Gammon Forum

See www.mushclient.com/spam for dealing with forum spam. Please read the MUSHclient FAQ!

I2C - Two-Wire Peripheral Interface - for Arduino

Postings by administrators only.

Refresh page

Posted by	**Nick Gammon** Australia (22,538 posts) bio *Forum Administrator*
Date	Wed 26 Jan 2011 03:50 AM (UTC) Amended on Wed 08 Oct 2014 12:10 AM (UTC) by Nick Gammon

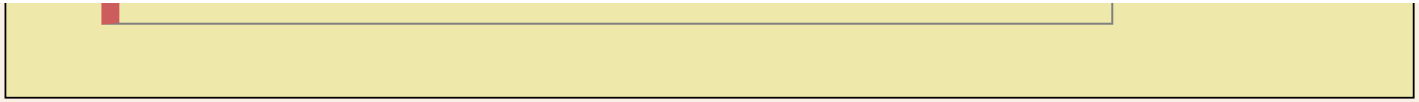
Message

> This page can be quickly reached from the link: http://www.gammon.com.au/i2c

Tip

A *summary* of everything shown below is available further down this page:

> This page can be quickly reached from the link: http://www.gammon.com.au/i2c-summary



This post describes how the I2C (Inter-Integrated Circuit, or "Two-Wire") interface works, with particular reference to the Arduino Uno which is based on the ATmega328P microprocessor chip. A lot of the details however will be of more general interest.

The Two-Wire interface is extremely useful for connecting multiple devices, as they can all share the same two pins (plus a ground return). This is because the devices are "addressable". Each device needs to have a unique address in the range 8 to 119. Address 0 is reserved as a "broadcast" address, addresses 1 to 7 are reserved for other purposes, and addresses 120 to 127 are reserved for future use.

Because of this you could have an LCD screen (say) at address 10, a keyboard at address 11, and so on.

More information about I2C at:

```
    http://en.wikipedia.org/wiki/I2c
```

More information about the Arduino Two-Wire interface at:

```
    http://www.arduino.cc/en/Reference/Wire
```

Other protocols

- Summary of protocols: http://www.gammon.com.au/forum/?id=10918

- 1-wire: http://www.gammon.com.au/forum/?id=10902
- Parallel interface: http://www.gammon.com.au/forum/?id=10903
- Serial (async): http://www.gammon.com.au/forum/?id=10894

- SPI (Serial Peripheral Interface): http://www.gammon.com.au/spi

Pinouts

On the Arduino Uno the pins you need are:

- **Analog** port 4 (A4) = SDA (serial data)
- **Analog** port 5 (A5) = SCL (serial clock)

On the Arduino **Mega**, SDA is **digital** pin 20 and SCL is **digital** pin 21 (they are marked SDA and SCL on the board itself).

On the Arduino **Leonardo**, the SDA and SCL pins are separate pins, so marked, on the board (next to AREF). They are also connected to D2 (SDA) and D3 (SCL).

These pins may require pull-up resistors (that is, connect them to +5v via something like a 4.7K resistor each).

The Atmega328 is configured to use internal pull-up resistors which may be adequate for short cable runs. **Warning:** for multiple I2C devices, or longer cable runs, the 4.7K pull-up resistor (for each line) is **recommended**.

Also see further down in this thread for some screen-shots of the effect of using different pull-up resistors.

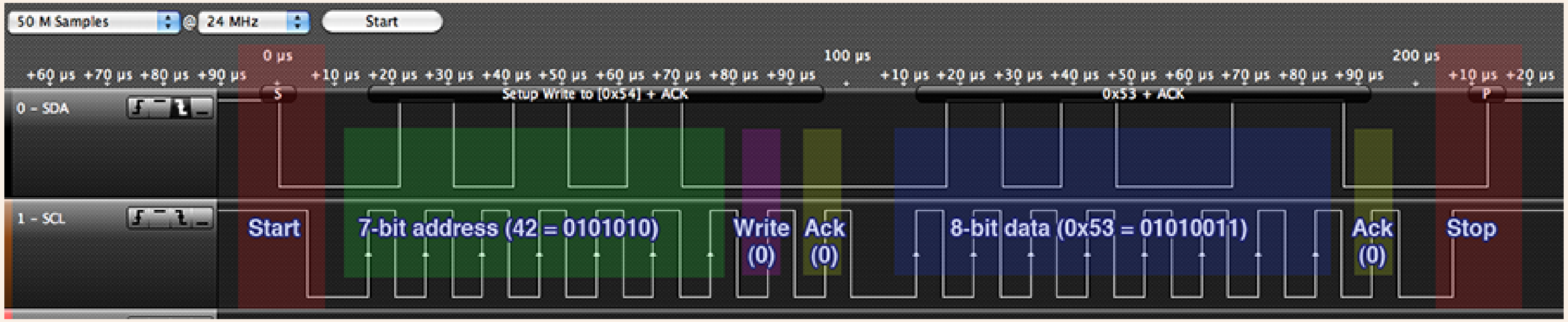
Of course, you also need to connect the GND (ground) pins to complete the circuit.

The pins should be connected **together** (that is, pin 4 to pin 4, and pin 5 to pin 5, if you are connecting Arduino Unos together). This is because the pull-up resistors keep the lines high until one of the devices wants to use it by pulling a line low. In other words, you don't swap pins (like you do with serial communications, where you connect Tx to Rx and vice-versa).

Note also that the Atmega specifies a maximum bus capacitance of 400 pf, so that would rule out long cable runs.

Sending data

Let's start with an image - this is a screenshot taken with a logic analyser. It shows the character "S" (0x53) being sent from the Arduino to the device at address 42.



From the above graphic note the following points of interest:

- The transmission starts with the "Start condition" (labelled Start). This is when SDA (serial data) is pulled low while SCL (serial clock) stays high.
- The 7-bit address of the required slave is then transmitted, most significant bit first. In this case it was 42 (0x2A or 0b0101010). The logic analyser reports the address as being 0x54 but that is really 0x2A (this is, 42) shifted left one bit, so that the "write" bit (0) is in the least-significant bit place.
- Then the read/write bit is transmitted. 0 means write (master to slave) and 1 means read (slave to master).
- The master then waits for the slave to pull the SDA line low which is an ACK (acknowledge) that a slave of that address exists and is ready to receive data. If there is no slave connected and powered on, or it does not have the required address, then it will ignore the address, leaving the SDA line high (by the pull-up resistor). This counts as a NAK (negative acknowledgement). This can be tested for in the software.
- Then the data byte (0x53 in this case) is transmitted, most significant bit first.
- Again, after the 8 bits of data, the master checks that the slave acknowledges by pulling the SDA line low. Thus, each byte is acknowledged.

- More data bytes could be transmitted, but are not shown here.
- The transmission is ended by the "Stop condition" (labelled Stop) which is sent by releasing the SDA line to allow it to be pulled up while SCL stays high.

The code to produce this (in Arduino's C++ language) was:

```
// Written by Nick Gammon
// February 2012

#include <Wire.h>

const byte SLAVE_ADDRESS = 42;
const byte LED = 13;

void setup ()
  {
  Wire.begin ();
  pinMode (LED, OUTPUT);
  }  // end of setup

void loop ()
  {
  for (byte x = 2; x <= 7; x++)
    {
    Wire.beginTransmission (SLAVE_ADDRESS);
    Wire.write (x);
    if (Wire.endTransmission () == 0)
      digitalWrite (LED, HIGH);
    else
      digitalWrite (LED, LOW);

    delay (200);
    }  // end of for loop
  }  // end of loop
```

[EDIT] Updated February 2012 to allow for version 1.0 of the Arduino IDE. This uses Wire.write rather Wire.send.

Tip:

Wire.endTransmission returns zero if the transmission was a success. It is a good idea to check that you got zero, otherwise you are sending "blind". If you do not get zero maybe you have the wrong slave address. Try running the "I2C scanner" described later in this thread.

The code above uses address 42 for the slave, and also uses the LED on pin 13 (which is standard on the Arduino Uno) to confirm visually that the transmission took place. If it succeeded then the LED is turned on, otherwise off. You can see this in operation by noting that the LED is only on if a slave of address 42 is connected and responding.

