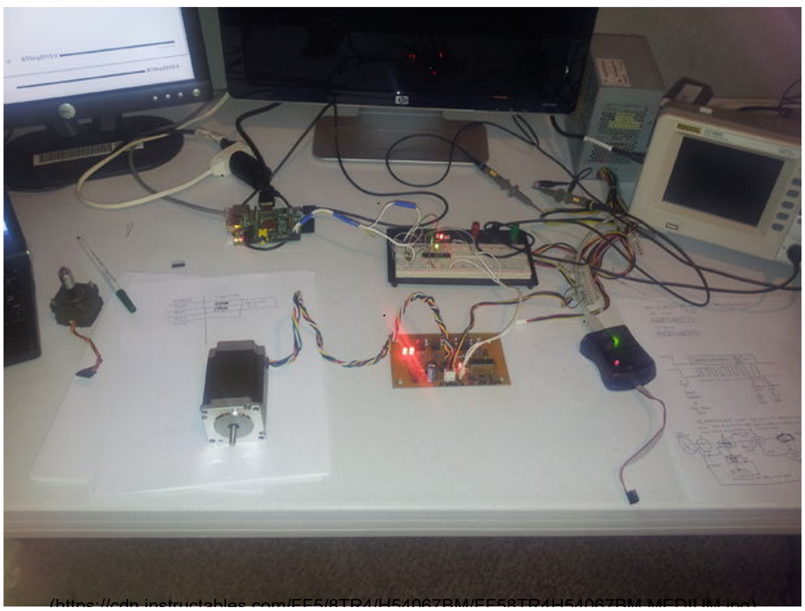
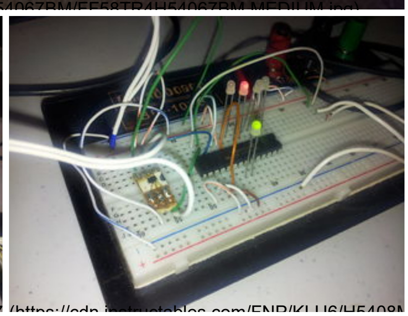
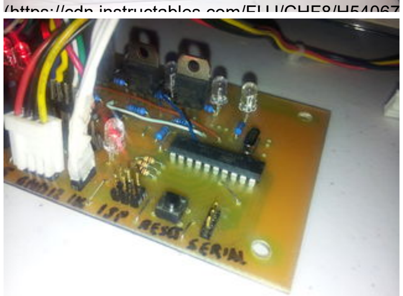
ATTiny USI I2C Introduction - a Powerful, Fast, and Convenient Communication Interface for Your ATTiny Projects!

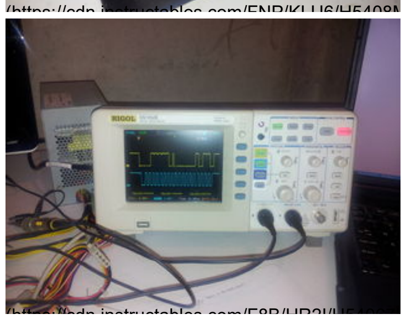
by CalcProgrammer1 (/member/CalcProgrammer1/) in raspberry-pi (/technology/raspberry-pi/)


(https://cdn.instructables.com/FF5/8TR4/H54067BM/FF58TR4H54067BM.MEDIUM.jpg)


(https://cdn.instructables.com/FLU/CHF8/H54067


(https://cdn.instructables.com/FNP/KLU6/H5408N


(https://cdn.instructables.com/FOC/O0FU/H5408


(https://cdn.instructables.com/F8P/HR2U/H5

About This Instructable

CalcProgrammer1 (/member/CalcProgramm
(https://plus.google.com/+Adaml
Follow 186
(/member/CalcProgrammer1/)

Bio: I finally graduated from Missouri University of Science and Technology (Missouri S&T, formerly University of Missouri Rolla) with a computer engineering degree. Originally from ...
More » (/member/CalcProgrammer1/)

ATTiny USI I2C Introduction - a Powerful, Fast, and Convenient Communication Interface for Your ATTiny Projects!

by CalcProgrammer1 (/member/CalcProgrammer1/) in raspberry-pi (/technology/raspberry-pi/)

I2C, it's a standard that's been around for around 20 years and has found uses in nearly every corner of the electronics universe. It's an incredibly useful technology for us microcontroller hobbyists but can seem daunting for new users. This tutorial will solve that problem, first by reviewing what I2C is and how it works, then by going in-depth on how to implement I2C on Atmel's ATTiny USI (Universal Serial Interface) hardware.

I2C is commonly used in GPIO expanders, EEPROM/Flash memory chips, temperature sensors, real-time clocks, LED drivers, and tons of other components. If you spend much time looking for new, cool parts you'll probably wind up with several I2C parts. Fortunately it is a protocol that is available on most microcontrollers, though it is a bit more complex than others. Learning it is tough at first, but once you know I2C it is a powerful tool.

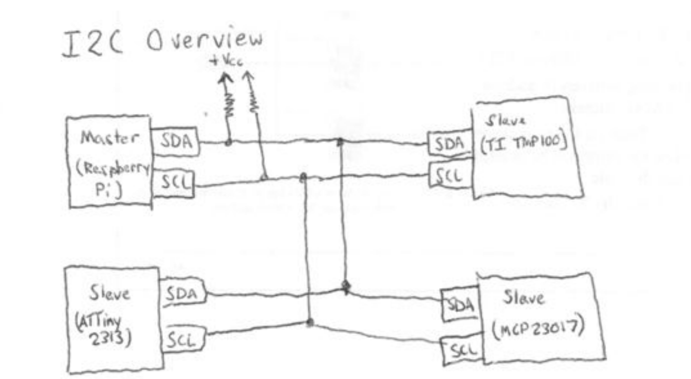
I2C Tools of Interest:
Before you dig too deep into I2C communications, you'll want to have some things on hand that will make your learning experience easier.

