

An Unofficial SoftWire User's Guide, Version 1.0.0

By R. G. Sparber

Protected by Creative Commons.¹

Purpose

Provide an understanding of [SoftWire](#) that will enable you to go from a device's spec sheet to a set of SoftWire function calls. This journey takes you from timing diagrams through voltage waveshapes and into software. Then, I take a sample spec sheet and turn it into a driver.

This document is a work in progress. I welcome your comments and suggested corrections.

Overview

SoftWire provides the functions to

- place information into a transmit buffer,
- process the transmit buffer as it sends this information to a designated slave,
- request information from a slave,
- collect that information and put it into a receive buffer,
- and read information from this receive buffer

Ways to Use This Document

If you wish to have the same understanding of SoftWire as I do, read this document sequentially. If you just want to create I2C device drivers, read the [Creating a Driver](#) section. If anything is confusing, refer to the corresponding section. If that doesn't help, contact me at rgsparber@AOL.com.

¹ This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by/4.0/> or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Creating a Driver

These steps should guide you in creating a driver for your selected device. Reading the example on page 33 may help you see how this works.

1. Find the spec sheet for the I2C device you wish to drive.
2. Identify if it expects 5 volt or 3.3 volt logic levels. Only connect it to an I2C bus with devices from the same voltage family. Furthermore, be sure that the total pull-up resistance is around 4.7K. Some breakout boards include optional pull-up resistors; you do not want more than one pair active on a bus.
3. Study the spec sheet carefully to find
 - a. The device's 7 bit address. This is sometimes called its base address.
 - b. What commands must you send the device, and how many bytes are returned? For example, a command to read temperature may return two bytes.
 - c. I assume your spec sheet contains one or more sequence diagrams showing what must pass between master and slave to accomplish the stated task. If it is in text, draw your sequence diagrams.
4. For each sequence diagram
 - a. Match the sequence of a SoftWire command to a segment of your sequence diagram. Pay close attention to the TransmissionInProgress flag state.
 - b. Form a list of these commands.
 - c. Place these commands into your code with tests, as needed, to prevent a faulty sensor from stopping execution.

Background

I have never liked using libraries that I don't understand. Recently, I needed to interface with an [I2C device](#) using SoftWire. Finding a library that included my device wasn't difficult, but it was designed for the built-in hardware I2C bus. I looked at the code, but it used many programming techniques that were foreign to me. I wanted to find a more straightforward and understandable way. I tried to use the functions defined for SoftWire.

The Reference section of Arduino provides a list of SoftWire functions, but I could not find anything like a User's Guide, so here is my attempt. It is based on studying the SoftWire.cpp and SoftWire.h files, plus applying this understanding to a driver for an I2C temperature sensor. All software was developed on a Sparkfun Pro Micro using the Arduino Integrated Development Environment. I used Notepad++ for code editing.

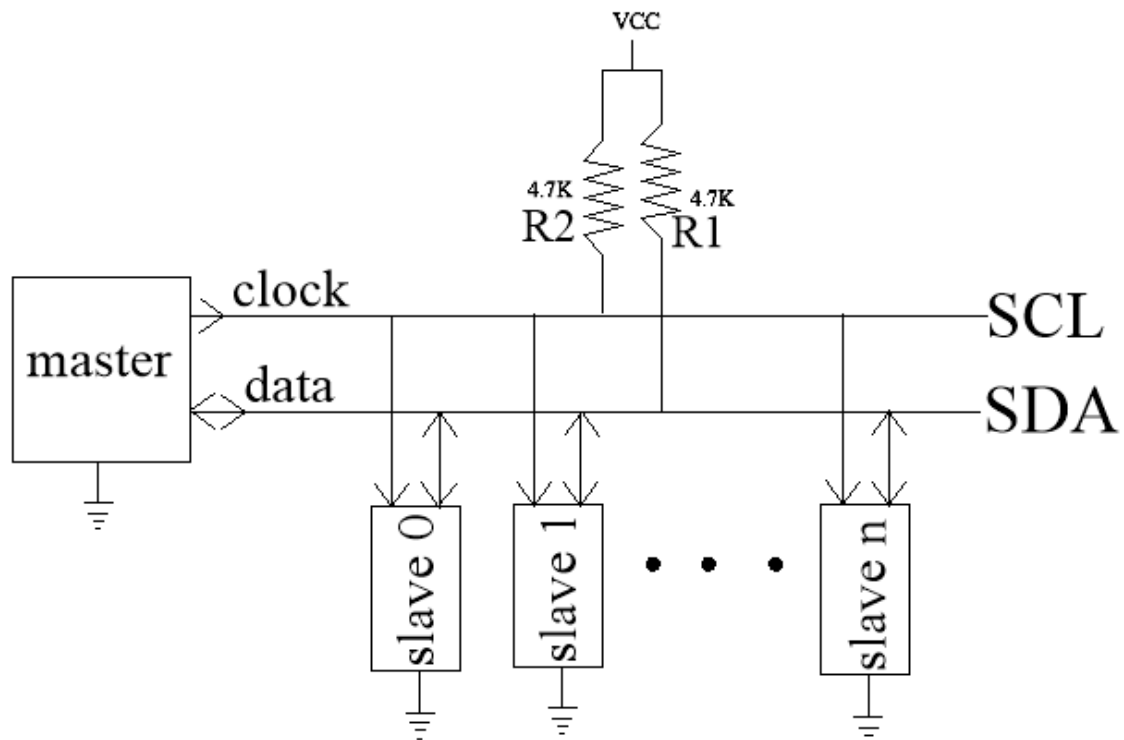
If you are unfamiliar with I2C, I suggest you read [this](#) excellent document.

If you go to [SoftWire](#), you will find a lot of software development-focused documentation. Maybe I missed it, but I couldn't find anything telling me how to use these functions to control an I2C device. Thankfully, SoftWire.h and SoftWire.cpp were clear enough for me to follow, given my level of software sophistication. The acid test was when I successfully applied this newfound understanding to writing a driver for the Si17021 Humidity and Temperature sensor that would read back temperature.

Contents

Purpose	1
Overview	1
Ways to Use This Document	1
Creating a Driver.....	2
Background	3
The Physical Layer.....	5
Ones, Zeros, and More.....	8
SoftWire	15
The SoftWire Functions.....	16
Key Low-Level Elements	17
begin()	17
beginTransaction()	17
write().....	17
endTransmission().....	18
requestFrom().....	20
read()	24
available().....	24
Example: The Si7021 Humidity and Temperature Sensor	26
Sample Code: Si7021 Device	31
Before setup():.....	31
In setup()	32
The I2C Driver	33
Appendix I: Analog Aspects of I2C.....	34
Comparing the Software to the Waveshapes	36
Acknowledgments.....	37

The Physical Layer



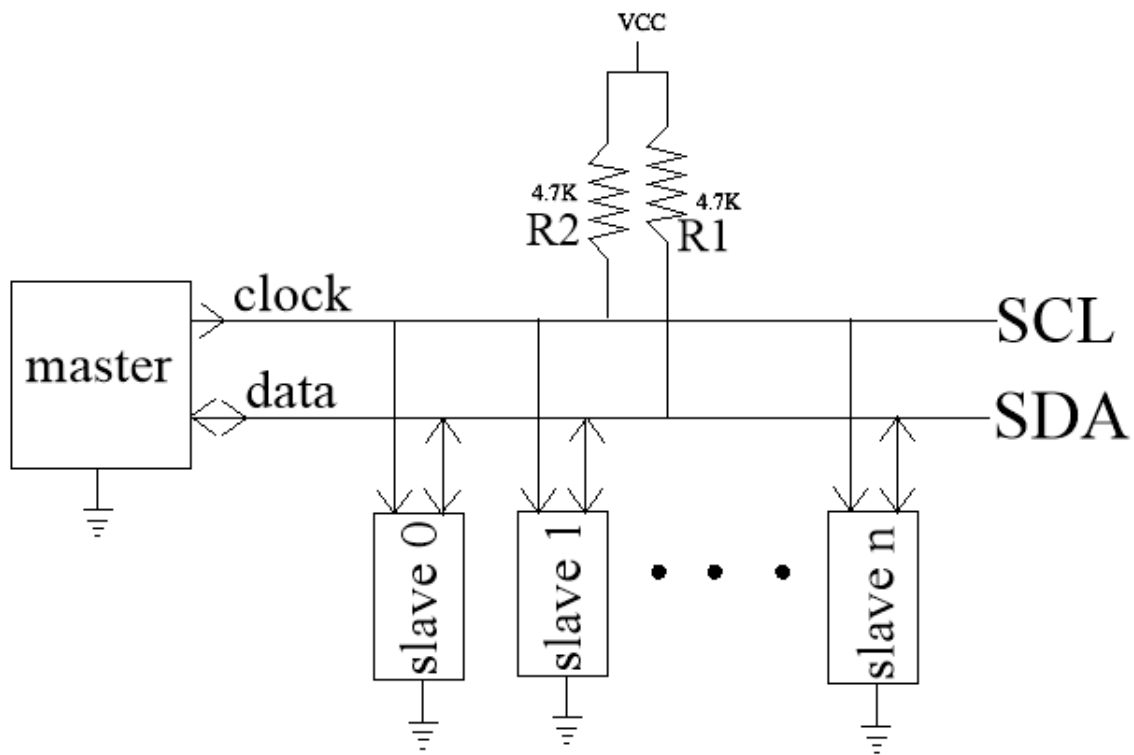
The I2C bus consists of a Serial Clock lead (SCL) and a Serial Data lead (SDA). If any boxes pull down on a lead, a logic 0 results. If no boxes are pulling down, you get a logic 1.

