

I2C Bus for ATtiny and ATmega

by **doctek** on January 1, 2009

Table of Contents

License: Attribution Non-commercial Share Alike (by-nc-sa)	2
Intro: I2C Bus for ATtiny and ATmega	2
step 1: What's All this I2C stuff anyway?	2
step 2: Order some I2C devices	3
step 3: I2C Drivers	3
step 4: Let's build!	4
File Downloads	8
step 5: Let's Code and Test!	8
File Downloads	10
step 6: Using I2C Memory	10
step 7: Web Resources	12
step 8: Notes for Geeks	12
Related Instructables	13
Advertisements	13
Comments	13

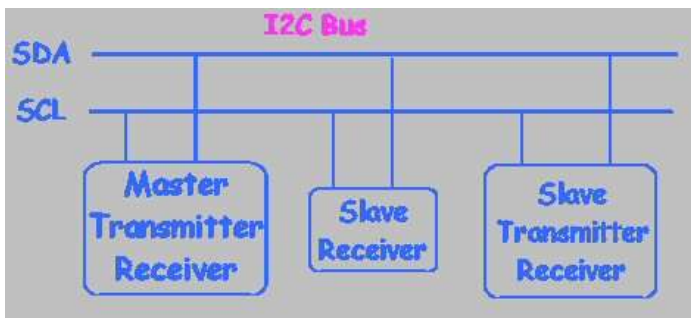
Intro: I2C Bus for ATtiny and ATmega

I love the Atmel AVR microcontrollers! Since building the Ghetto Development System described in this [Instructable](#), I've had no end of fun experimenting with the AVR ATtiny2313 and the ATmega168 in particular. I even went so far as to write an Instructable on using [switches as inputs](#), and extended the Ghetto Development System concept to [CPLDs](#).

During a recent project, I needed several switches for setting control values. The AVR's did not have enough I/O pins, so I had to think of something. I could have tried a complex input system with a keyboard and display, but the ATtiny2313 would have run out of resources. Fortunately, Atmel has provided a way around this problem by including an interface that can link to additional chips (such as memory or I/O ports) with a simple two wire interface. That's right, by using just two I/O pins on an AVR we can access many additional I/O pins, and other resources as well.

This two wire interface is formally known as the Inter-Integrated Circuit bus, or just the I2C bus and was invented by NXP when it was still Philips Semiconductors. If you're reading this Instructable then you've probably heard of the I2C bus and may even have used it on a PIC or other microcontroller. While conceptually very simple, and supported by hardware resources on the AVR's, software drivers are still necessary to use the I2C bus. Atmel provides Application Notes (see the Resources later in this Instructable), but these are incomplete and don't show any examples beyond communicating with another AVR device.

It is not the purpose of this Instructable to teach anyone how to create I2C drivers for the AVR's. Rather, I'll provide expanded versions of the Atmel drivers for ATtiny2313 and ATmega168 devices, I'll explain the requirements and restrictions that apply when using these, and I'll show you working examples of I2C devices. After you work through this Instructable you'll be able to use the I2C bus successfully in your AVR projects. Obviously, you can ignore the drivers for either tiny or MEGA if you're only interested in one of them. For those interested in learning more about the I2C bus, I'll provide links to appropriate material.



step 1: What's All this I2C stuff anyway?

The I2C bus is a simple, two-wire connection that can link multiple devices together and allow them to exchange data. In its simplest form there is one master device that communicates to multiple slave devices. All devices are connected in parallel to the two wires of the I2C bus. The two wires are known as SCL and SDA. SCL is the clock line and is controlled by the master device. SDA is the bi-directional data line. To transfer data, the master sends out a slave address combined with a one bit read/write flag. If a write is desired, the master will continue to send data to the addressed slave. If a read is requested, the slave will respond with data. To coordinate transactions, the SCL and SDA lines are manipulated by the master and the slave to signal several conditions. These include START, STOP, ACK (acknowledge) and NAK (no acknowledge). The details of these conditions are handled by the drivers. The true geeks among you can learn all the details in the links provided at the end of this Instructable.

The electrical requirements are pretty simple. The master and the slaves must use the same level for Vcc, the grounds must be connected, and the SCL and SDA lines must be pulled up to Vcc. The value of the pull-up resistors is precisely determined by a calculation based on the total capacitance on the bus, but practically can be pretty much any value between 1.8K and 10K. I start with 5.1K and use lower values until it works. This usually isn't an issue unless you have a lot of devices or long lengths of wire between devices.

The nominal data rate on the I2C bus is 100Kbits/second. Rates of 400Kbits/second, 1Mbits/second, and beyond are possible as well, but aren't supported by the drivers in this Instructable. All I2C devices will work at 100Kbits/second.

The ATtiny2313 and the ATmega168 each implement the I2C bus differently. ATtiny2313 uses the Universal Serial Interface (USI) hardware - which can also be used for the SPI bus. ATmega168 has dedicated hardware for the I2C bus known as the Two Wire Interface (TWI). Once the drivers are written, these differences are mostly transparent to the user. One significant difference is in the software: The ATmega168 I2C driver is interrupt driven while that for the ATtiny2313 is not. This means that an ATmega168 program does not have to wait for I2C data transfers to take place, but only needs to wait before initiating another transfer, or until data arrives from a read operation. The examples and discussion to follow should make this clear.

I2C addresses are 7 bits long, so up to 127 devices can be on the bus if each has a unique address. As shown in the figure, this 7 bit address is shifted left one bit and the least significant bit is used to flag a read or write of the device at the address. Thus the complete slave address is an 8 bit byte. The actual address is partially determined internally to the device and can't be changed (4 most significant bits), and partially determined by bits that may be connected to device pins (3 least significant bits) that can be tied high or low to set a specific address.

Sounds confusing, but an example will make this clear. The PCA8574A data sheet shows that the four most significant bits of the I2C address will always be 0111. The next three bits are determined by the settings on pins AD0, AD1 and AD2. These pins can be tied to ground or to the positive voltage supply (5 volts) to represent 0 or 1 respectively. So the range of possible addresses is 38 to 3F hexadecimal, as shown in the other figure from the PCA8574 data sheet. So by changing the address bit settings, up to 8 PCA8574As can be on the I2C bus at the same time. Each will respond to its specific slave address only. If even more I/O ports are needed, the PCA8574 can be used. The only difference between the PCA8574 and the PCA8574A is that the I2C slave address range of the PCA8574 is 20 to 27 hexadecimal.

Determining the address of a given device can be confusing since some data sheets consider the read/write bit to be part of the address. Read the data sheet carefully and keep in mind that the slave address will be 7 bits long. The read/write bit should be treated separately. Again, an example will help. The data sheet for the 24C16 EEPROM we'll experiment with says the first (most significant) four bits of the slave address are 1010. The next three bits can be determined by A0, A1 and A2; but note the data sheet also covers 24C01 through 24C08 which are smaller sized EEPROMs. The figure from the data sheet shows that the settings of these address bits are ignored as the size increases and are completely ignored for the 24C16. That is, the last three bits don't matter and the 24C16 really uses all I2C slave addresses 50 through 57 hexadecimal. The range of slave addresses will actually address different sections within the 24C16. The first 256 bytes are at address 50h, the next 256 at 51h, and so on up to the last 256 at 57h - for a total of 2K bytes. Since the address of the PCF8570 RAM we also experiment with is in this range, the 24C16 and the PCF8570 can't be used together.

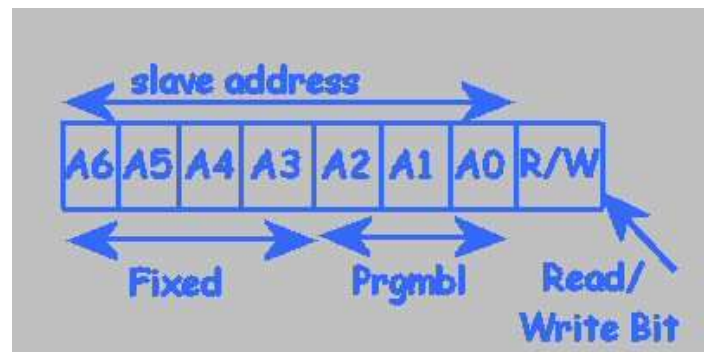
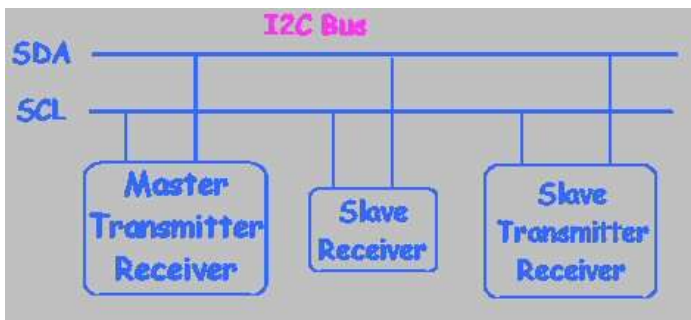


Table 5. PCA8574A address map

A6	A5	A4	A3	A2	A1	A0	Address
0	1	1	1	0	0	0	38h
0	1	1	1	0	0	1	39h
0	1	1	1	0	1	0	3Ah
0	1	1	1	0	1	1	3Bh
0	1	1	1	1	0	0	3Ch
0	1	1	1	1	0	1	3Dh
0	1	1	1	1	1	0	3Eh
0	1	1	1	1	1	1	3Fh

CAT24C16 / 08 / 04 / 02 / 01

NC / NC / NC / A ₀ / A ₀	1	8	V _{CC}
NC / NC / A ₁ / A ₁ / A ₁	2	7	WP
NC / A ₂ / A ₂ / A ₂ / A ₂	3	6	SCL
V _{SS}	4	5	SDA

step 2: Order some I2C devices

Now that you know a little about the I2C Bus and want to use it, why not order some I2C devices to experiment with now so they can be on the way to you while you're getting the software ready?