1. Various I2C-compatible parts - Anything goes, as long as it's I2C. If you're writing a master driver you need some things to talk to. I like Texas Instruments' TMP100 temperature sensor as it's cheap (free if you sample) and has a simple protocol (just send an I2C read command to get temp values). I more recently purchased some Microchip MCP23017 GPIO expanders which give you 16 bits of additional GPIO over the I2C bus.

2. Something that has a working I2C master - You'll want something to test/compare against if possible. An Arduino with the Wire library will work, but more recently I prefer my Raspberry Pi with the Linux i2ctools package. i2cdetect, i2cset, i2cget, and i2cdump are invaluable when writing code, especially slave-mode code.

3. Oscilloscope. I know this is a big one, but if you can work with one (either own one, borrow one, or go to a lab where you can use one) it's a super amazing help. I2C uses two wires, so a two-channel scope works great. I used my Rigol DS1052E (100Mhz modded) and it helped a TON. Of course, I did most of the work with it and am telling you what I learned, so hopefully it'll be easier for you.

Step 1: What Is I2C - 1



(https://cdn.instructables.com/FV6/6UZI/H54067EN/FV66UZIH54067EN.MEDIUM.jpg)

I2C (Inter-Integrated Circuit bus), originally developed by Phillips (now NXP Semiconductor) and also commonly known as TWI (Two Wire Interface) by Atmel and other companies who don't want to get into trademark issues, is a **two-wire synchronous serial bus**. Let's take a look at what each of those words means:

Two wire - This one's easy, I2C uses two wires (in addition to ground, of course!) They're called **SDA** (serial data) and **SCL** (serial clock). These are wired in an **open-drain** configuration, which means that the outputs of all connected devices cannot directly output a logic-level 1 (high) and instead can only pull low (connect to ground, outputting a 0). To make the line go high, all devices release their pull on the line and a **pull-up resistor** between the line and the positive rail pulls the voltage up. A good pull-up resistor is **1-10K ohms**, low enough that the signal can be seen as a high level by all devices but high enough that it can easily be shorted out (pulled down) and not cause damage or significant power usage. There is one pull-up resistor on SDA and one on SCL.

Synchronous - This means that data transfer is synchronized via a **clock signal** that is present to all connected devices. This is generated by the master. To contrast, an asynchronous serial system does not have a clock signal. Instead, it uses a pre-determined time-base, or baud rate. An example of asynchronous serial is RS-232 (the common serial port on many computers).

Serial - Data transferred serially means that one single bit is transferred at a time over a single wire. To contrast, parallel data transfer has multiple wires, each carrying one bit, which are all sampled at once to transfer multiple bits in parallel.

Bus - A bus is a system which allows many devices to communicate to each other over a **single set of wires**. While it may be called a bus, USB is not a true bus at the hardware level, as connecting multiple devices requires a hub. A bus such as I2C allows new devices to be added simply by attaching their SDA and SCL connections to the existing line. Busses (I2C, USB, PCI, etc) all use an **addressing system**, in which each device has a **unique address**. An address in this case is simply a binary number, and all messages to that device must be sent with that address.

Step 2: What Is I2C - 2

(https://cdn.instructables.com/FV1/DSA5/H54067F8/FV1DSA5H54067F8.MEDIUM.jpg)

On an I2C bus, there are **masters** and there are **slaves**. A master initiates a connection while a slave must wait for a master to address it before it sends or receives anything. I2C has **multi-master** capability, which means that more than one master may exist, and if two masters attempt a transmission at the same time, they must perform **arbitration** to correct the problem. This tutorial will not cover multi-master configurations, but it should be noted that they do exist.

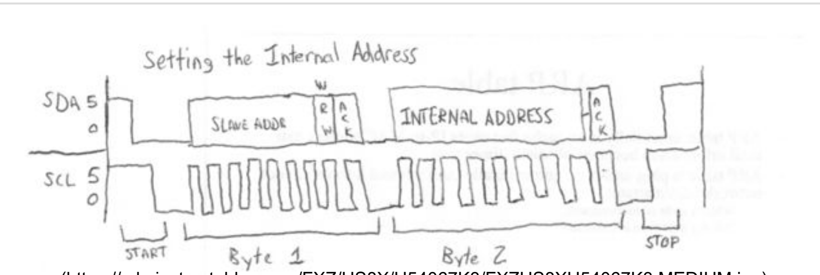
A master may either request to **send** or **receive** data from a slave. During a **send**, the master writes data to the bus and the slave reads the data off the bus and stores it in its memory. During a **receive**, the master reads the bus for data sent out by the slave. In both situations, the **master provides the clock signal** on SCK.

At the end of every byte (which is 8 bits) transmitted on the I2C bus, the receiving device must provide an **acknowledgement** (ACK). The only time in which this does not happen is when the master is receiving data from the slave, in which it ends the transmission with a **not-acknowledgement** (NACK or NAK) indicating that the slave should stop sending data. An ACK is represented by a low (pulled-down or 0) state while a NACK is represented by a high (not pulled-down or 1) state. Since the default state of the bus is high, an ACK is a confirmation that the other device is present and has successfully processed the transmission.

In addition to ACK's and NACK's, I2C has two additional framing conditions known as a **start condition** and a **stop condition**. A start condition is transmitted by the master to indicate the start of a transmission. During a start transition, the SDA line first transitions from high to low and then, after a noticeable amount of time, the SCL does the same. A stop condition, which is issued by the master at the end of a transmission, is the reverse. First the SCL line goes from low to high, then the SDA does the same. Note that the SDA and SCL lines both are high when the bus is inactive.