At any given time, there can only be one “master.” During a communication session, the master sources clock to all “slaves” as shown by the arrows going from the master to all the slaves. The master can dialog with any slave by calling the unique address built into each slave. When called upon, any slave can use the clock to place data on the SDA line.

From a software standpoint, a maximum of 127 slaves can exist on one I2C bus. Due to electrical² and failure mode constraints³, this number is typically fewer than 10.

² The longer the wires carrying clock and data, the more capacitance exists from each conductor to ground. This capacitance and its related pull-up resistor form an RC circuit. If you try to run the clock too fast, the resistor cannot charge up the capacitance fast enough to reach a logic 1. See Appendix I, page 34 for details.

³ You can imagine the difficulty of find which of 127 devices was pulling down on the clock or data lead because it was faulted.



From an electrical standpoint, when the master or a slave wants to send a logic 0, it pulls that lead to ground. When the master and all slaves are not pulling to ground, the lead can rise up to a logic 1 thanks to the pull-up resistors R1 and R2, which connect to a voltage source, VCC.

You will find that some I2C breakout boards have pull-up resistors with the option of disconnecting them. Do not have more than one set of pull-up resistors on a given I2C bus.

Notice that the pull-up resistors connect to VCC, usually the symbol for the device's power supply. In most cases, the voltage used to set logic 1 via the pull-ups is the same as the device's power supply voltage. For example, you can have a 5 volt powered device, and it expects to see a logic 1 equal to 5 volts. Consult the spec sheet for your device to be sure.

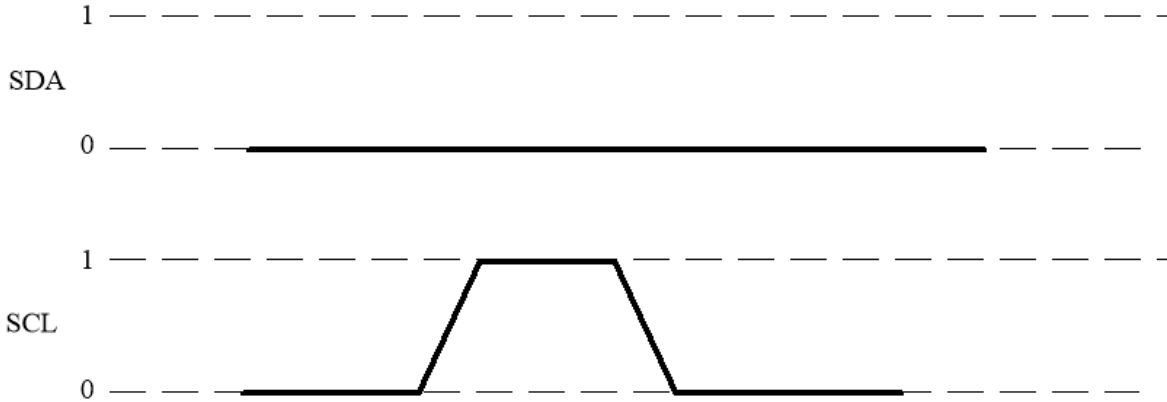
All I2C devices recognize ground as a logic 0 regardless of the supply voltage.

What do you do if some devices need 5 volt logic and the rest need 3.3 volt logic? You will need two I2C buses.

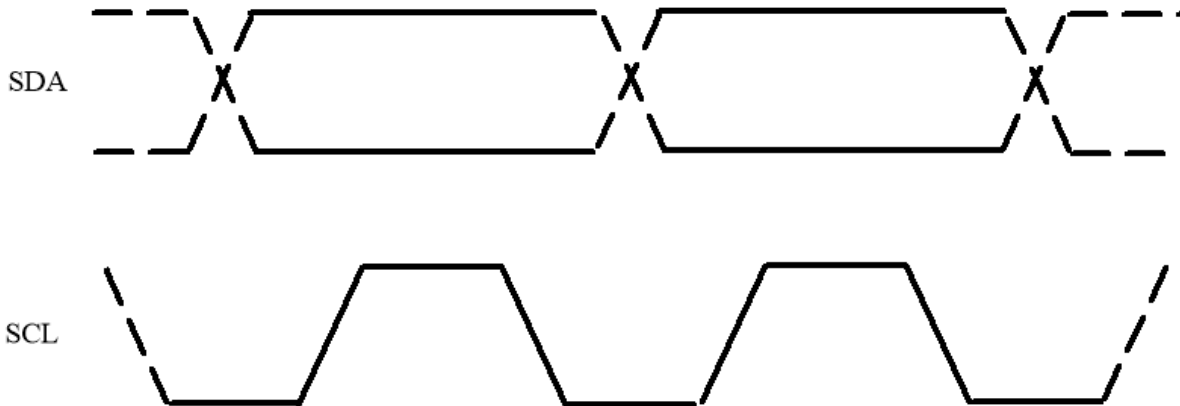
What do you do if you have two slaves with the same address? You will need two I2C buses to prevent them from responding simultaneously.

Most Arduino-compatible computers have only one hardware-based I2C bus, so you may need to employ SoftWire for extra I2C busses. It permits you to have many I2C buses. For every two GPIO pins, you get one I2C bus. You can only have one of these buses active at a time, but that is a small price to pay.

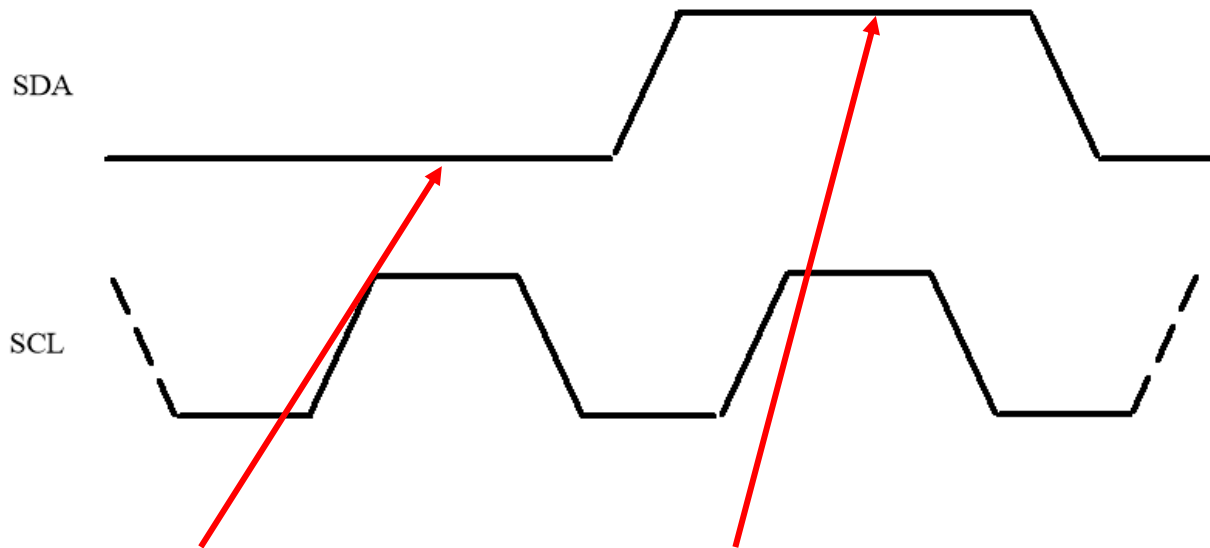
If SDA was at logic 0 during a clock pulse, a 0 is present.



The digital value only exists when the clock is at logic 1. During this time, SDA must be steady. SDA is free to change when SCL is at logic 0.