Appropriate devices include an I/O Interface Expander (my favorite), a Static Ram, and an EEPROM. There's lots more, but these are a great start. The AVR processors we'll use are the ATtiny2313 and the Atmega168 (used in Arduino). If you're new to these, then have a look at this great [Instructable](#) to learn about them and build your Ghetto Development System. The schematic of the ATmega168 in the present Instructable shows how to implement the Ghetto Development System for this processor. The parallel port cable is the same as the one for the ATtiny2313. (I haven't tried the USB version of the Ghetto Development System, so I'm not sure how the I2C bus is accessed on it. Same for the Arduino.)

Here are Digikey part numbers.

Port Expander:

IC I2C I/O EXPANDER 568-4236-5-ND

Ram:

IC SRAM 256X8 W/I2C 568-1071-5-ND

EEPROM :

IC EEPROM SERIAL 16K CAT24C16LI-G-ND

step 3: I2C Drivers

Here are the descriptions of the driver functions for the I2C bus. These were developed using the Atmel Apps Notes for starters. I couldn't have done this without them as a base to build on. Development was done using WinAVR and the gcc C compiler.

Clock rate restrictions are described below for each processor. Since I am not able to test all the processor flavor / clock rate combinations possible, I'll just stick to what I actually can test and try to indicate the restrictions and limitations.

Here are the driver functions and how to use them. Please look at the examples for more details and to see the functions in use in complete programs.

For the ATtiny2313:

Clock Requirement:

The drivers are designed for a clock rate of 1MHz (the default rate) for ATtiny2313. If you want to run at other rates, then you'll have to adjust constants in the drivers.

Email me if you need help doing this. You can also get some hints from the Atmel apps notes in the links in the Resources Step.

USI_TWI_Master_Initialise()

This function initializes the USI hardware for I2C mode operation. Call it once at the start of your program. It returns void and there are no arguments.

USI_TWI_Get_State_Info()

This function returns I2C error information and is used if an error occurred during an I2C transaction. Since this function only returns an error code, I use the function TWI_Act_On_Failure_In_Last_Transmission(TWIerrorMsg) to flash an error LED. The error codes are defined in USI_TWI_Master.h. Here's how to call it:

TWI_Act_On_Failure_In_Last_Transmission(USI_TWI_Get_State_Info())

USI_TWI_Start_Read_Write()

This function is used to read and write single bytes to I2C devices. It is also used to write multiple bytes. There are 6 steps to using this function.

1)Declare a message buffer in your program to hold the slave address and the data byte to be sent or received.

unsigned char messageBuf (MESSAGEBUF_SIZE);

2)Put the Slave Address as the first byte in the buffer. Shift it one bit left and OR in the Read/Write bit. Note the Read/Write bit will be 1 for a Read and 0 for a Write. This example is for a Read.

messageBuf(0) = (TWI_targetSlaveAddress<<TWI_ADR_BITS) | (TRUE<<TWI_READ_BIT);

3)When doing a Write, put the byte to be written into the next location in the buffer.

4)Call the USI_TWI_Start_Read_Write function with the message buffer and the message size as arguments.

temp = USI_TWI_Start_Read_Write(messageBuf, 2);

5)The returned value (temp in this case) can be tested to see if an error occurred. If so, it is handled as discussed above. See examples in the programs.

http://www.instructables.com/id/I2C_Bus_for_ATtiny_and_ATmega/

6) If a Read was requested, the byte read will be in the second location in the buffer.

If multiple bytes are to be written (such as to a memory device), this same routine can be used. Setting up the buffer and calling the routine are slightly different. The second byte in the buffer will be the starting memory address to which to write. The data to be written will be in subsequent bytes. The message size will be the size including all the valid data. So if 6 bytes are to be written, then the message size will be 8 (slave address + memory address + 6 bytes of data).

`USI_TWI_Start_Random_Read()`

This function is used to read multiple bytes from an I2C device, typically it's only meaningful for a memory of some sort. Using this routine is very similar to the previous routine, with two exceptions.

The setting of the Read/Write bit doesn't matter. Calling this routine will always cause a Read operation.

The messageSize should be 2 plus the number of bytes to be read.

If no errors occurred, the data will be in the buffer beginning at the second location.

For the ATmega168:

Clock Requirement:

The drivers are designed for a clock rate of 4MHz for ATmega168. The example code shows how to set this clock rate. If you want to run at other rates, then you'll have to adjust constants in the drivers. Email me if you need to do this.

`TWI_Master_Initialise()`

This function initializes the TWI hardware for I2C mode operation. Call it once at the start of your program. It returns void and there are no arguments. Be sure to enable interrupts by calling `swi()` after initializing.

`TWI_Get_State_Info()`

This function returns I2C error information and is used if an error occurred during an I2C transaction. Since this function only returns an error code, I use the function `TWI_Act_On_Failure_In_Last_Transmission(TWIerrorMsg)` to flash an error LED. The error codes are defined in `TWI_Master.h`, but are modified for signaling on an error LED. See the example code for details. Here's how to call it:

`TWI_Act_On_Failure_In_Last_Transmission(TWI_Get_State_Info())`

Note that error checking is done by making sure that the I2C transaction is complete (function described below) and then testing a bit in the global status word.

`TWI_Start_Read_Write()`

`TWI_Start_Random_Read()`

These two functions work the same as the corresponding functions described above but with a few exceptions.

They don't return any error values.

Data read is not transferred into the buffer. Doing this will be done with the function described next.

When calling `TWI_Start_Random_Read`, the messageSize should be the number of data bytes requested plus one, not two.

The I2C driver for the ATmega168 is interrupt driven. That is, the I2C transactions are started and then execute independently while the main routine continues to run. When the main routine wants data from an I2C transaction that it started, it must check to see if the data is available. The situation is the same for error checking. The main routine must be sure that the I2C transaction is complete before checking for errors. The next two functions are used for these purposes.

`TWI_Transceiver_Busy()`

Call this function to see if an I2C transaction is complete before checking for errors. The example programs show how to use this.

`TWI_Read_Data_From_Buffer()`

Call this function to transfer data from the I2C driver's receive buffer into the message buffer. This function will make sure the I2C transaction is complete before transferring the data. While a value is returned by this function, I find checking the error bit directly to be more reliable. Here's how to call it. The message Size should be one greater than the number of data bits desired. The data will be in messageBuf starting at the second location.