The first byte in an I2C transmission is the address byte. This is sent by the master and is used to determine what slave to talk to and whether to perform a send or receive (also known as write and read, respectively). A slave address is 7 bits long, and there are several reserved addresses. One such reserved address is 0x00, which is often considered a global write (write to all slaves). You usually configure the slave device's address by tying address select pins high or low, though on a microcontroller you set the address programmatically as we will do on the ATTiny2313. The least-significant bit of the address byte is the Read/Write bit which indicates whether to perform a read or write. If one, the operation is a read, if zero a write.

Step 3: What Is I2C - 3



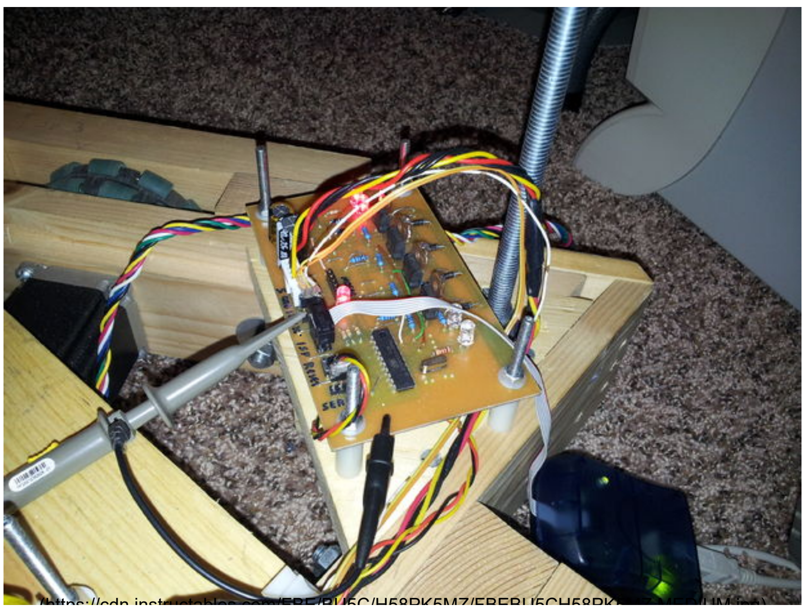
(https://cdn.instructables.com/FXZ/HS8Y/H54067K3/FXZHS8YH54067K3.MEDIUM.jpg)

That basically covers the I2C protocol itself, in that a master can initiate a read or a write, and transfer continues until the master sends a stop condition. When the master is reading from a slave, it will issue a NACK instead of an ACK on the last byte, before the stop condition, to indicate that it is done receiving. From here, with a proper implementation thus far, you can communicate to all the devices you want. However, there is one extra thing that I want to point out, as it is used quite often.

On some I2C devices (or should I say most, it's very common), the access protocol is set up as a **register bank**. To read or write from these registers, you must first write an **internal address** which is the address of the register you wish to read or write. After writing an internal address, you may read or write multiple bytes and the internal address will increment with each byte. This is the **preferred protocol** for almost all I2C memory devices as well as most sensors and I/O expanders. While it is possible to have a protocol that doesn't follow the register bank protocol, the vast majority of devices do and many I2C tools are built around it. As such, it is worth pointing out. It is also the protocol that I will implement on the ATTiny2313.

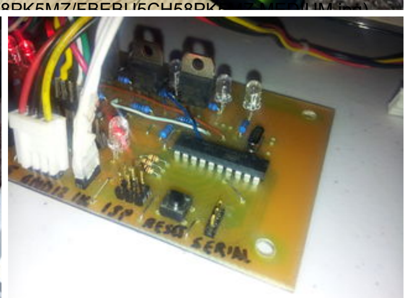
As mentioned before, before reading or writing any register you must send the **device internal address**, which is done by performing a write operation of one byte, which contains the internal address. For write operations, the transmission may continue with data values, the first of which will be stored in the desired address and any additional bytes will increment upwards by one each time. For reads, the master will send a stop condition, then start a new transmission for reading. This is because **you can not have both a write and a read in the same transmission**. In some cases, a repeated start may be sent instead of a stop then start. A **repeated start** is a high-to-low transition on SDA while SCL is high.

Step 4: ATTiny USI I2C Code Implementation - Overview



(https://cdn.instructables.com/FRF/RU5C/H58PK5MZ/FRFRU5CH58PK...

(https://cdn.instructables.com/F4K/I5H1/H58P2...



(https://cdn.instructables.com/FOG/O0FU/H5408...

At this point in the tutorial, you should be at least basically familiar with the I2C protocol. Now I'm going to go into detail on the actual I2C protocol implementation for the ATTiny USI hardware.

To preface this, the board shown on this step's main picture is a custom unipolar stepper motor controller that I designed as part of my senior project this Spring. The board is capable of driving a single unipolar stepper motor with PWM, variable speed, and three different stepping modes (single-stepping, power stepping, and half-stepping). It also operates in bursts, as the controller will only run the motor for the given number of steps. For continuous operation, the stepping counter must be reloaded by whatever is driving the board before it reaches zero. None of this is important for the tutorial.

The important part here is that the robot these boards power has three wheels (omnidirectional wheels, arranged in a triangular pattern). I wanted to build three identical boards but only use a single RS-232 serial interface from the robot's main computer (a laptop) to control all three. The idea I came up with was to use the serial port for the computer interface and the I2C bus to connect all three boards. In this setup, the board connected to the PC takes on the master role in addition to being a slave node. The PC then sends I2C formatted messages onto the bus for the three boards to operate upon.

