Digital Communications

Talking to Electronics

Richard Lin

**Application:** You’ve played with microcontrollers, but all we've done so far is blink LEDs. However, it can do so much more - we just need a bit more hardware, and a way to communicate with it.

**On our cars:** There's only so much a microcontroller alone can do. Devices like the graphic display, battery monitor, and even simple sensors require external hardware and the code to interface with it.

Contents

[Revision History 2](#_Toc369223313)

[Licensing 2](#_Toc369223314)

[Code 2](#_Toc369223315)

[Parts List 3](#_Toc369223316)

[Introduction 4](#_Toc369223317)

[Basics: Communications 4](#_Toc369223318)

[Basics: Analog and Digital Signals 4](#_Toc369223319)

[Reading Waveform Plots 4](#_Toc369223320)

[Digital Communications 5](#_Toc369223321)

[Digital Communications Standards 6](#_Toc369223322)

[Digital Communications Standards 7](#_Toc369223323)

[UART, Universal Asynchronous Receiver-Transmitter 7](#_Toc369223324)

[SPI, Serial Peripheral Interface 8](#_Toc369223325)

[I2C, Inter-Integrated Circuit 11](#_Toc369223326)

[Other Protocols 12](#_Toc369223327)

[Higher Levels 12](#_Toc369223328)

[On microcontrollers 12](#_Toc369223329)

[Lab 0: Warm-up, Computer-Microcontroller UART 13](#_Toc369223330)

[Circuit Schematic: 13](#_Toc369223331)

[Lab 1: Basic Microcontroller-Microcontroller UART 16](#_Toc369223332)

[Lab 2: Transmitting Data 17](#_Toc369223333)

[Extra for Experts: Accelerometer SPI 19](#_Toc369223334)

[Introduction 19](#_Toc369223335)

[Circuit Schematic: 19](#_Toc369223336)

[Accelerometer Introduction 19](#_Toc369223337)

[Lab Manual 20](#_Toc369223338)

[Conclusion 21](#_Toc369223339)

# Revision History

|  |  |  |
| --- | --- | --- |
| **Version** | **Date** | **Changelist** |
| 1.0 | Fall 2011 | Initial release |
| 1.1 | 26 Sep 2012 | Reformatting, various typo fixes, diagram improvements  Fixed code |
| 1.2 | 30 Sep 2012 | Split Lab 1 into Labs 1, 2 |
| 1.3 | 11 Oct 2013 | Update diagrams and code for BRAIN |

# Licensing

Creative Commons License

This document (including source elements, like diagrams), is licensed under the Creative Commons Attribution-ShareAlike 3.0 Unported License (<http://creativecommons.org/licenses/by-sa/3.0/deed.en_US>).

While commercial use is not prohibited under the terms of the license (the drawbacks outweigh the benefits, nor is a solar car team going to start suing people for obscene amounts over copyright infringement), you shouldn’t be shamelessly trying to turn a profit off someone else’s work. What exactly this means is completely up to you – but please behave like responsible adults. Keep in mind that you must license any derivative works under a similar license – which gives recipients redistribution rights.

# Code

You can download the code used in the lab on GitHub either through the online interface:

<https://github.com/CalSol/training-code>

or a local git client:

git clone https://github.com/CalSol/training-code.git

# Parts List

|  |  |  |  |
| --- | --- | --- | --- |
| **Item** | **Image** | **Function** | **Notes** |
| **Breadboard** | C:\Users\Ducky\Desktop\calsol-training\ee1\images\breadboard-internals.png  (showing internal connectivity) | Prototyping work area |  |
| **BRAIN** | https://calsolbrain.googlecode.com/svn/wiki/img/Brain2Pins.jpg | Microcontroller |  |
| **Resistor**  **330 ohm** |  | Current limiting for LEDs |  |
| **Switch** | C:\Users\Ducky\Desktop\calsol-training\ee1\images\comp_switch.png | Digital input |  |
| **Potentiometer** | **C:\Users\Ducky\Desktop\calsol-training\ee1\images\comp_pot.png** | Analog input |  |
| **LED** |  | Pretty lights! |  |
| **Extra for Experts** | | | |
| **Digital Accelerometer** |  | Sensor with digital interface | These sensors sense acceleration (including gravity) in one, two, or three axes |

# Introduction

## Basics: Communications

So far, you have been able to do some interesting stuff with a microcontroller, like blinking an LED or reading the state of a switch. But, within the chip, there's only so much hardware - for example, you can't communicate on the Internet, detect motion, or even read the temperature.

Therefore, you need to go off-chip with external hardware. Which also means that you need a way to talk with that hardware...

## Basics: Analog and Digital Signals

Electronics talk with each other through **signals** - an electrical quantity (such as voltage or current) which is varied to convey information. You've already played with a simple signal by varying the voltage at an analog input pin using a potentiometer.

Signals can be divided into two types: analog and digital.

**Analog signals** are ones that can vary over a continuous range. The aforementioned potentiometer generates an analog signal - as you turn the potentiometer, the voltage changes continuously. Even if you turn the knob ever so slightly, the voltage will also change ever so slightly[[1]](#footnote-1).

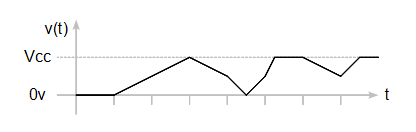
**Digital signals**, however, only vary over a discrete range. In electronics, digital usually means 2 levels: either high or low (0 or 1). The switch is an example: it is either open or closed (there is no in between) - and consequently, the voltage at the input pin is either near the positive supply voltage (5v) or ground (0v).

Notice how I said "near". In reality, voltage is an analog quantity, and due to the presence of various factors (like noise), the voltage at the input pin is not exactly 0v or 5v. Therefore, hardware exists inside the chip to determine if the input is closer to a logic 0 or logic 1, and produce the corresponding output. In this sense, digital signals can be seen as more resistant to interference or noise.

