Lab 1: Assembly Implementation of I^2C Protocol

Ryan French
ryanfrench3@montana.edu
Graduate Student, Department of Physics
Montana State Physics University - May 3, 2020

1. Introduction

It is commonly said that the most important invention in the past century is the transistor. This revolutionized the way we developed computers, as transistors led to both size reduction and improved speed. However, sometimes overlooked is the development and improvements in communication theory, specifically concerning digital communications.

In this lab, a form of digital communication, I^2C, will be implemented from the most basic principles possible. Using the Texas Instruments' flavor of assembly language on the MSP430FR2355, a bit-bang method was developed to allow the master MCU to write data (no reads were implemented).

2. Theory

2.1. I^2C Protocol

The I^2C (inter-integrated circuit) communication protocol is a means of synchronous communication amongst several masters and slaves. The I^2C bus consists of two digital lines: SDA (data line) and SCL (clock line). When sending data, the clock is toggled at a constant frequency, while the data line is selectively toggled to send the intended bits. This protocol sends data with MSB first.

The full explanation of sending data follows:

1. To start the transaction, the SDA is pulled low while SCL remains high
2. The SCL is then also pulled low
3. The address of the intended target is sent (this is usually a 7-bit number)
   (a) Each bit of the data is sent on a rising clock edge, so the SDA line must be set low or high first before setting the SCL line low.
4. A read/write bit must follow. Since this is a write operation, a low signal must be sent
5. Next, the master must switch its SDA line to input and listen to the data line. If the slave sends an ACK bit (logical low), then the master can proceed sending data
6. With SDA as output again, the master can send 8 bits of data, as described above
7. After each 8 bits of data sent, it is necessary to listen for an ACK signal from the slave
8. To stop and close the bus, a stop condition must be sent. The stop condition is performed by pulling the SCL line high and then pulling the SDA line high (thus putting the bus in idle mode again)

3. Microntroller Implementation

To see a flowchart of this implementation, see appendix A.

Knowing how the process works, based on the above explanation, it is straightforward to implement the protocol in ASM. Subroutines were written for the following important pieces of the communication:

- Idle
  - Set both SDA and SCL high (mostly for initialization)
- Start
  - SDA high, delay, SCL high
- Stop
  - SDA low, delay, SCL high, delay, SDA high
- Read ACK
  - Set SDA input. SCL high, read SDA. If NACK, keep trying, else continue
- Send high/low
  - SDA high/low, delay, SCL high, delay, SCL low

Using these subroutines, it was easy to finish the demos.

3.1. Demo 1

The first demo was simply sending an arbitrary address and analyzing the resulting waveform with an oscilloscope. This was straightforward and doesn't really merit any discussion.

3.2. Demos 2-3

These demos were slightly more involved: following an address, 10 bytes of data are sent and analyzed with the oscilloscope: the integers from 0 to 9. Again, not much else to say on this demo.

3.3. Demos 4-5

These demos are intended to show a full read-write process wherein a master requests data from a slave device and then reads the resulting data. To do this, a DS3231 real-time clock was connected to the master by I^2C. Consulting the RTC's datasheet, the address and the data registers were determined.

First, it was required to request temperature data from the RTC. This was done by writing the byte necessary to access the RTC's temperature sensor. This temperature value is sent back to the master and the hex value of the temperature was read using the oscilloscope.

Next, it was required to initialize and request time from the RTC. To initialize, it was required to set the appropriate bits in the RTC's time registers. The time chosen was arbitrary. Next, requests were made (much like reading the temperature) to read select time registers on the RTC: the hours, minutes, and seconds registers. Using a loop of requests and analyzing the data on the oscilloscope, it was seen that the seconds data incremented as is expected.

4. Discussion

While the I^2C theory presented here seems simple, there are a lot of more intricate details on this communication method that were not implemented in this lab. Examples of these features:

- Clock stretching
  - A slave may hold the SCL line low to indicate that it not ready to receive more data.
- Arbitration
  - When multiple masters are communicating over the same bus, it is necessary for each master to poll the data line to avoid sending data while the bus is busy. Sometimes, however, two masters may start to send data at the same time. This is when it's necessary to arbitrate: as the masters send data, they also monitor their SDA line. If one master sets its SDA to high but reads a low on its SDA line, it knows that another master is using the bus and it will thus back off. Because of this, if the two masters are addressing two different slaves, only the master communicating with the lower-addressed slave will actually send data.

Even though I^2C is a basic communication protocol which is slower than other methods, implementation can be difficult. Some issues that commonly occur are floating data and long rise times. Floating data will plague any system that does not apply pull-up resistors on both the SDA and SCL lines. These pull-up resistors can vary in value, but in general, pull-ups of 4.7-10 kΩ are recommended. This is tricky though, as will be discussed next.

