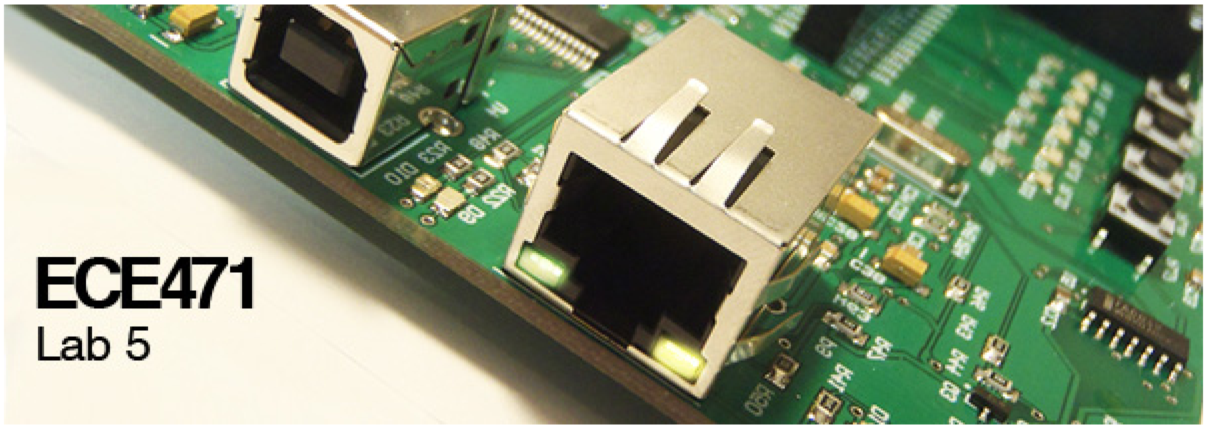


NDSU Embedded Systems II

I2C bitbang

I2C bitbang communication with 24LC00 EE PROM

Objective: To obtain familiarity with the following:

- I2C protocol

By
Nathan Zimmerman

Requirements for lab 5:

Introduction: I2C is an intrachip communication protocol popular in industry for its simplistic hardware interface. I2C's significant advantage over other protocols is that it is composed of only 2 lines which are SDA and SLC. SDA is a unidirectional data line which is synchronized by SCL. In this lab we will attempt to thoroughly understand I2C by implementing a bit banged version of I2C.

Requirements for Lab 5:

1. Implement **Software** based I2C via software bitbang.
2. Utilize software bit bang to read & to 24LC00 EEPROM

Result should look similar to Lab5.axf . You should be able to get one of these chips from Jeff.

Extra Credit:

1. Use hardware I2C to perform a read/write operation on the 24LC00 EEPROM

Useful Resources:

I2C:

http://en.wikipedia.org/wiki/I%C2%B2C

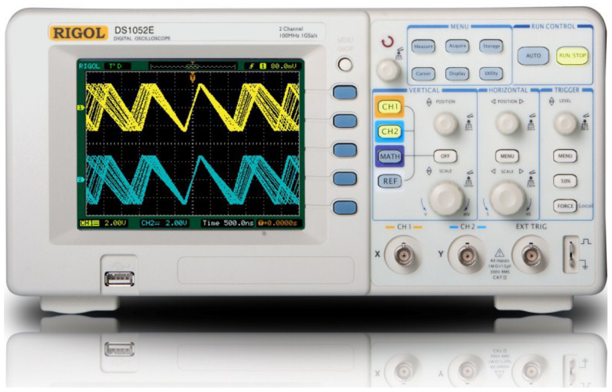
Software I2C tutorial:

http://www.robot-electronics.co.uk/acatalog/I2C_Tutorial.html

Official Standard

http://www.nxp.com/documents/other/39340011.pdf

Oscilloscope: Highly recommended you work with oscilloscopes in the lab for this assignment



I2C or I^2C: (Inter-Integrated Circuit)