## Reading Waveform Plots

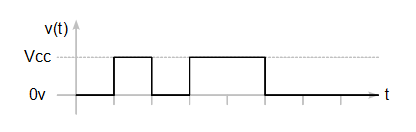
Before we go into any more detail about communications, it is important that you understand signals and how they are graphically represented.

As stated before, a signal is an electrical quantity which varies to convey information. The most common way to achieve this is to vary voltage with time, which can be graphically depicted as below:

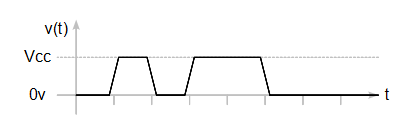


The Y (vertical) axis represents the voltage on the wire, while the X (horizontal) axis represents time. In the waveform above, the voltage starts at zero, then slowly increases to (supply voltage), then decreases back down to zero, and so on. Since the signal isn't neatly confined between two values, it is an example of an analog signal.

Digital signals are similar, except they take on a finite number of values (usually 2 - high, , and low, ). An example digital signal graph:



However, one thing to note is that realistically, signal transitions (low-to-high or high-to-low "edges") are not actually that sharp. Therefore, digital signals are also sometimes represented like this:



Although the signal takes on values between and , those are approximated to one or the other by receiving hardware.

## Digital Communications

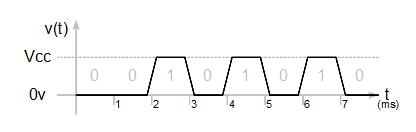
While either reading a 1 or 0 is useful for a switch, it doesn't scale. What if you needed to send an 8-bit number?

One way to do it is to have 8 digital lines, one for each bit. This is called a **parallel** line.

But what if you don't want to use 8 lines? This is where the **time-varying** signals comes in. A constant 1 or 0 isn't very interesting, but if you could send the bits of the number one after the other, then you can do a lot more. This is called a **serial** line.

As an example, suppose you wanted to send an 8-bit number, say 42. In binary, 42 is equal to 00101010, so a simple way to transmit that would be to send one bit every millisecond. The receiver would then **sample** (read) the line every millisecond, determining whether a 1 or 0 was sent, and reconstruct the data accordingly.

Here is that that signal would look like, assuming you transmit starting with the most significant bit:



## Digital Communications Standards

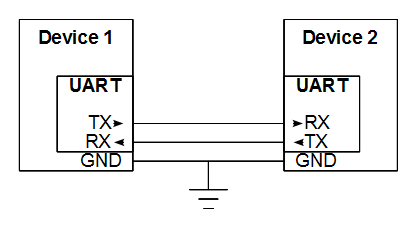
We have a way to send complex digital quantities now. But that is just one way. What if you thought a millisecond was way too slow since you want to send several million bits per second? What if you thought 8 bits wasn't large enough? As you can see, there are many variations on digital communications. Several common ones you are likely to encounter will be discussed here.

Note that the information here is just a high-level overview, and if you are truly interested, you can find more information on Wikipedia.

# Digital Communications Standards

## UART, Universal Asynchronous Receiver-Transmitter

In this scheme, bidirectional communications uses two lines per device: TX (transmit) and RX (receive). These should be pretty straightforward - to have two devices talk, connect the transmit port of one device to the receive port of the other.



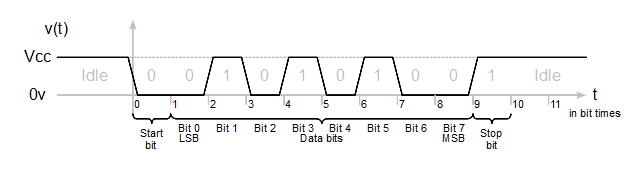
Note that in addition to connecting the RX and TX lines, you need to make sure that both devices have a common ground.

Also, note that the UART is typically a module within a larger device, like a microcontroller.

When the line is idle, it is at a high level. When a device wants to communicate, it first sends out a start bit (low level). This is followed by data, which is sent out one bit at a time, starting with the least significant bit (LSB). After that, it ends with one or more stop bits (high level).

The number of bits per second (which consequently determines the time per bit) is specified by the **baud rate** (commonly 9600, 19200, and 38400), and the number of bits per unit of data sent (including any overhead) is the **frame length**. Usually, one byte (8 bits) of data is transferred at a time with one start bit and one stop bit. To work properly, both devices must be configured with the same transmission parameters (like baud rate) beforehand.

The signal waveform looks like this:



## SPI, Serial Peripheral Interface

SPI, also called "four-wire serial," is a **master/slave bus**, where the master device initiates communications, and the slave devices only communicate when requested to. It is also a **shared bus** interface, where some lines can be shared between multiple devices. However, there can only be one master per bus.

The 4 signals in a SPI interface are CS (chip select), SDI (serial data in), SDO (serial data out), and SCK (clock). Variations on these names exist. In particular, note that the data in / out are from the devices point of view, so another naming convention is MISO (master in, slave out) and MOSI (master out, slave in).

The **CS line** indicates when a device is being talked to and is controlled by the master device. It is high when idle, and goes low when the master wants to send data to a particular device. There must be one CS line per slave; all other lines may be shared.

The **MISO and MOSI lines** are used for data transfers. During a data transfer, both are active at once (both the master and slave send data), and therefore SPI is a **full-duplex** bus.

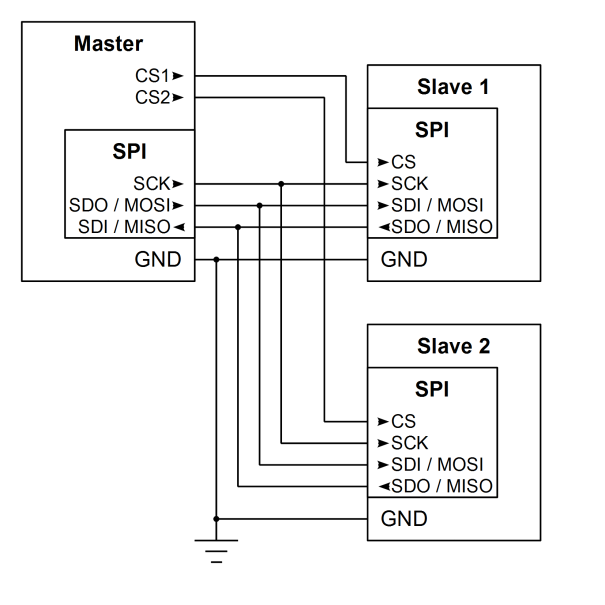
The **SCK line** is used for clocking and is controlled by the master device. The data on MISO and MOSI are either changed on the rising edge or falling edge (configurable parameter) of SCK. Additionally, the clock signal can be set to either high when idle or low when idle.