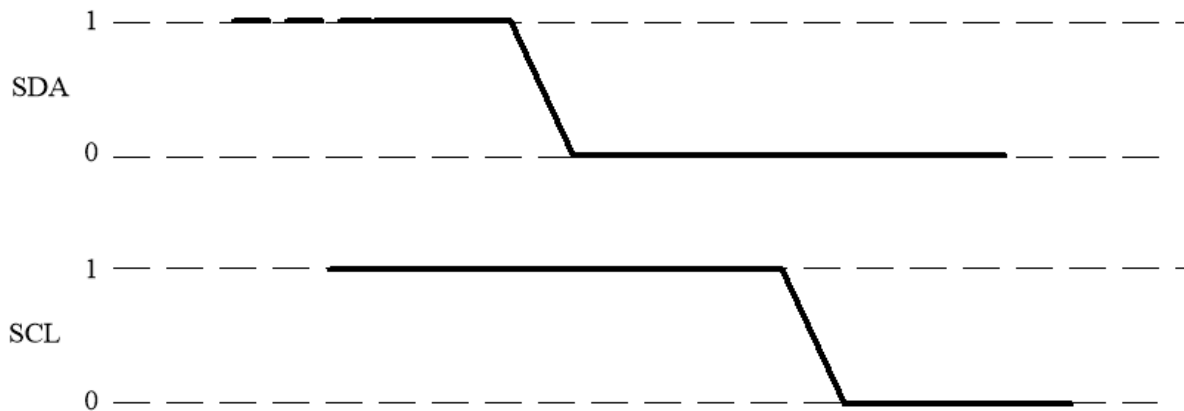


To transmit 01, we would see

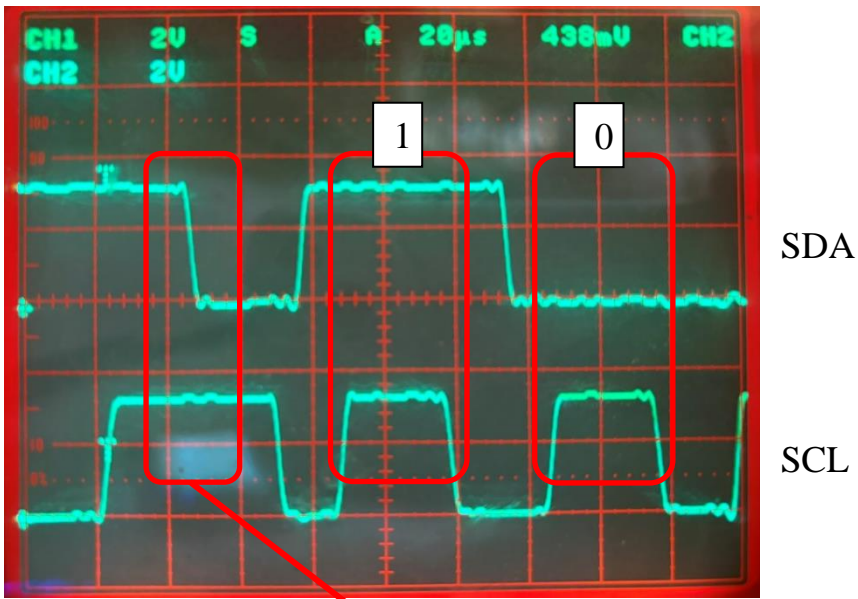


SDA is at logic 0 for the first clock pulse and logic 1 for the second. If you understand the rules so far, you know that SDA will not change while SCL is high.

To signal the start of a series of bits, we use a “START condition.”



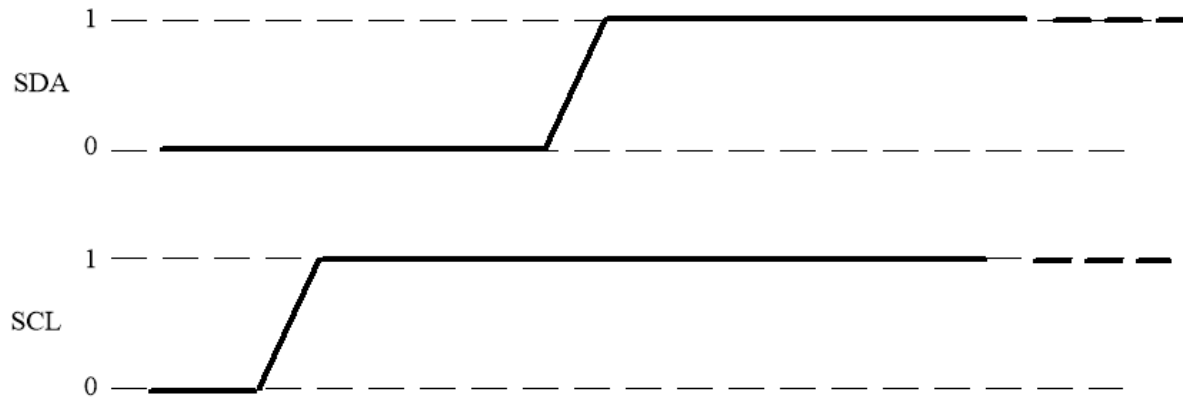
When SDA changes from logic 1 to logic 0 while SCL is high, we get a START condition. In [the standards](#), START is marked with an “S.” If subsequent START conditions occur, they can be marked with “Sr” or simply “S.”



First⁴, we have the START condition. Followed by 10.

⁴ The picture was taken with an iPhone of a Tektronix 2430A oscilloscope.

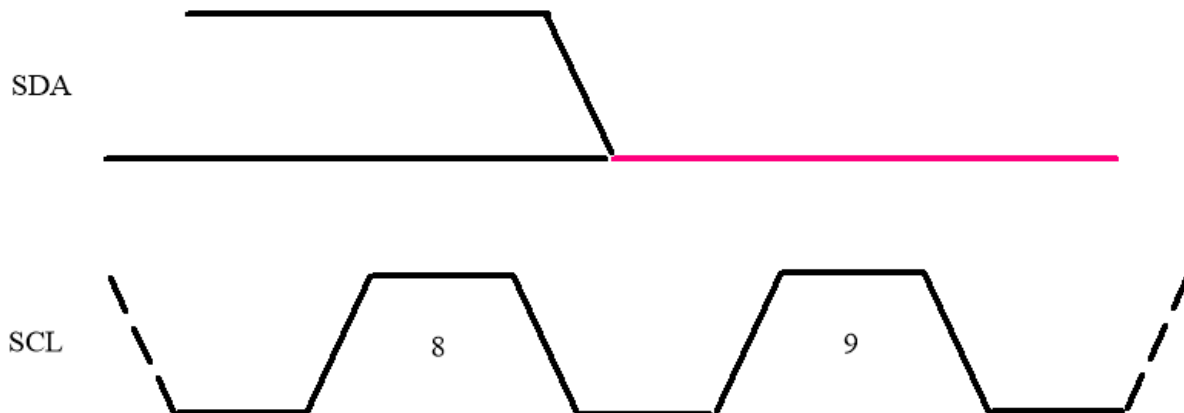
Similarly, we need a way to signal the end of a series of bits, a STOP condition.



When SDA changes from a logic 0 to a logic 1 while SCL is at logic 1, we have a STOP condition.

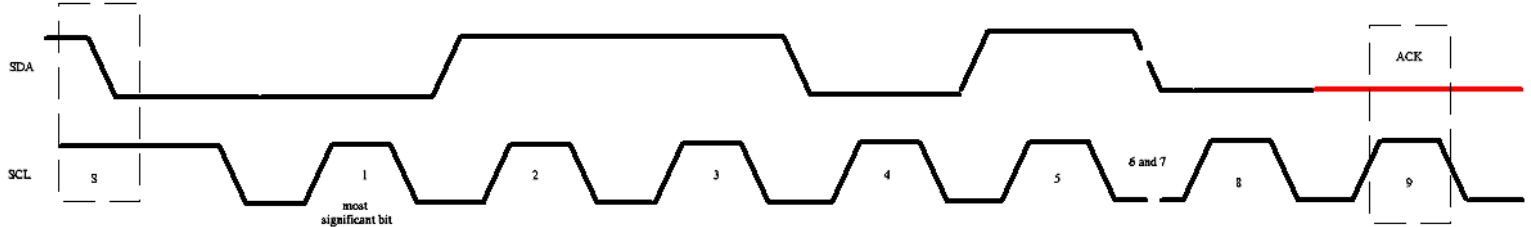
In the standards, STOP is designated with a “P.”

There is one more piece to this puzzle. After eight bits, the receiver takes control of the SDA lead for an acknowledgment (ACK) that it received the previous byte.



During clock pulse 8, SDA can be a logic 0 or 1. Then, whoever sent those bits changes from driving the SDA line to listening to it. The end that received the byte changes to driving the SDA line (in red) and sets it to logic 0 to indicate ACK. If the receiver doesn't successfully receive the byte, it is clueless, so it remains in the listening mode, causing the pull-up resistors to put logic 1 on SDA. This is called No-ACK or NACK.

Time to put this all together.

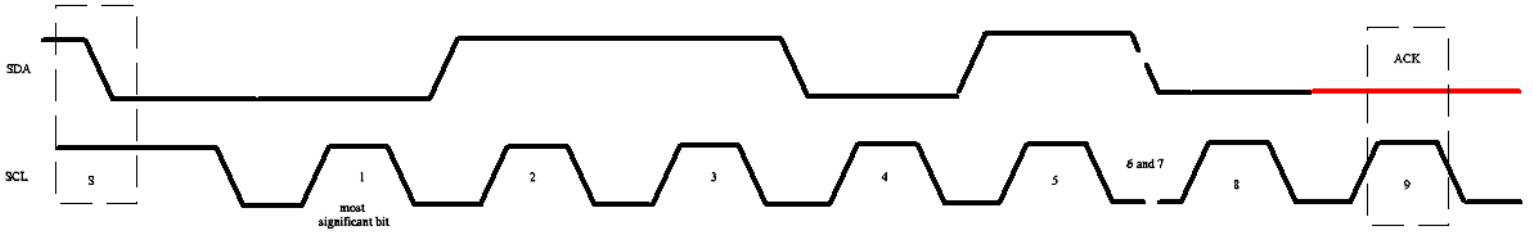


Before bits start to fly, SDA and SCL are sitting at logic 1 because neither the master nor any slaves pull it down to logic 0. Then, the master pulls SDA to logic 0 while leaving SCL high to give us a START condition.