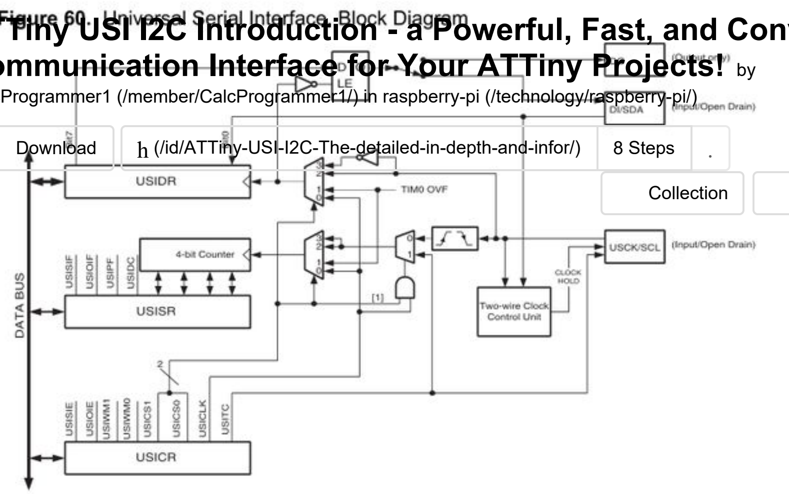
For this task, my boards would have to support **both** I2C modes, being able to operate as both a **slave** and a **master** depending on serial port operations. Knowing very little about the USI hardware and only slightly more about the I2C protocol in general, I set out to master the I2C protocol, to make it my slave and command it to do my data transmissions. And that I did, and it worked well for the project.

All up until I got my Raspberry Pi at least, because when I finally got around to playing with the Pi, I tried hooking up my I2C motor boards to its I2C port in an attempt to have a Pi-powered robot. Unfortunately, no matter what commands I sent, the Pi could not make communication happen. Since I had never validated my protocol beyond my own master code talking to my own slave code, I figured I wasn't implementing the protocol just right, and sat down to make it all work correctly. That I did, with the new code much more streamlined, organized, and understandable (comments galore for anyone who wants to learn!). Since my journey into the world of I2C was rough, I decided to post here for all to see, and to go into as much detail as I could to make I2C's functionality clear.

In the next few steps I will talk about the USI hardware and how it works as both a master and a slave. I have also attached my USI code files. I want people to have a good USI implementation and I also want them to read how it works, knowing exactly what's going on is crucial when dealing with a complex, low-level system, so I've commented my files thoroughly.

Step 5: ATTiny USI I2C Code Implementation - USI Hardware

Figure 60. Universal Serial Interface, Block Diagram



(https://cdn.instructables.com/FB3/Y33M/H5035SB5/FB3Y33MH5035SB5.MEDIUM.jpg)

So, before we look at code, let's look at the **datasheet**. Specifically I'm looking at the ATTiny2313 datasheet as that's the chip I'm using, but the same USI hardware can be found in many different ATTiny models. Note that the output pins may be different between chips, but otherwise the hardware works the same way and has the same registers.

The USI hardware has three **pins**:

DO - Data Output, used for three-wire (SPI) communication mode only

DI/SDA - Data Input/Serial Data, used as SDA in I2C configuration

USCK/SCL - Clock, used as SCK in I2C configuration

In addition, the USI hardware has three **registers**:

USIDR - USI Data Shift Register - Shifts data in and out of the USI hardware

USISR - USI Status Register - Has state flags and **4-bit counter** (more on this below)

USICR - USI Control Register - Has interrupt enables, clock modes, and software clock strobe functions

4-Bit Counter

The 4-bit counter occupies the lower 4 bits of USISR and is used to time the overflow interrupts when operating in slave mode as well as to help generate SCK clock pulses in master mode. Being a 4-bit counter, it counts upwards from 0 to 15 before overflowing. Upon overflowing, it can trigger an interrupt (USI_OVERFLOW_vect, enabled by a bit in USICR). This is used for the USI slave state table to keep track of transmission as it switches between transmission states (more on this in the slave code section).

When acting as a master, the 4-bit counter is used along with the clock strobe bit in USICR to generate the SCK clock. You set the counter to overflow in the number of clock pulses you wish to generate (generally 8 or 1, with 8 being a data transmission and 1 being an ACK/NACK transmission). Then, you loop until the counter overflows, continuously setting the clock strobe bit and performing a delay wait. More on this can be seen in the master code section.

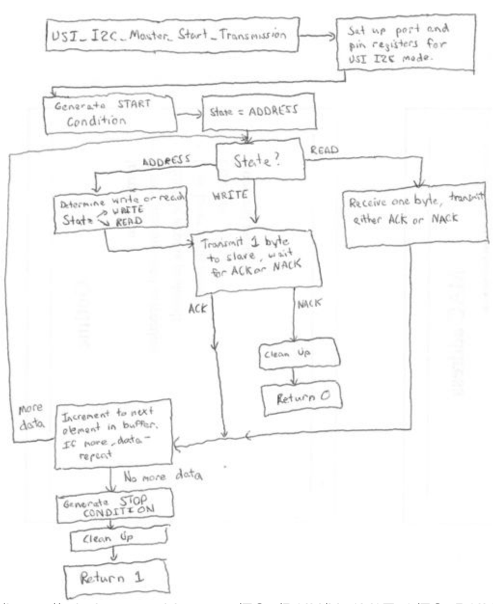
Two Wire Clock Control Unit (Start Condition Detector)

The TWI Clock Control Unit is a module in the USI that monitors the SCK line for start and stop conditions. Its primary purpose is the **start condition detector**, which, when enabled, generates a USI_START_vect interrupt whenever it detects a valid start condition. This interrupt handler is the starting point for slave-mode USI I2C transmission handling, and must set up the 4-bit counter to overflow after the address transmission has occurred. From there on, the overflow interrupt manages the rest of that I2C message and resets the start condition detector for the next message.

