variable that is defined in the firmware can be made publicly available. The type can be int, double, or string; the string is far by the most capable. **Function**: A function that takes a string and returns an integer; it can be made publicly available. (Note that the argument type is Arduino’s String type).

**Subscribe**: A device can register to receive any events that start with certain characters. When an event is received, the topic and the data are sent to a function. Particle broadcasts standard events involving your devices. By using Webhook responses the device can also be provided with information requested from the internet, but this is a more advanced use. **Publish**: A device can broadcast an event with a topic and data. Using Webhooks or IFTTT.com, a service that allows web services to work together, allows actions to be triggered when certain events are broadcasted.

**Webhooks**: Webhooks are a little complicated but very helpful. Webhooks cause a message to be sent to a website with are particular payload (data/information). Webhooks can also capture the response and provide it to the device subscribed to a particular event.

# Conclusion

The I2C and CAN protocol were investigated. I2C proved to be easy to implement; there are also many commercial devices that are ready to communicate with this protocol. The CAN protocol was more complicated to implement, but is suitable for more complicated systems. The cloud communication is much different that the previously mentioned protocols; this communication is more abstract than the others.

The investigation on these communication protocols has been educational, but also brought future opportunities to light. The demonstrations provided in this paper could be viewed as a proof of concept for something much greater. It would be possible for a CAN or I2C capable peripheral device to be connected to a Back-End service. This Back-End service could allow an end-user to control the device, or provide the user with analytics derived from the data connected from the device.

1. <http://www.edn.com/design/analog/4371297/Design-calculations-for-robust-I2C-communications> [↑](#endnote-ref-2)
2. <http://marco.guardigli.it/2010/10/hacking-your-car.html> [↑](#endnote-ref-3)