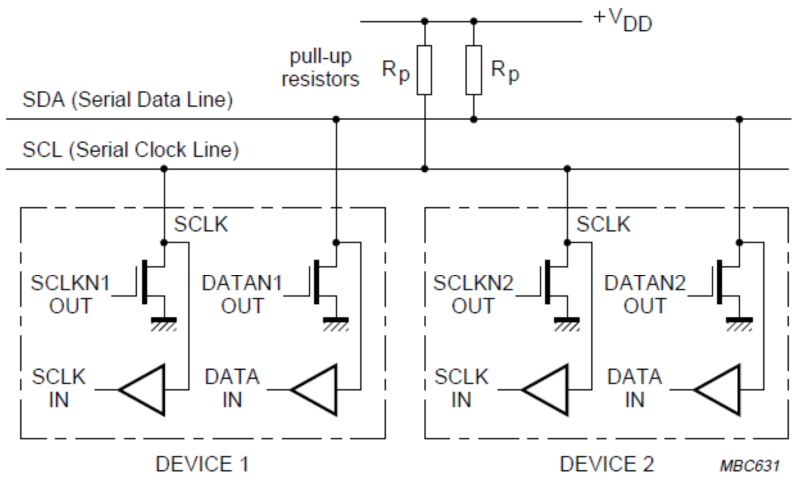
Specs

Lines: (2)	**Packet Size:** 8 bits + ACK
SDA: Serial Data Line	**Packet Overhead:** Address byte
SCL: Serial Clock	**Standards:** Distributed by NXP/Phillips
Baud rates: <400kHz generally	**CLK:** asynchronous

Introduction: As mentioned previously, I2C is a popular intrachip communication protocol which is popular due to its utilization of only 2 wires. In addition, due to the rigid definitions of the standard, implementation at times can be easier than other protocols such as SPI where the protocol is redesigned per chip implementation.

Schematic:

(image from NXP Spec)



Above is a primitive representation of how an I2C bus is implemented. There are a couple important things to note about the image above. First and foremost, both the master and the slave have the ability to drive both the SDA line and the SCL line. Likewise, it is also important to note that neither the master nor the slave have the ability to drive the bus lines high. An external pullup resistor is required on the line in order for a VCC voltage level to be seen on the bus. When the master outputs SDA/SCL as low, a VCC gate voltage is applied on the internal FET attached to the SDA/SCL line. Consequently, the master's FET pulls the potential of the line to GND/VSS. When the master outputs SDA/SCL as high, no voltage is applied to the gate of the internal FET effectively making the Input/Output SDA/SCL line act as high impedance as far as the host is concerned. This method of driving is frequent in industry and is called **Open Drain.** This method allows for unidirectional dataflow without any dangerous current surges as a result bus contention between the master and the slaves.

I2C Continued: (Inter-Integrated Circuit)

Layout:

(image from NXP Spec)

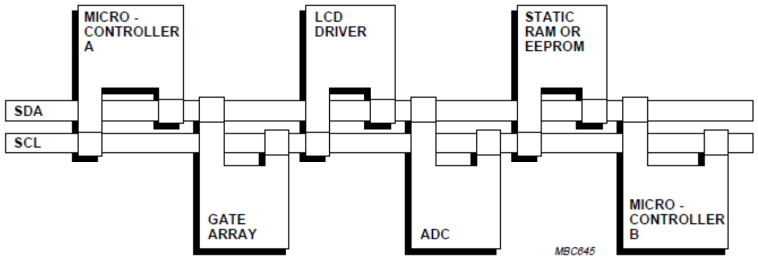


Fig.2 Example of an I^{2}C-bus configuration using two microcontrollers.

Note that I2C is generally a single master implementation with multiple slaves. However, there exists the possibility for multiple masters. However, these implementations are rare. Slave designation is determined through an address byte which is sent initially on the bus.

Protocol: When idle, both the SDA and SCL lines are high. The master begins communication on the bus by instigating a start condition. The start condition occurs when SDA is asserted low followed by SCL being asserted low. Similarly, a stop condition occurs at the end of a transition and is represented by SCL being asserted high followed by SDA being asserted high. See image below:

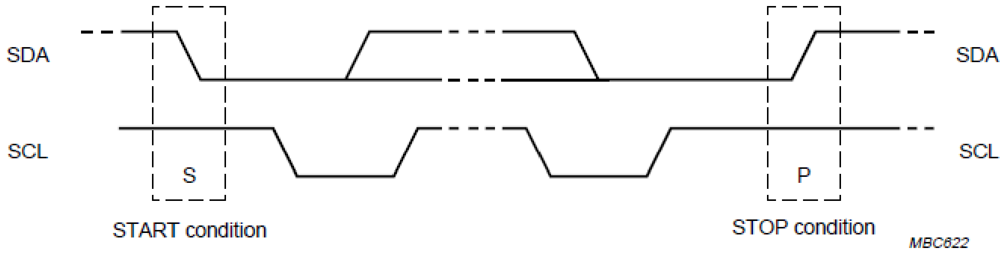
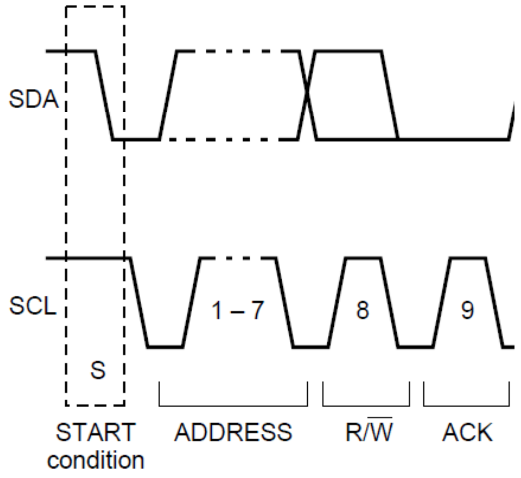