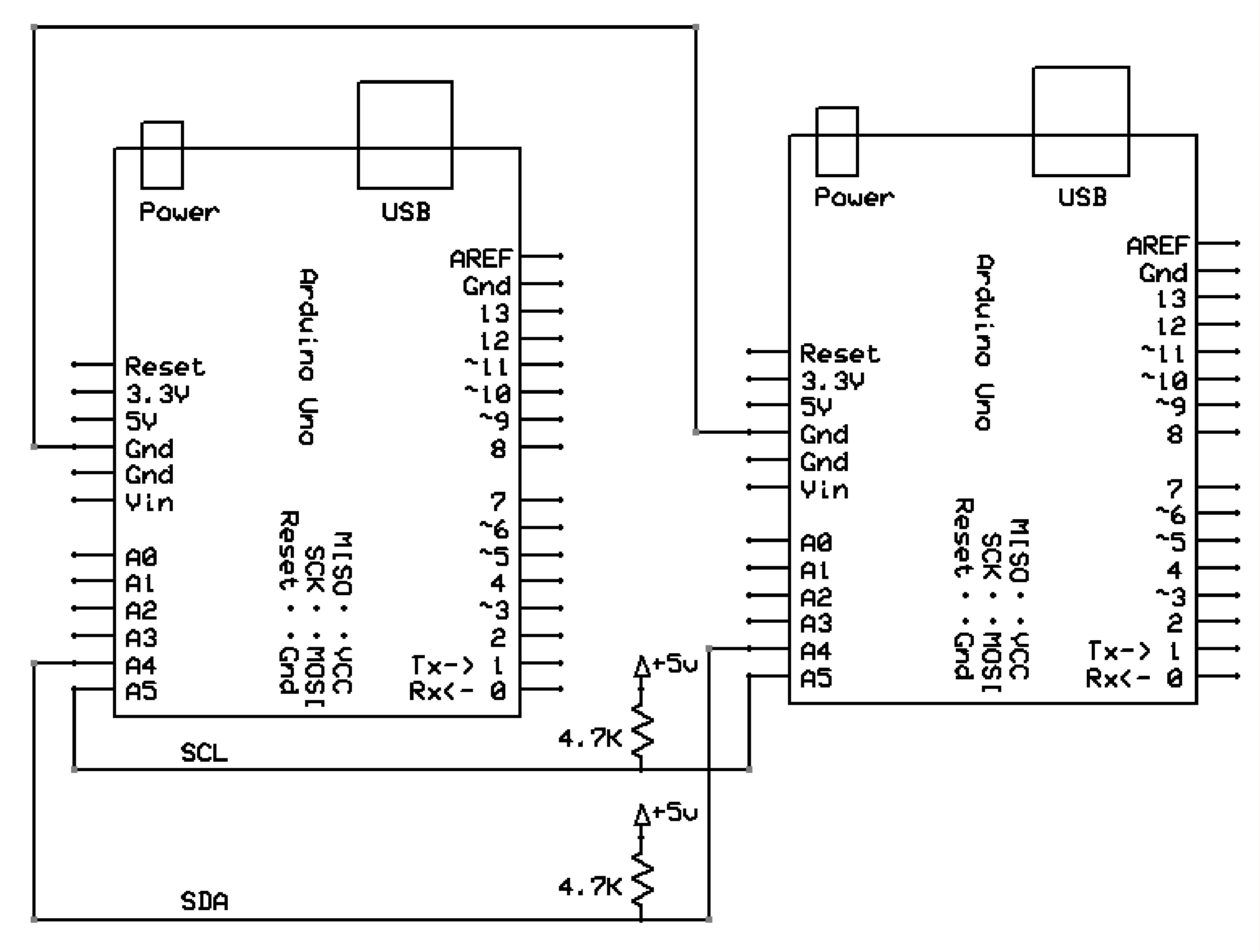
The code for the slave in my case was:

```
// Written by Nick Gammon
// February 2012

#include <Wire.h>

const byte MY_ADDRESS = 42;

void setup ()
  {
  Wire.begin (MY_ADDRESS);
  for (byte i = 2; i <= 7; i++)
    pinMode (i, OUTPUT);
  // set up receive handler
  Wire.onReceive (receiveEvent);
  }  // end of setup

void loop()
  {
  // nothing in main loop
  }

// called by interrupt service routine when incoming data arrives
void receiveEvent (int howMany)
  {
  for (int i = 0; i < howMany; i++)
    {
    byte c = Wire.read ();
    // toggle requested LED
    if (digitalRead (c) == LOW)
```

```
      digitalWrite (c, HIGH);
    else
      digitalWrite (c, LOW);
    }  // end of for loop
}  // end of receiveEvent
```

The slave code has nothing in the main loop, because the two-wire interface generates an interrupt when data arrives. This was displayed on LEDs plugged into pins D2 through to D7 using an appropriate resistor in series with each one (eg. 470 ohms).

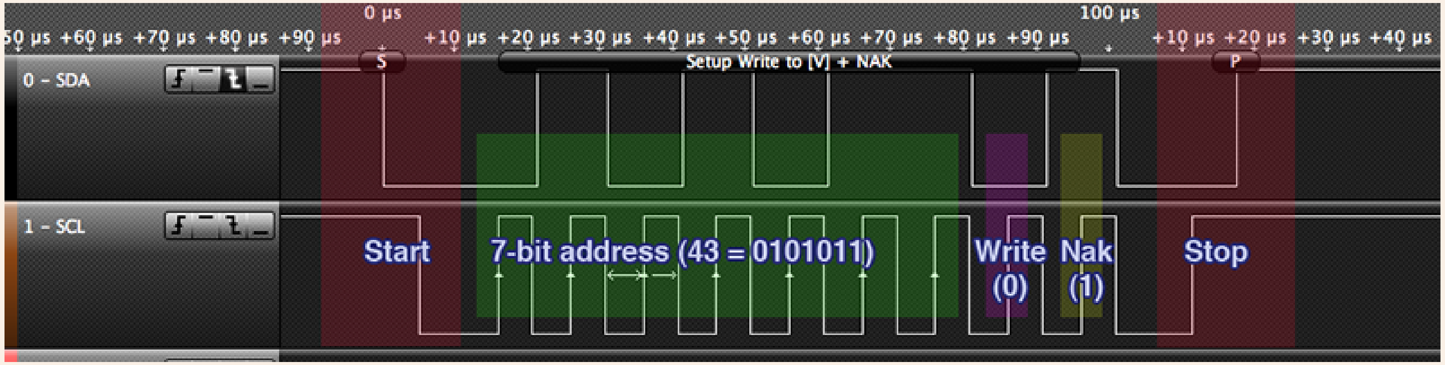
[EDIT] Updated February 2012 to allow for version 1.0 of the Arduino IDE. This uses Wire.read rather Wire.receive.



Example of connecting two Unos together to communicate via I2C

ACK or NAK

The above graphic shows what happens if the slave device responds. Below is what happens if it doesn't. In this case I changed the address from 42 to 43, so that the slave would ignore the attempt to communicate with it.


0 µs
100 µs
50 µs +60 µs +70 µs +80 µs +90 µs
+10 µs +20 µs +30 µs +40 µs +50 µs +60 µs +70 µs +80 µs +90 µs
+10 µs +20 µs +30 µs +40 µs
0 - SDA
1 - SCL
S
Setup Write to [V] + NAK
P
Start
7-bit address (43 = 0101011)
Write (0)
Nak (1)
Stop

This condition can be tested for in Wire.endTransmission(). That returns zero on success and various error codes on failure.

Timing

The timing for the entire transaction above (address byte and data byte) was that from the start condition to the stop condition took 0.2125417 milliseconds. Thus at that rate you could send around 4705 bytes per second. However that is a bit inefficient because half of it was the address. The time taken to send one byte was 96.3333 microseconds, which translates to 10,380 bytes per second. This is reasonable enough if you are just driving a LCD display or similar.

You can increase the clock speed by adding this line after "Wire.begin ();", like this:

```
Wire.begin ();
TWBR = 12;
```

With that in place, the speeds are 4 times as fast. So a single byte then takes around 28 microseconds.

Sample TWBR values

The formula for converting TWBR into frequency is:

```
freq = clock / (16 + (2 * TWBR * prescaler))
```

The default prescaler is 1, and the default value for TWBR (on the Uno etc.) is 72. Thus:

freq = 16000000 / (16 + 144) = 100000

```
TWBR   prescaler   Frequency

  12       1       400   kHz  (the maximum supported frequency)
  32       1       200   kHz
  72       1       100   kHz  (the default)
 152       1        50   kHz
  78       4        25   kHz
 158       4        12.5 kHz
```

To set the prescaler to 4 you need to set the bit TWPS0 in TWSR, so for example to have a clock of 12.5 kHz:

```
  Wire.begin ();
  TWBR = 158;
  TWSR |= bit (TWPS0);
```

[EDIT] Updated 17 June 2012 to add the table of TWBR values and related frequencies.

Buffer Length

It isn't currently mentioned in the documentation, but the internal buffer used for I2C communications is 32 bytes. That means you can transfer a maximum of 32 bytes in one transaction.

It also isn't particularly clear, but the functions Wire.beginTransmission and Wire.write don't actually send anything. They simply prepare an internal buffer (with a maximum length of 32 bytes) for the transmission. This is so that the hardware can then clock out the data at a high rate. For example:

```
Wire.beginTransmission (SLAVE_ADDRESS);  // prepare internal buffer
Wire.write ("hello world");              // put data into buffer
byte result = Wire.endTransmission ();   // transmission occurs here
```

After calling Wire.endTransmission, if zero was returned, you know the call was a success.

Communicating with other devices

So all this I2C stuff is great, you can use just two wires (plus ground) to talk to up to 119 devices. But what if you have a device (like a keypad) that doesn't support I2C? Well, the simple answer is that you can use something that does as an interface. For example, a second Arduino. My examples above do just that, using one as the master and second one to display text on LEDs.

