

I2C Bit-Banging Tutorial: Part I

POSTED ON AUGUST 19, 2016

Introduction

I2C (or Inter-Integrated Circuit) is a pretty neat communications protocol. It needs only two wires (plus ground!), it's synchronous, it supports multiple masters and slaves, and it's fairly straight-forward to implement without relying on dedicated microcontroller hardware. Lately I've been taking advantage of that last advantage: implementing I2C on a microcontroller using software, a.k.a. bit-banging.

The need for bit-banging arises when, as mentioned, the microcontroller you're working with has no dedicated I2C hardware or said hardware is in use with other pin functions. For example, I've been using the [PIC16LF18323](http://www.microchip.com/wwwproducts/en/PIC16LF18323) (<http://www.microchip.com/wwwproducts/en/PIC16LF18323>) as the Tx controller in the [musicFromMotion](https://calcium3000.wordpress.com/projects/mfm/) (<https://calcium3000.wordpress.com/projects/mfm/>) project which has a Master Synchronous Serial Port (MSSP) module, which can operate in either I2C or SPI mode. Though SPI is typically simpler to bit-bang (especially in master mode) it can take a toll on the data rate, which was more important in my application. Thus I can use the dedicated MSSP for SPI, then bit-bang I2C on any two free GPIO pins. The device we're speaking to is an [MMA8653](http://www.nxp.com/products/sensors/accelerometers/3-axis-accelerometers/2g-4g-8g-low-g-10-bit-digital-accelerometer:MMA8653FC?lang_cd=en) (http://www.nxp.com/products/sensors/accelerometers/3-axis-accelerometers/2g-4g-8g-low-g-10-bit-digital-accelerometer:MMA8653FC?lang_cd=en) accelerometer to capture the 'motion' of the musicFromMotion.

Like any good brick-layer, we'll start from the ground up. First, we'll enjoy a quick tutorial on the I2C protocol in Softwareland and Hardwareland; next, we'll lay out basic input/output (I/O) functions followed by basic I2C functions. In Part II we'll use these basic functions to speak with an example device — an MMA8653 accelerometer — and look to where we can go next. In later posts we'll explore some troubleshooting techniques, and even develop a debugging project! Joy!

I2C Protocol

Here's a conversation between me and my boss:

RECENT POSTS

New mini project: nRF24L01+ Raspberry Pi adapter

Albuquerque Mini Maker Faire!

Keyboard Keyboard in Pure Data

Data from nRF24L01+ to Pure Data

I2C Bit-Banging Tutorial: Part I

CATEGORIES

Learning (2)

Maker Faire (2)

Press (1)

Projects (3)

Search

"Initiate conversation. Calvin, I want to give you orders," says he.
"Acknowledged," I reply.
"Clean your desk."
"Gotcha."
"End conversation."

Simple enough, yeah? We've just witnessed an I2C transaction! First, my boss (the 'master') initiates the conversation and addresses me (the 'slave'). I acknowledge, he gives me a command. I acknowledge, he ends the conversation. This describes an I2C *write byte* (or *send byte*) function, and it's one of the simplest I2C transactions there is. We commonly use such a function for basic, no-response transactions such as clearing faults or setting modes on the slave device.

For a bit more complexity, we can simply repeat the command-acknowledge lines to form an I2C *write byte data* function:

"Initiate conversation. Calvin, I want to give you orders," says he.
"Okay then," I reply.
"That desk cleaning project I have you on."
"Yes."
"Put all of your pens in the blue jar."
"Will do."
"End conversation."