The clock signal means the devices don't need to agree on timing beforehand – they only need to monitor this line. However, all devices have frequency limits, above which data will be corrupted.

To add more slave devices to the line, just connect the device to the SCK, MISO, and MOSI lines. However, keep in mind that each device needs its own CS line so it knows when it’s being addressed.

The number of bits per data transfer is usually 8, but other sizes (like 12-bit and 16-bit) also exist.

A SPI circuit looks like:

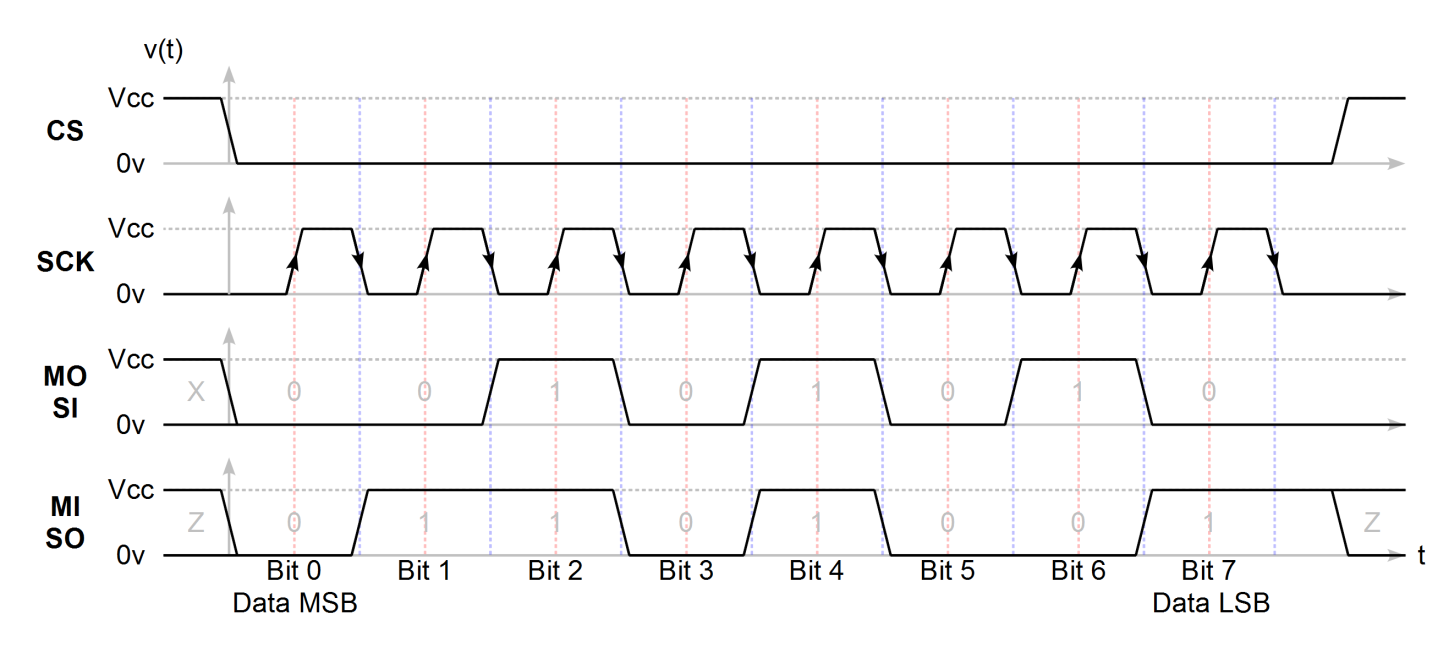


Note that on the master, the CS output pins are not part of the SPI module. This is usually the case on microcontrollers - the CS pin is just a normal GPIO (general purpose input/output) pin and is controlled in software.

On slave devices, however, the CS pin is part of the SPI module as it indicates when data on the bus is intended for a particular device.

A SPI data transfer looks like:

(note: settings are clock low when idle and data changed on falling edge, also known as mode 0, 0)



You may see some new data symbols here:

* **X** means an indeterminate value
* **Z** means a floating value - the line is neither pulled high or low. The Z stands for high-Z, or high-impedance.

At the beginning, the master pulls CS low, selecting the slave device. At this point, the slave device actively drives the MOSI pin.

Then, the master starts sending clocks (pulses on the SCK line). In this mode, data is read from the line every rising edge (red lines), and new data (the next bit) is put on the line every falling edge (blue lines). After 8 edges, the data transfer is complete, and the master pulls CS high, deselecting the slave device.

Data is sent most significant byte (MSB) first, so in the diagram above the master sends the decimal digit 42.

## I2C, Inter-Integrated Circuit

I2C, also called "two-wire serial," is also a master/slave bus. It is also a shared bus interface, but unlike SPI, it only uses two wires, both of which are shared between slave devices.