In fact, the "master" can also act as a slave, since you can have multiple masters on one wire. The example below shows how you can send data from one Arduino to another, whilst waiting for information to be sent from the second back to the first.

Master

```
// Written by Nick Gammon
// February 2012

#include <Wire.h>

const byte MY_ADDRESS = 25;
const byte SLAVE_ADDRESS = 42;
const byte LED = 13;

void setup()
  {
  Wire.begin (MY_ADDRESS);
  Wire.onReceive (receiveEvent);
  pinMode (LED, OUTPUT);
  }  // end of setup

void loop()
  {

  for (int x = 2; x <= 7; x++)
    {
    Wire.beginTransmission (SLAVE_ADDRESS);
    Wire.write (x);
    Wire.endTransmission ();
    delay (200);
    }  // end of for

  }  // end of loop

void receiveEvent (int howMany)
 {
  for (int i = 0; i < howMany; i++)
    {
    byte b = Wire.read ();
    digitalWrite (LED, b);
    }  // end of for loop
} // end of receiveEvent
```

[EDIT] Updated February 2012 to allow for version 1.0 of the Arduino IDE. This uses Wire.read rather Wire.receive, and Wire.write rather than Wire.send.

Slave

```
// Written by Nick Gammon
// February 2012

#include <Wire.h>

const byte MY_ADDRESS = 42;
const byte OTHER_ADDRESS = 25;
```

```
void setup ()
  {
  Wire.begin (MY_ADDRESS);
  for (byte i = 2; i <= 7; i++)
    pinMode (i, OUTPUT);
  Wire.onReceive (receiveEvent);
  }  // end of setup

void loop()
  {
  int v = analogRead (0);
  Wire.beginTransmission (OTHER_ADDRESS);
  Wire.write (v < 512);
  Wire.endTransmission ();
  delay (20);
  }  // end of loop

// called by interrupt service routine when incoming data arrives
void receiveEvent (int howMany)
 {
  for (int i = 0; i < howMany; i++)
    {
    byte c = Wire.read ();
    // toggle requested LED
    if (digitalRead (c) == LOW)
      digitalWrite (c, HIGH);
    else
      digitalWrite (c, LOW);
    }  // end of for loop
  }  // end of receiveEvent
```

[EDIT] Updated February 2012 to allow for version 1.0 of the Arduino IDE. This uses Wire.read rather Wire.receive, and Wire.write rather than Wire.send.

The examples above use addresses 25 and 42. Each device "registers" its own address with:

```
Wire.begin(MY_ADDRESS);
```

It also registers an interrupt handler to be called when the other end wants to send some data:

```
Wire.onReceive(receiveEvent);
```

Now one can send a stream of data to the second, while the second sends back a message if the value on analog port 0 drops below 512. Thus we have two-way communication.

Dedicated I2C expanders

Another (cheaper) approach is to simply use "16-port I/O expander" chips (such as the MCP23017). These connect to an I2C bus and provides 16 digital ports which can be configured as inputs or outputs depending on the chip. Most of these allow you to set some address bits (eg. via jumpers) so you might use 8 of the expanders (if you needed to!) connected to a single I2C line, with addresses like 000, 001, 010 etc.

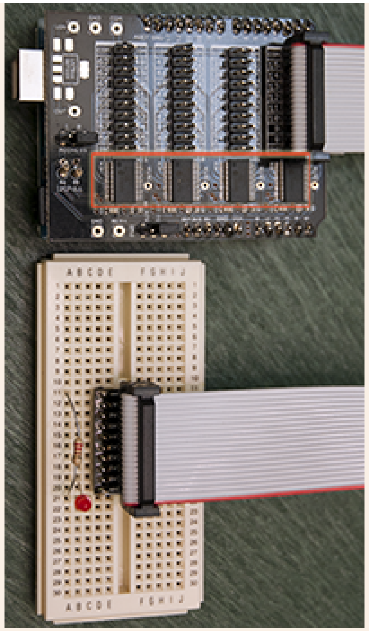
I have seen these for sale at around $US 1.20, which is pretty cheap.

See this post for an example of driving an LCD screen with an I/O expander chip:

http://www.gammon.com.au/forum/?id=10940

Alternatively, the Atmega328 chip on its own is only $6, so the suggested approach above of using another microprocessor isn't all that expensive an option.

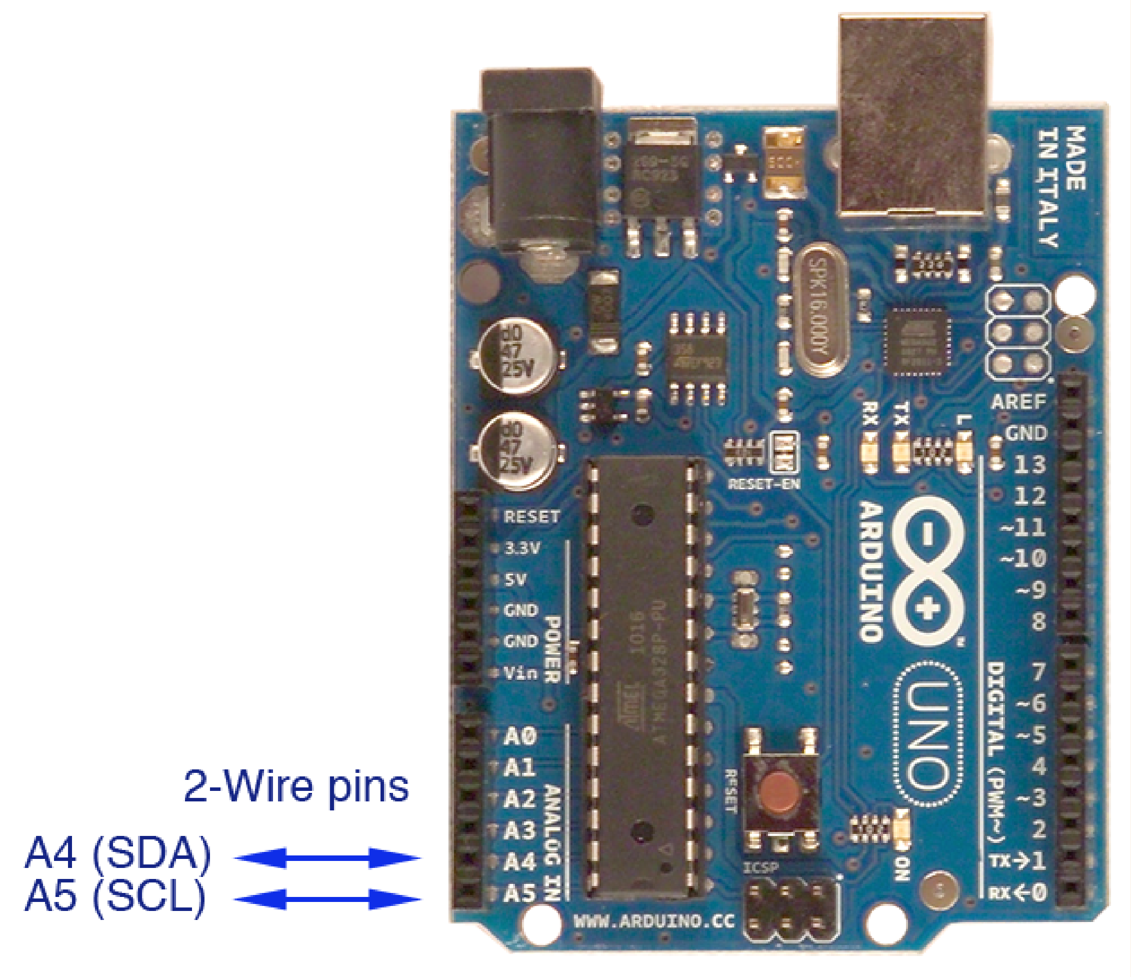
Here is an example of an expander board (the "Centipede"). This uses 4 x MCP23017 chips to give a whopping 64 I/O ports, at the expense of only two pins on the Arduino. This board comes with headers that let you connect cables and run to other boards where you might have measuring devices, or output devices.



Arduino Library

```
#include <Wire.h>
```

Pinouts on the Arduino Uno



Note that more recent Arduino Unos (R3 boards and above) have dedicated (extra) I2C pins (near the USB connector) as shown on the photo on this page: http://www.gammon.com.au/uno

That is in *addition* to being able to use A4/A5 (they are connected together).

[EDIT] Updated on 1 February 2013 to mention more reserved I2C addresses (1 to 7).

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

Posted by **Nick Gammon** Australia (22,538 posts) bio *Forum Administrator*

Date Reply #1 on Sun 06 Feb 2011 02:22 AM (UTC)

Amended on Thu 28 Feb 2013 04:52 AM (UTC) by Nick Gammon

Message

Broadcasting

A bit of experimenting has shown that the broadcast capability is not turned on by default. Broadcasting is done by sending to "address 0". If so configured, then all slaves receive the message, rather than a single slave. This could be useful where one device wanted to notify all other devices of an event, without having to address them all individually. This could save a substantial amount of time, and also save needing to know the addresses of all the other devices that want to know about this event.