Then, the most significant bit arrives. In our example, SDA is at logic 0, so the first bit is a 0. During clock pulse 2, SDA is at logic 1, so a 1 exists. SDA remained at logic 1 during clock pulse 3, so we get another 1. During clock pulse 4, SDA has transitioned to a logic 0 so we get a 0. Clock pulse 5 witnesses a logic 1, so we get a 1. Skipping clock pulses 6 and 7 because I think you get it by now, we arrive at clock pulse 8, our least significant bit. SDA is at logic 0, so our last bit is a 0.

The master then changes from driving SDA to listening to it, while the slave changes from listening to SDA to driving it and puts out a logic 0. This is the ACK, which remains low until clock pulse 9 is gone.

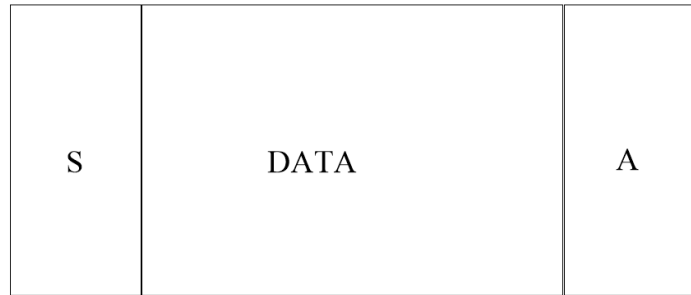
We would then either receive another byte of data followed by another ACK or, if done, receive a STOP condition as shown on the previous page.



To recap, we have the following pieces used to build I2C messages

- START
- Data: 0 or 1
- ACK or NACK after each byte of data
- STOP

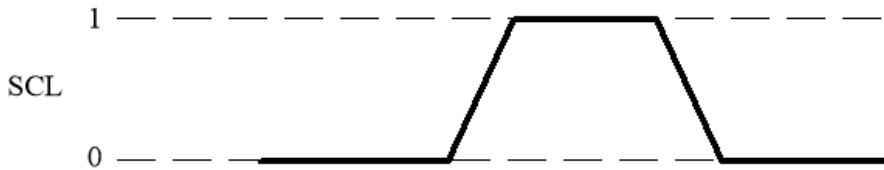
In a spec sheet you may see
START, DATA, and ACK
represented like this:



You will see these concepts again as we talk about SoftWire.

SoftWire

The SoftWire library consists of SoftWire.h and SoftWire.cpp. This code performs what is commonly called “bit banging.” While [Wire](#) accomplishes I2C by having hardware change the state of SDA and SCL, in SoftWire, it is all in the code.



To generate a clock cycle, we start with SCL at logic 0, then write a 1 to the SCL port, wait for $\frac{T}{2}$ microseconds, write a 0, and $\frac{T}{2}$ again. The clock “period” is then T microseconds.

If every clock cycle passed one bit of data, the baud rate would equal the reciprocal of the clock period. For example, if the clock was at logic 1 for 20 microseconds and then low for 20 microseconds, the baud rate would be $(\frac{1}{40 \text{ microseconds}}) = 25,000$ bits per second or $\frac{25,000 \text{ bits per second}}{8 \text{ bits per byte}} = 3125$ bytes per second.

But since each byte is accompanied by an acknowledgment, which takes one clock cycle, the baud rate is reduced to $\frac{25,000 \text{ bits per second}}{9 \text{ bits per byte}} = 2778$ bytes per second.

I will introduce you to the commands and then detail each one. Then, I will show you how they fit together to satisfy the interface of a specific I2C device.

The SoftWire Functions

`begin()`

`beginTransaction()`

`write()`

`endTransmission()` This is a big one.

`requestFrom()`

`read()`

`available()`

`stop()`

Currently, I'm not addressing the clock stretching feature.

At the bottom of each description is a list of return values.

Key Low-Level Elements

TransmissionInProgress is a flag that tells various commands how to behave.

The Transmit Buffer is an array that holds bytes to be sent to the designated slave.

The Receive Buffer is an array that holds bytes read from the designated slave.

Functional Descriptions of Each Function

begin()

begin(void) sends a STOP condition plus sets the **TransmissionInProgress flag to false**. It returns a value of 0. It is typically only in setup() and called once per bus. For example, if I want two SoftWire busses:

```
sw0.begin();
sw1.begin();
```

Return value: It does not return anything.

beginTransmission()

beginTransmission(<slave's address>) prepares for a new transmission by clearing the transmit buffer. This address is 7 bits. It saves the slave address for other functions to use. **It does not modify the TransmissionInProgress flag.**

Return value: It does not return anything.

write()

We have two flavors:

- write(<byte>) puts the byte into the transmit buffer.
- write(<an array's name>, <number of bytes>) reads the array and puts the specified number of bytes into the transmit buffer. For example, say we have the array Qaz[33] and want to put the first two elements into the transmit buffer. Then we would enter `write(Qaz, 2)`. **It does not modify the TransmissionInProgress flag.**

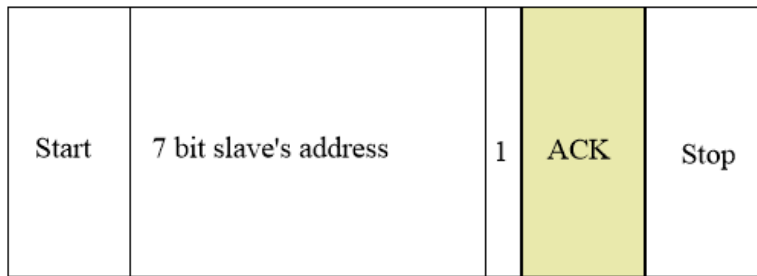
Return value: If the transmit buffer is full, it returns 0. If data was written, it returns a 1.

endTransmission()

We have a few possible behaviors:

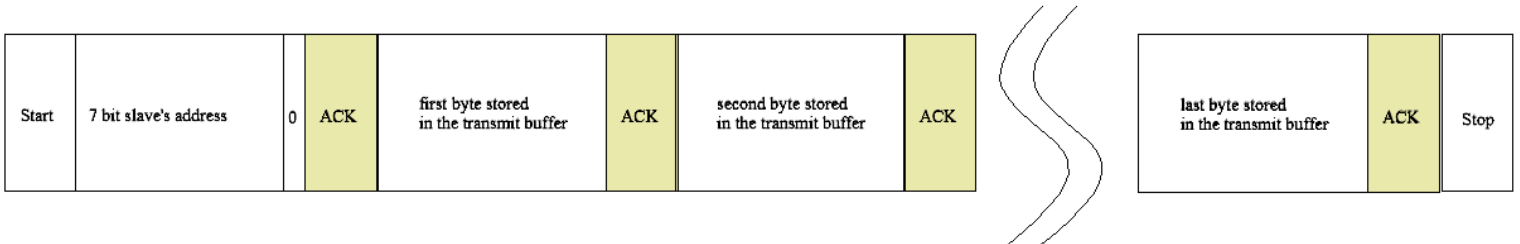
- `endTransmission(true)` first looks at the `TransmissionInProgress` flag.
 - If the `TransmissionInProgress` flag is true, it puts out
 - START,
 - the slave's 7 bit address is concatenated with a 1. This "1" signifies a read.
 - It then looks for an ACK from the slave. At this level of detail, assume the ACK was received.
 - It then sends STOP.

In graphical form:



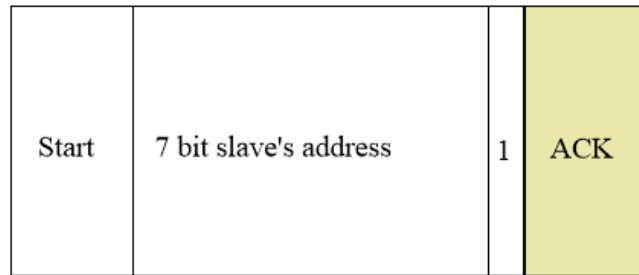
- If the `TransmissionInProgress` flag is false, it puts out
 - START,
 - the slave's 7 bit address is concatenated with a 0. This "0" signifies a write.
 - It then looks for an ACK. At this level of detail, assume the ACK was received.
 - Then it puts all bytes stored in the transmit buffer on the bus looking for ACK between each byte.
 - It then sends STOP.

In graphical form:



- endTransmission(**false**)
 - If the TransmissionInProgress flag is **true**, it puts out
 - START,
 - the slave's 7 bit address is concatenated with a 1. This "1" signifies a read.
 - It then looks for an ACK. At this level of detail, assume the ACK was received.