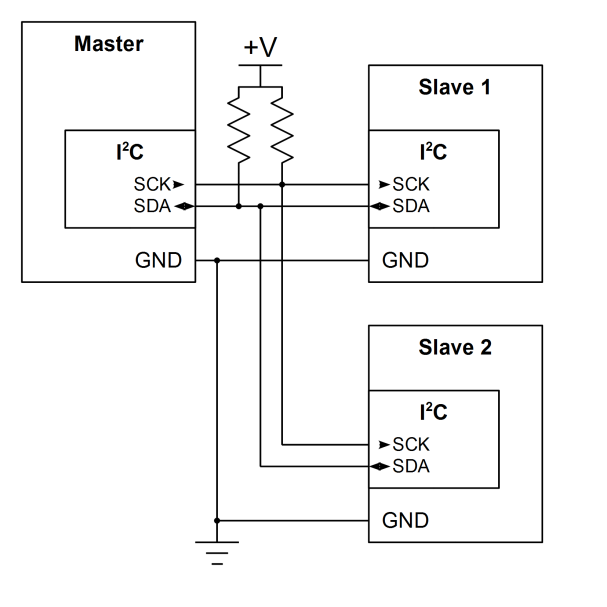
The two lines are SDA (serial data) and SCL (serial clock).

The **SCK line** is a clock line just like SPI. Unlike SPI, the clock edge transition on which new data is put on the data line is defined. The master sends clock signals.

The **SDA line** is a bidirectional data line. Both master and slave devices communicate on this line, and the protocol specifies who transmits when.

Both lines are connected to through a pull-up resistor. When transmitting, devices modulate the line by pulling it low or doing nothing. In this scheme, it is impossible to have a dangerous bus conflict where two devices attempt to drive a line to opposite voltages (though, a conflict condition may result in data corruption).

An I2C circuit looks like:



Unlike SPI, there is no individual chip-select per device. Instead, each device now has a 7-bit or 10-bit address which is transmitted at the beginning of each message.

The format of each message is as follows:

**Start condition** (abbreviated **S**, SDA high-to-low transition while SCL high) indicates the start of a message.

**Address** of the slave device. Like all I2C data, the data bit is only valid when SCL is high and bits are shifted out MSB first.

**Directionality bit** indicating if the rest of the message is a write operation (master to slave, directionality bit low) or read operation (slave to master, directionality bit high).

**ACK or NACK**, where the addressed slave drives SDA low to acknowledge.

One or more bytes of **data**. Each byte is followed by an ACK from the receiver. In the case of a slave-to-master transfer, a NACK to end the transfer so that the master can assert a stop condition.

**Stop condition** (abbreviated **P**, SDA low-to-high transition while SCL high) indicates the end of a message. At this point, the master releases the bus, and other masters can transmit.

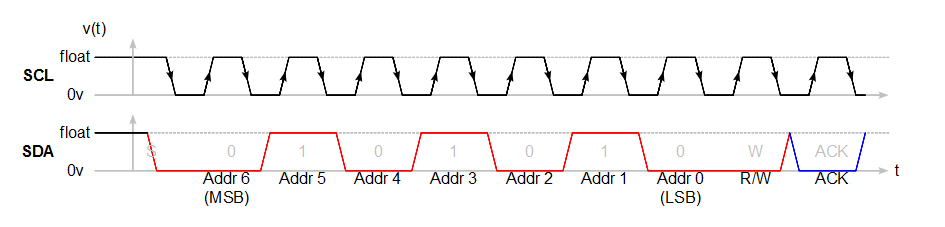
Note that instead of a stop condition, the master may send a **repeated start** (abbreviated **R**, same waveform as start condition) to transmit another message without giving up the bus. This is useful when writes and reads need to be atomic – like when a write sets a memory address to read.

Don’t worry about remembering all the little details of the I2C protocol. Most datasheets for chips communicating over I2C will specify what is expected, as long as you know the basics.

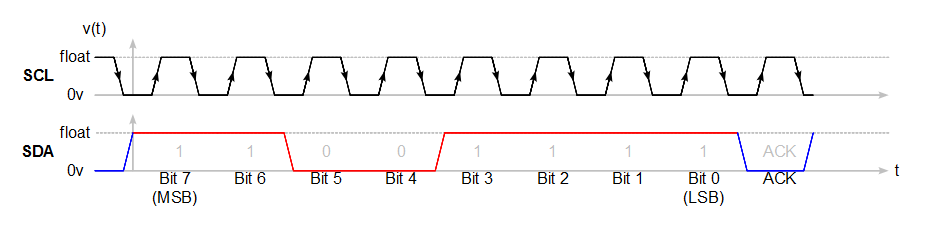
Since all that is a bit abstract, here are some waveform diagrams:

Notation: SDA is **red** when master is transmitting or **blue** when slave is transmitting.

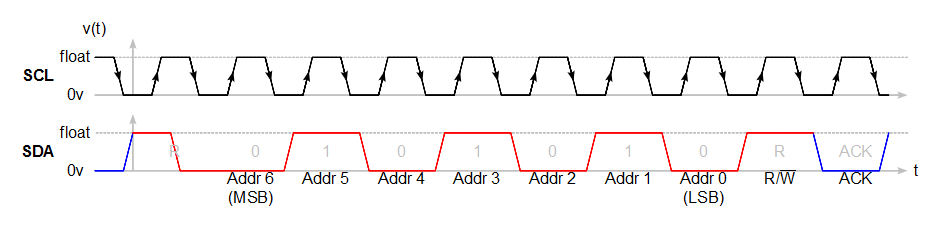
**Addressing** (to device address 0x2A, master-to-slave write mode, slave ACK):



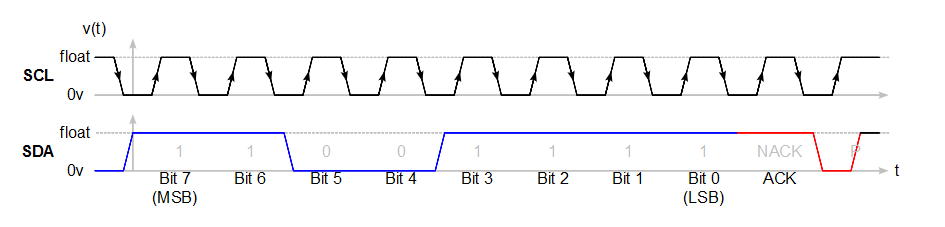
**Data Transfer** (send 0xCF, slave ACK):



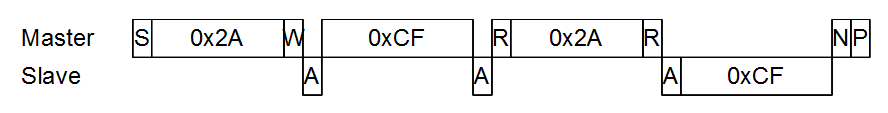
**Repeated Start** (to device address 0x2A, slave-to-master read mode):



**Data Transfer and Stop** (receive 0xCF, master NACK, stop):



Sometimes, I2C messages are represented in a more compact form using abbreviations. The above transfers could also look like:



## Other Protocols

UART, SPI, I2C are the most common protocols you will encounter on CalSol’s boards. Parallel is also another one, but the high pin requirement makes it less common.