Fig.5 START and STOP conditions.

I2C Continued: (Inter-Integrated Circuit)

Protocol Continued: Slave Addressing

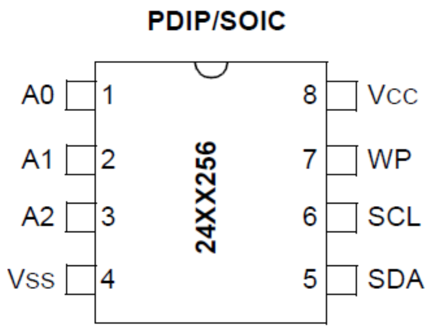
After the start condition occurs, the master must address which slave it wishes to communicate with. This occurs by the transmission of an address over the SDA line. This address is usually 7 bits long followed by a R/W bit and what is called an "ACK". The R/W bit signifies if the master wishes to write a register on the slave or to read a register. When R/W = VCC, the master whishes to perform a read. When R/W = 0, the master wishes to perform a write.



The **"ACK"** or acknowledgement bit is a bit transmitted by the **slave** to confirm it accepts being addressed. In addition to ACK-ing after every address, the slave will ACK after every byte of data generally Consequently, upon the master's completion of sending the address, the master must de assert control of the SDA line allowing the addressed slave to pull SDA low. The master will look for this bit to confirm that the slave is present and willing to accept data. If the master does not see an ACK, the master should instigate a stop condition and repeat the process.

Slave varying addresses :

Each slave generally has a 7 bit address. Consequently, there lies the possibility for conflict if an I2C slave has a fixed address. Consequently, many ICs have pins dedicated to the I2C address of the device. Meaning when depending on whether these pins are pulled up to VCC or low to GND, the address of the device changes allowing for design flexibility. An example I2C EEPROM pinout can be seen to the right where 3 pins can be optionally strapped to change the slaves address.

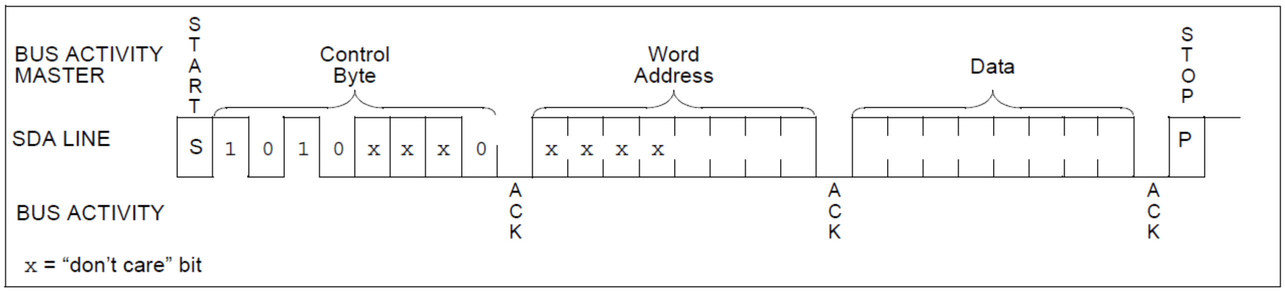


I2C Continued: (Inter-Integrated Circuit)