Read the Datasheet

I won't go into detail on each of the bits in each of these registers, but if you're looking at writing some USI code it is essential that you **read these sections of the datasheet**. I recommend reading the **whole** Universal Serial Interface - USI section (pages 142-150 of the ATTiny2313 full datasheet). This will give you all the information you'll need in addition to what I've pointed out here.

Step 6: ATTiny USI I2C Code Implementation - USI I2C Master



(https://cdn.instructables.com/FQ7/P0IN/H58V8T53/FQ7P0IN

The USI I2C Master (usi_i2c_master.c/h) library provides I2C **master mode** capabilities using the USI hardware. There are two important functions that the user should be familiar with. The first is the initialization function which sets up the SDA/SCL pins and USI hardware, and the other is the **transmission** function, which performs either a read or a write operation on an I2C message, returning 1 (true) if successful and 0 (false) if an error (received a NACK) occurred. A third function, the **transfer** function, is used by the transceiver function to send and receive data. The transfer function should not be used outside of the usi_i2c_master library.

The transmission function takes **two arguments**. The first is a pointer to a **data buffer** in which to send or receive data from. It is assumed that the first byte in this buffer is the ADDRESS+R/W byte (upper 7 bits address, LSB is R/W). This byte, by protocol, is always transmitted and never received. The rest of the buffer will either be sent out or filled by received data according to the R/W bit (1 for read, 0 for write). The second argument is the total size of the buffer (including the address byte).

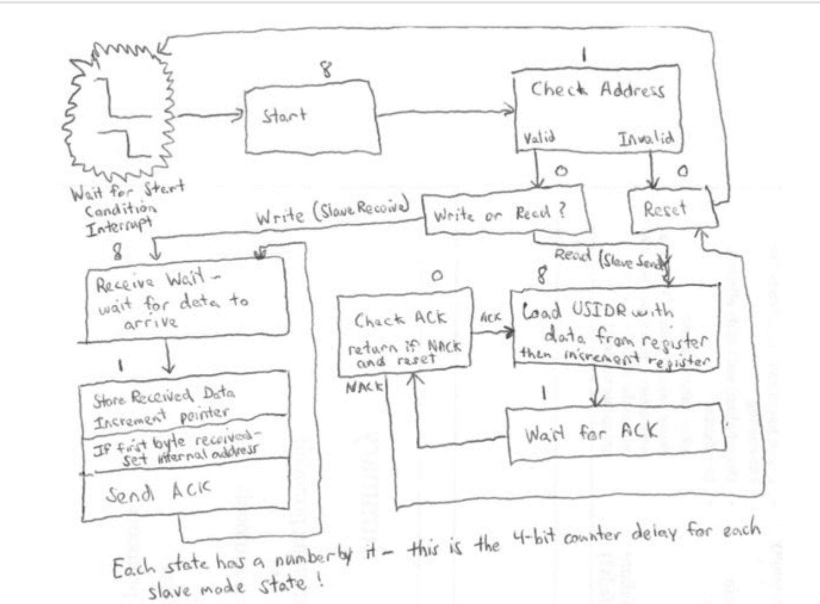
Here's a brief example. Let's say we want to output the value **0x70** to the internal address **0x12** of the device with address **0x40**. First, we must create a buffer to store our transmission:

```
char i2c_transmit_buffer[3];
char i2c_transmit_buffer_len = 3;

i2c_transmit_buffer[0] = (0x40 << 1) | 0  //Or'ing with 0 is unnecessary, but for clarity's sake this sets the R/W bit for a write.

i2c_transmit_buffer[1] = 0x12;  //Internal address

i2c_transmit_buffer[2] = 0x70;  //Value to write

//Transmit the I2C message
USI_I2C_Master_Start_Transmission(i2c_transmit_buffer, i2c_transmit_buffer_size);
```

There you have it, message transmission complete! That was easy! If you want more information on the inner workings of the USI_I2C_Master code, just take a look inside the **usi_i2c_master.c** file where I've commented the states, transfer function, and other interesting sections. I've made use of one-line #define macros to make it more clear what each line's purpose is.

In the next step, I will introduce the slave-mode code, which is significantly more complex but also easy to use from an end-user standpoint. I've taken a different approach on implementing the slave code that I haven't seen in any other tutorials, it's interesting and useful!

Step 7: ATTiny USI I2C Code Implementation - USI I2C Slave



(https://cdn.instructables.com/FAC/DOKB/H4VOF0TL/FACDOKBH4VOF0TL.MEDIUM.jpg)

Unlike the master code, the USI I2C Slave code (usi_i2c_slave.c/h) is implemented **almost entirely** using the USI interrupts. As mentioned earlier, the USI module has two interrupts, one generated upon detection of a START condition and the other based on the overflow of the 4-bit counter. The counter is crucial for the slave code to work correctly, and was not explained very well in tutorials and code I read. In the flowchart image I have noted numbers for each state in the logic. These numbers (8's, 1's, and 0's) are counter count values indicating how many ticks the counter should count before transitioning to the next state. As the counter is clocked using SCL clocks, these values tell how many SCL clock pulses must occur before the next state. In general, something that waits 8 clock pulses is waiting on transmission/reception of a data byte while something that waits 1 clock pulse is waiting on transmission/reception of an ACK/NACK. A few things wait 0 clocks, meaning that they instantaneously continue to the next state or are an expanded part of the previous state (in the case of Write or Read?).