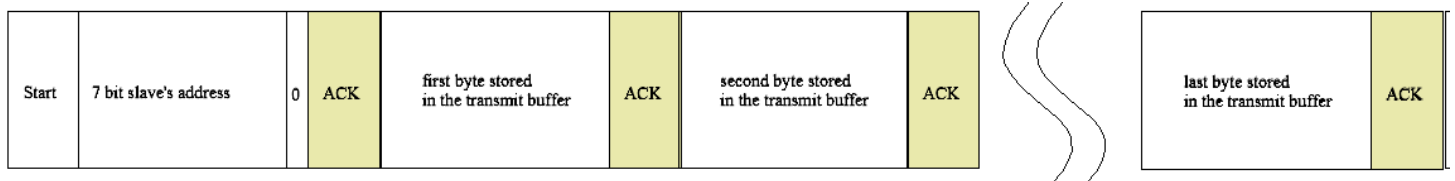
In graphical form:



Notice that no Stop exists on the end.

- If the TransmissionInProgress flag is **false**, it puts out
 - START,
 - the slave's 7 bit address concatenated with a 0. This "0" signifies a write.
 - It then looks for an ACK. At this level of detail, assume the ACK was received.
 - Then, it puts all bytes stored in the transmit buffer on the bus, looking for ACK between each byte.
 - It then sets the TransmissionInProgress flag to **true**.

In graphical form:



Notice that no Stop exists at the end.

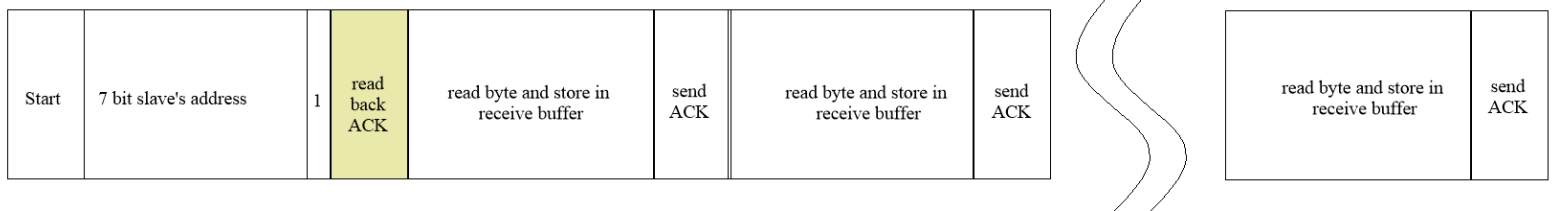
Return value: If it doesn't receive anything from the slave during the ACK clock cycle, it returns a NACK which is 2. Timeout is related to clock stretching and isn't covered in this document.

requestFrom()

`requestFrom()` has a few possible behaviors:

- `requestFrom(<slave's address>, <quantity>, false5)` has two possible behaviors:
 - If the `TransmissionInProgress` flag is **true**,
 - It clears the receive buffer
 - The 7 bit `<slave's address>` is concatenated with a "1", which means read, to become "Addr+1"
 - It puts out onto the bus
 - START
 - Addr+1
 - look for an ACK. At this level of detail, assume the ACK was received.
 - For each byte specified by `<quantity>`
 - Read back from the bus a bytes
 - send an ACK to the slave
 - put the received byte into the receive buffer.
 - `requestFrom()` returns the number of bytes read

In graphical form:

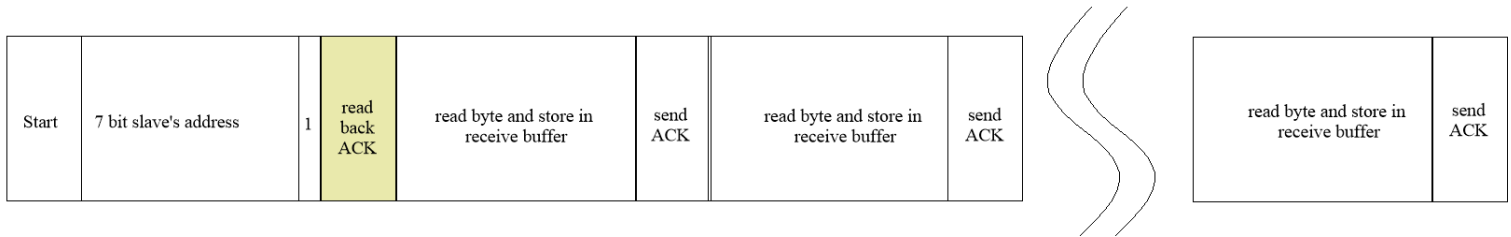


⁵ This is the `sendStop` flag, so making it false means we do not want to send a STOP condition.

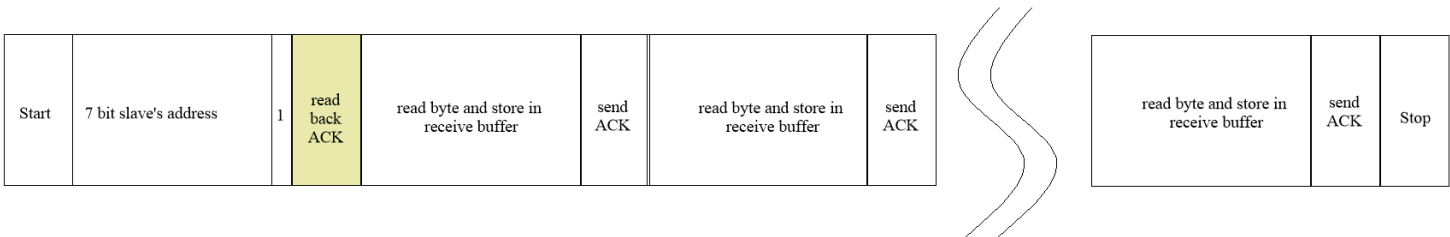
requestFrom(<slave's address>, <quantity>, **false**) continued

- **If the TransmissionInProgress flag is false,**
 - It clears the receive buffer
 - The 7 bit <slave's address> is concatenated with a "1", which means read, to become "Addr+1"
 - It puts out onto the bus
 - START
 - Addr+1
 - look for an ACK. At this level of detail, assume the ACK was received.
 - For each byte specified by <quantity>
 - Read back from the bus a bytes
 - send an ACK to the slave
 - put received byte into the receive buffer.
 - **It then sets the TransmissionInProgress flag to true.**
 - requestFrom() returns the number of bytes read

In graphical form:



- requestFrom(<slave's address>, <quantity>, true⁶) has two possible behaviors:
 - If the TransmissionInProgress flag is true,
 - It clears the receive buffer
 - The 7 bit <slave's address> is concatenated with a "1", which means read, to become "Addr+1"
 - It puts out onto the bus
 - START
 - Addr+1
 - look for an ACK. At this level of detail, assume the ACK was received.
 - For each byte specified by <quantity>
 - Read back from the bus a bytes
 - send an ACK to the slave
 - put the received byte into the receive buffer.
 - Put STOP onto the bus
 - It then sets the TransmissionInProgress flag to false⁷.
 - requestFrom() returns the number of bytes read
- In graphical form:



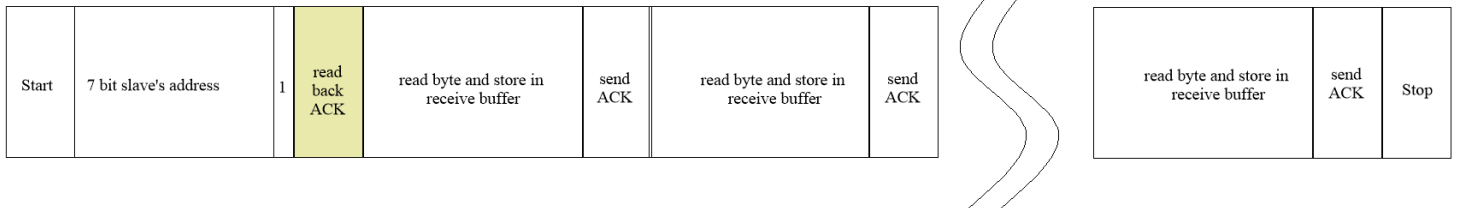
⁶ This is the sendStop flag, so making it true means we want to send a STOP condition.

⁷ Since we are sending a STOP condition, it means a transmission is not in progress so we set the TransmissionInProgress flag to false. Other functions need to know this.

requestFrom(<slave's address>, <quantity>, **true**) continued

- If the **TransmissionInProgress** flag is **false**,
 - It clears the receive buffer
 - The 7 bit <slave's address> is concatenated with a "1", which means read, to become "Addr+1"
 - It puts out onto the bus
 - START
 - Addr+1
 - look for an ACK. At this level of detail, assume the ACK was received.
 - For each byte specified by <quantity>
 - Read back from the bus a bytes
 - send an ACK to the slave
 - put received byte into the receive buffer.
 - Put **STOP** onto the bus
 - **It then sets the TransmissionInProgress flag to true.**
 - requestFrom() returns the number of bytes read

In graphical form:



Return value: the total number of bytes read

read()

Return value: This function returns one byte from the receive buffer and decrements the buffer pointer. This is a first-in, first-out buffer. If no byte is present, it returns -1.

Example:

Given you are expecting to get up to 16 bytes:

```
int qaz[16];
for(byte j = 0; j < 15; j++)
{
    qaz[j] = read();
    if(qaz[j] < 0) break; //if nothing in buffer, stop
    reading from it
}
```

available()

Return value: This function returns the number of bytes not yet read from the receive buffer.