Many other protocols exist, including CAN (which will be covered in a future lab), USB, PCIe, FireWire, and much more.

## Higher Levels

The above protocols only define how data is exchanged, and not the meaning of the data exchanged. That is one abstraction level higher, and is not covered by the standards.

For slave devices which communicate through one of these standards, the specifics are usually well documented in the chip’s datasheet so that you can implement an interface with them.

## On microcontrollers

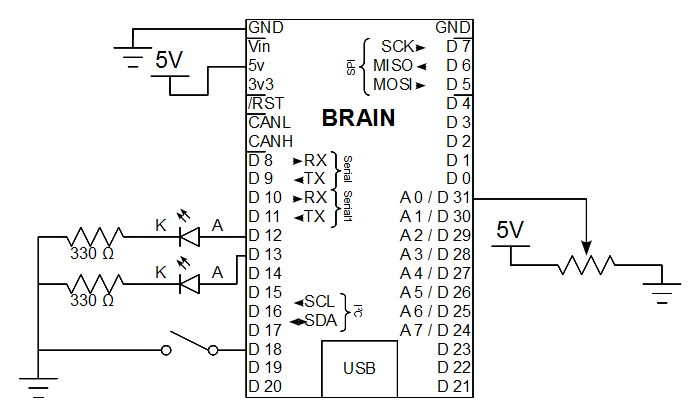
Most microcontrollers have modules built-in to communicating through UART, SPI, and I2C. Typically, you only deal with transmitting or receiving an entire data word, and won't have to worry about the fine details of when a bit changes.

In this lab, you will work the three most common inter-chip communications protocols: UART, I2C, and SPI.

# Lab 0: Warm-up, Computer-Microcontroller UART

1. Set up your hardware as follows:
   1. Connect LEDs to pins D12 D13. Make sure you have a 330 ohm resistor in series with them - otherwise you might kill the LED.
   2. Connect a switch to pin D18 such that the pin is low when the button is pressed.
   3. Connect the potentiometer (set up as a voltage divider between the positive supply and ground) to A0.
   4. We recommend setting up your hardware on a mobile (not tied down) breadboard, as in the next section you will be connecting two of these together.
2. Deploy the following piece of code (next page) onto your BRAIN. Reminder: code is available on GitHub – you do not need to manually copy and paste code from this document.

## Circuit Schematic:



If you need a reminder of how breadboards or other components are connected internally, check the diagrams in the parts list near the beginning of the document.

Make sure you connect all the +5V power lines and the ground lines together. It is recommended that you use the terminal strips on the side of your breadboard for these.

**Code: ee3\_lab0.pde**

const int PIN\_LED1 **=** 12**;**

const int PIN\_LED2 **=** 13**;**

const int PIN\_SWITCH **=** 18**;**

const int PIN\_POT **=** 0**;**

const int PIN\_UART1\_RX **=** 10**;**

void setup**()** **{**

Serial**.**begin**(**9600**);**

Serial1**.**begin**(**9600**);**

pinMode**(**PIN\_LED1**,** OUTPUT**);**

pinMode**(**PIN\_LED2**,** OUTPUT**);**

pinMode**(**PIN\_SWITCH**,** INPUT**);**

digitalWrite**(**PIN\_SWITCH**,** 1**);** // pull-up for switch

digitalWrite**(**PIN\_UART1\_RX**,** 1**);** // pull-up for external UART RX

Serial**.**println**(**"Ready"**);**

**}**

void loop**()** **{**

static int led1Val **=** 0**;**

static int led2Val **=** 0**;**

**if** **(!**digitalRead**(**PIN\_SWITCH**))** **{**

Serial**.**print**(**"1"**);**

Serial1**.**print**(**"1"**);**

// Do a delay before and after to filter out switch bounce.

delay**(**50**);**

**while** **(!**digitalRead**(**PIN\_SWITCH**))** **{**

delay**(**50**);**

**}**

**}**

// Note: the BRAIN has a separate UART port for external communications

// and the USB console. This additional code processes from both, making it

// equivalent to the Arduino's external / console UART.

**while** **(**1**)** **{**

char inByte**;**

**if** **(**Serial**.**available**())** **{**

inByte **=** Serial**.**read**();**

**}** **else** **if** **(**Serial1**.**available**())** **{**

inByte **=** Serial1**.**read**();**

**}** **else** **{**

**break;**

**}**

**if** **(**inByte **==** '1'**)** **{**

led1Val **=** **!**led1Val**;**

**}** **else** **if** **(**inByte **==** '2'**)** **{**

led2Val **=** **!**led2Val**;**

**}** **else** **{**

led1Val **=** **!**led1Val**;**

led2Val **=** **!**led2Val**;**

**}**

**}**

digitalWrite**(**PIN\_LED1**,** led1Val**);**

digitalWrite**(**PIN\_LED2**,** led2Val**);**

**}**

Can you see what this code does?

The stuff in setup() should be pretty straightforward.

* The only thing which might be confusing is why we are writing a HIGH to a digital input pin. The reason is that when the switch is not pressed, the pin is not connected to anything (floating). Writing a HIGH value enables the internal pull-up, which weakly biases the pin high.
* Similarly, writing HIGH to the UART input pin enables the internal pull-up, preventing noise from creating spurious data.