In the "slave" example below I have enabled receiving broadcasts by setting the low-order bit of the TWAR register. Omit that line if you want a slave to ignore broadcasts. This is documented in the Atmega manual on page 234 (at present) in section 21.7.3 (Slave Receiver Mode).

The examples send a 16-bit number from the master to one or all clients. The master splits the number into two bytes. An alternative approach would be to use the "send" variant which takes a pointer and a length.

Master:

```
// Written by Nick Gammon
// February 2011

#include <Wire.h>

const byte MY_ADDRESS = 25;      // me
const byte SLAVE_ADDRESS = 42;   // slave #42

void setup ()
  {
  Wire.begin (MY_ADDRESS);  // initialize hardware registers etc.
  }  // end of setup

void loop()
  {
  unsigned int value = 1234;  // ie. 0x04 0xD2

  Wire.beginTransmission (0);  // broadcast to all
  Wire.write (highByte (value));
  Wire.write (lowByte (value));
  byte err = Wire.endTransmission  ();  // non-zero means error

  delay (100);  // wait 0.1 seconds
  }   // end of loop
```

[EDIT] Updated February 2012 to allow for version 1.0 of the Arduino IDE. This uses Wire.write rather Wire.send.

In this example the master has an address (25) so it could actually be a slave too. In that case one of the other Arduinos could send to address 25. Of course you then need to add the receiveEvent handler to the "master" as well.

Slave:

```
// Written by Nick Gammon
// February 2011

#include <Wire.h>

const byte MY_ADDRESS = 42;   // me
const byte LED = 13;          // LED is on pin 13

byte ledVal = 0;

void receiveEvent (int howMany)
 {

 // we are expecting 2 bytes, so check we got them
 if (howMany == 2)
   {
```

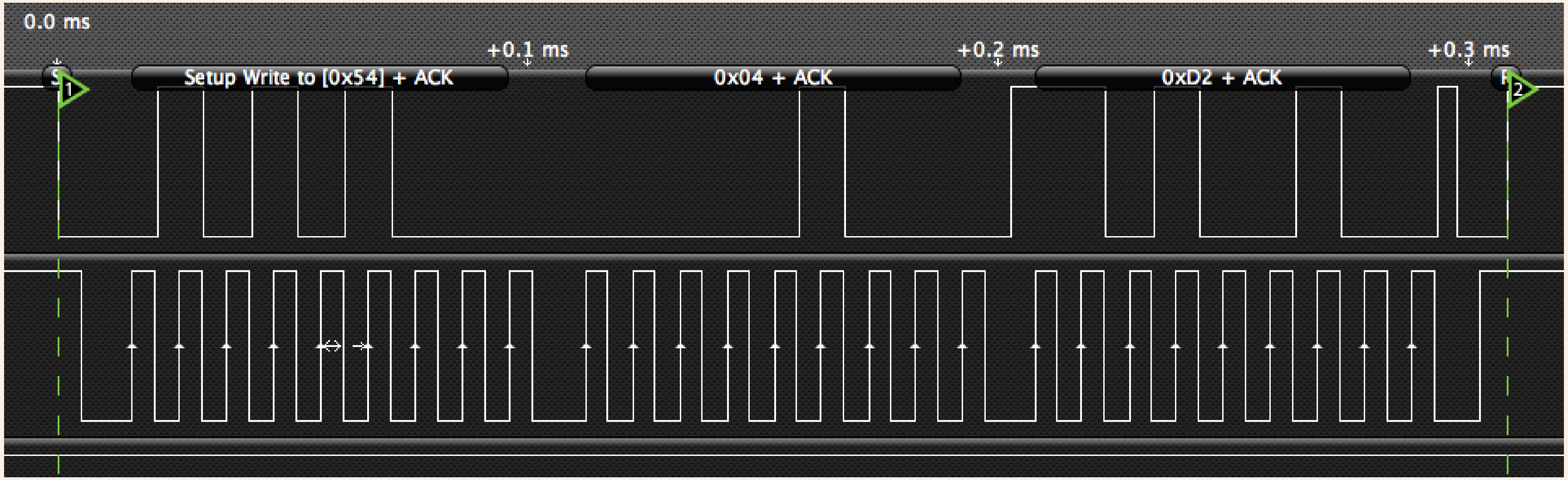
```
   int result;

   result = Wire.read ();
   result <<= 8;
   result |= Wire.read ();

   // do something with result here ...

   // for example, flash the LED

   digitalWrite (LED, ledVal ^= 1);   // flash the LED

   }  // end if 2 bytes were sent to us

  // throw away any garbage
  while (Wire.available () > 0)
    Wire.read ();

  }  // end of receiveEvent

void setup ()
  {
  Wire.begin (MY_ADDRESS);  // initialize hardware registers etc.

  TWAR = (MY_ADDRESS << 1) | 1;  // enable broadcasts to be received

  Wire.onReceive(receiveEvent);  // set up receive handler
  pinMode(LED, OUTPUT);          // for debugging, allow LED to be flashed
  }  // end of setup

void loop ()
  {
  }  // end of loop
```

[EDIT] Updated February 2012 to allow for version 1.0 of the Arduino IDE. This uses Wire.read rather Wire.receive.

Timing

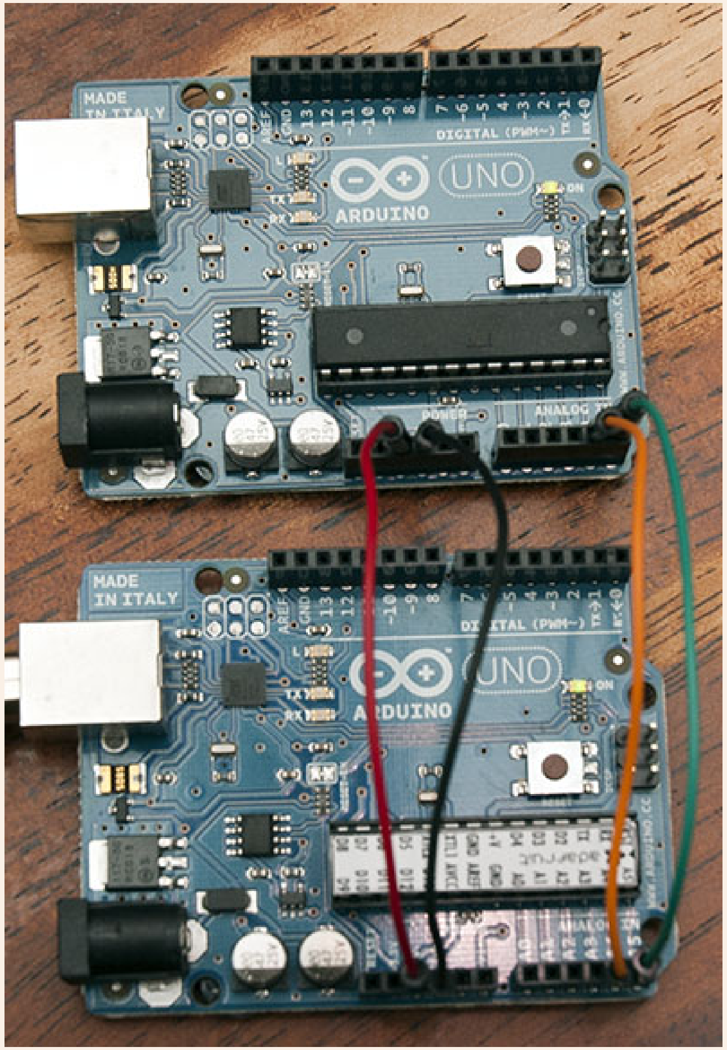
The screenshot below shows that the time taken to address the slave, and send two bytes (a 16-bit number) was just over 0.3 milliseconds (in fact: 0.30775 ms from the start condition to the stop condition).

Of course, if you used the broadcast address you could send to 100 slaves in the same time.



Connections

The photo below shows the two Arduinos wired together for the above test. Note that the SDA (A4) and SCL (A5) ports are wired together. The other two wires provide a ground return, and power for the second Arduino.



- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

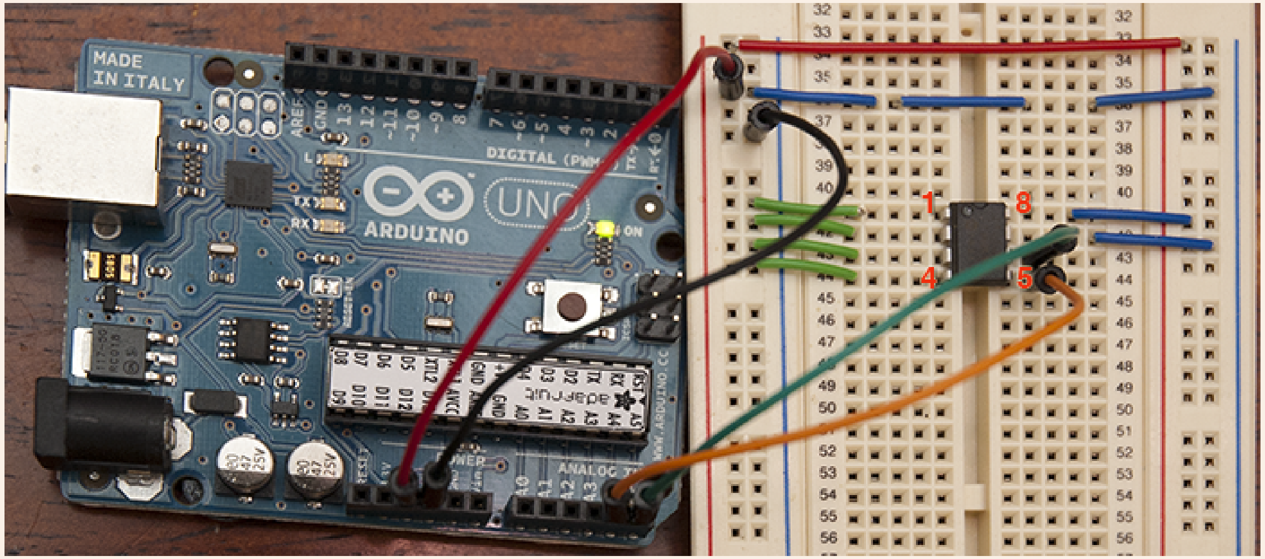
Posted by	**Nick Gammon** Australia (22,538 posts) bio *Forum Administrator*
Date	Reply #2 on Tue 08 Feb 2011 06:10 AM (UTC) Amended on Sun 22 Feb 2015 08:39 PM (UTC) by Nick Gammon

Message

Example of I2C to access EEPROM

One use of I2C is to easily connect to EEPROM (Electrically Erasable Programmable Read-Only Memory). This could be used to save data (eg. for logging purposes) because the data in it is retained even after power is removed. Effectively, it is like a "flash" drive or USB stick.

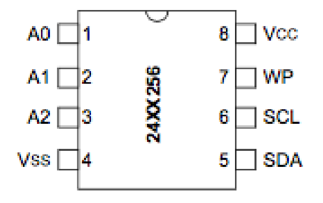
The photo below illustrates doing this using a 24LC256 chip which can be purchased for around $US 2. This stores 256 K bits (32 K bytes).



Wiring is pretty straightforward. The first four pins (on the left) are all connected to ground. Pins 1, 2 and 3 are the address-select lines, so if you had multiple identical chips you could jumper some of those to +5v to turn them from a 0 to a 1. With all connected to ground the chip's address is 0x50. If you connected pin 1 to +5v (pin 1 being A0 address select) then the chip's address would be 0x51, and so on for all 8 combinations of addresses.

Pin 8 is connected to +5v for the "power supply". Pin 7 is write-protect. Connect that to ground to enable writing. The remaining two pins are the SCL (pins 6) and SDA (pin 5). SCL connects to A5 on the Arduino, and SDA connects to A4, as shown. Pin 4 is the chip's ground. To summarize:

- Pin 1 : A0 (address select)
- Pin 2 : A1
- Pin 3 : A2
- Pin 4 : GND (ground)
- Pin 5 : SDA (serial data)
- Pin 6 : SCL (serial clock)
- Pin 7 : WP (write protect: low to enable writing)
- Pin 8 : +5v (power)



With this set-up we can now do a trial program to write to the chip and read the data back:

```
// Written by Nick Gammon
// February 2011

// Note: maximum address is 0x7FFF (32 Kbytes)

#include <Wire.h>

const byte rom = 0x50;    // Address of 24LC256 eeprom chip

void setup (void)
  {
  Serial.begin (9600);   // debugging

  Wire.begin ();

  // test: write the number 123 to address 0x1234
  writeEEPROM (rom, 0x1234, 123);

  // read back to confirm
  byte a = readEEPROM (rom, 0x1234);

  Serial.println (a, DEC);  // display to confirm

  // test: write a string to address 0x1000
  byte hello [] = "Hello, world!";

  writeEEPROM (rom, 0x1000, hello, sizeof hello);

  // read back to confirm
  byte test [sizeof hello];
  byte err = readEEPROM (rom, 0x1000, test, sizeof test);

  Serial.println ((char *) test);  // display to confirm
  }  // end of setup

void loop() {}  // no main loop


// write len (max 32) bytes to device, returns non-zero on error
//  return code: 0xFF means buffer too long
//             other: other error (eg. device not present)

// Note that if writing multiple bytes the address plus
//  length must not cross a 64-byte boundary or it will "wrap"

byte writeEEPROM (byte device, unsigned int addr, byte * data, byte len )
  {
  byte err;
  byte counter;

  if (len > BUFFER_LENGTH)  // 32 (in Wire.h)
    return 0xFF;  // too long

  Wire.beginTransmission(device);
  Wire.write ((byte) (addr >> 8));    // high order byte
  Wire.write ((byte) (addr & 0xFF));  // low-order byte
  Wire.write (data, len);
  err = Wire.endTransmission ();

  if (err != 0)
    return err;  // cannot write to device

  // wait for write to finish by sending address again
  //  ... give up after 100 attempts (1/10 of a second)
  for (counter = 0; counter < 100; counter++)
    {
```

```
    delayMicroseconds (300);  // give it a moment
    Wire.beginTransmission (device);
    Wire.write ((byte) (addr >> 8));    // high order byte
    Wire.write ((byte) (addr & 0xFF));  // low-order byte
    err = Wire.endTransmission ();
    if (err == 0)
      break;
    }

  return err;

  } // end of writeEEPROM

// write one byte to device, returns non-zero on error
byte writeEEPROM (byte device, unsigned int addr, byte data )
  {
  return writeEEPROM (device, addr, &data, 1);
  } // end of writeEEPROM

// read len (max 32) bytes from device, returns non-zero on error
//  return code: 0xFF means buffer too long
//               0xFE means device did not return all requested bytes
//             other: other error (eg. device not present)

// Note that if reading multiple bytes the address plus
//  length must not cross a 64-byte boundary or it will "wrap"

byte readEEPROM (byte device, unsigned int addr, byte * data, byte len )
  {
  byte err;
  byte counter;

  if (len > BUFFER_LENGTH)  // 32 (in Wire.h)
    return 0xFF;  // too long

  Wire.beginTransmission (device);
  Wire.write ((byte) (addr >> 8));    // high order byte
  Wire.write ((byte) (addr & 0xFF));  // low-order byte
  err = Wire.endTransmission ();

  if (err != 0)
    return err;  // cannot read from device

  // initiate blocking read into internal buffer
  Wire.requestFrom (device, len);

  // pull data out of Wire buffer into our buffer
  for (counter = 0; counter < len; counter++)
    {
    data [counter] = Wire.read ();
    }

  return 0;  // OK
  }  // end of readEEPROM

// read one byte from device, returns 0xFF on error
byte readEEPROM (byte device, unsigned int addr )
  {
  byte temp;

  if (readEEPROM (device, addr, &temp, 1) == 0)
    return temp;

  return 0xFF;

  }  // end of readEEPROM
```

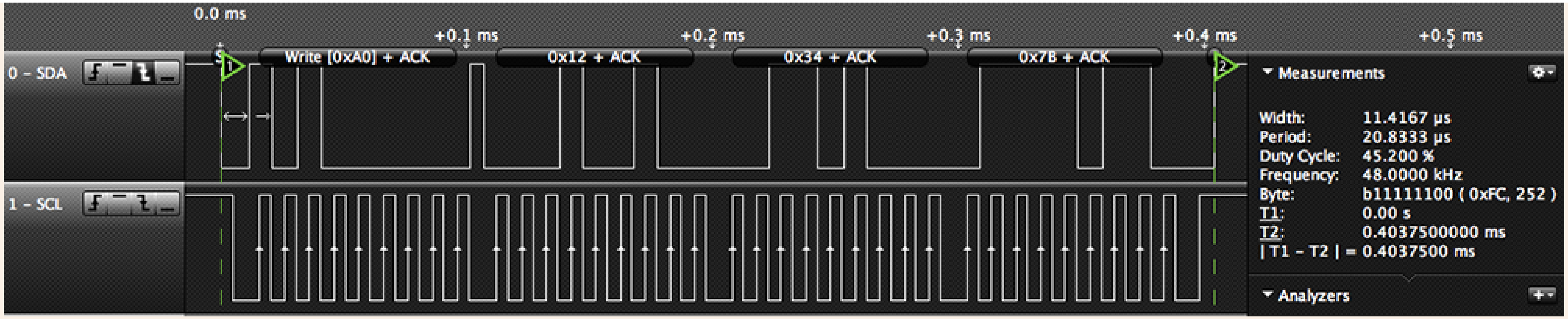
The writeEEPROM function writes a single byte, or a string of up to 32 bytes, to the specified address.