`temp = TWI_Read_Data_From_Buffer(messageBuf, messageSize);`

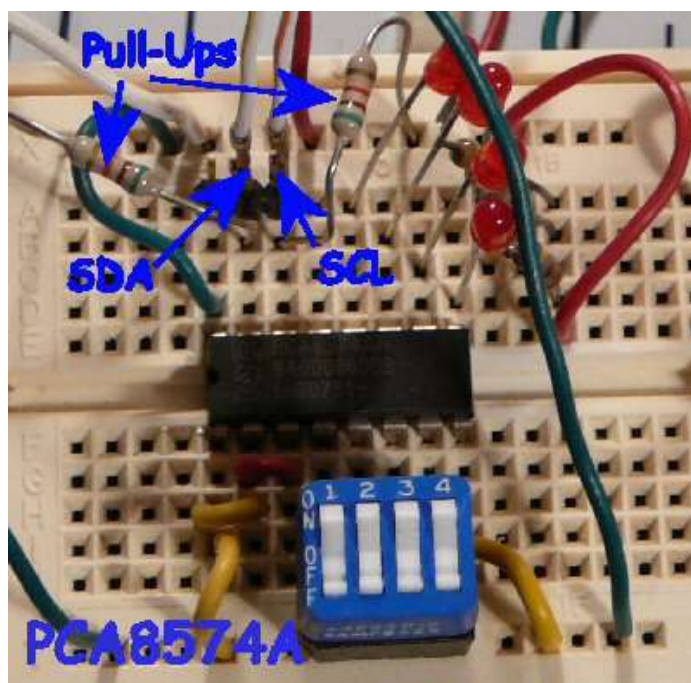
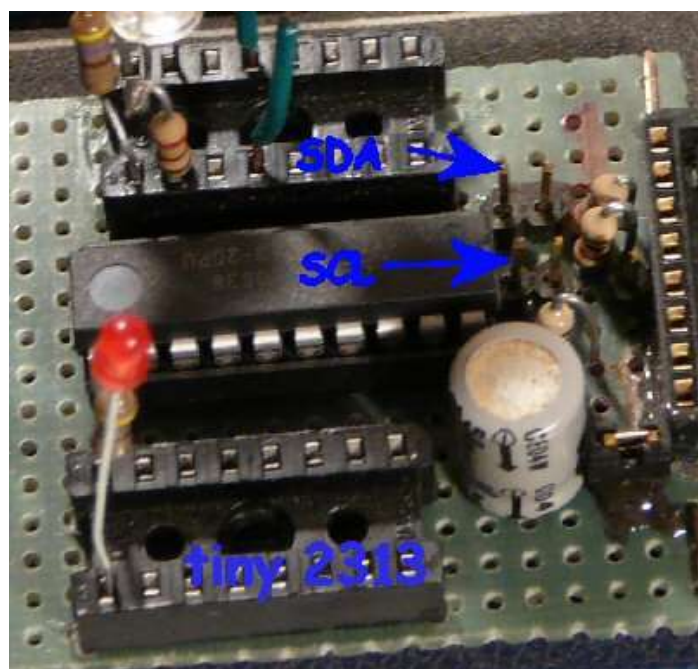
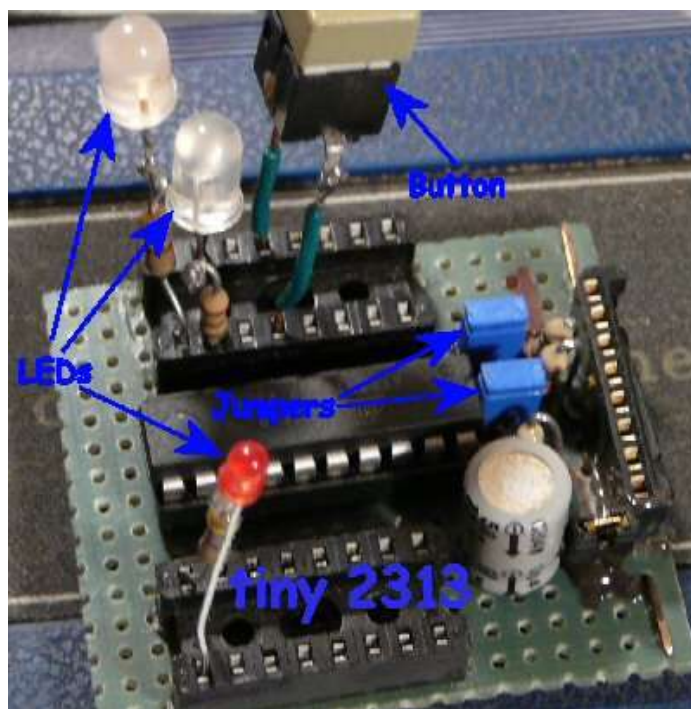
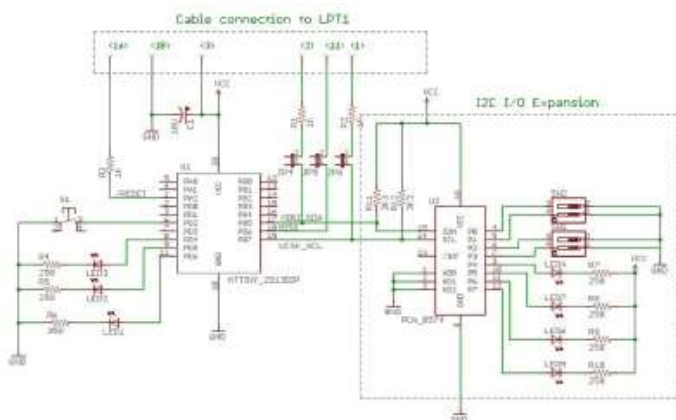
step 4: Let's build!

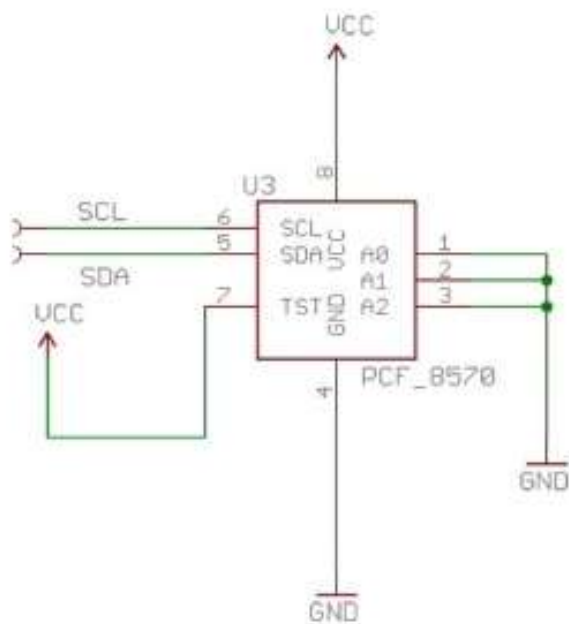
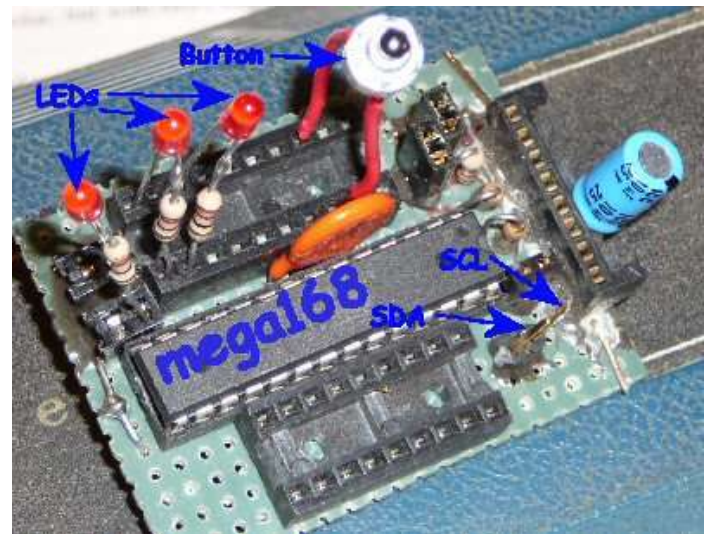
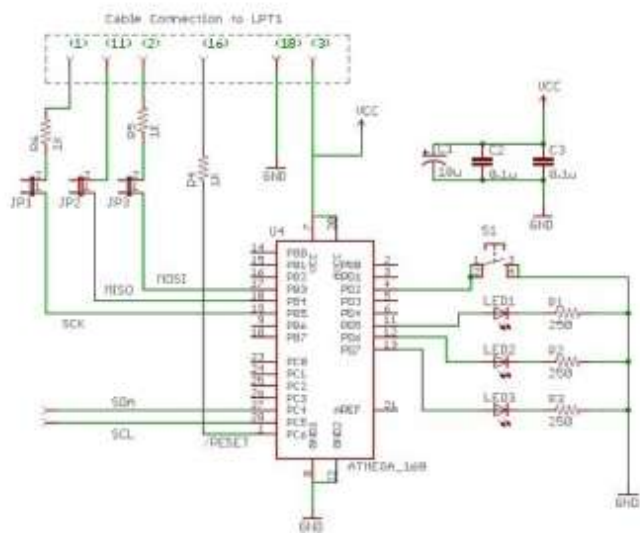
Start by downloading the file I2C Schematics.zip. You may want to create an I2C folder in your work area to hold the schematics and the example program files. Unzip the schematics into this directory. You'll find a folder called I2C Schematics. Open the file named tiny I2C.pdf. This schematic shows the ATtiny2313 Ghetto Development System, and the PCA8574A I/O Port Expander (has the large dashed box around it). The Port Expander circuit is built on a breadboard. Have a look at the photos to see what these circuits look like. They're really pretty simple.