As for loop(), there are two major parts:

* Reading the switch pin
  + A zero value means the button is pressed. When this happens, the program prints out the character '1' to both the serial console and external UART.
  + Afterwards, it waits for the pin to go high (switch release).
  + The delays are necessary since the switch **bounces** when you press it and release it - the level changes several times before settling to its final level. The delay **debounces** the button so the extra transitions do not register as button presses.
* Checking the serial input
  + Serial.available() returns the number of bytes the serial receive has queued up. If it is greater than 0, that means there is new data waiting to be read. The code checks for data from both Serial (console) and Serial1 (external UART, which will be used in the next section).
  + Serial.read() returns the earliest byte received which hasn't been read yet.
  + Note that when the code checks the read byte, it checks against a numeral in single-quotes. This indicates that we are not comparing the number itself, but rather the ASCII (text) representation of the number. When you enter a 1 on the serial console, it does not send the byte 0x01; it rather sends 0x31, the ASCII encoding for the character '1'.

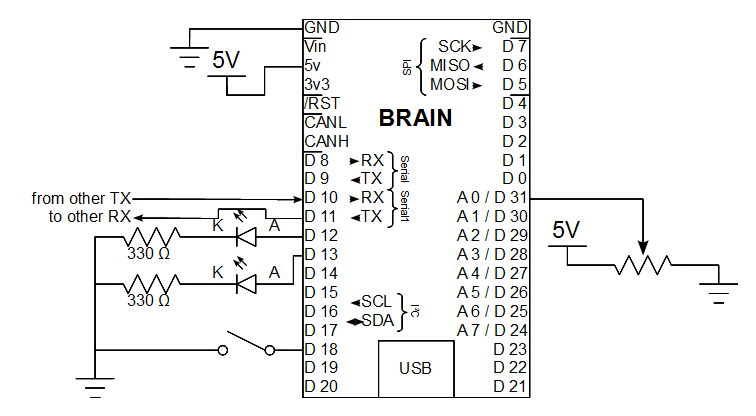
Play around with the code. Open up the serial terminal and try sending the characters 1 or 2. Note what happens to the LEDs when you send those characters: '1' should toggle a LED, '2' should toggle another LED, and any other character should toggle both LEDs. Additionally, when you press the switch, a '1' should show up on your terminal.

# Lab 1: Basic Microcontroller-Microcontroller UART

So far, you can communicate with your computer through magical device known as "Serial". On the microcontroller side, that magical device is just a UART, which can be used to communicate with other electronics. And this code does just that!

1. Now, find another group who is also ready to work on Lab 1.
2. Make sure the code from **Lab 0** is already deployed on both boards. The Serial1 hardware will also be used to communicate between the two boards in this lab.
3. Now, connect the TX pin of one BRAIN to the RX pin of the other BRAIN (and vice versa).
   1. This connects the UART transmitter of one BRAIN to the UART receiver of the other BRAIN. When you Serial.print() a byte from one BRAIN, it will appear in Serial.read() of the other BRAIN.
   2. When you press the switch on one board, it should toggle a LED on the other board.
4. Connect the grounds of both boards.
5. Blinking one LED isn't that fun. Change the code such that pressing the button toggles *both* LEDs on the other board. This should require a change of only one character to the transmitter.

**Circuit Schematic**

****

Hint: the only difference between this circuit and the Lab 0 circuit is the addition of the receive and transmit lines. Don’t forget to run a ground line between the two boards as well.

# Lab 2: Transmitting Data

Keep in mind that UART doesn't necessarily have to transmit text - it just transmits a byte of data. What that data means is completely up to the programmer.

To demonstrate that, complete the following code so that changing the potentiometer on one board will vary the LED brightness on the other.

Recall what these functions do:

* analogRead(analogIn) reads the voltage of analog input analogIn. This outputs a value from 0 to 1023, inclusive.
* analogWrite(pwmPin, value) adjusts the PWM duty cycle of pin pwmPin to value, where value is between 0 and 255. If a LED is connected to pwmPin, this varies the brightness of the LED, although brightness is nonlinear to value.

You will also need to use the Serial1.print(value) function, which puts a raw 8-bit binary number value onto the external UART line. You may also want to duplicate the code onto Serial.print(value) if you want to see the data on the Serial Monitor. If you do this, you should see gibberish (or nothing) most of the time: you are seeing the serial monitor trying to represent raw binary data as ASCII-encoded characters - and failing miserably.

Note that in order to simplify the control code in this lab, all commands only span one transmitted byte. This does not have to be the case (and usually is not the case in real applications). Transmitting these multi-byte commands is much easier than parsing received multi-byte commands.

We will explore that further in the next sections covering I2C and SPI.

**Code: ee3\_lab2.pde**

const int PIN\_LED1 **=** 12**;**

const int PIN\_LED2 **=** 13**;**

const int PIN\_SWITCH **=** 18**;**

const int PIN\_POT **=** 0**;**

void setup**()** **{**

Serial**.**begin**(**9600**);** // Initialize the USB console

Serial1**.**begin**(**9600**);** // Initialize the external UART port

pinMode**(**PIN\_LED1**,** OUTPUT**);**

pinMode**(**PIN\_LED2**,** OUTPUT**);**

pinMode**(**PIN\_SWITCH**,** INPUT**);**

digitalWrite**(**PIN\_SWITCH**,** 1**);**

Serial**.**println**(**"Ready"**);**

**}**

void loop**()** **{**

int potIn **=** analogRead**(**PIN\_POT**)** **/** 4**;**

// Transmit data here!