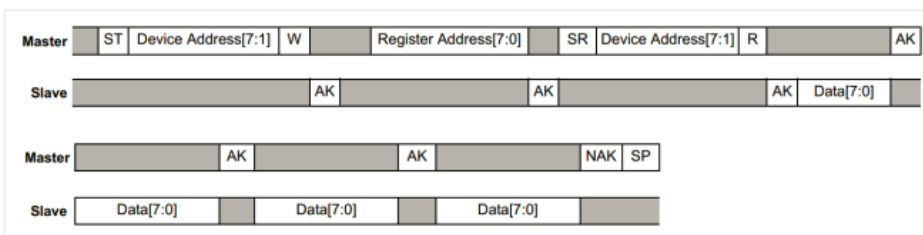
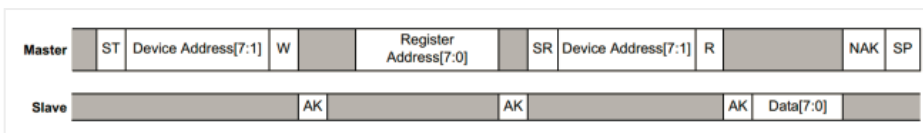
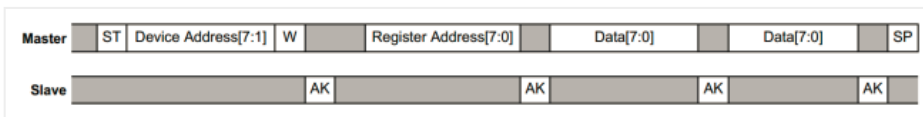
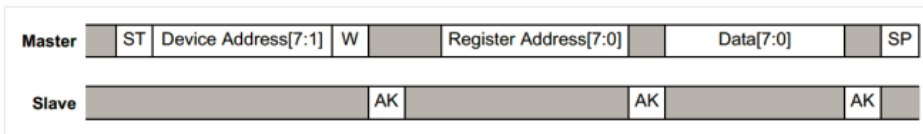
Here my boss is initiating the conversation, addressing me, and I acknowledge just like before. But then he gives me a context — a *register* or *command code* — and I acknowledge, then he gives me a command related to that context (*data*), I acknowledge, and he ends the conversation. This kind of transaction is used more frequently than plain old *write byte* since most devices have multiple memory registers that need to be addressed before a command can be executed.

Now if my boss ever tires of giving me orders and wants my input, things get a bit trickier. Since he's the master, I can only speak when he requests information (live is tough for integrated circuits). So the most basic way to answer his request is via I2C *read byte*:

"Initiate conversation. Calvin, I want to give you orders," he orders.
"Surely," I say.
"I am interested in the color pens in that blue jar."
"Right-o."
"Re-initiate conversation. Calvin, I want you to tell me."
"Red."
"Not acknowledged. End conversation."

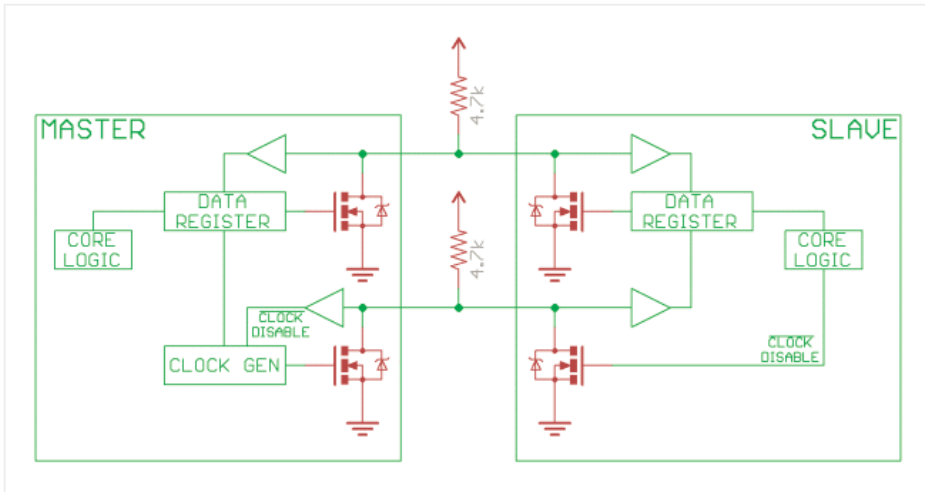
Now in English this sounds pretty silly, but this is the world of I2C. The first two lines are the usual, but then my boss gives me context of what he wants from me (pens in the blue jar). I acknowledge, but don't tell him anything yet. He then re-initiates the conversation (called a *repeated start condition*) and addresses me with a request for information. I give it to him, he not-acknowledges me (how rude!) and ends the conversation. Such a transaction is used to receive data from, say, a sensor or peek into the device's configuration. Extending to receiving multiple