So, as an end-user, you probably are more interested in how to interface the library to your own code! This is **easy**, and here's why. I've done away with the receive/transmit buffers that were used in other USI I2C implementations (mainly those based on AVR312 app-note) and instead implemented the **register bank protocol** described at the beginning of this tutorial. The bank is stored as an array of **pointers**, not data values, so you must **attach local variables in your code** to **memory addresses in the I2C register bank** by setting the pointers to point to your variables. This means that your main-line code doesn't ever have to poll I2C buffers or handle data arrivals, the values are updated instantaneously whenever they arrive. It also allows program variables to be polled at any time by the I2C interface without affecting the main-line code (other than the delay due to interrupt). It's a pretty neat system. Let's again take a brief example.

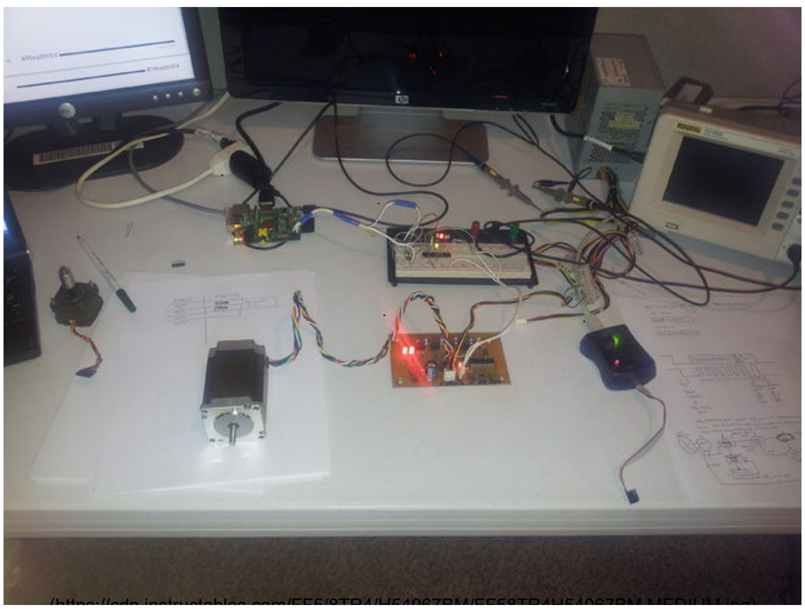
For example, let's say we have a very basic software PWM generator that is driving an LED. We want to be able to change the PWM value (a **16-bit value**, just for the sake of learning about pointers) without making the main loop complicated. With the magic of asynchronous I2C slave, we can do just that!

```
#include "usi_i2c_slave.h"

//Define a reference to the I2C slave register bank pointer array
extern char* USI_Slave_register_buffer[];

int main()
{
//Create 16-bit PWM value
unsigned int pwm_val = 0;

//Assign the pwm value low byte to I2C internal address 0x00
//Assign the pwm value high byte to I2C internal address 0x01
USI_Slave_register_buffer[0] = (unsigned char*)&pwm_val;
USI_Slave_register_buffer[1] = (unsigned char*)(&pwm_val)+1;

//Initialize I2C slave with slave device address 0x40
USI_I2C_Init(0x40);

//Set up pin A0 as output for LED (we'll assume that whatever chip we're on has
pin A0 available)
DDRA |= 0x01;

while(1)
```

```
{
PORTA |= 0x01; //Turn LED on
for(unsigned int i = 0; i < pwm_data; i++)
{
PORTA &= ~(0x01); //Turn LED off
}
}
}
```

And there you have it! The main loop makes **no reference** to I2C at all, but upon sending a PWM value to the 16-bit location in I2C internal address 0x00/0x01, we can totally control the PWM of the LED! For added stability (to make sure only the pointer values you're using are available and to prevent stray pointers) I suggest you **change the #define USI_SLAVE_REGISTER_COUNT to be the number of register pointers you need**, no more, no less. When an access (read or write) is attempted on a register index outside of the range 0x00 to USI_SLAVE_REGISTER_COUNT - 1, nothing is written and zero is returned.

Step 8: ATTiny USI I2C Code Implementation - Git the Code!



(https://cdn.instructables.com/FE5/8TB4/H54067BM/FE58TB4H54067BM.MEDIUM.jpg)

To get (git) the code, head to my GitHub page! This code was written as part of my Senior Project Stepper Motor Controller, so you can check out my implementation of the motor controller using my I2C drivers. I also have a decent interrupt-driven USART serial driver in there you can use as well.

https://github.com/CalcProgrammer1/Stepper-Motor-Controller/tree/master/UnipolarStepperDriver (https://github.com/CalcProgrammer1/Stepper-Motor-Controller/tree/master/UnipolarStepperDriver)

Note that the names and functionality used in the I2C drivers may vary slightly due to bug fixes and updates that I implement. If it's anything serious I'll edit the tutorial, but for now it should be very accurate.

Now that you're armed with the knowledge of I2C, you're ready to go out and talk to stuff! With I2C's bus design, you can connect a lot (theoretically up to 128, limited by address restrictions though) of devices to the network!

Also, if you haven't got one yet, get yourself a Raspberry Pi, they're awesome. if you order one today, you might have it by the end of the year, but it'll be a very merry Christmas if that's the case. As far as I2C goes it's been a great help and I can't wait to build a robot around it.

Comments

HashimSakkaf (/member/HashimSakkaf/) 2017-06-23 Reply

Hi

Thank you for this tutorial, I have few questions:

- How to specify the address of each slave?

- How to know the address of the master?