Then it attempts to address the device repeatedly, until it gets an ACK. This is "acknowledge polling" as described in the data sheet for the 24LC256 chip. It says that while the chip is busy actually writing to memory it will NAK (ignore, actually) any attempts to address it. Thus, when we get back an ACK we know the data is written and we can safely proceed. The loop contains a 300 microsecond delay to avoid flooding the I2C wire with failed attempts to address the chip.

According to the chip specification, the maximum delay a write should take is 5 milliseconds.

In practice I found during my test that it took 3.47 ms for the chip to become ready for another write.

Write timing



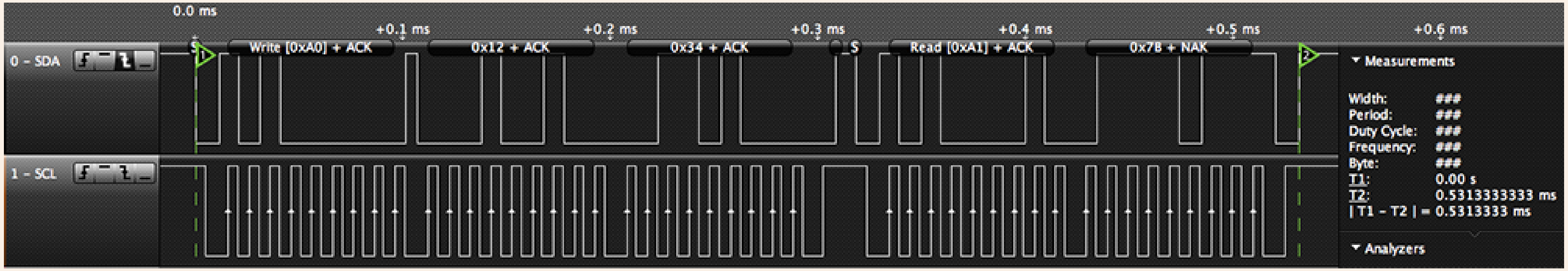
```
Width:        11.4167 µs
Period:       20.8333 µs
Duty Cycle:   45.200 %
Frequency:    48.0000 kHz
Byte:         b11111100 ( 0xFC, 252 )
T1:           0.00 s
T2:           0.4037500000 ms
| T1 - T2 | = 0.4037500 ms
```

This shows that it took 0.404 ms (404 microseconds) to write a single byte. It would be more efficient to write more bytes at a time (the chip can handle 64 bytes, and the I2C library can buffer 32 bytes, thus up to 32 bytes would be reasonable). The second form of the writeEEPROM function writes an array of bytes, so you could write a structure (eg. time and temperature).

A test shows that writing 4 bytes only took 0.690 ms, which is not much more than writing one byte. After writing the 4 bytes, it still only took just over 3 ms for the chip to be ready to write to again.

It would be sensible then to buffer up writes if possible, to spread out the settling time of 3 ms over more bytes.

Read timing



```
Width:        ###
Period:       ###
Duty Cycle:   ###
Frequency:    ###
Byte:         ###
T1:           0.00 s
T2:           0.5313333333 ms
| T1 - T2 | = 0.5313333 ms
```

This shows the time taken to read a byte back. First we send out the requested address (hex 1234) and then go into "read" mode. The device sends back the contents of the requested address (in this case 0x7B or 123 in decimal), to which we reply NAK to tell the device that we don't need any more data.

In this case it took 0.531 ms (531 microseconds) to get a single byte back. A further test shows that it only took 0.820 ms to read back 4 bytes, so again it is faster to deal in more than one byte if possible.

The second form of the readEEPROM function reads an array of bytes, so you could read back a structure (eg. time and temperature).

Timing summary

If reading/writing single bytes:

Writing: Write plus settling time: 3.87 ms.

Thus, you could write 258 bytes per second.

Reading: Read time: 0.531 ms.

Thus, you could read 1883 bytes per second.

This might seem a bit slow, but after all writing a single byte to a hard disk is also not very efficient.

If reading/writing four bytes:

Writing: Write plus settling time: 4.16 ms.

Thus, you could write 962 bytes per second.

Reading: Read time: 0.820 ms.

Thus, you could read 4878 bytes per second.

Caution about multiple bytes

According to the specs, the chip is organized into "pages" of 64 bytes. Attempts to read or write more than one byte at a time will cause an internal address counter to wrap around on a page boundary. So for example if you were writing to address 62, and wrote four bytes, you would actually write to addresses 62, 63, 0 and 1.

Thus if you are planning to save time by doing multiple writes you would need to make sure that your writes don't wrap page boundaries. One way would be to write in multiples of 4, for example. That way you would cross over the 64-byte boundaries between writes, not in the middle of one.

Life of chip

The spec for the 24LC256 says that the chip should be able to sustain over a million erase/write cycles. This should certainly be adequate for data logging or saving game data.

However for situations like where you were planning to save a player's game status, it would probably be wise to only do that every minute or so (rather than every second) as otherwise you might wear out the chip prematurely.

Another approach would be to save important events (eg. if the player gained loot), whilst keeping fairly unimportant events (like changing rooms) to be saved less frequently.

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

Posted by **Nick Gammon** Australia (22,538 posts) bio *Forum Administrator*

Date Reply #3 on Tue 15 Feb 2011 10:47 PM (UTC)

Message

Using I2C to drive an LCD display

See this forum post for a lengthy description of how I connected up a 128x64 pixel LCD display using I2C to communicate with it:

http://www.gammon.com.au/forum/?id=10940

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

Posted by **Nick Gammon** Australia (22,538 posts) bio *Forum Administrator*

Date Reply #4 on Thu 17 Feb 2011 08:51 PM (UTC)

Amended on Fri 28 Aug 2015 04:25 AM (UTC) by Nick Gammon

Message

Request/response

These examples illustrate how a master can request data from a slave (this part is similar to how you communicate with the EEPROM chip).

First, the master, which asks the slave for information:

Master

```
// Written by Nick Gammon
// Date: 18th February 2011

#include <Wire.h>

const int SLAVE_ADDRESS = 42;

// various commands we might send
enum {
    CMD_ID = 1,
    CMD_READ_A0  = 2,
    CMD_READ_D8 = 3
    };

void sendCommand (const byte cmd, const int responseSize)
  {
  Wire.beginTransmission (SLAVE_ADDRESS);
  Wire.write (cmd);
  Wire.endTransmission ();

  Wire.requestFrom (SLAVE_ADDRESS, responseSize);
  }  // end of sendCommand
```

```
void setup ()
  {
  Wire.begin ();
  Serial.begin (9600);  // start serial for output

  sendCommand (CMD_ID, 1);

  if (Wire.available ())
    {
    Serial.print ("Slave is ID: ");
    Serial.println (Wire.read (), DEC);
    }
  else
    Serial.println ("No response to ID request");

  }  // end of setup

void loop()
  {
  int val;

  sendCommand (CMD_READ_A0, 2);
  val = Wire.read ();
  val <<= 8;
  val |= Wire.read ();
  Serial.print ("Value of A0: ");
  Serial.println (val, DEC);

  sendCommand (CMD_READ_D8, 1);
  val = Wire.read ();
  Serial.print ("Value of D8: ");
  Serial.println (val, DEC);

  delay (500);
  }  // end of loop
```

In this case we ask the slave for its "ID". Then ask for a sensor reading from A0, and then D8. For simplicity I haven't checked if we actually got a response from the slave in the main loop. In practice you would check because otherwise the slave might be disconnected and you wouldn't know.

Tip:

Wire.requestFrom does not return until the data is available (or if it fails to become available). Thus it is **not necessary** to do a loop waiting for Wire.available after doing Wire.requestFrom.

For example:

```
Wire.requestFrom (SLAVE_ADDRESS, responseSize);
while (Wire.available () < responseSize) { }  // <---- not necessary
```

~~In fact, Wire.requestFrom returns the number of bytes obtained, so a better method is to do this:~~

```
byte bytesReceived = Wire.requestFrom (SLAVE_ADDRESS, responseSize);
if (bytesReceived < responseSize)
  {
  // handle error - we did not get everything we wanted
  }
else
  {
  // data received, now use Wire.read to obtain it.
  }
```

Correction. See Reply #10 below. This is not correct. Wire.requestFrom actually returns the number of bytes requested, or zero.

Therefore the only reasonable way of doing the Wire.requestFrom is:

```
if (Wire.requestFrom (SLAVE_ADDRESS, responseSize) == 0)
  {
  // handle error - no response
  }
else
  {
  // data received, now use Wire.read to obtain it.
  }
```

Plus, as suggested below, some sort of error checking to make sure that all of the data was received correctly.

The slave is a bit more interesting. It has to be able to reply to any request without even knowing the address of the device making the request.

It does this by setting up an interrupt handler (by calling Wire.onRequest). This gets called any time a master wants a response. We also need the Wire.onReceive handler, to get the initial "command" - that is, we need to know **what** data is wanted.