/\*YOUR CODE HERE\*/

delay**(**100**);**

**while** **(**Serial1**.**available**()** **>** 0**)** **{**

// Read received data here!

char inByte **=** /\*YOUR CODE HERE\*/

// Process received data here!

analogWrite**(**PIN\_LED1**,** /\*YOUR CODE HERE\*/**);**

**}**

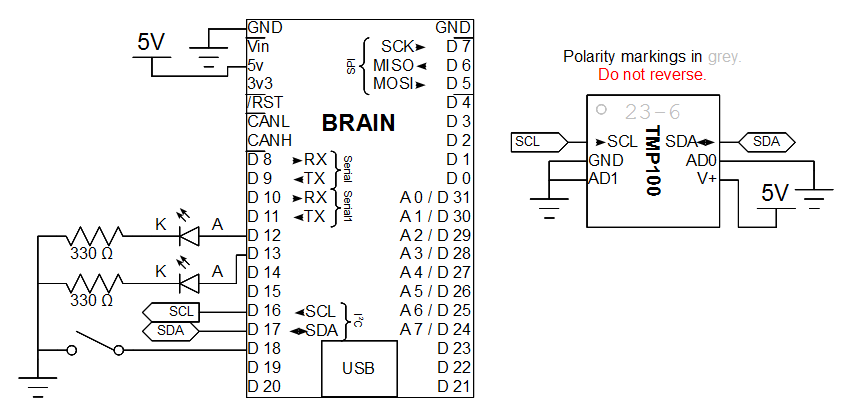
**}**

# Lab 3: I2C Thermometer

Even though UART is commonly used as a serial console, it is less commonly used as an inter-chip interface. For that, you’ll usually see I2C and SPI. In this section, you will read the temperature from an external temperature sensor through I2C.

1. Set up your hardware as follows:
   1. Ask one the lab staff for the I2C temperature sensor board. Note that this is a surface-mount chip soldered onto a breakout (adaptor) board so it is breadboard compatible.
   2. Plug the I2C temperature sensor into your breadboard. Warning: this is a polarized device – if you wire it up backwards, hardware damage may occur. The diagram below is pin-accurate; just ensure the white silkscreen dot is on the top-left corner and the “23-6” label is on top.
   3. Connect the power connections.
   4. Connect the address pins, AD0 and AD1, to ground. These pins set part of the device’s I2C address, allowing several of them to coexist on a single bus. With AD1 and AD0 grounded, the device’s address is 1001000, or 0x48.
   5. Make the straight-through SCL and SDA connections – wire each pin from the BRAIN directly to the corresponding pin on the sensor board. We will use the internal pull-up resistors on the BRAIN on the SCL and SDA lines instead of adding external ones.
2. Deploy the example code, **ee3\_lab3.pde**. This should read data from the sensor and print out the temperature, in degrees Celsius, to the console.

**Circuit Schematic**



**Code: ee3\_lab3.pde**

#include <Wire.h>

const int PIN\_SCL **=** 16**;**

const int PIN\_SDA **=** 17**;**

// I2C address of TMP100 with A0=0, A1=0

const int TMP100\_I2C\_ADDR **=** 0x48**;**

const int TMP100\_REG\_TEMPERATURE **=** 0x00**;**

const int TMP100\_REG\_CONFIG **=** 0x01**;**

float digitalOutputToTemp**(**int16\_t val**)** **{**

**return** **(**float**)**val **/** 16**;**

**}**

void setup**()** **{**

// Initialize serial console

Serial**.**begin**(**9600**);**

// Enable internal I2C pull-up resistors

digitalWrite**(**PIN\_SCL**,** HIGH**);**

digitalWrite**(**PIN\_SDA**,** HIGH**);**

// Initialize I2C in master mode

Wire**.**begin**();**

// Setup configuration registers on TMP100

Wire**.**beginTransmission**(**TMP100\_I2C\_ADDR**);**

Wire**.**send**(**TMP100\_REG\_CONFIG**);**

Wire**.**send**(**0x60**);** // Set 12-bit mode

Wire**.**endTransmission**();**

// Set pointer at temperature

Wire**.**beginTransmission**(**TMP100\_I2C\_ADDR**);**

Wire**.**send**(**TMP100\_REG\_TEMPERATURE**);**

Wire**.**endTransmission**();**

Serial**.**println**(**"Ready"**);**

**}**

void loop**()** **{**

Wire**.**requestFrom**(**TMP100\_I2C\_ADDR**,** 2**);**

**if** **(**Wire**.**available**()** **==** 2**)** **{**

int16\_t val **=** Wire**.**receive**();**

val **=** val **<<** 8**;**

val **|=** Wire**.**receive**();**

val **=** val **>>** 4**;**

Serial**.**println**(**"Read temperature"**);**

Serial**.**println**(**digitalOutputToTemp**(**val**));**

**}** **else** **{**

Serial**.**println**(**"Invalid response"**);**

**}**

delay**(**500**);**

**}**

Can you understand what the code does, given explanations of the library functions and the sensor?

* The Wire library is for I2C communications.
  + Write.begin() activates the I2C module in master mode.
  + Wire.beginTransmission(addr) begins a message addressing addr in the master-to-slave (write) direction. This generates the start condition, address bits, and directionality bit.
  + Wire.send(data) transmits data to the slave.
  + Wire.endTransmission() generates the stop condition.
  + Wire.requestFrom(addr, numBytes) requests numBytes from addr. This is a completely self-contained I2C message, generating the start condition, address bits, directionality bit, and transfer bytes. Once done, data can be read out using Wire.read(), and Wire.available() returns the number of bytes left remaining.
  + Note that we digitalWrite the SCL and SDA pins high, enabling the internal pull-ups.
* The temperature sensor communicates as follows:
  + Its address, given the address pin connections in the circuit, is 0x48.
  + It contains an internal register pointer, which indicates which register read and write operations affect.
  + The first byte of a write operation sets the internal register pointer. Subsequent bytes write to the target register.
  + The temperature is a 12-bit value in the register at address 0, left-aligned. The funky code with the bitwise shift and or operations packs the two read bytes into one integer.
  + The configuration register is at address 1 and controls things like shutdown and the sensor resolution.

Now, remember what we said before about I2C being a shared bus? Time to implement that. Add another sensor to the I2C bus (with a different address configuration, of course), and write the code to read both. You may find it useful to refactor the code and break out the sensor initialization and read code into their own functions, each taking an I2C address as an argument.