Example:

```
if(available() = 2) //I am expecting 2 bytes
{
    byte qaz = read();
    byte wsx = read();
}
```

Now, available() will return 0.

stop(false)⁸

This function puts a STOP condition on the bus and sets the **TransmissionInProgress** flag to **false**.

Return value: 0 unless it is doing a clock stretch and has a failure.

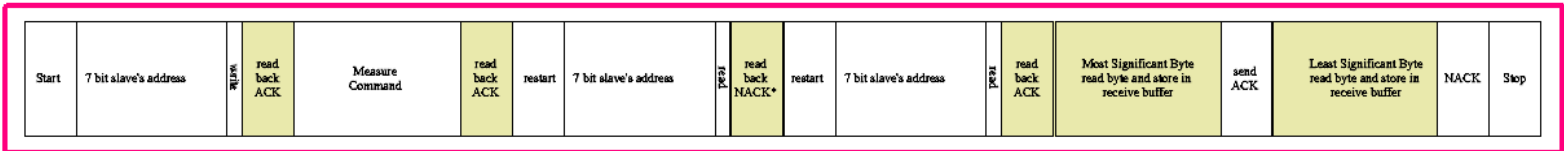
⁸ The function stop(true) involves clock stretching which I'm not addressing in this version of the document.

Example: The Si7021 Humidity and Temperature Sensor

I will periodically refer to [this](#) spec sheet from SILICON LABS.

Table 2 shows that it can take up to 10.8 milliseconds to do a 14-bit temperature conversion. Table 3 shows the I2C interface specs, but we will not be nearly this fast using SoftWire. Table 5 says the accuracy is within +/- 1 °C from -40 °C to +115 °C. Between -10 °C and 85 °C is +/- 0.4 °C maximum.

From Table 7, the storage temperature is from -65 °C to 150 °C, so it will survive my application where it will see -55 °C even though the accuracy will suffer at this temperature. It also specifies a maximum supply voltage of 4.2V while the voltage on any input pin is no more than the supply voltage plus 0.3V. We will run at 3.3V, so no input pin should get above 3.6V. **Note that 5 volts on the I2C pull-up resistors violates the spec and can damage the part.**



From section 5, we get the I2C Interface specs. The red border reminds you that this is from the device spec.

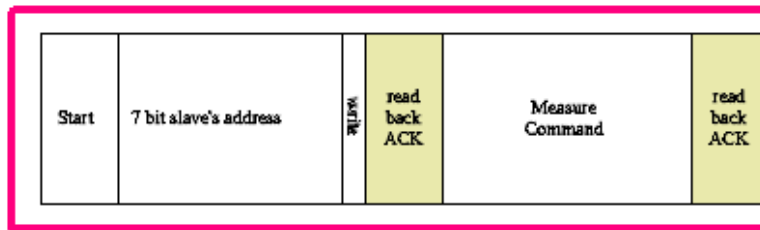
The 7 bit slave address is **0x40**. To measure temperature without “Hold Master Mode,” we will use command code **0xF3**. Hold Master Mode is tied to clock stretching, which I won't be using. Instead, I will wait the maximum time before asking for a reading.

From section 5.1⁹, we get the required details of a transaction. They are using terminology consistent with the I2C standards.

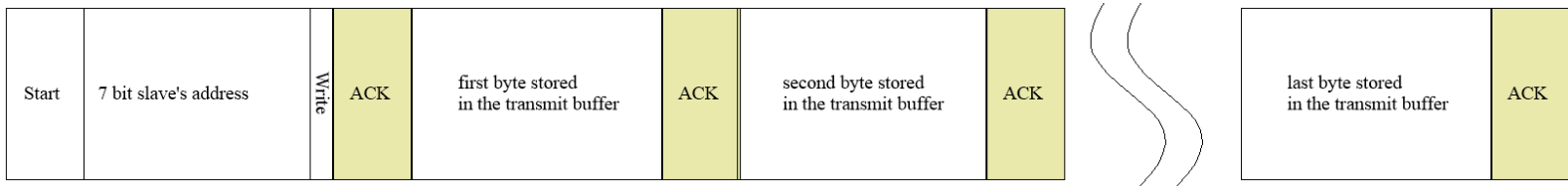
The white boxes are what the master sends, and the gray boxes are what the sensor sends.

⁹ I was unable to gain permission to reproduce their figures so had to draft a new one.

First, we must send START, the slave address, 0x40, a write bit, wait for an ACK, and then the measurement command, 0xF3, and receive another ACK.



Looking through the graphical forms, I found



I only need to send one byte, the Measurement command. This graphic corresponds to:

```
endTransmission(false)
TransmissionInProgress flag is false
```

But before we can call `endTransmission(false)`, we must put the Measurement command into the transmit buffer. The command `write(<byte>)` will do the trick.

So far, we have:

```
write (0xF3);
endTransmission(false);
```

Don't forget that we must have the `TransmissionInProgress` flag set to **false**.

`begin()` goes into `setup()`, where it will only be called once and will set the `TransmissionInProgress` flag set to **false**.

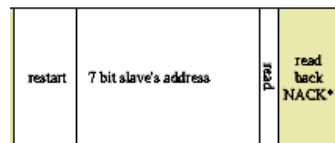
`beginTransaction(0x40)` clears the receive buffer and saves the slave address for other functions.

So far, we have

```
Setup()
{
    begin();//sets TransmissionInProgress to false
}

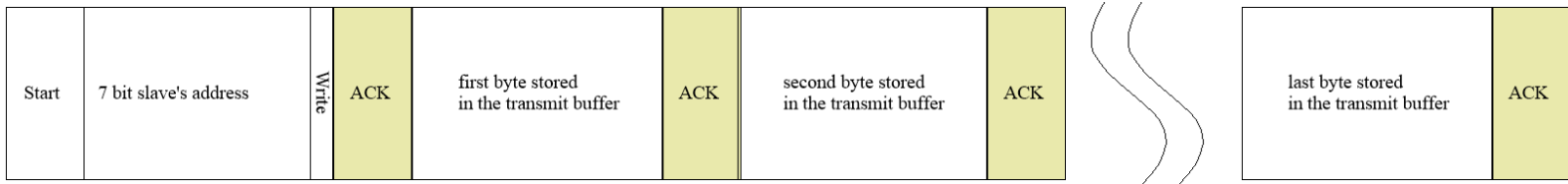
void TempSensor()
{
    beginTransmission(0x40); //sets TransmissionInProgress false
    write (0xF3);//has no effect on the TransmissionInProgress flag
    endTransmission(false);// TransmissionInProgress flag is false
}
```

which tells the sensor we want a temperature reading.



These sequences match, except we expect an ACK from the sensor, and the spec sheet says “NA*.” Following the asterisk brings us to

*Note: Device will NACK the slave address byte until conversion is complete.



In other words, we will get a NACK until the sensor is done with its conversion. When the conversion is completed, we will get our ACK. We just have to be sure our timer accommodates this delay in receiving an ACK. Using `setTimeout(1000)` when we set up the bus, we can tolerate up to a 1000 millisecond delay in receiving ACK. This delay will be compatible with all sensors I've seen yet not delay a failed sensor's detection by more than 1 second.

The above graphic comes from `endTransmission(false)` on page 19.

So now we have:

```

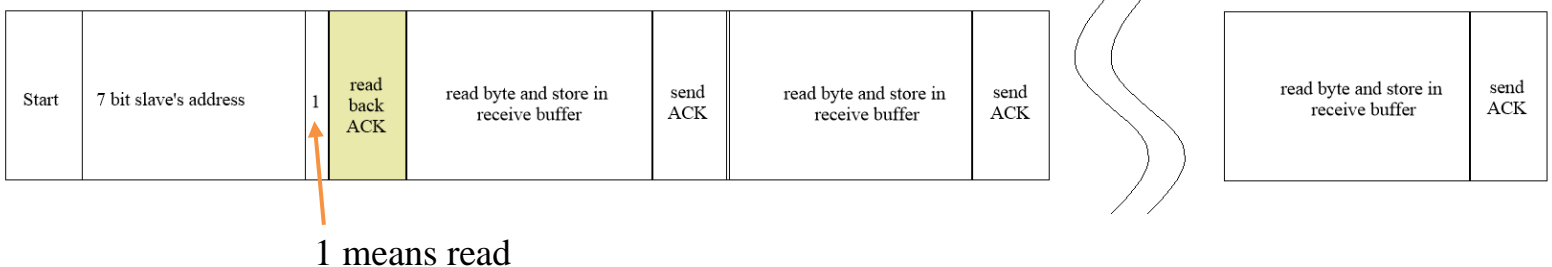
Setup()
{
    begin();//sets TransmissionInProgress to false
}

void TempSensor()
{
    beginTransmission(0x40); //sets TransmissionInProgress false
    write (0xF3);//has no effect on the TransmissionInProgress flag
    endTransmission(false);// TransmissionInProgress flag is true
}
    
```

Next, we need to ask for the two byte temperature reading.



On page 20, I found:



These match, except that the spec sheet says the master will send a NACK after reading back the second byte, while SoftWire will send an ACK - not a serious discrepancy, as was confirmed during my testing.