TroyB2 (/member/TroyB2/) 2015-02-25 Reply

Nice work! I've used the I2C slace code in order to be able to read logging messages from an ATtiny85 which doesn't come with a USART (after adding the appropriate pinout definitions to usi_i2c_slave.h)

I read data using python on Raspberry Pi using smbus library. It works perfectly if I use the single byte method:

bus.read_byte_data(DEVICE_ADDRESS, REG_ADDRESS)

but didn't work when I tried to read a block of bytes in one call:

data = bus.read_i2c_block_data(DEVICE_ADDRESS, REG_ADDRESS)

which returns a list of 32 bytes - the 1st is correct, followed by 31 0xff s

Not a problem, as reading bytes works fine, just wondered if there's some way to define the I2C registers in usi_i2c_slave.c to handle blocks of bytes(?)

I've tried a couple of other I2C libs and yours is the only one to work for me with ATtiny85.

ATTiny USI I2C Introduction - a Powerful, Fast, and Convenient Communication Interface for Your ATTiny Projects! by

AlecM5 (/member/AlecM5/) . TroyB2 (/member/TroyB2/) 2015-04-03 Reply

I am trying to use an ATtiny85 as well with i2c tools on a raspberry pi. I have changed the pin definitions, but for some reason it will not show up when I run **i2cdetect** on the pi. I think it might be the clock speed, what is yours set at for the microcontroller?

osunderdog (/member/osunderdog/) . AlecM5 (/member/AlecM5/) Reply 2017-02-01

I've read that the SDA & SCL lines need to be pulled up to improve odds of successful detection. I wasn't getting i2cdetect to work until I added pullups .. 470k Ohms

TroyB2 (/member/TroyB2/) . AlecM5 (/member/AlecM5/) 2015-04-03 Reply

Hi Alec

Yes, I think I had to reduce the clock speed below about 40000 - I've got it set to 20000 (the default is 100000).

The default can be configured by editing/creating a file

/etc/modprobe.d/i2c_bcm2708.conf

and adding a line

options i2c_bcm2708 baudrate=20000

Reboot/Test:

cat /sys/module/i2c_bcm2708/parameters/baudrate
20000

rokenbuzz (/member/rokenbuzz/) 2017-01-15 Reply

I'm looking at the slave code. It looks like the main code that uses the slave code could be accessing the memory buffer that the ISR writes to. Shouldn't there be a mutual exclusion mechanism in place to prevent this? As an example, in your two byte PWM sample, say there are only two possible PWM values being sent by the master, 0x11 0x11 and 0x22 0x22. Even if the code read two bytes as a 16-bit, the underlying assembly could read 0x11, get interrupted as the incoming 0x22 0x22 is written by the ISR, and then continue to read the second byte as 0x22. So it would have read 0x11 0x22. Even if the architecture did atomic reads of 16-bit, it would still be an issue if the buffer was bigger, say 32 bytes, and the main code was only halfway through reading the bytes when the ISR wakes up and changes values. I'd like to use this code, so hopefully I'm missing something.

steveo98501 (/member/steveo98501/) 2015-10-03 Reply

God I hate that these wortheless pedantic instructibles with low information density end up at the top of Googles hit list and I end up clicking on them before reading that the site is Instructibles.com

rokenbuzz (/member/rokenbuzz/) . steveo98501 (/member/steveo98501/) Reply 2017-01-13

This instructable was exactly what I needed. Google worked well and the instructable was great.

szvenigorodsky (/member/szvenigorodsky/) . steveo98501 (/member/steveo98501/)

2016-03-09 Reply

Nice job.

ATTiny USI I2C Introduction - a Powerful, Fast, and Convenient Communication Interface for Your ATTiny Projects! by

CalcProgrammer1 (/member/CalcProgrammer1/) in raspberry-pi (/technology/raspberry-pi/)

Download | h (/id/ATTiny-USI-I2C-The-detailed-in-depth-and-infor/) | 8 Steps | .

Collection | I Made it! | Favorite | Share

(https://cdn.instructables.com/FE8/IMMS/ILJWG8E2/FE8IMMSILJWG8E2.LARGE.jpg)

(https://cdn.instructables.com/FEY/ZNYC/ILJWG8E1/FEYZNYCILJWG8E1.LARGE.jpg)

WVvan (/member/WVvan/) 2015-06-16 Reply

This is most excellent. Thanks for the Instructable.

TroyB2 (/member/TroyB2/) 2015-04-11 Reply

I think the reason that the slave wouldn't send more than 1 byte at a time to the master is that the slave code always interprets the master's response as NACK even when it sends ACK. This can be fixed by changing in usi_i2c_slave.c

case USI_SLAVE_SEND_DATA_ACK_CHECK:
if(USIDR **& 0x01**)
{

So that it only test the LSB of USIDR - which is the 1bit that the master has just written to SDA for ACK (SDA low) or NACK (SDA high). I can now read multiple bytes correctly using the python smbus library.

---

wilkolunenburg (/member/wilkolunenburg/) 2014-10-01 Reply

Thanks. Very useful and good to follow.

---

mspinks (/member/mspinks/) 2013-10-06 Reply

Thank you so much. I've used an I2C module before, but never the USI in the ATTiny's. Now I finally understand what the heck that 4-bit counter is for. I haven't looked at the code entirely, but I'm guessing you use the 4-bit counter overflow flag to check when the transfer is done. I've been staring at the datasheet all day, but now it finally makes sense! I will probably write my own libraries to get a good understanding of it, but I will base it heavily off of yours.

CalcProgrammer1 (/member/CalcProgrammer1/) . mspinks (/member/mspinks/)

2013-10-11 Reply

You're welcome!

I based my driver heavily off the Atmel app note for USI I2C, even though that one worked it was really hard to understand and I wrote mine to clean it up and figure out how it all works. Definitely try writing your own if you want to really understand how it works!