For reference, the device addresses of the TMP100 sensors are (as taken from the device’s datasheet <http://www.ti.com/lit/ds/sbos231g/sbos231g.pdf>):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ADD1** | **ADD0** | **I2C Address** | **ADD1** | **ADD0** | **I2C Address** |
| **0** | **0** | 1001000 | **1** | **Float** | 1001101 |
| **0** | **Float** | 1001001 | **1** | **1** | 1001110 |
| **0** | **1** | 1001010 | **Float** | **0** | 1001011 |
| **1** | **0** | 1001100 | **Float** | **1** | 1001111 |

# Extra for Experts: Accelerometer SPI

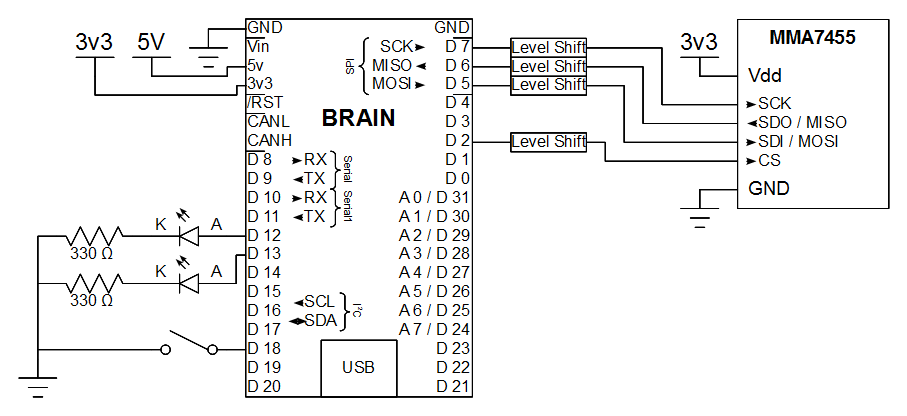
Note: We only have one accelerometer to go around, so you will be deploying your code onto a pre-built reference circuit.

## Introduction

Accelerometers are devices which sense acceleration, including the acceleration due to gravity. However, these devices are rarely (probably never) integrated onto microcontrollers, and usually sold as external devices. Communications standards supported by these include analog (output voltage proportional to sensed acceleration), SPI, I2C, and other less common ones.

Today, we will use the MMA7455 SPI accelerometer to demonstrate communications with external devices. The idea will be to change the LED's brightness based on the acceleration on the accelerometer's long axis.

## Circuit Schematic:

****

**See skeleton code: ee3\_extra.pde on the training website**

## Accelerometer Introduction

The MMA7455 accelerometer interface consists exclusively of **register read and register write** operations - the microcontroller sends instructions to the accelerometer to read from or write to a portion of the accelerometer's memory. This memory is where things like the accelerometer's state, configuration, and output is stored - by writing particular addresses, you can turn the accelerometer on/off, set the measurement range (in g's - standard gravity - 1g is the force from gravity on Earth's surface).

## Lab Manual

1. The initialization code has already been done for you. It sets the accelerometer into measurement mode, and sets the full scale at .
   1. Note that the relevant registers have been defined at the top. This is good programming practice, as something like MMA\_MCTL makes more sense than 0x16. In general, "**magic numbers**" (where you have a numerical constant in the middle of your code) are discouraged in favor of named constants.
   2. Note that it also does a sanity check, by ensuring that you can read out data written to the accelerometer. This can help detect wiring and coding mistakes, and may save you hours of banging your head against the table.
   3. The accelerometer interface code has also been done for you:  
      aclReadReg(reg) reads address reg through SPI commands and returns its value.  
      aclWriteReg(reg, value) writes value to address reg on the accelerometer using SPI commands.
   4. The code to implement the SPI interface is also included - and you can see what is going on under the hood. For each command, the microcontroller drives CS low to indicate it wants to talk with the accelerometer, then sends a byte using the microcontroller's SPI hardware. For every byte sent, another byte is received - but this byte is not always meaningful and is sometimes discarded.
2. The part to read the relevant accelerometer register has also been done for you. The code runs as-is, but as you may notice, the LED's brightness does not vary linearly with the PWM duty cycle.
3. Let's fix that. What you need to do is already outlined for you in the code.
   1. The idea is that instead of setting the read value as the duty cycle, do some processing on the data.
   2. First, the data is a **signed char**, which has values ranging from -128 to 127. Since you can't make your LED absorb light, make that positive by taking the absolute value of it. The function abs(val) returns .
   3. Then, cap the value at 1g = 64, the highest you are likely to see under the force of gravity.
   4. Now, if you've done the extra for experts on the previous lab, you will remember that linearly varying the duty cycle is a bad idea if you want to linearly scale brightness. A better way is to vary the duty cycle with the square of the input. Since the duty cycle for PWM is in the range from 0 to 255, and the square root of 255 is around 16, you need to scale the input from 0-64 to 0-16. This should be a simple divide operation.
   5. That's it. Try the code. Tilt the accelerometer to change the force of gravity between axes. As you tilt the long axis up and down, you should see the LED change brightness.
4. Now, repeat the same so that the Z-axis is displayed on the second LED. This should simply be a matter of copying and pasting the code, and changing variable names and register addresses. The 8-bit Z-axis output is stored in the register defined by MMA\_ZOUT8 = 0x08.

# Conclusion

You've implemented a simple communications scheme between two BRAINs today and possibly also wrote code to interact with an accelerometer. This forms the basis of how to communicate with external devices, opening the door to using things like external memory, sensors, and other fun gadgets. We commonly use these things because the microcontroller does not have everything that we need built-in.

1. You may notice that although the voltage may change ever so slightly, the digitized value won't. Through an analog-to-digital converter (ADC), the hardware changes the analog voltage into a discrete digital quantity valued from 0 to 1023. This digital quantity is what the processor works with. [↑](#footnote-ref-1)