Translating these conversations to I2C jargon, these ‘initiate’ and ‘end conversation’ sentences represent *start* and *stop conditions*, typically abbreviated as *S* and *P* or *ST* and *SP*, respectively. The funky ‘re-initiate conversation’ is called a *repeated start* (*RS* or *SR*) condition and is functionally no different than a start condition, but its place in the transaction lends it its name. *Acknowledges* (‘right-o,’ ‘surely,’ ‘gotcha’) and *not-acknowledges* are abbreviated as *ACK* and *NAK* or *A* and *N*, respectively. My name is an *address*, and the ‘give you orders’ versus ‘tell me things’ represent *write* (*W*) and *read* (*R*) commands. Below is a nice visual reference for these transactions, courtesy of [NXP \(http://www.nxp.com/\)](http://www.nxp.com/):



Electrically, I like to think of I2C as being a pessimistic protocol. No device ever holds a bus line up — it can only bring it down. This is because I2C uses an open-drain type of communication, meaning that instead of outputting a positive voltage for a logical 1 the device goes into a high-impedence (Hi-Z) state, effectively removing its pin from the bus. For a logical 0, however, the device actively shorts the line to ground. What a stick in the mud.

What keeps ones at a positive voltage are pull-up resistors on each bus line, so that when a pin 'secedes' the other device reads the value of the line as a positive voltage. Since the pull-up resistor tends to be at least a few kilo-Ohm, when a device pulls a line low the line falls to just about zero Volts (not exactly zero due to internal resistance of the microcontroller pin drivers, but good enough for us).



- Figure 5: electrical connections for basic I2C communication (courtesy of Sparkfun).

As mentioned earlier, I2C is a two-wire protocol (plus ground!), a.k.a. a two-wire interface (TWI), that includes one data line (*SDA*) and one master-generated clock line (*SCL*). Thus we only have two pins to keep track of! So let's dive into how to twiddle *SCL* and *SDA* to make Figures 1-4 happen!

Beginning Bit Banging

Alright then, let's do this! I'll try to make this as universal as possible, meaning that I'll provide code examples in C for PIC and MSP430 microcontrollers. (Note: for PIC users I'm using the PICkit 3 with MPLAB X IDE v3.55; check out the Instructable [Programming PIC Microcontrollers](http://www.instructables.com/id/Programming-PIC-Microcontrollers/?ALLSTEPS) (<http://www.instructables.com/id/Programming-PIC-Microcontrollers/?ALLSTEPS>) for initial setup of the project.)

Initializing

Before executing any I2C transactions we'll need to configure the microcontroller's appropriate pins. Be sure to change the appropriate pin numbers and port names to those used in your project! For now just skim over the code; all will be revealed by the end of this post:

```
// Initializing microcontroller for I2C with a PIC:
#include <xc.h>
#include <stdint.h>
#include <stdbool.h>

#define _XTAL_FREQ      8000000    // crystal frequency [Hz], needed
#define I2C_DELAY       5    // microseconds
#define SCL              0
#define SDA              1

void i2c_init(void)
{
    TRISC |= (1 << SCL) | (1 << SDA);    // set as inputs (Hi-Z)
    ANSEL &= ~(1 << SCL) | (1 << SDA));    // set as digital
    __delay_us(I2C_DELAY);    // compiler-specific function: MPLAB X
}
```

```
// Initializing microcontroller for I2C with MSP430:
#include <msp430g2553.h>

#define I2C_DELAY       10    // clock cycles (5 microseconds at 8MHz)
#define SCL             BIT0
#define SDA             BIT1

void i2c_init(void)
{
    P2SEL &= ~(SCL | SDA);    // I/O enabled
    P2SEL2 &= ~(SCL | SDA);
    P2DIR &= ~(SCL | SDA);    // set as inputs (Hi-Z)

    __delay_cycles(I2C_DELAY);
}
```

Setting and Clearing Bits

The most basic operations we'll need are setting (logical 1) and clearing (logical 0) bits. You may laugh, but due to the open-drain nature of I2C this isn't wholly trivial. Instead of just outputting ones and zeros willy nilly, we'll have to set the desired pin as an input when we want to set a bit. This may seem a bit bizarre, but when microcontrollers are set as inputs their pin states are set to high impedance, or open-drain. Then we'll need to set the pin as an output and set it low to clear a bit.