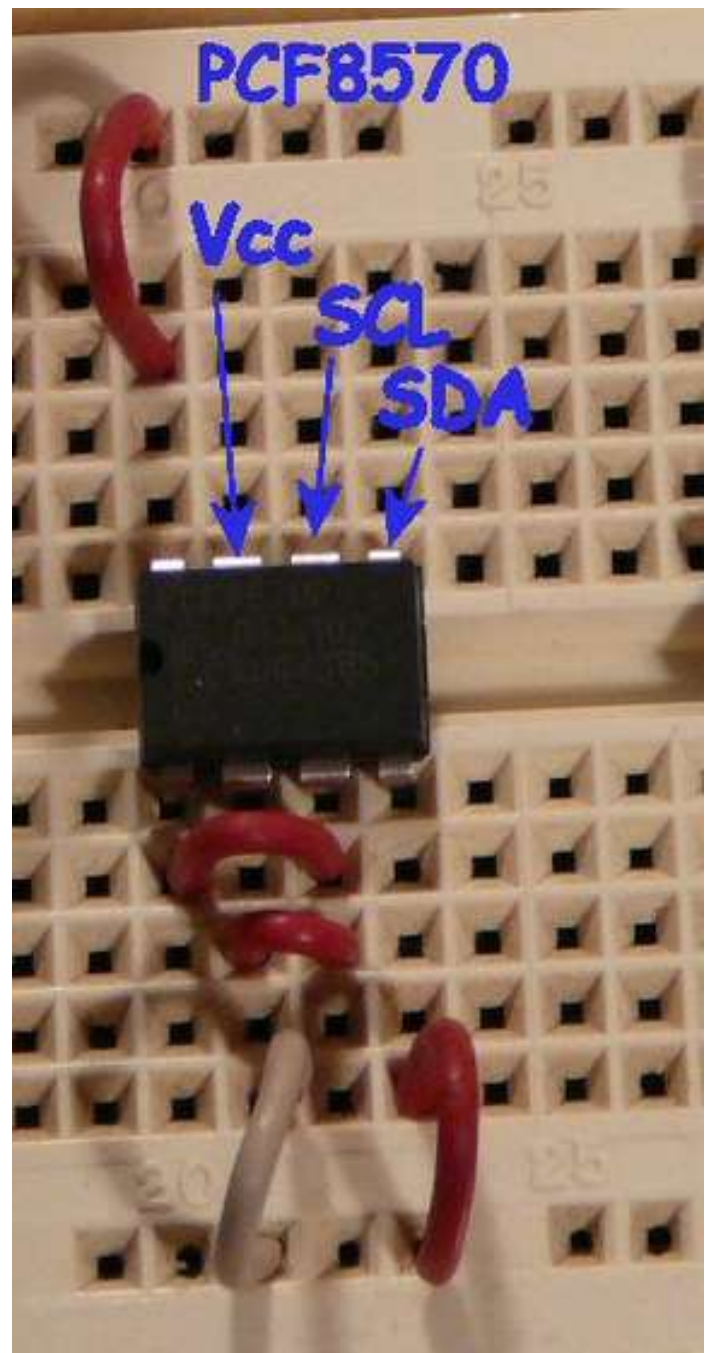
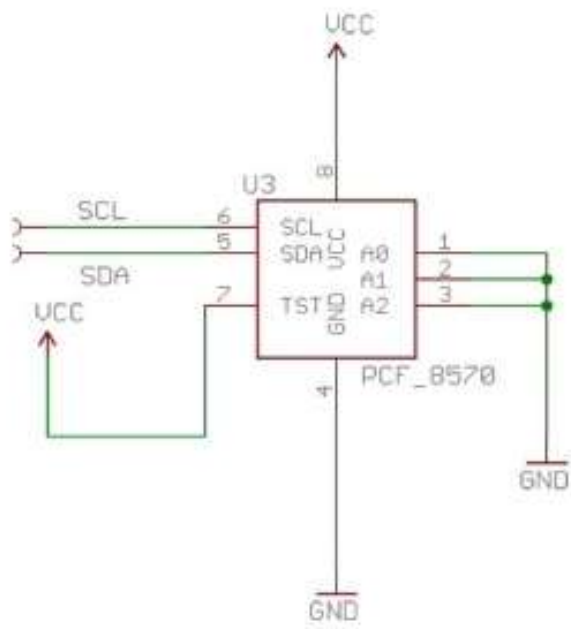
The ATtiny2313 portion of the schematic is just the Ghetto Development System with three blinkenlights (LED1, 2, and 3, plus R4, 5, and 6) and a pushbutton (S1) hooked to it, plus one additional detail. That detail is the addition of jumpers (JP4, 5, and 6) which can be removed to allow connection of the I2C bus SCL and SDA lines. The jumpers must be in place for programming, then removed so SCL and SDA can be connected. The photos show the jumpers in place and removed. Placement of these jumpers is up to you, you just have to put them on your Ghetto Development System if you want to use the I2C bus. The I2C bus must be disconnected and the jumpers put in place for programming. Note that you only really need to be concerned with JP4 and JP6 for the I2C bus. Put in JP5 if you think you'll ever want to use the SPI bus.

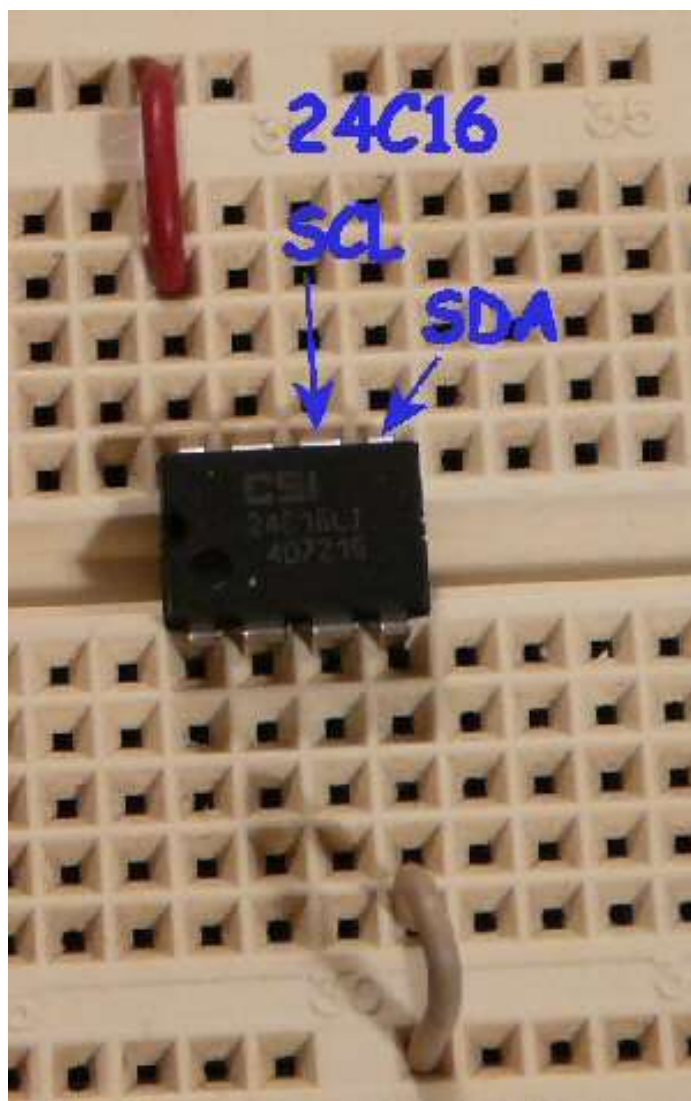
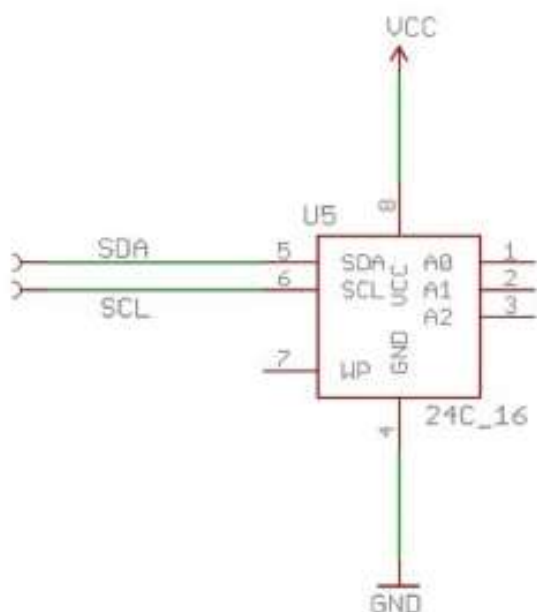
Breadboarding the PCA8574A I/O Port Expander is very simple. Provide Vcc (+5 volts) and Gnd (ground) connections and connect AD0, 1, and 2 to ground (makes the I2C slave address 38 hex). Then connect 4 blinkenlights, and 4 DIP switches. (If you don't have DIP switches you can just use wires. Tie to ground or leave floating to signal on or off respectively.) Finally, connect the pull-up resistors (R11 and 12) from SDA and SCL to Vcc. These are shown as 3.3K, but any value from 1.8K to 5.1K should work (maybe up to 10K but I haven't tried that). Once you've programmed the ATtiny2313 you can remove the jumpers and connect SDA and SCL for testing.

Now for the ATmega168. The only wrinkle here is that you may not have built a Ghetto Development System for this processor. If that's the case, then the schematic I provide (MEGA I2C.pdf) will show you how. This is just a permutation of the ATtiny2313 version. If you plan ahead you can make sure your programming cable will fit both systems. The main difference is the addition of C2 and C3. See the pictures for placement of these, they should be very close to the chip; one of them is actually under the chip. These help keep noise out of the analog to digital converter in particular. You don't need to put in the jumpers unless you plan to use the SPI bus since they're not needed for the I2C bus on this chip. Note that the PCA8574A breadboard will be unchanged. You'll just hook up SDA and SCL and away you go! Easy, huh?