The corresponding command is

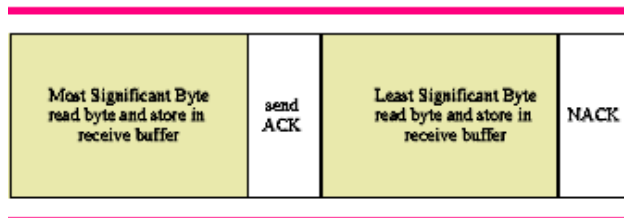
`requestFrom(0xF3, 2, false)` with the `TransmissionInProgress` flag **true**.

We now have

```
Setup()
{
    begin();//sets TransmissionInProgress to false
}

void TempSensor()
{
    beginTransmission(0x40); //sets TransmissionInProgress false
    write (0xF3);//has no effect on the TransmissionInProgress flag
    endTransmission(false);// TransmissionInProgress flag is false
    requestFrom(0xF3,2, false);//TransmissionInProgress flag is true.
```

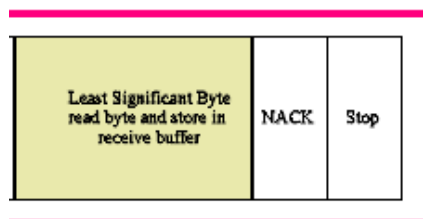
We should now have two bytes in our receive buffer.



We read the two bytes by calling read() twice. Notice that the first byte into the buffer was the Most Significant byte, and the second byte was the Least Significant byte. We read out in the same order.

```
byte MSB = read();//first in, first out buffer
byte LSB = read();
```

The final action is to end the conversation between the master and the slave. This is accomplished with a STOP condition, designated “P,” in the spec sheet.



All we need to do is call stop(**false**). It also sets the TransmissionInProgress flag to **false**.

Sample Code: Si7021 Device

Before setup():

```
#include <SoftWire.h>
#include <AsyncDelay.h> //used by SoftWire
int sda0Pin = 8; //define logical pins used for bus 0
int scl0Pin = 9;
int sda1Pin = 16; //define logical pins used for bus 1
int scl1Pin = 10;
SoftWire sw0(sda0Pin, scl0Pin); //define the two I2C busses
SoftWire sw1(sda1Pin, scl1Pin);
// These buffers must be at least as large as the largest read or write you
perform.
char sw0TxBuffer[16]; //used by SoftWire bus 0
char sw0RxBuffer[16]; //used by SoftWire bus 0
char sw1TxBuffer[16]; //used by SoftWire bus 1
char sw1RxBuffer[16]; //used by SoftWire bus 1
AsyncDelay readInterval; //required by SoftWire docs
byte Si7021AddressByte = 0x40; //this is the 7 bit address of our sensor
byte measureTempByte = 0xF3; //this is the 8 bit command for reading temp
```

The first `#include` brings in `SoftWire` header file, which defines functions and variables within `SoftWire.cpp`, which it includes. `SoftWire` uses `AsyncDelay`, so that is our second `#include`.

Next, define which logical pins on our Pro Micro processor will be used for each of the two `SoftWire` buses. Then, we make two copies of `SoftWire`. One is called `sw0` with its `SDA` and `SCL`, while the other is called `sw1` with its `SDA` and `SCL`. Next, the transmit and receive buffers for each I2C bus are defined.

We make one copy of `AsyncDelay`, which we included, and call it `readInterval`. The last two lines are specific to the I2C device we will drive.

In setup()

```

setup()
{
    sw0.setTxBuffer(sw0TxBuffer, sizeof(sw0TxBuffer)); //set up bus 0
    sw0.setRxBuffer(sw0RxBuffer, sizeof(sw0RxBuffer));
    sw0.setDelay_us(5); //from SoftWire docs
    sw0.setTimeout(1000); //from SoftWire docs
    sw0.begin(); //called only once, at setup

    sw1.setTxBuffer(sw1TxBuffer, sizeof(sw1TxBuffer)); //set up bus 1
    sw1.setRxBuffer(sw1RxBuffer, sizeof(sw1RxBuffer));
    sw1.setDelay_us(5);
    sw1.setTimeout(1000);
    sw1.begin();

    readInterval.start(2000, AsyncDelay::MILLIS); //from SoftWire docs
}

```

SoftWire requires the next block of code. I didn't change any of it. Notice that I used the same commands for each bus, only changing the bus name.

Then comes our first SoftWire operational command: `begin()`. It sets the initial conditions on each bus by sending out a STOP condition and setting the `TransmissionInProgress` flag to false.

Although we have set up two I2C buses, SoftWire can only operate one at a time.

This last line is required by SoftWire.

This concludes the setting up of the environment to run my I2C sensor driver.

The I2C Driver

I am driving a [Si7021 Humidity and Temperature sensor](#) connected to SoftWire bus 0. I will read the temperature and place the raw result into an array.