A major design consideration in I^2C circuits is capacitance. While it is a slower protocol, it is still very suceptible to slow digital transitions. While capacitance usually isn't a big deal in simple circuits, it becomes important when placing multiple devices on the bus or when designing PCBs with digital communication traces. When in the latter situation, the first goal should be determining the major sources of capacitance in the circuit. Examples and mitigation strategies follow:

1. Pin capacitances
   - Obviously, the best way to fix this is to use fewer ICs, as each pin usually contributes 10 pF to the bus.
2. Trace capacitances
   - Remember that capacitance comes in two flavors: self-capacitance and mutual capacitance. Both of these must be considered when there are 2(+) conductors that are being dealt with.
   - Though these two types are often thought of being separate ideas, that is not true. With multiple conductors, we can not use the generic Q=VC equation (this is only mutual capacitance). Instead, we must couple the types of capacitances through a capacitance matrix. This matrix, once determined, determines the true capacitance of a system (fun fact: because of this matrix, it is found that for symmetric "plate" capacitors, the positive plate will hold more charge than the negative plate).
   - Minimizing self-capacitance is done by simply changing the geometry of the conductor. So when implementing the data bus, keeping the traces as short as possible will cut down on bus capacitance.
   - Minimizing mutual-capacitance is more complicated. Data traces should be kept as far apart as possible if they must be routed long distances (alternatively, placing VDD and GND traces between them will reduce this effect).
   - In addition, it is necessary to consider capacitance between the data line and ground. This can be minimized by choosing PCB material with a lower dielectric constant and placing the ground plane farther away from the bus lines (i.e., placing a layer between the bus and ground)

After minimizing the capacitance, it will be easy to choose a pull-up resistor value (usually chosen by considering the characteristic time, $\tau = R\,C$). Of course, if capacitance is still an issue, it's also possible to use I^2C buffers!

A. Flowchart

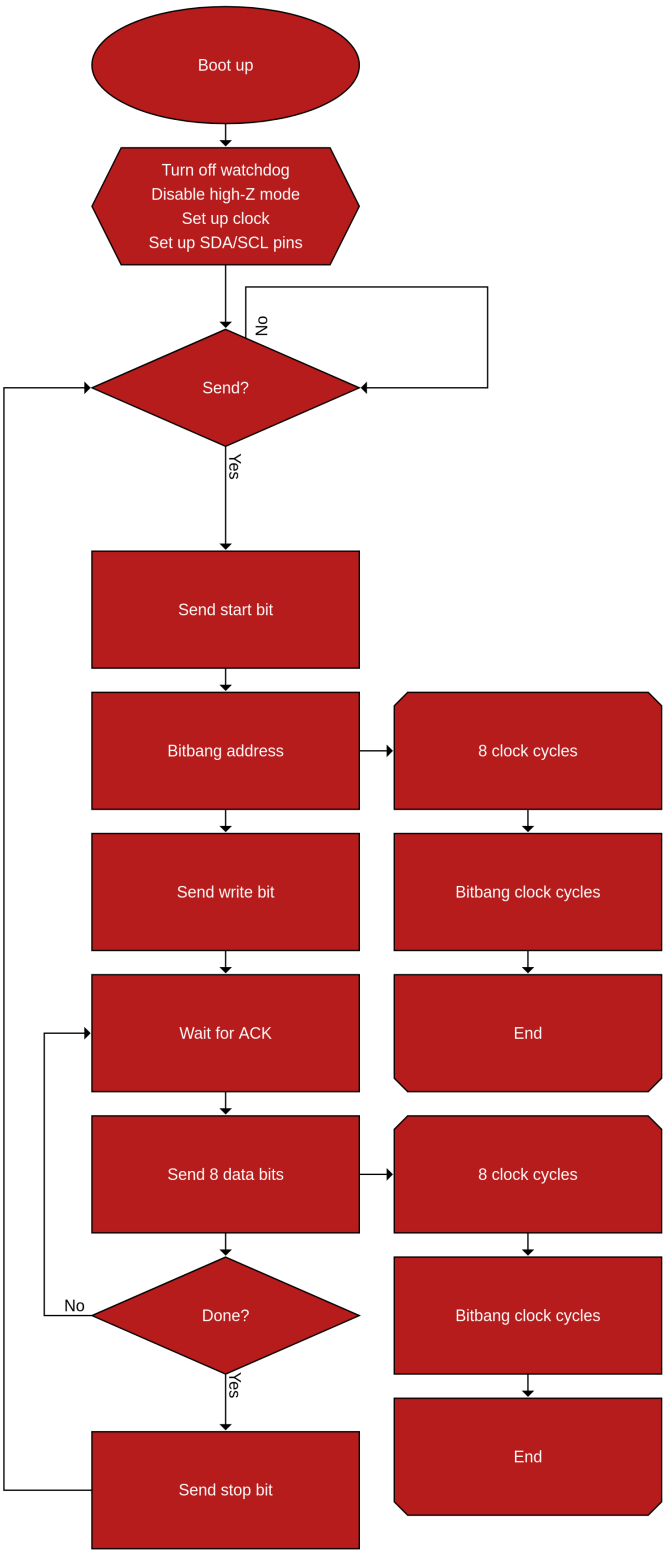

Figure 1: *Flowchart of basic I^2C writing.*