Slave

```
// Written by Nick Gammon
// Date: 18th February 2011

#include <Wire.h>

const byte MY_ADDRESS = 42;

// various commands we might get

enum {
    CMD_ID = 1,
    CMD_READ_A0  = 2,
    CMD_READ_D8 = 3
    };

char command;

void setup()
  {
  command = 0;

  pinMode (8, INPUT);
  digitalWrite (8, HIGH);  // enable pull-up
  pinMode (A0, INPUT);
  digitalWrite (A0, LOW);  // disable pull-up

  Wire.begin (MY_ADDRESS);
  Wire.onReceive (receiveEvent);  // interrupt handler for incoming messages
  Wire.onRequest (requestEvent);  // interrupt handler for when data is wanted

  }  // end of setup


void loop()
  {
  // all done by interrupts
  }  // end of loop

void receiveEvent (int howMany)
  {
  command = Wire.read ();  // remember command for when we get request
  } // end of receiveEvent


void sendSensor (const byte which)
  {
  int val = analogRead (which);
  byte buf [2];

    buf [0] = val >> 8;
    buf [1] = val & 0xFF;
    Wire.write (buf, 2);
  }  // end of sendSensor

void requestEvent ()
  {
  switch (command)
     {
     case CMD_ID:      Wire.write (0x55); break;   // send our ID
     case CMD_READ_A0: sendSensor (A0); break;  // send A0 value
     case CMD_READ_D8: Wire.write (digitalRead (8)); break;   // send D8 value

     }  // end of switch

  }  // end of requestEvent
```

The example above assumes a single-byte command, which we save in the variable "command". Then when the requestEvent handler is called, we check what the most-recent command was, and send an appropriate response.

Warning - because of the way the Wire library is written, the requestEvent handler can only (successfully) do a single send. The reason is that each attempt to send a reply resets the internal buffer back to the start. Thus in your requestEvent, if you need to send multiple bytes, you should assemble them into a temporary buffer, and then send that buffer using a single Wire.write. For example:

```
void requestEvent()
  {

  byte buf [2];

  buf [0] = 42;
  buf [1] = 55;

  Wire.write (buf, sizeof buf);  // send 2-byte response
  }
```

Tip:

You can use a struct to send multiple things at once. For example:

```
void requestEvent ()
  {

  struct
    {
    byte command;
    byte argument;
    } response;

  response.command = 42;
  response.argument = 55;

  Wire.write ((byte *) &response, sizeof response);

  }  // end of requestEvent
```

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

Posted by **Nick Gammon** Australia (22,538 posts) bio *Forum Administrator*

Date Reply #5 on Thu 17 Feb 2011 11:17 PM (UTC)

Amended on Tue 26 Apr 2011 04:52 AM (UTC) by Nick Gammon

Message

Pull-up resistors

An interesting point was raised on the Arduino forum about unreliability of devices connected with I2C without pull-up resistors. A very interesting reference was made to this site:

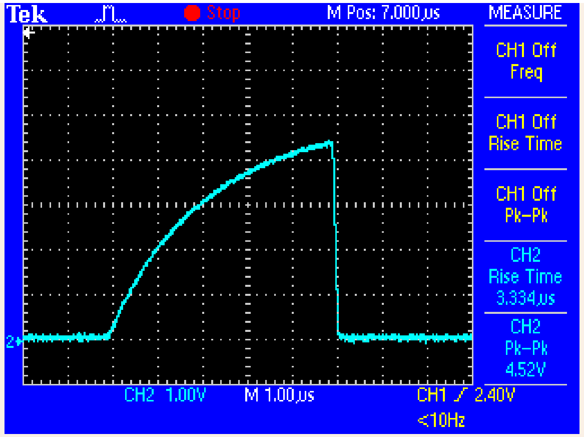
http://www.dsscircuits.com/articles/effects-of-varying-i2c-pull-up-resistors.html

In case that ever goes down, I made some measurements of my own, connecting two Arduinos together for the "Request/response" example above and then measured the signals on the SCL (clock) line.

This time, instead of using a logic analyzer I used a digital oscilloscope. Whilst logic analyzers are great for, well, analyzing logic, they tend to hide whether you are getting nice "clean" logic signals or not.

I disabled the pull-up resistor on one of the Arduinos, enabled the other one, and then measured with this result:

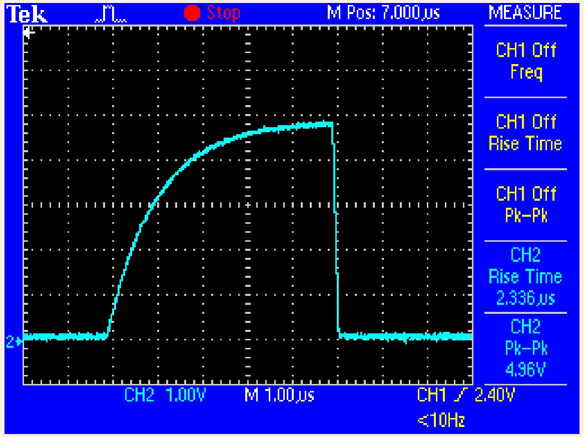
One pull-up resistor enabled internally



Note how far from being a nice square wave the signal looks quite ratty, with a very curved and long rise time.

Two pull-up resistors enabled internally

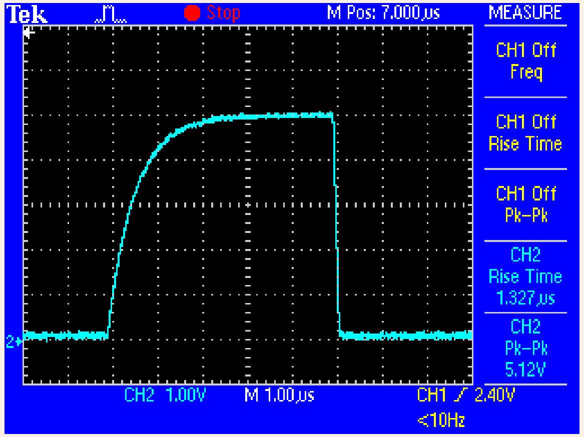
In my case I was able to enable the pull-up resistors on both the sending and receiving end, as both ends were an Arduino, which gave a slighly better result:



10K external pull-up

Now with both internal pull-ups disabled, I tried various external pull-up values. Initially a value of around 33K gave the same results as the internal pull-up. This sounds about right as the spec for the ATmega says the internal pullup is in the range 20K to 50K.

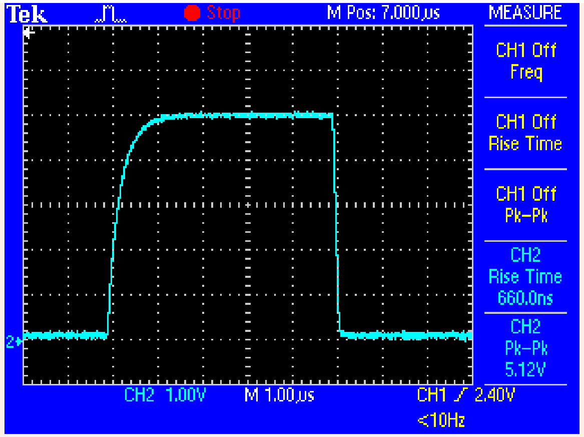
However the image below is for a 10K pull-up:



That is looking a bit more square.

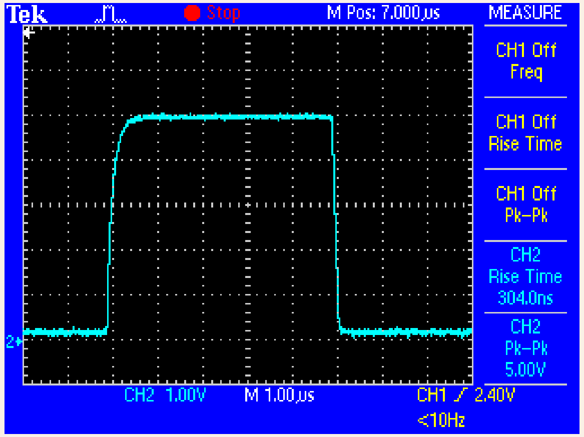
4.7K external pull-up

Dropping the pull-up value down to what I have seen recommended elsewhere, namely 4.7K, gives a somewhat cleaner, square-er result:



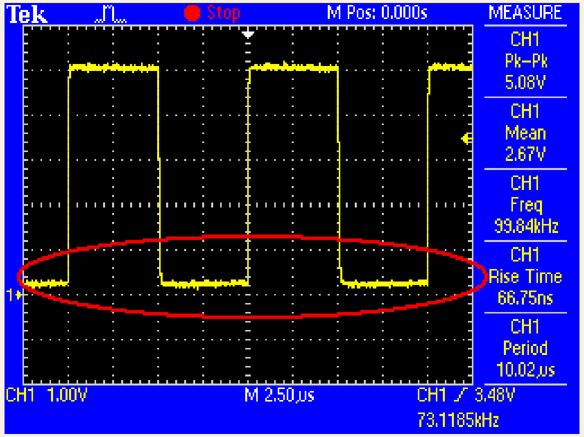
2.2K external pull-up

The 2.2K resistor gives a nice square-looking signal.