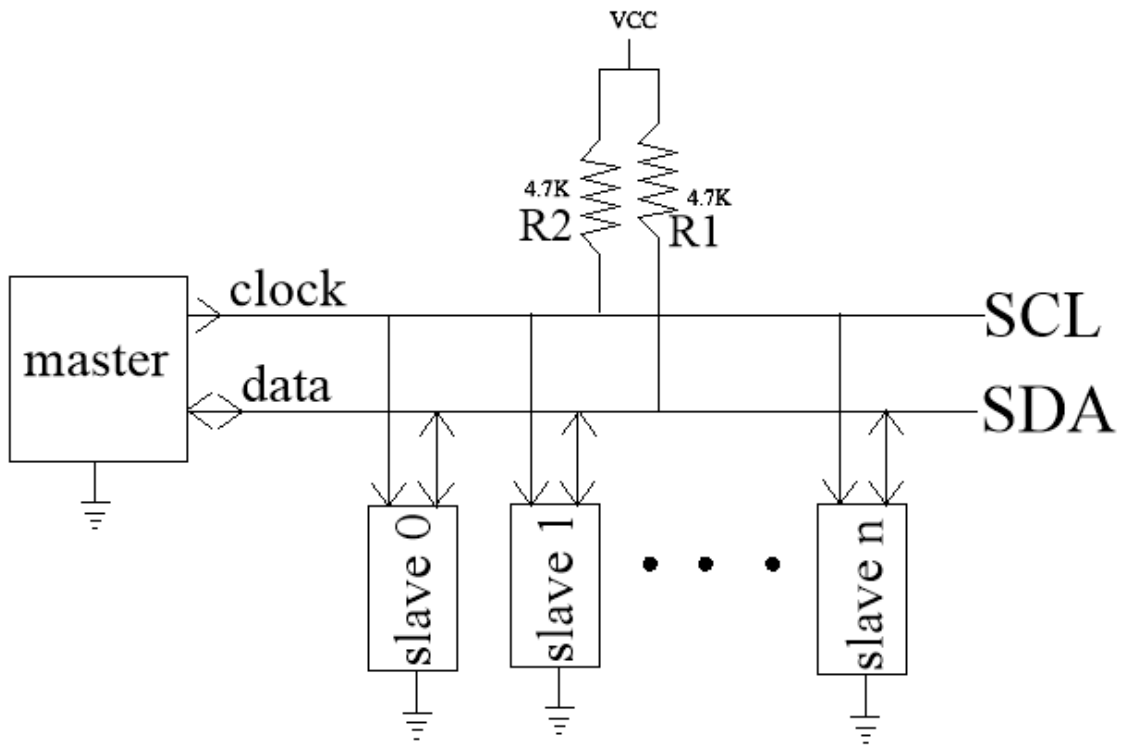
```

void Si7021Driver()
{
sw0.beginTransaction(Si7021AddressByte);//bus 0. Sensor address is 0x40
    if(!sw0.write(measureTempByte)//returns 0 for fault and 1 if OK
    {
        with failure
            receivedDataByte[16] = failedByte;//populate status byte
            return;
    }
    if(sw0.endTransmission(false) != 0)
    {;//send slave address, write and measurement command
        with failure
            receivedDataByte[16] = failedByte;//populate status byte
            return;
        }
    }

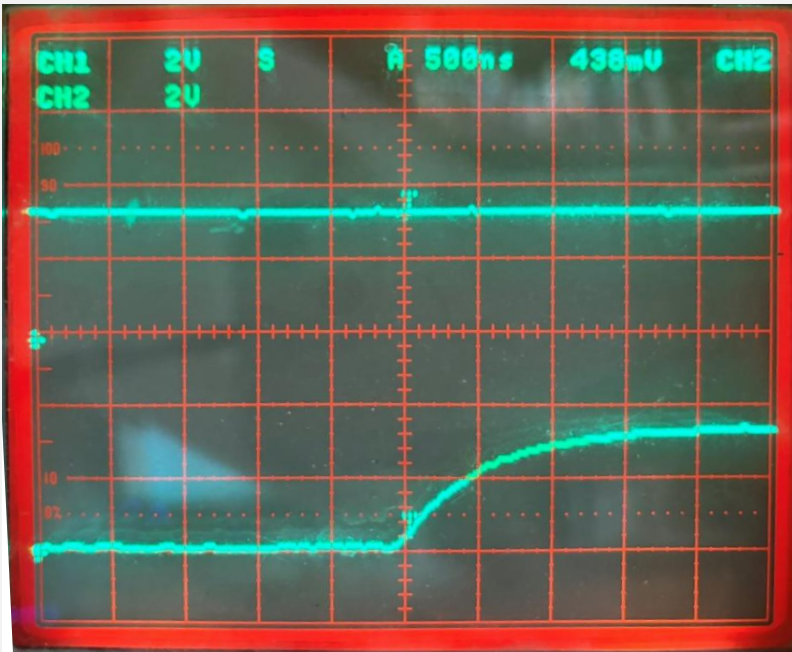
    delay(20);//time for sensor to take a reading plus software
    overhead
    sw0.requestFrom(Si7021AddressByte,2, false);//command a read of 2
    bytes of data
    unsigned long startTimeULong = millis();
    byte byteCountByte = 0;
    while(1)
    {;//read the MS Byte and LS Byte
        if(sw0.available() >= 2)
        {;//our two bytes are waiting to be read
            receivedDataByte[0] = sw0.read();//MSB
            receivedDataByte[1] = sw0.read();//LSB
            receivedDataByte[16] = successByte;//set status byte
            return;
        }
        }else
        {
            if((millis() - startTimeULong) > 200)
            {;//we timed out waiting for our temp reading
                receivedDataByte[16] = timedOutByte;//4
                return;//give up on sensor after waiting 200 ms
            }
        }
    }
    sw0.stop(false);
}

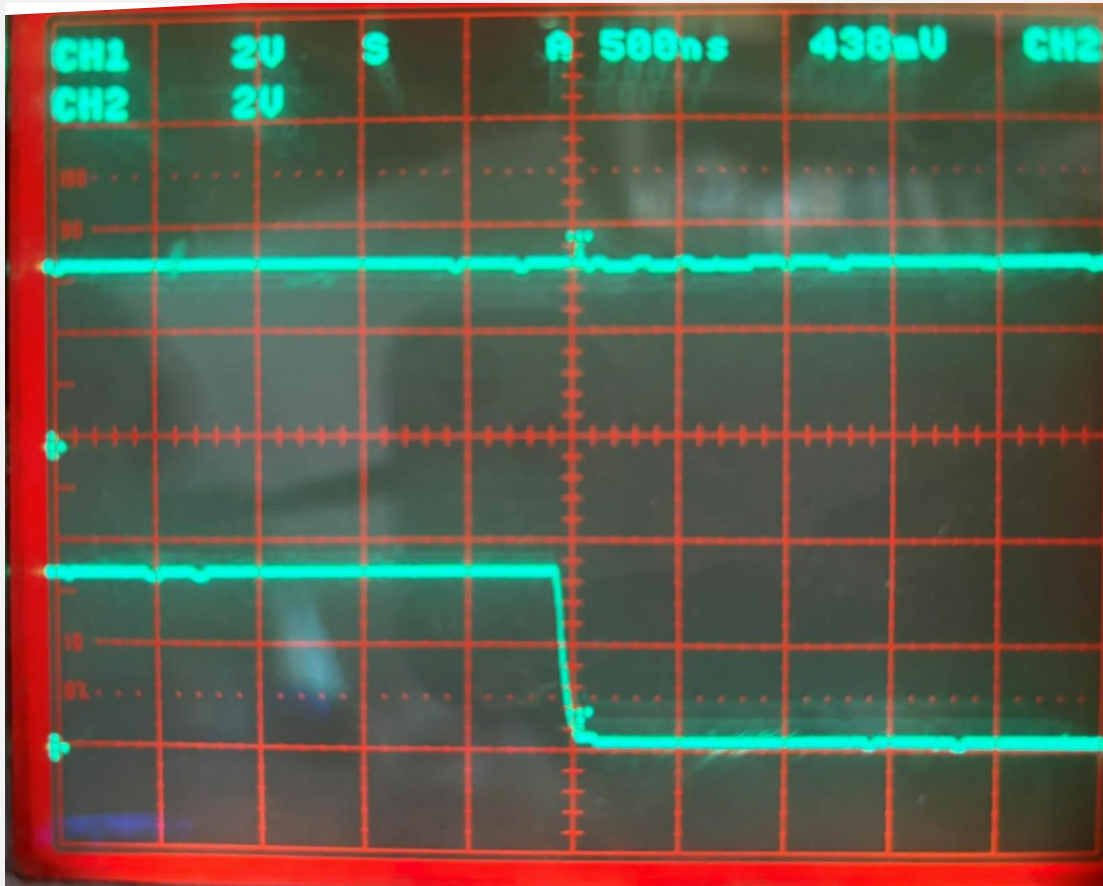
```

Appendix I: Analog Aspects of I2C



The slow rise time due to the capacitance of the SCL lead and the 4.7K pull-up are evident here. It takes about 1.5 microseconds to go from a logic 0 to a logic 1.





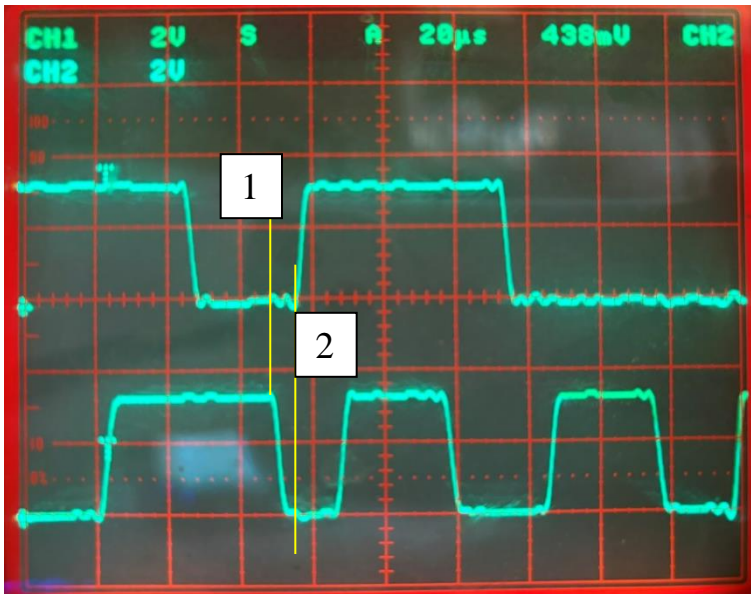
Here, we see the negative edge of the clock. The Pro Micro is pulling down to a logic 0 and discharging the capacitance of the line. It can sink up to about 40 mA, so the transition from logic 1 to 0 takes place in about 100 ns.

We can reduce our rise time by using smaller pull-up resistors, but this will reduce the available current during the fall time because the device must sink both the capacitive and pull-up current.

As a rough check, note that our rising clock edge follows $v = 3.3e^{-\frac{t}{\tau}}$ volts. Note that we are at 2 volts after 400 ns. This gives us a $\tau = 800$ ns, which equals $R \times C$. With $R = 4.7k$, we can calculate the total capacitance on the node: 170 pf.

The initial fall time is about 2 volts in 50 ns and is approximately linear so that we can say $I = C \frac{dv}{dt} = 170 \text{ pf} \frac{2V}{50 \text{ ns}} = 6.8 \text{ mA}$, which is sunk by the Pro Micro. That is far less than the maximum rated current of 40 mA.

Comparing the Software to the Waveshapes



Within `SoftWire.cpp` is a function called `IICWrite()`¹⁰. You can see how it controls SCL and SDA to form the needed signals on the I2C bus. Periodically, it adds a delay set to 5 microseconds (μs) in my application.

Near the top of this function, they drive SCL low, marked as line 1. Then, we have an `if()` that releases SDA if true (line 2). Note that the two lines are around 8 μs apart. This is the apparent time needed to execute the code. Later in the scope trace, I found a delay of about 10 μs while the code specified 5 μs .

This added delay tells me `SoftWire` is running near its maximum baud rate. The software delays would dominate if we reduced the specified delay to zero. These delays are a function of how many lines of code are executed between events, plus the type of commands. Even at 5 μs , you can see variation in the time SCL is low.

I see a maximum clock period of around 60 μs . To pass one byte plus the ACK takes 9 clock periods, 540 μs . Assuming we pass a slave one byte for its address, one byte for accessing an internal register, and then two bytes passed back; we are talking about a total of four bytes or about 2 milliseconds (ms). The spec sheet for the device studied says it takes no more than 10.8 ms to take a reading, but empirically, we have found the system needs a delay of 20 ms. If the I2C bus took zero time, we would still be looking at 18 ms, so getting the bus to run faster wouldn't help the response time much. There is little value in running the I2C bus faster.

¹⁰ The "I" stands for low level and not eleven.

Acknowledgments

Thanks to the authors of SoftWire for writing code that I could comprehend.

I welcome your comments and questions.

If you want me to contact you each time I publish an article, email me with "Subscribe" in the subject line. In the body of the email, please tell me if you are interested in metalworking, software plus electronics, kayaking, and/or the Lectric XP eBike so I can put you on the right distribution list.

If you are on a list and have had enough, email me "Unsubscribe" in the subject line. No hard feelings.

Rick Sparber
rgsparber@AOL.com
Rick.Sparber.org