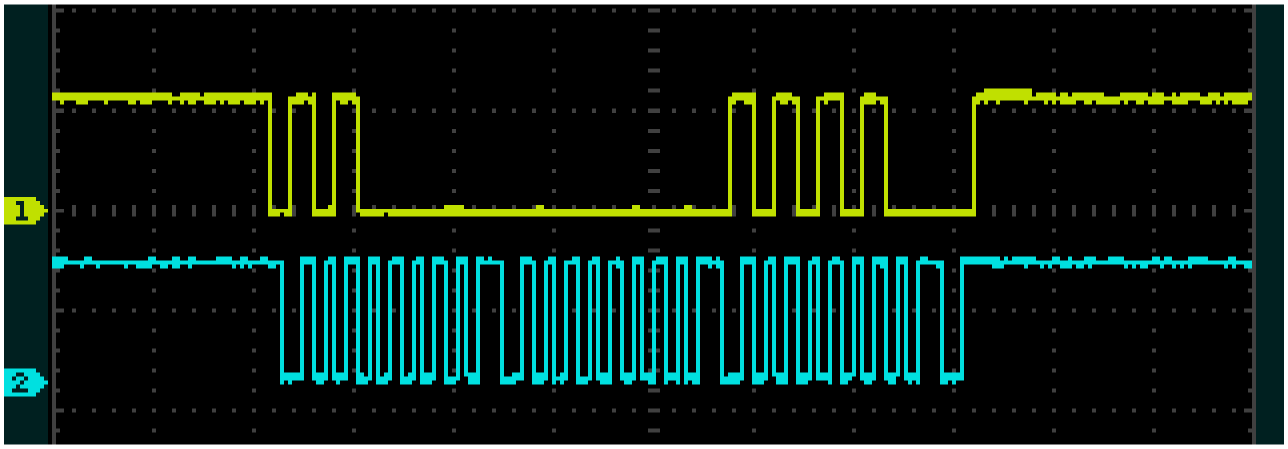
Protocol Continued: Writing to a reg example:

Once the slave has been address and the master has confirmed the ACK from the slave, the next step is to transmit the register location that is to be written to. After the register has been specified, the master will then pump out byte or bytes of data. If multiple bytes are sent out, they will write to registers incrementing from the initial address. See the below example for writing to a single register on the 24LC00 EEPROM. Keep in mind that this image is not displaying the SCL line. The SCL line must be clocking in every byte.

FIGURE 7-2: BYTE WRITE



Scope shot of write: Slave Address 0xA0, Reg = 0x0, Data = 0xAA

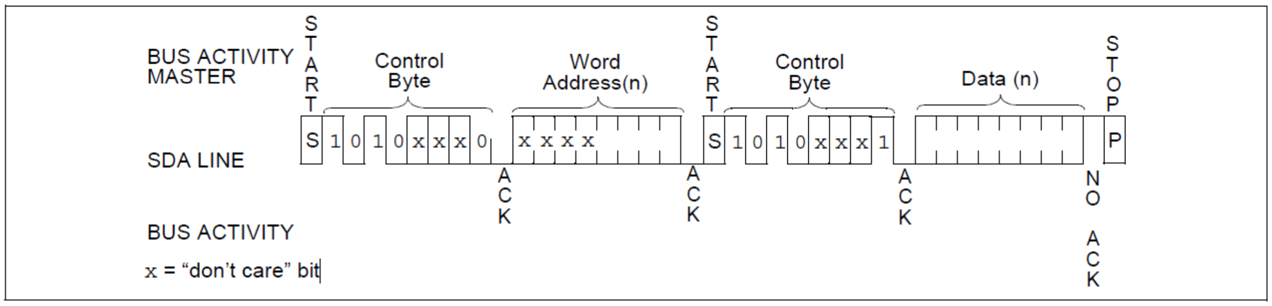


I2C Continued: (Inter-Integrated Circuit)

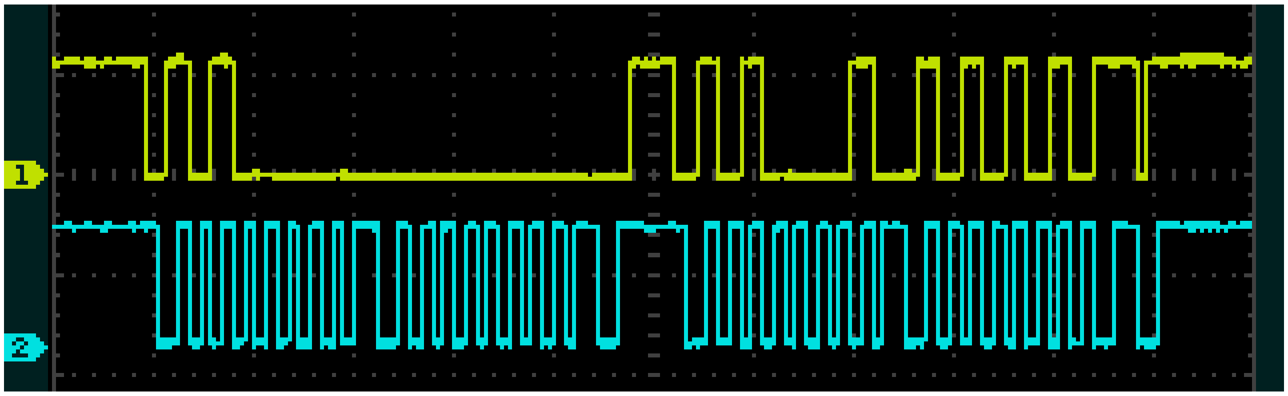
Reading a reg example:

Reading a register begins the same way writing a register does. The master will perform a start condition, addresses the slave with R/W bit set to 0, and then write a register or word address it wishes to read from. This is followed by a stop and then a start condition. The master will then send the address byte (Control Byte in image) yet again except this time the R/W bit will be set to 1. The master will then release control of the SDA line completely and clock in 9 bits of data. For our chip specifically, there is no ACK on the 9th bit of the data read in. This may not always be the case. Reference image below:

FIGURE 8-2: RANDOM READ



Scope shot of read: Slave Address 0xA0, Reg = 0x0, Data received = 0xAA



I2C Continued: (Inter-Integrated Circuit)

Other considerations: SLC slave control

As observed in the schematic diagram of the line driving scheme, both the master and the slave can control both lines. So far we have covered as to why the slave needs control of the SDA line. This a course is for ACKs as well as sending data back to the master. However, we have not looked at as to why the slave should need control of the clock line. This further complicates the protocol because the slave actually has the ability to hold the clock line low if it needs more time to process incoming data. Therefore, upon driving the SCL line high, the master must recheck the line to ensure that it has indeed been asserted high. If the line is not high, the master must wait for the slave to release control of the clock line before clocking in any new data. However, for the chip that we are using, provided we have a decently slow CLK speed, we can dangerously assume that the EEPROM will react in time to reduce coding complexity.

Considerations for implementing bit banged on the LPC1769 :

As mentioned previously, the SDA and SCL lines must be of open drain topology. Our fancy micro a course has the ability to set just about any line to be configured as an open drain driving line. This can be done in the PINMODE_OD register. Read the user manual and configure both your SDA and SCL pins as open drain! The internal pullups are on by default, therefore you will not need to add any to the lines. You can get away with pullups weaker than the recommended 10k. The following defines should be helpful:

```
#define I2C_SDA          P1_19
#define I2C_SCL                          P1_24
#define delay            delay_ms(2)

#define SCL_HIGH         (LPC_GPIO1 ->FIOSET = I2C_SCL)
#define SCL_LOW          (LPC_GPIO1 ->FIOCLR = I2C_SCL)
#define SDA_HIGH         (LPC_GPIO1 ->FIOSET = I2C_SDA)
#define SDA_LOW          (LPC_GPIO1 ->FIOCLR = I2C_SDA)
#define SCL_CLK          SCL_HIGH;delay;SCL_LOW;
#define SDA_OUT          (LPC_GPIO1 ->FIODIR |= I2C_SDA)
#define SDA_IN                           (LPC_GPIO1 ->FIODIR &= ~I2C_SDA)
#define SCL_OUT          (LPC_GPIO1 ->FIODIR |= I2C_SCL)
#define SCL_IN                           (LPC_GPIO1 ->FIODIR &= ~I2C_SCL)
#define SDA_ACK          (((LPC_GPIO1 ->FIOPIN)&I2C_SDA))
```

Lab Report Guidelines:

1. Upload your code files to your assigned GIT repo. You must first create an account on github and email me your account name in order to gain access to your repo.

2. If you are having trouble getting GIT to work initially, email me your code at NDSU.ECE471@GMAIL.COM. The code should be attached in the form of .c and .h files or in a complete zip archive . Your email title should be : Lab_Number : Myname. For example: Lab_1 : Nathan Zimmerman. This method of turning in assignments is cumbersome and will not be available long term so attempt to get GIT working ASAP.

   You are allowed to work in groups but you are all expected to turn in your own individual code. You should write the code yourself and it should not be a duplication of someone elses code.

Grading:

100% **: Code runs and by majority meets requirements**

50% **: Code does not meet requirements or does not compile but a reasonable effort was made to complete the lab.**

0% **: Code was not turned in or significantly falls short of meeting requirements.**