ATTiny USI I2C Introduction - a Powerful, Fast, and Convenient Communication Interface for Your ATTiny Projects! by

CalcProgrammer1 (/member/CalcProgrammer1/) in raspberry-pi (/technology/raspberry-pi/)

Download | h (/id/ATTiny-USI-I2C-The-detailed-in-depth-and-infor/) | 8 Steps | Collection | I Made it! | Favorite | Share

mspinks (/member/mspinks/) . CalcProgrammer1 (/member/CalcProgrammer1/) 2013-10-12 Reply

Everyone on the inter-webs kept telling me that the AVR app note had a major flaw, but no one cared to actually tell me what it was, and instead just insisted that I download Don Blake's code. Well I finally found someone to explain it for me. If anyone else is interested: http://www.aca-vogel.de/TINYUSII2C_AVR312/APN_TINYUSI_I2C.html#mozTocId854460

It's always a good idea to try and write one. You will learn a lot! For instance, I learned that the USI code doesn't actually check that the start condition actually completed. I originally did it without the while loop at the beginning of the start vector. I didn't want my routine to use interrupts, so I didn't think it was necessary. I would only work once. Every subsequent read would only return 0x01. o_O

Eloyucu (/member/Eloyucu/) 2013-08-20 Reply

I don´t unsderstand this line:
i2c_transmit_buffer[0] = (0x40 << 1) | 0 //Or'ing with 0 is unnecessary, but for clarity's sake this sets the R/W bit for a write.
Why do you make the displacement <<1?? Aren´t you sending 0x80?? and it´s a diferent address that you wanted...
About the OR´ing with 0... if you want to put a 0 at the last bit (to write on the salve... I imagine), don´tn you need to make somethig like (address & 0xfe)??
In the contrary case, if you want to read from the slave, you need to put a 1 on the last bit, so (I think) you need make something like (address | 0x01)... or am I worng?

CalcProgrammer1 (/member/CalcProgrammer1/) . Eloyucu (/member/Eloyucu/) 2013-08-21 Reply

The i2c slave address is shifted to the left by 1. That line sets the address to 0x40 (0b1000000, i2c addresses are 7 bits long) and then sets the R/W bit to zero (write mode). The or'ing with zero is pointless, I just did it for context, to show readers that the write bit is zero and the read bit is one (where I use | 1). Since the operation already shifts left by 1 the 0 bit will never actually be one, so the | 0 part can be removed. An optimizing compiler will do this for you.

---

xianic (/member/xianic/) 2013-05-01 Reply

Thank you for making this instructable, I have the I2C Slave code running on an ATTiny 167 clocked at 8 MHz. It doesn't acknowledge it's address when the bus is running at 100kHz but it all works beautifully on a 5kHz bus. Have you any Idea why this is?

Kinnishian (/member/Kinnishian/) . xianic (/member/xianic/) 2013-07-13 Reply

Have you tried more intermediate speeds? My first guess is to assume you're running the attiny on the internal RC and it's pretty inaccurate, making more timing issues at a higher speed. (Non-ECE person here, just hobbyist)

---

peterrog61 (/member/peterrog61/) 2013-06-20 Reply

ATTiny USI I2C Introduction - a Powerful, Fast, and Convenient Communication Interface for Your ATTiny Projects!

by CalcProgrammer1 (/member/CalcProgrammer1/) in raspberry-pi (/technology/raspberry-pi/)

Download (/id/ATTiny-USI-I2C-The-detailed-in-depth-and-infor/) 8 Steps .

Collection I Made it! Favorite Share

I have been trying to use your slave code to read the 2 byte ADC value from an ATTiny45. The first problem I found is the default clock of 8MHz is not fast enough to allow the ACK to take place before the clock edge. A 16MHz clock works with about 1uS to spare. The code at case USI_SEND_DATA needed 3 lines adding as follows:

```
USISR = (USI_SLAVE_CLEAR_START_USISR);
PORT_USI |= (1 << PORT_USI_SDA);
USI_SET_SDA_OUTPUT();
```

at case USI_SLAVE_SEND_DATA_ACK_WAIT you need to add

```
USIDR = 0;
```

after this the code works fine. So thanks for this code which is well written and very useful.

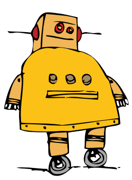
arjun_29 (/member/arjun_29/) 2013-02-08 Reply

It's not working. I tried the code. But when function switches from transceiver function to Transfer function pointer loses its reference. If possible post main function & updated code

pfred2 (/member/pfred2/) 2012-08-01 Reply

Serial communications always struck me as pretty complicated so I never messed with the stuff. From what you've described I2C sounds even more involved. So congratulations for figuring out how to get it to work for you. I would like to see more about the stepper motor driver you made. A Raspberry PI might be better if the odds of getting one wasn't like hitting the lottery.

More Comments



Newsletter

Let your inbox help you discover our best projects, classes, and contests. Instructables will help you learn how to make anything!

enter email I'm in!

About Us

Who We Are (/about/)
Advertise (/advertise/)
Contact (/about/contact.jsp)
Jobs (/community/Positions-available-at-Instructables/)
Help (/id/how-to-write-a-great-instructable/)

Find Us

Facebook (http://www.facebook.com/instructables)
Youtube (http://www.youtube.com/user/instructablestv)
Twitter (http://www.twitter.com/instructables)
Pinterest (http://www.pinterest.com/instructables)
Google+ (https://plus.google.com/+instructables)

Resources

For Teachers (/teachers/)
Residency Program (/pier9residency)
Gift Premium Account (/account/give?sourcea=footer)
Forums (/community/?categoryGroup=all&category=all)
Answers (/tag/type-question/?sort=RECENT)
Sitemap (/sitemap/)