Let's take a look at setting a bit:

```
// Setting a bit with a PIC:
```

```
void set_bit(uint8_t pin)
{
    TRISC |= (1 << pin);
}
```

```
// Setting a bit with MSP430:
```

```
void set_bit(uint8_t pin)
{
    P2DIR &= ~pin;    // set as input
}
```

And clearing a bit may look more familiar:

```
// Clearing a bit with a PIC:
```

```
void clear_bit(uint8_t pin)
{
    TRISC &= ~(1 << pin);
    LATC &= ~(1 << pin);    // PORTC can be used on PICs without ou
}
```

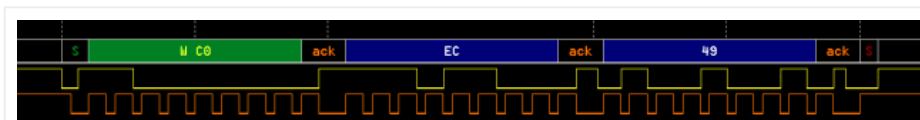
```
// Clearing a bit with MSP430:
```

```
void clear_bit(uint8_t pin)
{
    P2DIR |= pin;    // set as output
    P2OUT &= ~pin;    // set low
}
```

Idling and Start/Stop Conditions

First, a quick definition: idle. An idle bus means that both SCL and SDA are high — this is the state of the bus when no transactions are taking place. Remember that lines on a bus being high (logical 1) means that no device is actively pulling a line low (logical 0).

When the bus is idle and the master now wants to initiate an I2C transaction, she begins with a start condition. Here, she pulls and holds down the SDA line (clear SDA), waits, then pulls and holds down SCL (clear SCL), and waits again. This signals to the slave that a transaction is about to take place on the bus. See Figure 6 for a visual representation.



- Figure 6: example I2C write transaction. The orange trace represents SCL, the yellow SDA, and the I2C interpretation rests on top.

To write a start condition function we'll use our previous *set_bit()* and *clear_bit()* functions:

```
// Initiating a start condition with a PIC:
void start_condition(void)
{
    set_bit(SCL);
    set_bit(SDA);
    __delay_us(I2C_DELAY);
    clear_bit(SDA);
    __delay_us(I2C_DELAY);
    clear_bit(SCL);
    __delay_us(I2C_DELAY);
}
```

```
// Initiating a start condition with MSP430:
void start_cond(void)
{
    set_bit(SCL);
    set_bit(SDA);
    _delay_cycles(I2C_DELAY);
    clear_bit(SDA);
    _delay_cycles(I2C_DELAY);
    clear_bit(SCL);
    _delay_cycles(I2C_DELAY);
}
```

Stop conditions are sort of start conditions in reverse. The master pulls SDA low, waits, pulls SCL high, waits, pulls SDA high, and waits. Then we're back in idle, ready for more I2C action!

```
// Initiating a stop condition with a PIC:
void stop_condition(void)
{
    clear_bit(SDA);
    __delay_us(I2C_DELAY);
    set_bit(SCL);
    __delay_us(I2C_DELAY);
    set_bit(SDA);
    __delay_us(I2C_DELAY);
}
```

```
// Initiating a stop condition with MSP430:
void stop_cond(void)
{
    clear_bit(SDA);
    _delay_cycles(I2C_DELAY);
    set_bit(SCL);
    _delay_cycles(I2C_DELAY);
    set_bit(SDA);
    _delay_cycles(I2C_DELAY);
}
```

Reading and Writing Bits

Masters and slaves always read bit values when SCL is high, so it's the job of the writers to make sure SDA has a stable value for the entire duration of this state. To ensure this, the writer typically changes the SDA line immediately after SCL goes low so there's plenty of time for the signal to stabilize. We can observe this in Figure 6, where SDA is pulled low immediately after the second SCL pulse so that its value is no doubt zero for the third clock pulse.

Let's see the code for writing a bit:

```
// Writing a bit with a PIC:
void write_bit(uint8_t b)
{
    if(b > 0) set_bit(SDA);
    else clear_bit(SDA);

    __delay_us(I2C_DELAY);
    set_bit(SCL);
    __delay_us(I2C_DELAY);
    clear_bit(SCL);
}
```

```
// Writing a bit with MSP430:
void write_bit(uint8_t b)
{
    if(b > 0) set_bit(SDA);
    else clear_bit(SDA);

    _delay_cycles(I2C_DELAY);
    set_bit(SCL);
    _delay_cycles(I2C_DELAY);
    clear_bit(SCL);
}
```