File Downloads



I2C Schematics.zip (75 KB)

[NOTE: When saving, if you see .tmp as the file ext, rename it to 'I2C Schematics.zip']

step 5: Let's Code and Test!

It's time to build the drivers and the example programs. We'll start with the ATtiny2313 and the PCA8574A breadboard that we just built. Download the file I2C.zip into your I2C work directory and unzip it. You'll have a new folder called I2C. In it you'll find USI_I2C (for ATtiny2313) and TWI_I2C (for ATmega168). In USI_I2C, you'll find the I_O Port folder. That folder contains the code for our first example program, and the USI_I2C drivers.

Using WinAVR, compile and load the code into the ATtiny2313. Take a deep breath and turn on the power. Here's what to expect:

At power on, LED 1 on port PD6 of the ATtiny2313 blinks twice.

Nothing else will happen until you push the button (S1). Each time the button is pressed, the switches are read and their setting will be displayed on the LEDs connected to the PCA8574A. Change the value of the switches, press the button, and the LEDs should change. Keep doing this until you get over the thrill of seeing it work. If (God forbid!) things don't work as expected, carefully check over your wiring. I2C errors will be signaled by blinks on LED3 (PD4) and probably mean you need to check that SDA and SCL are connected to the correct pins and are pulled up correctly. If things still don't work, read the rest of this section to learn about debugging.

Now go back and let's have a look at the code. Open the file USI_I2C_Port.c. This is the code for the example program. (USI_TWI_Master.c and USI_TWI_Master.h contain the drivers - you can ignore them unless you're curious.) Use the example to guide your own I2C applications.

Mostly, the program shows you how to initialize and use the I2C drivers, including setting up the slave address and the rest of the message buffer, and getting the data out of it. You'll also see how I debounce the button and set up the while loop. There are a few details of the program worth mentioning. Note that the data from the switches is inverted before it is written to the LEDs on the Port Expander. Also note that the input ports on the Port Expander must be written as High to make them work properly. Those details are described in the PCA8574A data sheet. Always read the data sheets carefully!

Of more interest is the use of conditional debugging. Near the start of the program file is the statement `//#define DEBUG` and sprinkled throughout the code are `#ifdef DEBUG` statements. As long as `DEBUG` is not defined (the two slashes make the line a comment and keep it from being defined), the code within the `#ifdef` to `#endif` statements will not be compiled. But if things don't work as you expect, recompile and reload the code with `#define DEBUG` uncommented. You'll get a lot more blinks on the LEDs which you can decode to follow the execution of your program and aid you in finding exactly where things go wrong. In fact, I recommend you try this just to see what happens.

What you'll see is that LED 2 (on PD5) will blink as execution progress through the program. The value read from the switches will be blinked on LED 1 (PD6) before it is displayed on the Port Expander LEDs. You should be able to track the program as it runs by using these LEDs.

http://www.instructables.com/id/I2C_Bus_for_ATtiny_and_ATmega/

We'll work with the ATmega168 next; skip this section if you're only interested in the ATtiny2313. Still with me? Good. Move to the TWI_I2C folder, change your working directory to IO_Port, and compile and load TWI_I2C_Port.c into the ATmega168. Disconnect the SDA and SCL lines from the ATtiny2313 and connect them to ATmega168. Hook up power and ground, and power up. The operation should be the same! Play until the thrill subsides, then let's look at the code.

Open TWI_I2C_Port.c. The code is nearly identical except for error handling and accommodating interrupt driven drivers. Here are the differences:

Note that the clock must be set to 4MHz for the I2C bus to work properly.

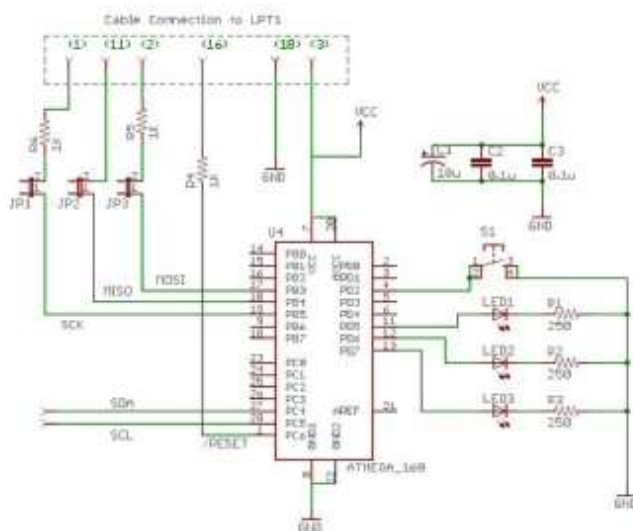
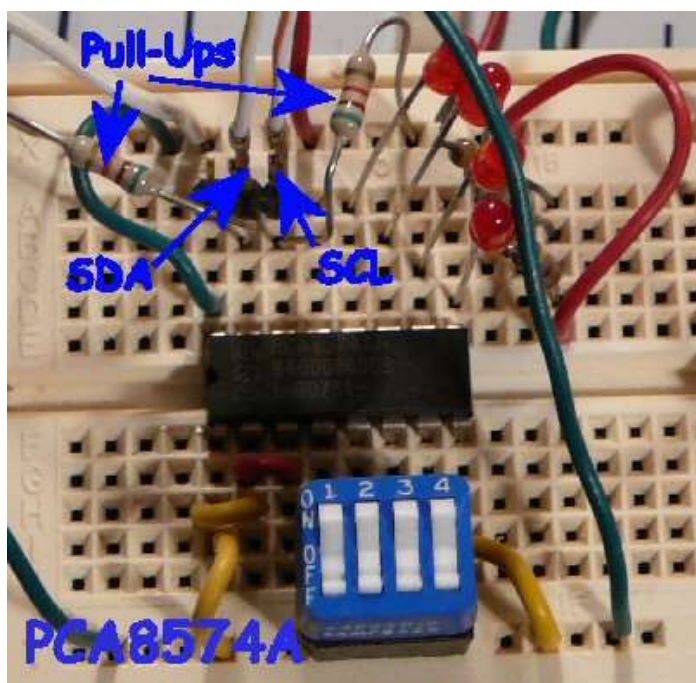
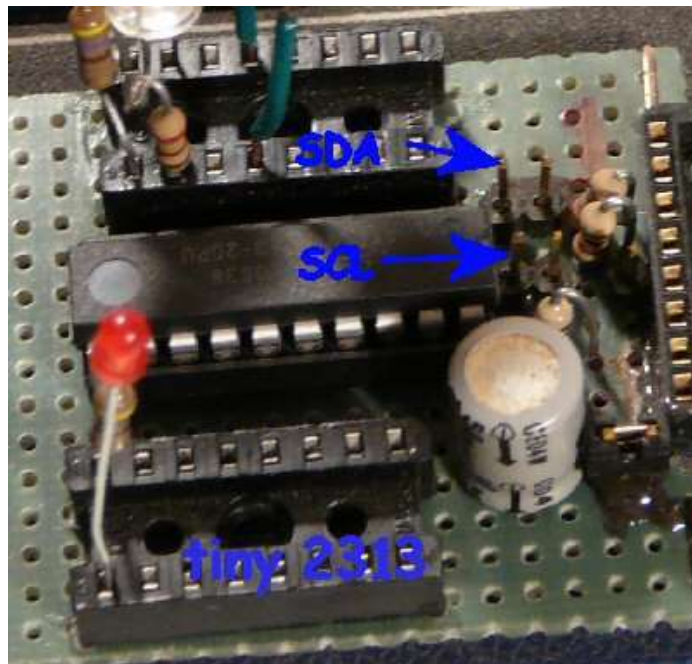
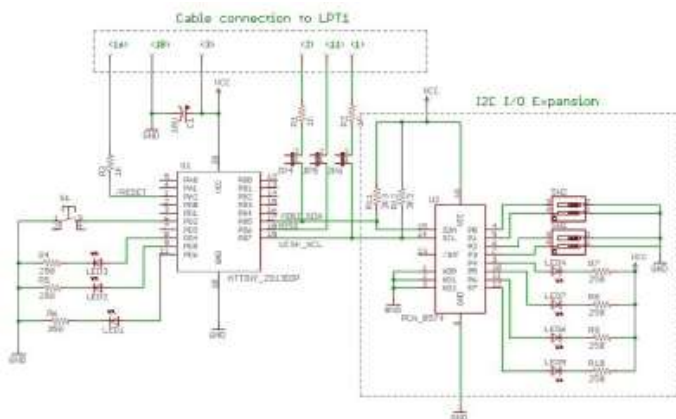
The sei(); statement turns on interrupts after initialization of the I2C drivers.