Let's continue with reading a bit:


```
// Reading a bit with a PIC:
uint8_t read_bit(void)
{
    uint8_t b;

    set_bit(SDA); // recall that we're setting SDA as an input here
    __delay_us(I2C_DELAY);
    set_bit(SCL);
    __delay_us(I2C_DELAY);

    b = (PORTC >> SDA) & 0x01; // read bit

    clear_bit(SCL);

    return b;
}
```



```
// Reading a bit with MSP430:
uint8_t _read_bit(void)
{
    uint8_t b;

    set_bit(SDA);
    _delay_cycles(I2C_DELAY);
    set_bit(SCL);
    _delay_cycles(I2C_DELAY);

    // SDA already set as input
    if((P2IN & SDA) > 0) b = 1;
    else b = 0;

    clear_bit(SCL);

    return b;
}
```

Allow me now to impose a stop condition! If you're having a hard time following at this point: stop here, follow the functions with your finger, compare with Figure 6, and repeat. Make sure you're up to speed before moving on, for the waves only get bigger.

Acknowledging or Not

But first, an easy discussion about acknowledging (ACKing). Recall that after every byte of data sent the other device with either ACK to say, 'All is good, carry on,' or NACK to express the opposite. A NACK can also occur if the master attempts to talk to a slave using an incorrect address, or if the device is simply non-responsive. Finally, a master will use a NACK to signal a slave to stop sending it data.

Writing a Byte

We're now encroaching on full I2C transactions! Besides calling *write_bit()* eight times we can lump in other function calls that frequently precede or proceed sending a byte. We'll include the options to precede with a start condition, proceed with a stop condition, and check whether the recipient ACKed.

But first, another discussion: [endianness](https://en.wikipedia.org/wiki/Endianness) (<https://en.wikipedia.org/wiki/Endianness>). Silly word, eh? It refers to the order of bits and bytes sent. I2C uses big-endian bit order, so the most-significant bits are sent before the least-significant. It may have never crossed your mind, but the idea of endianness may save you in the future (e.g. SMBus, an I2C protocol, uses little-endian byte order, so least-significant bytes are sent first). Anyhoo, back to writing bytes:

```
// Writing a byte with a PIC:
bool write_byte(uint8_t B,
                bool start,
                bool stop)
{
    uint8_t ack = 0;

    if(start) start_cond();

    for(uint8_t i=0; i<8; i++)
    {
        write_bit(B & 0x80);    // write the most-significant bit
        B < 0) return false;
    else return true;
}
```

```
// Writing a byte with MSP430:
bool write_byte(uint8_t B,
                bool start,
                bool stop)
{
    uint8_t ack = 0;

    if(start) start_cond();

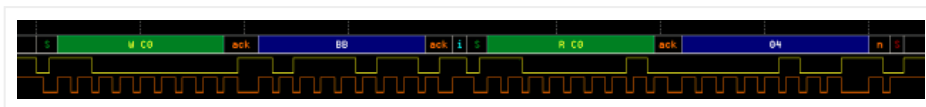
    volatile uint8_t i;
    for(i=0; i<8; i++)
    {
        write_bit(B & 0x80);
        B < 0)
    {
        stop_cond();
        return false;
    }

    if(stop) stop_cond();

    return true;
}
```

Reading a Byte

Hark back to Figure 3 above, and take a look at the example reading transaction in Figure 7:



— Figure 7: example I2C read transaction. The orange trace represents SCL, the yellow SDA, and the I2C interpretation rests on top.

The second blue byte in Figure 7 is the only byte read by the master, which is why I reference Figure 3 as well to disambiguate the distinction. There isn't quite as much going on compared with writing a byte, but we'll borrow the stop condition option. And instead of checking for an ACK, we'll include the option to ACK ('send me more data!') or not ('no more, please!'). Oh, and of course we'll be reading bits instead of writing:

```
// Reading a byte with a PIC:
uint8_t read_byte(bool ack,
                  bool stop)
{
    uint8_t B = 0;

    for(uint8_t i=0; i<8; i++)
    {
        B <<= 1;
        B |= read_bit();
    }

    if(ack) write_bit(0);
    else write_bit(1);

    if(stop) stop_cond();

    return B;
}
```

```
// Reading a byte with MSP430:
uint8_t read_byte(bool ack,
                  bool stop)
{
    uint8_t B = 0;

    volatile uint8_t i;
    for(i=0; i<8; i++)
    {
        B <<= 1;
        B |= read_bit();
    }

    if(ack) write_bit(0);
    else write_bit(1);

    if(stop) stop_cond();

    return B;
}
```

Reading and Writing I2C Data

Woohoo! We've made it to creating actual I2C transactions! We'll begin, like always, with the simplest: *i2c_read_byte()* and *i2c_write_byte()*.

I2C Read Byte and Write Byte

While the latter has occasional utility, the former is really only useful in debugging (unless the slave has only a single function that executes every time that it's addressed). Nonetheless, neither are too difficult to concoct or analyze. Beginning with writing a byte:

```
// I2C writing a byte with a PIC:
bool i2c_write_byte(uint8_t address,
                    uint8_t data)
{
    if(write_byte(address << 1, true, false)) // start, send address
    {
        if(write_byte(data, false, true)) return true; // send data
    }

    return false;
}
```

```
// I2C writing a byte with MSP430:
bool i2c_write_byte(uint8_t address,
                    uint8_t data)
{
    if(write_byte(address << 1, true, false))
    {
        if(write_byte(data, false, true))
        {
            return true;
        }
    }

    stop_cond();
    return false;
}
```

And reading a byte:

```
// I2C reading a byte with a PIC:
uint8_t i2c_read_byte(uint8_t address)
{
    if(write_byte((address << 1) | 0x01, true, false))    // start,
    {
        return read_byte(false, true);
    }

    return 0;    // return zero if NACKed
}
```

```
// I2C reading a byte with MSP430:
uint8_t i2c_read_byte(uint8_t address)
{
    if(write_byte((address << 1) | 0x01, true, false))
    {
        return read_byte(false, true);
    }

    stop_cond();
    return 0xff;
}
```

I2C Read Byte Data and Write Byte Data

Finally, we've arrived at the summit. These two functions will be the powerhouses of your I2C transactions. The extra 'data' means we're sending an extra byte to the slave to tell it which register we're interested in. For I2C writing a byte of data, this simply translates to sending one more byte than with *i2c_write_byte()*:

```
// I2C writing a byte of data with a PIC:
bool i2c_write_byte_data(uint8_t address,
                        uint8_t reg,
                        uint8_t data)
{
    if(write_byte(address << 1, true, false)) // start, send address
    {
        if(write_byte(reg, false, false)) // send desired register
        {
            if(write_byte(data, false, true)) return true; // send data
        }
    }

    return false;
}
```

```
// I2C writing a byte of data with MSP430:
bool i2c_write_byte_data(uint8_t address,
                        uint8_t reg,
                        uint8_t data)
{
    if(write_byte(address << 1, true, false))
    {
        if(write_byte(reg, false, false))
        {
            if(write_byte(data, false, true))
            {
                return true;
            }
        }
    }

    stop_cond();
    return false;
}
```

I2C reading a byte of data is more like extending *i2c_write_byte()* with *i2c_read_byte()* immediately afterward. Now we could do just this, but I'm going to avoid using these previous functions in case we want to delete them (or just one) later:

```
// I2C reading a byte of data with a PIC:
uint8_t i2c_read_byte_data(uint8_t address,
                           uint8_t register)
{
    if(write_byte(address << 1, true, false)) // start, send addr
    {
        if(write_byte(register, false, false)) // send desired re
        {
            if(write_byte((address << 1) | 0x01, true, false)) //
            {
                return read_byte(false, true); // read data
            }
        }
    }

    return 0; // return zero if NACKed
}
```

```
// I2C reading a byte of data with MSP430:
uint8_t i2c_read_byte_data(uint8_t address,
                           uint8_t reg)
{
    if(write_byte(address << 1, true, false))
    {
        if(write_byte(reg, false, false))
        {
            if(write_byte(address << 1, true, false))
            {
                return read_byte(false, true);
            }
        }
    }

    stop_cond();
    return 0xff;
}
```