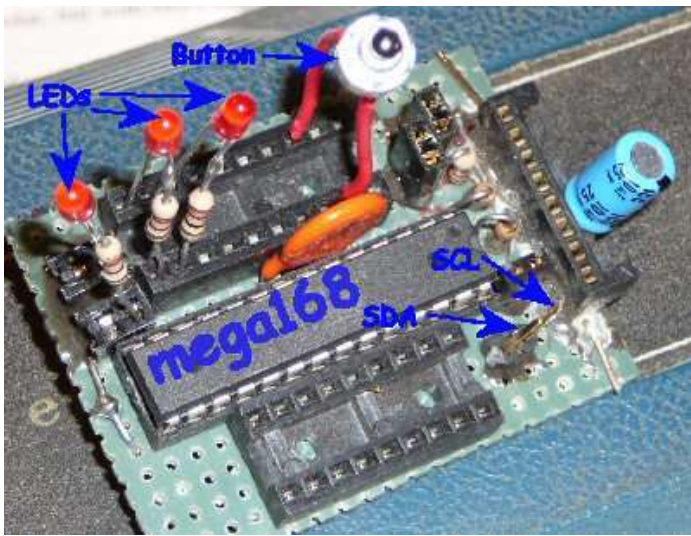
To check for errors, a specific status bit is tested.

During a read, the TWI_Read_Data_From_Buffer function must be called to transfer the data read into the message buffer.

During a write, while (TWI_Transceiver_Busy()) must be used to be sure the transfer is complete before checking for errors.

These last two functions are described above in the description of the drivers. Other than that, the code's pretty much the same as for the ATtiny2313. DEBUG works the same also if you want to experiment with that.





File Downloads



I2C.zip (47 KB)

[NOTE: When saving, if you see .tmp as the file ext, rename it to 'I2C.zip']

step 6: Using I2C Memory

Now that we've learned to use the I2C bus to read and write an I/O Port Expander, let's move on to using I2C memories, both RAM and EEPROM. The main difference is that multiple bytes can be read to or written from memories with a single I2C command.

To get ready for these experiments, we need to modify hardware slightly and build a couple of new circuits on the breadboard. Keep the Port Expander circuit since we'll use it to display some memory values. Remove the DIP switches from the PCA8574A and put blinkenlights on those pins. If you don't have enough blinkenlights, move the ones on P4 thru P7 over to P0 thru P3. (Values to be displayed are small enough.)

Now look at the schematic I2C Ram.pdf and hook up the PCF8570 on the breadboard. Have a look at the picture also. Be sure to tie pin 7 to Vcc. Run wires for SDA and SCL from the PCA8574A. No additional pull-up resistors are required.

If you're also interested in the EEPROM, build that circuit also using I2C EEPROM.pdf for the 24C16, but be warned that the example uses the ATmega168. This circuit is really simple. As discussed above, the address bits should be ignored. Just hook up power and ground. Don't connect SDA and SCL just yet since we haven't finished experimenting with the Ram.

We'll start our memory experiments with the ATtiny2313 connected to the PCA8574A Port Expander and to the PCF8570 Ram. The program will write some numbers to the Ram, then read them back and display them on the Port Expander.

Change your working directory to RAM under USI I2C. Use the make file to compile and download USI_I2C_RAM.c. Note that the I2C driver files are identical to those we used earlier. Hook up the power and you should see a single blink on LED 1 (PD6). Data will be written to the first 4 bytes of memory. Press the button and two bytes will be read back and displayed. You should see one LED light on the Port Expander (P0), a two second pause, then two LEDs light (P0 and P1). Another two second pause and the LEDs should turn off. Press the button again to start the sequence over. Debugging is similar to the method described above.

Let's take a look at the code. Open USI_I2C_RAM.c. It should look pretty similar to the previous code. The main differences are the details of reading and writing memory. Look at the way the message buffer is loaded before the call that actually does the write. The first byte is the slave address with the read/write bit set appropriately. But the next byte is the memory address at which to start writing data. Then comes the actual data bytes which will be sequentially loaded into memory starting at the address we specified. We specify the message size as 6. So we start writing at address 00 and write the values 01, 03, 02 and 06 into the memory locations 00 to 03.

To read the data back from the memory we must use the USI_TWI_Start_Random_Read function. The message buffer gets the slave address in the first byte and the starting address in the second byte. Then call the function with the message size set to the number of bytes to read plus 2. Note that the read/write bit doesn't matter since a read will be done regardless. The data returned will start in the second location in the message buffer. Once the data is read in, it's inverted for display on the Port Expander and written one byte at a time to it with a pause between the values. Finally, the Port Expander LEDs are turned off. The writes to the Port Expander are identical to what was done in the previous examples. For fun, you can uncomment the #define DEBUG statement as above and see lots of blinking LEDs.

Flushed with excitement after another successful experiment, let's move to the ATmega168 and an EEPROM. Change your working directory to EEPROM under TWI I2C. Use the make file to compile and download TWI_I2C_EEPROM.c. Note that the I2C driver files are identical to those we used earlier for the PCA8574A. To test the program, disconnect the ATtiny2313 and connect the ATmega168. Leave the I2C bus hooked to the Ram and power up. The results are different since we're now writing and reading more data. LED 1 on PD7 should blink at initialization. Press the button and data will be read back from the memory and displayed. The LEDs on the PCA8574 should blink the following sequence: P1, P0 & P2, (all off), P0 & P1, P1 & P2. Finally the Port LEDs should all go off. Press the button again to repeat this.

Oh, but wait, you say. Isn't this program for the EEPROM? Since we're accessing a memory device at the same I2C address, the same program works for both the Ram and the EEPROM. Power down and move SDA and SCL from the Ram to the EEPROM and run the program again. It should work exactly the same. Note that the EEPROM and the Ram can't be connected to the I2C bus at the same time since they share the same address. (The clever ones among you may consider changing the programmable address bits on the Ram, but that still won't work. The 24C16 uses the entire block of addresses that can be programmed for the Ram.)

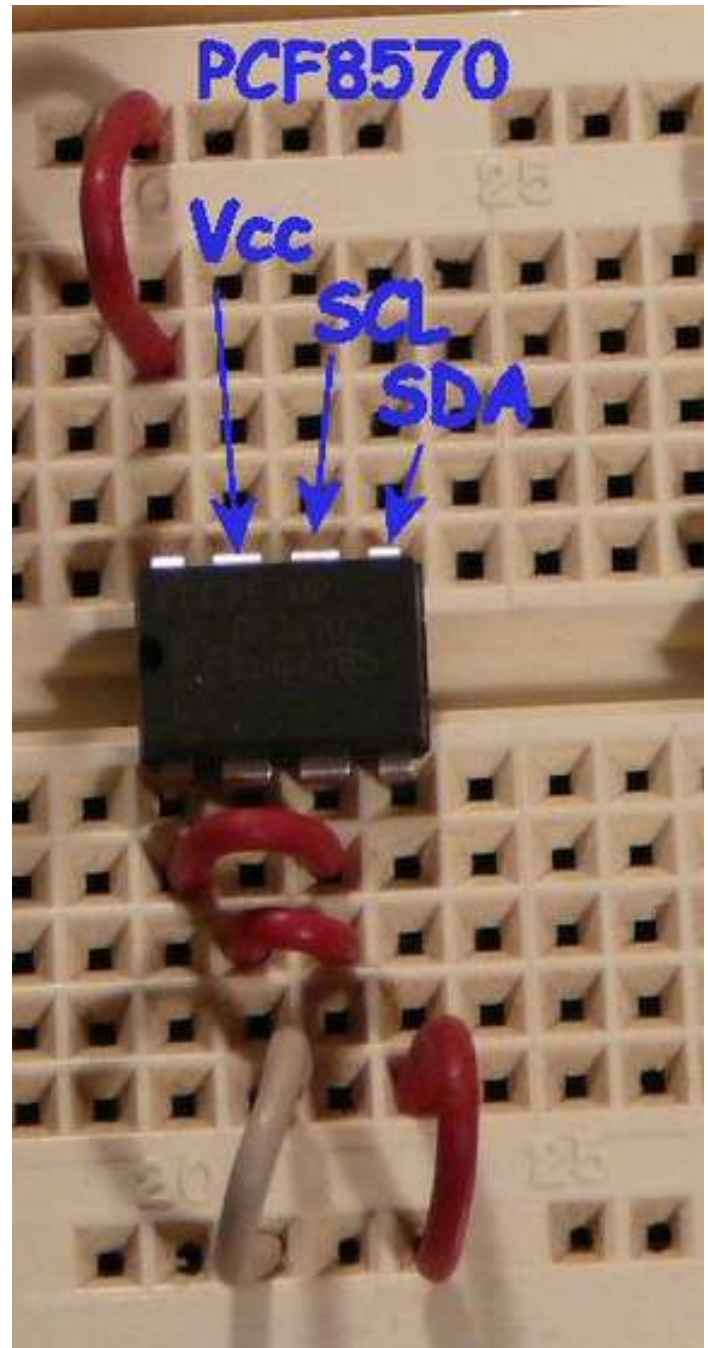
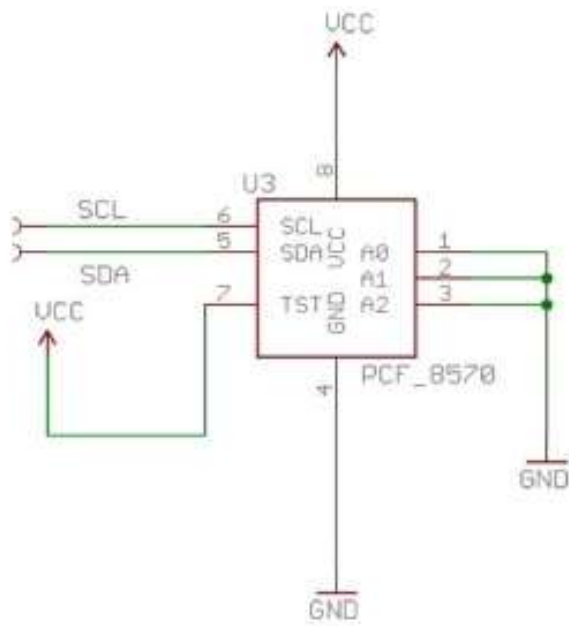
OK, let's look at this last program. Open TWI_I2C_EEPROM.c. The first thing to notice is that I've indicated how to address the complete 24C16 EEPROM. It can be accessed in 256 byte chunks at 8 different I2C slave addresses. See how MEMORY_ADDR is defined as the starting address at 50 hexadecimal; that's why the Ram worked. If you want to access other blocks of the 24C16, then use the other addresses as I've indicated. Have a look at how I set up to write to the memory. First the slave address with the read/write bit set is put in the buffer, then the starting address of 00, then 16 bytes of data. The function TWI_Start_Read_Write is called to write the data (as before) with the message size set to 18. When the button is pressed, we use TWI_Start_Random_Read and TWI_Read_Data_From_Buffer to read the data back. Every third byte is displayed on the Port Expander LEDs. Finally, the LEDs are turned off to await the next button press.