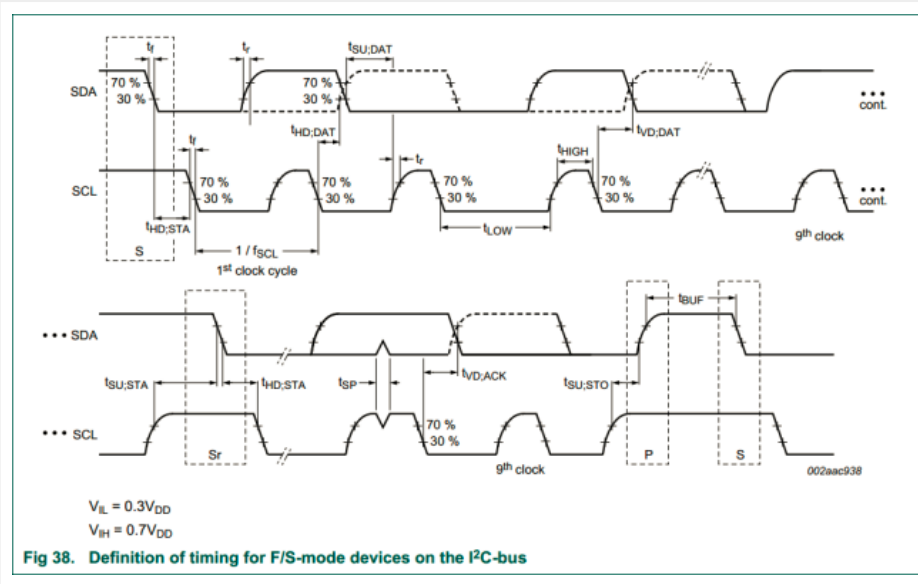
Timing

We've kicked this can down the road since the beginning, but now it's time to talk timing. Per the latest [I2C specification](http://www.nxp.com/documents/user_manual/UM10204.pdf) (http://www.nxp.com/documents/user_manual/UM10204.pdf) (April 2014), I2C bit rates range from 100 thousand bits per second (kbps) in Normal Mode to 400kbps

in Fast Mode and up (3.4Mbps!). Remember, since I2C is synchronous we aren't coerced into transmitting data at rigid baud rates. (However, some off-the-shelf devices include timeout functions while waiting for data so we're not quite free to choose arbitrarily-low data rates whilst interfacing with these devices. Check the datasheet!) So what's stopping us from transmitting at 0.05bps (3 bits per minute)? Well, let's take a look at the I2C specification:

Symbol	Parameter	Conditions	Standard-mode		Fast-mode		Unit
			Min	Max	Min	Max	
f _{SCL}	SCL clock frequency		0	100	0	400	kHz
t _{HD,STA}	hold time (repeated) START condition	After this period, the first clock pulse is generated.	4.0	-	0.6	-	µs
t _{LOW}	LOW period of the SCL clock		4.7	-	1.3	-	µs
t _{HIGH}	HIGH period of the SCL clock		4.0	-	0.6	-	µs
t _{SU,STA}	set-up time for a repeated START condition		4.7	-	0.6	-	µs
t _{HD,DAT}	data hold time ^[2]	CBUS compatible masters (see Remark in Section 4.1)	5.0	-	-	-	µs
		I ² C-bus devices	0 ^[3]	- ^[4]	0 ^[3]	- ^[4]	µs
t _{SU,DAT}	data set-up time		250	-	100 ^[5]	-	ns
t _r	rise time of both SDA and SCL signals		-	1000	20	300	ns
t _f	fall time of both SDA and SCL signals ^{[3][6][7][8]}		-	300	20 × (V _{DD} / 5.5 V)	300	ns
t _{SU,STO}	set-up time for STOP condition		4.0	-	0.6	-	µs
t _{BUF}	bus free time between a STOP and START condition		4.7	-	1.3	-	µs
C _D	capacitive load for each bus line ^[10]		-	400	-	400	pF
t _{VD,DAT}	data valid time ^[11]		-	3.45 ^[4]	-	0.9 ^[4]	µs
t _{VD,ACK}	data valid acknowledge time ^[12]		-	3.45 ^[4]	-	0.9 ^[4]	µs
V _{NL}	noise margin at the LOW level	for each connected device (including hysteresis)	0.1V _{DD}	-	0.1V _{DD}	-	V
V _{NH}	noise margin at the HIGH level	for each connected device (including hysteresis)	0.2V _{DD}	-	0.2V _{DD}	-	V

— Figure 8: I2C timing specification table (adapted, courtesy of NXP).



— Figure 9: I2C timing specification figure (courtesy of NXP).

Notice there are a lot of hyphens in the 'Max' columns — this means there is no limit for that parameter. The only maximum times we need to be wary of are rise/fall times and data valid times. The formers are dictated by our microcontroller's slew rate (typically less than a few hundred nanoseconds) and the bus capacitance (typically dominated by trace/wire length); the latter can be

configured in software — since we change the SDA value immediately after a falling SCL edge these values are minimal (a few microcontroller instruction cycles). In short: we're good. We can transmit as slow as we want!

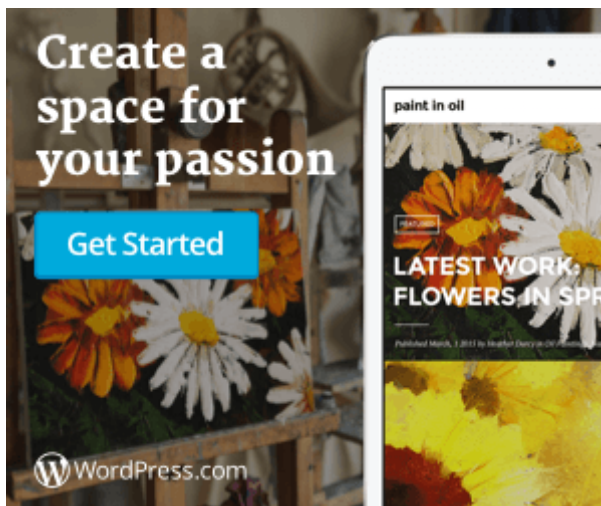
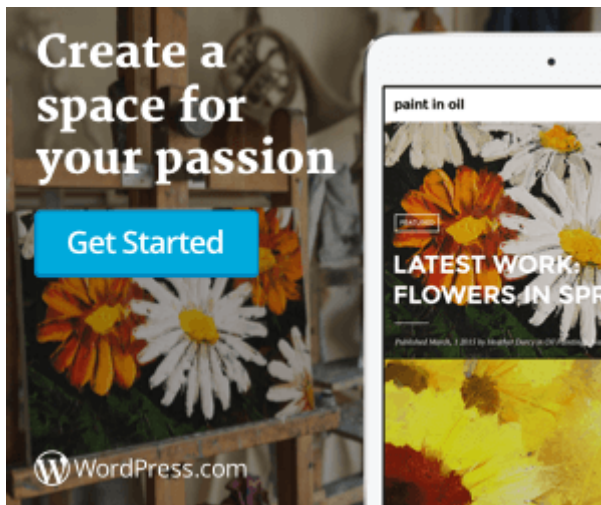
Now for the upper end of data rates we must be wary of minimum on-times. Scroll between Figure 8 and Figure 9 and we can see that the maximum delay time we encounter is 5µs (data hold time). For simplicity we've set our *I2C_DELAY* variable to this maximum delay time, though a few microseconds per transaction can be saved by defining several delay values. (But beware! Most delay functions take integers as arguments, so sending '4.7' will be truncated as 4.)

Wrapping Up

Wait, what if we want to send TWO bytes of data? Or read two? Well, now it's your turn! Look at Figures 2 and 4 and see if you can figure it out yourself.

In the next post in this series we'll look at implementing these functions on a real device and get into troubleshooting techniques. But for now I'll let you digest and celebrate — think of how far you've come! And how free you are! No more relying on dedicated hardware! And once you've done the grunt work you can reproduce this code forever! Your grandchildren can inherit your bit-banged I2C libraries! Joy!

For more tutorials and explanations of I2C, check out [Sparkfun](https://learn.sparkfun.com/tutorials/i2c) (<https://learn.sparkfun.com/tutorials/i2c>) and [tronixstuff](http://tronixstuff.com/2010/10/20/tutorial-arduino-and-the-i2c-bus/) (<http://tronixstuff.com/2010/10/20/tutorial-arduino-and-the-i2c-bus/>). For more advanced characteristics and operation, including some gnarly troubleshooting techniques, see the [I2C Bus](http://www.i2c-bus.org/) (<http://www.i2c-bus.org/>) homepage.



Share this:



Loading...

POSTED IN LEARNING | TAGGED I2C, MSP430, PIC | LEAVE A COMMENT |

← Milwaukee Maker Faire!

Data from nRF24L01+ to Pure Data →

Leave a Reply

Enter your comment here...