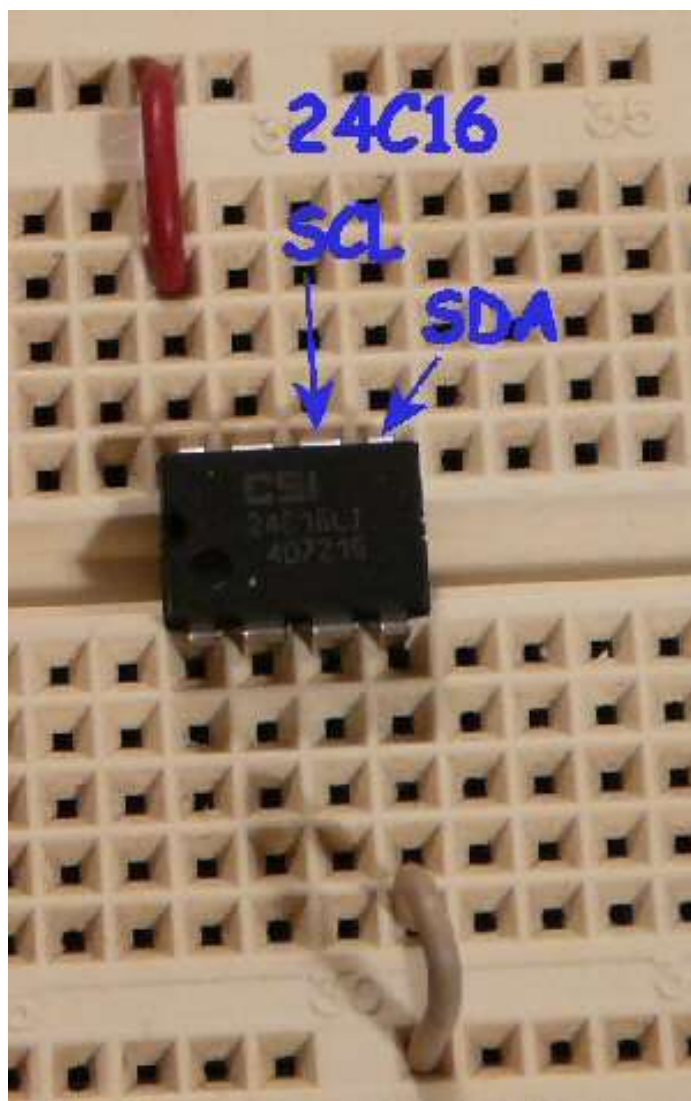
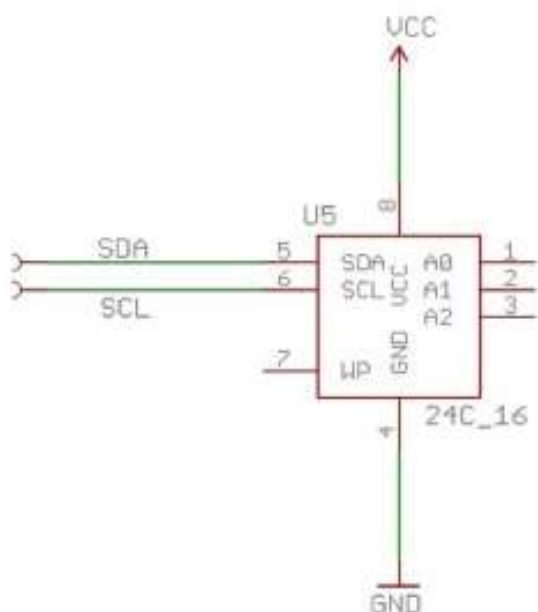
You might wonder why I chose to write 16 bytes. If you read the data sheet carefully, you'll see that the 24C16 does a write cycle whenever it receives 16 bytes even if

http://www.instructables.com/id/I2C_Bus_for_ATtiny_and_ATmega/

more bytes are being sent. So that seemed like a nice number to use. If you choose to increase this, you'll have to change the size of MESSAGEBUF_SIZE. You'll also have to change the value TWI_BUFFER_SIZE in TWI_Master.h. This is because the driver copies the data from the message buffer for use by the interrupt service routine.

Congratulations! You're now ready to use the I2C bus in your own projects!





step 7: Web Resources

Here are the links to the datasheets for the parts used for the experiments. You should definitely get these if you get nothing else.

Port Expander
Ram
EEPROM

Being the creator of I2C, NXP (Philips) has a ton of great stuff. (They like to use square brackets in their URLs, so I can't properly include them here. Sorry.) To get to the I2C area, select Interface from the Products list. You'll be able to get to their I2C site and access to all the datasheets and apps notes they offer. The I2C bus description and technical details in particular are [here](#).

Get the ATtiny2313 and ATmega168 datasheets (data books?) from [Atmel](#).

The Atmel application notes are [here](#). Look at AVR310 and AVR315. Grab the code also.

Have a look [here](#) for lots more I2C stuff.

step 8: Notes for Geeks

For the true geek who wants to know the details, here are some things to keep in mind if you look at the Atmel Apps Notes and driver code:

- The method of addressing and commanding an I2C device is not part of the spec! Other than the slave address and read/write bit, commands, modes, etc. are not specified and are specific to a given device. To make this very clear, note that the scheme used in the Atmel example only applies to that example, and is pretty much non-standard.
- The USI implementation differs from the TWI implementation in a few important ways.
- + With USI, clocking is provided by software; with TWI it is provided by a Bit Rate Generator.
- + The USI method does not use interrupts; the TWI does. This makes a certain amount of sense since the Mega family (using TWI) could be doing lots of other things and shouldn't be hogged by I2C transfers. An interrupt driven version for USI is certainly possible, it's just not implemented in this Instructable.
- + The USI hardware is not optimized for I2C and can only handle 8 bit transfers. This means that two transfers are required to send the ninth bit (either NACK or ACK). The TWI hardware handles this automatically. This makes the USI driver implementation a little more complicated.
- + Error detection for the TWI is handled in hardware. The USI requires handling in software which complicates things somewhat.
- + The TWI hardware controls the configuration of the port directly. The USI hardware requires that the port bits be configured before the port can be used. You'll see this in the Master_Initialize routine for the USI.

- Atmel claims it's possible to use AVR port pull-ups for the I2C bus pull-ups. I haven't figured out a way to make that approach work. Using two external resistors seems like a pretty simple scheme, so I didn't spend a lot of time on this.

Related Instructables



I2C Controlled 7 Segment LED Display by nolte919



I2C between Arduino's by cornelam



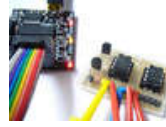
How to fix dead atmega and attiny avr chips by manekinen



World's Cheapest I2C (I-Squared-C) Adapter by pburgess



HD44780 LCD to I2C adapter board for the Bus Pirate by ian



Bus Pirate 3EEPROM explorer board by ian



Interactive Ambient Light by Hazard



Extreme Surface Mount Soldering by doctek

Comments

20 comments [Add Comment](#)



gremlin_1 says:

May 4, 2010. 12:15 AM [REPLY](#)

I really appreciate the author spending the time to write this tutorial. I have built the ATTINY2313 and PCA8574 I/O expander circuit. However, I have one question on the schematic. The LEDs on the PCA 8574 seems to be backward. They will never turn on. Maybe the common VCC connection is wrong and it should be GND instead? Can the author clarify it? Thanks.



doctek says:

May 4, 2010. 9:33 AM [REPLY](#)

Nice catch! The LEDs on the I/O expander schematic are indeed backwards and would never turn on. Reversing them makes things work correctly. Do not connect the common connection to ground unless you want the LEDs to default to on instead of off. The PCA8574's outputs are initially high, not low. So the circuit is intended to turn the LEDs on when an output is taken low. Thus, the common connection is to VCC.

Thanks for your attention to detail!



tissit says:

Apr 13, 2010. 1:38 PM [REPLY](#)

Atmel claims it's possible to use AVR port pull-ups for the I2C bus pull-ups. I haven't figured out a way to make that approach work.

Have you tried setting the pin and then twiddling the DDR instead of the actual pin. ISTR that being the usual way to do open collector-ish lines.



tissit says:

Apr 13, 2010. 1:20 PM [REPLY](#)

Not all avr's have TWI, though. It would be nice to see some great writeup on developing a bitbang driver. That'd be useful for other controllers as well.



nolte919 says:

Feb 9, 2010. 6:29 PM [REPLY](#)

I used this great instructable and referenced it a couple times in my instructable I2C Controlled 7 Segment LED Display . I used the LED display as part of a digital thermometer. I really like I2C. I also used external 4.7k pullup resistors on each I2C line rather than relying on the AVR internal ones.



doctek says:

Feb 9, 2010. 8:47 PM [REPLY](#)

Nice job with your Instructable! I'm glad you found mine useful and I appreciate the shout-out. The external pull-ups are shown in my schematic for the 2313 and are discussed in the second paragraph of step one. The value you found useful (4.7K) is right in the range I suggest: 1.8 to 5.1K, so I'm glad that worked for you. The internal pull-ups are fine for switches, but I just didn't quite trust them for this application.

Your comments remind me that I need to update this Instructable with my latest code that now handles addresses longer than two bytes. (See discussion with p2otoole.)



nolte919 says:

Feb 10, 2010. 8:57 PM [REPLY](#)

Too cool. The project I'm working on right now will have an AVR as an I2C slave to put a 16x2 LCD character display on an I2C bus. To try and make an LCD module that's super easy to add to a project. I'm going off the slave versions of the Atmel app. notes you used for this project.



doctek says:

Feb 11, 2010. 12:17 PM [REPLY](#)

Sounds like a great project! I'm eager to see the Instructable. Of course there are already some I2C 16x2 displays out there, but that shouldn't keep you from making a better one!

If you use the code from Atmel for the slave, be aware that they handle addressing in a non-standard way. Have a look at the data sheets for the standard approach. The data sheets for the parts (EEPROM and Port Expander) I used are a good place to start. You could also have a look at some of the I2C displays that exist to see what they do. Or invent your own scheme. It's your project after all.



nolte919 says:

Feb 11, 2010. 9:08 PM [REPLY](#)

Yeah, the one I plan on making is considerably cheaper than any I could find and I plan on making the user interface easier too. And as I'm sure you understand, I'm as much interested in learning and teaching how to make a slave I2C device as I am in making a better I2C LCD display. I think the I2C protocol is so versatile and over 100 devices with only 2 pins is hard to beat.



dtr2 says:

Dec 23, 2009. 12:42 PM [REPLY](#)

Do you have a "twi slave" code for for ATTINY2313? I want to control multiple of those from a master ARDUINO...



roagile-2007 says:

Nov 24, 2009. 1:06 PM [REPLY](#)

Hey my friends i need make the arduino send a message to the nxt please how do i make it.I want to program the nxt to use nxc conncted to the arduino,the arduino will be connected to the rfid reader.the rfid tag will be read and the arduino compares with the tags already stored in its memory,if it corresponds it sends the msg to the nxt,plz help is vry vry agent



LYNTRAN says:

Oct 21, 2009. 9:12 AM [REPLY](#)

Dear Developer of this source code,

I have a question about the source code.

I went over the code written for ATtiny2313. You select External, positive edge for the Shift register and Software clock strobe (USITC) for 4-bit Counter (USICS1=1, USICS0=0, USICLK=1). In the function unsigned char USI_TWI_Start_Transceiver_With_Data() you define the constant below.

```
unsigned char const tempUSISR_8bit = (1<<USISIF)|(1<<USIOIF)|(1<<USIPF)|(1<<USIDC)| // Prepare register value to: Clear flags, and
                                (0x0<<USICNT0); // set USI to shift 8 bits i.e. count 16 clock edges.
```

Why the program has to count 16 clock edges for 8-bit transfer? I thought the program has to count 16 clock edges for 8-bit transfer when USICS1=1, USICS0=0, USICLK=0 or when USICS1=1, USICS0=1, USICLK=0.

Please explain.

Thank you so much,



doctek says:

Oct 21, 2009. 5:21 PM [REPLY](#)

This part of the code (as I point out) is from Atmel. Have a look at the Apps Notes and data sheets I reference in Step 7. Frankly, the code worked fine for me and I was satisfied with that. If this question is of real concern to you and the resources I mention don't help, then you might post your question at the AVR Freaks site.



fabelizer says:

Jul 12, 2009. 6:23 AM [REPLY](#)

Great experiment, if I can only get it to compile in AVR Studio4! Does anyone have any tips on opening this project in AVR Studio 4? Even though Studio4 allows me to manually open the USI_I2C_Port.c file, when made into a project file, it complains: >Coordinator: None of the available object file readers can read the specified >object file. Please check the format of the object file. >Error loading object file I:\My Documents\AVR\Programs\I2C\USI_I2C\I_O>Port\USI_I2C_Port.c and stops. Suggestions? Thanks, -fab



fabelizer says:

Jul 12, 2009. 11:32 AM [REPLY](#)

Ok, I got it compiled. Apparently the USI_TWI_Master.c has to be put into the source files area in the project, then there were a few minor version related changes, but the code is now in the tn2313! -fab



p2otoole says:

Feb 2, 2009. 8:42 PM [REPLY](#)

I tried using the program TWI_I2C_EEPROM.c, with the circuit hooked up according to the schematic, of course, and got **(1) warning: return type of 'main' is not 'int'. (2) error: 'CLKPR' undeclared (first use in this function)**

Other than that, this program was what I have been looking for, and I know its gonna work for me,

Thank you



doctek says:

Feb 3, 2009. 10:59 AM [REPLY](#)

The warning about main seems to come with the gcc compiler. I think there's a way to get rid of it, but I haven't been successful at doing so. Hints I've found about how to do it haven't worked. It can be safely ignored, however. CLKPR is the register for the ATmega168. What processor are you compiling for? You may have to change the name.



p2otoole says:

Feb 3, 2009. 11:44 AM [REPLY](#)

I was trying to upload to an atmega8, which has the same pin layout and number of pins as an at-168. Also the EEPROM ckt that I have is a 24LC512; two strikes against me right there, I reckon. But I do have an atmega168 that I could use. I don't have the other ckts. that you discussed in this tutorial (port extender, etc) I would really like to get this program going, as it is one that I have been searching for; it will fit right into my plan.

Thank you



doctek says:

Feb 3, 2009. 1:34 PM [REPLY](#)

The parts you are interested in should work, but there's a couple of details you should know about. I downloaded the data sheets and here's what I found.

The ATmega8 does not have a CLKPR register. To set the clock prescaler to give you a 4MHz clock you have to blow fuse bits. With that done, the TWI looks like it would work the same.

The 24LC512 lives in the I2C address block from 50h to 57h. You pick the exact address by tying A0-A2 high or low to select the three least significant bits. That is, all lo = 50h, A0 hi, rest lo = 51h, etc.

But there's an additional wrinkle (isn't there always? ;)) The 24LC512 uses two address bytes to address the 64K memory space. Mostly not a problem, you just put in an extra address byte and make the buffer 1 byte bigger. Skip one additional byte when you try to read from the buffer as well. The block read and write size is 128 bytes, so that needs to change if you want to take advantage of that capability. Finally, the TWi_Start_Random_Read read routine must be modified since you have to send two address bytes instead of one address byte before sending the second START signal.

Unfortunately I don't have a 24LC512 to experiment with so I can't guarantee this will work, but I *think* all that needs to be done is to pass in an additional argument to the function telling it how long the address is. Right now that value is always 2, but if you add the argument then you could pass in either 2 or 3. If you want to contact me privately I'll send you modified code to try out. If it works, we could post it as an update.



BrianCatlin says:
Impressive. Well written!

Jan 4, 2009. 10:36 AM **REPLY**