1K external pull-up

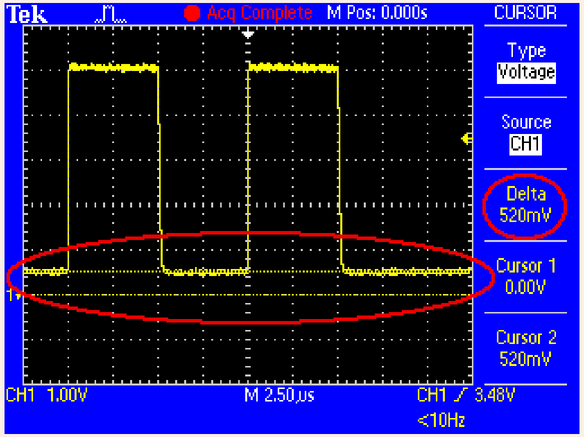
A 1K resistor gives a very square-looking signal, however notice now that the 0V edge (the bottom) is now lifting off the bottom slightly (circled). Also the power through the pull-up is now getting more significant (I = E/R, so it is drawing 5mA).



---

470 ohm external pull-up

The 470 ohm pull-up gives a square signal, but it is no longer reaching 0V. The delta figure on the image shows that the bottom is now at 520mV (half a volt). Also the power through the pull-up is now getting more significant (I = E/R, so it is drawing 10.6mA).



There's a bit more to it than that, as the ideal value would depend on the number of devices connected, cable length, I2C clock speed and so on. However the screen shots above should help show the effect of choosing different values.

[EDIT] Edited 26 April 2011 to add 1K and 470 ohm pull-up images, which is why they are in different colours.

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

Posted by **Nick Gammon** Australia (22,538 posts) bio *Forum Administrator*

Date Reply #6 on Wed 20 Apr 2011 04:54 AM (UTC)

Amended on Thu 09 Jan 2014 10:37 AM (UTC) by Nick Gammon

Message

I2C Scanner

If you have a device you aren't sure of the slave address of, you can run the sketch below. It tries every possible address (from 1 to 119) and if a device responds, prints that address.

```
// I2C Scanner
// Written by Nick Gammon
// Date: 20th April 2011

#include <Wire.h>

void setup() {
  Serial.begin (115200);

  // Leonardo: wait for serial port to connect
  while (!Serial)
    {
    }

  Serial.println ();
  Serial.println ("I2C scanner. Scanning ...");
  byte count = 0;

  Wire.begin();
  for (byte i = 8; i < 120; i++)
  {
    Wire.beginTransmission (i);
    if (Wire.endTransmission () == 0)
      {
      Serial.print ("Found address: ");
      Serial.print (i, DEC);
      Serial.print (" (0x");
      Serial.print (i, HEX);
      Serial.println (")");
      count++;
      delay (1);  // maybe unneeded?
      } // end of good response
  } // end of for loop
  Serial.println ("Done.");
  Serial.print ("Found ");
  Serial.print (count, DEC);
  Serial.println (" device(s).");
}  // end of setup

void loop() {}
```

Example output:

```
I2C scanner. Scanning ...
Found address: 42 (0x2A)
Done.
Found 1 device(s).
```

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

Posted by **Nick Gammon** Australia (22,538 posts) bio *Forum Administrator*

Date Reply #7 on Mon 09 Jan 2012 07:54 PM (UTC)

Amended on Tue 13 Jan 2015 02:55 AM (UTC) by Nick Gammon

Message

Alternative I2C library

Some people have reported on the Arduino forum that I2C "hangs" under certain circumstances. This may well be because the inbuilt "Wire" library expects certain interrupts to happen, and loops until they do. If the interrupt is lost, the library hangs.

An alternative library is described here:

http://dsscircuits.com/articles/arduino-i2c-master-library

In case that site ever goes down, a copy of the the actual library zip file is here:

http://www.gammon.com.au/Arduino/I2C_Rev5.zip

Note that this is a library for the "master" end only, not the slave.

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

Posted by **Nick Gammon** Australia (22,538 posts) bio *Forum Administrator*

Date Reply #8 on Tue 08 May 2012 03:23 AM (UTC)

Amended on Fri 26 Jan 2018 09:57 PM (UTC) by Nick Gammon

Message

Send and receive any data type

The core I2C library just sends and receives bytes. Often enough you might want to send a float, int or long type. The small I2C_Anything library helps solve that.

The library basically consists of this file:

I2C_Anything.h

```
// Written by Nick Gammon
// May 2012

#include <Arduino.h>
#include <Wire.h>

template <typename T> unsigned int I2C_writeAnything (const T& value)
  {
  Wire.write((byte *) &value, sizeof (value));
  return sizeof (value);
  }  // end of I2C_writeAnything

template <typename T> unsigned int I2C_readAnything(T& value)
  {
    byte * p = (byte*) &value;
    unsigned int i;
    for (i = 0; i < sizeof value; i++)
          *p++ = Wire.read();
    return i;
  }  // end of I2C_readAnything
```

It uses templates to convert any data type into a stream of bytes. For example, to send a float and long to another Arduino ...

Master

```
// Written by Nick Gammon
// May 2012

#include <Wire.h>
#include <I2C_Anything.h>

const byte SLAVE_ADDRESS = 42;

void setup()
{
  Wire.begin ();
}  // end of setup

void loop()
{

 long foo = 42;

 for (float fnum = 1; fnum <= 10; fnum += 0.015)
    {
    Wire.beginTransmission (SLAVE_ADDRESS);
    I2C_writeAnything (fnum);
    I2C_writeAnything (foo++);
    Wire.endTransmission ();

    delay (200);
    }  // end of for

}  // end of loop
```

The two lines starting I2C_writeAnything use the template to send fnum and foo as the appropriate series of bytes.

Then to receive them:

Slave

```
// Written by Nick Gammon
// May 2012

#include <Wire.h>
#include <I2C_Anything.h>

const byte MY_ADDRESS = 42;
```

```
void setup()
{
  Wire.begin (MY_ADDRESS);
  Serial.begin (115200);
  Wire.onReceive (receiveEvent);
}  // end of setup

volatile boolean haveData = false;
volatile float fnum;
volatile long foo;

void loop()
{
  if (haveData)
    {
    Serial.print ("Received fnum = ");
    Serial.println (fnum);
    Serial.print ("Received foo = ");
    Serial.println (foo);
    haveData = false;
    }  // end if haveData

}  // end of loop

// called by interrupt service routine when incoming data arrives
void receiveEvent (int howMany)
 {
 if (howMany >= (sizeof fnum) + (sizeof foo))
   {
   I2C_readAnything (fnum);
   I2C_readAnything (foo);
   haveData = true;
   }  // end if have enough data
 }  // end of receiveEvent
```

The lines starting with I2C_readAnything read the correct number of bytes for the data type, and assemble them back into the variable passed as an argument.

Library

Just put the raw file I2C_Anything.h into a folder "I2C_Anything" and put that into your Arduino "libraries" folder. Then restart the IDE and you are ready to use it.

https://github.com/nickgammon/I2C_Anything

The version at GitHub may be more up-to-date than the code above, and possibly any issues with it may be discussed.

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

Posted by **Nick Gammon** Australia (22,538 posts) bio *Forum Administrator*

Date Reply #9 on Sun 31 Mar 2013 10:08 PM (UTC)

Amended on Tue 19 May 2015 09:02 PM (UTC) by Nick Gammon

Message

I2C summary

This post summarizes the most important aspects of the above posts.

> This page can be quickly reached from the link: http://www.gammon.com.au/i2c-summary

I2C address

The slave address is a 7-bit address, thus is in the range 0 to 127 decimal (0x00 to 0x7F in hex). However addresses 0 to 7, and 120 to 127 are reserved.

If you see a number larger than 127 (0x7F) quoted, then that is the **8-bit address**, which includes the read/write bit. You need to divide an 8-bit address by two (shift right one) to get the correct address for the Wire library. For example if a datasheet says to use address 0xC0 for writing and 0xC1 for reading, that includes the read/write bit. Drop the "1" and divide by two, giving the "real" address of 0x60.

If you are not sure, use the I2C scanner (see reply #6 above) to determine which address your device actually uses.

Initializing the Wire library

Master

```
#include <Wire.h>
...
Wire.begin ();
```

Slave

```
#include <Wire.h>
...
const byte SLAVE_ADDRESS = 0x68;  // eg. DS1307 clock
...
Wire.begin (SLAVE_ADDRESS);
```

Changing I2C speed

Master only, as that sends out the clock pulses.

After doing:

```
Wire.begin ();
```

Choose another speed using TWBR (Two Wire Bit Rate) register:

```
TWBR = 12;  // 400 kHz (maximum)
```

or:

```
TWBR = 32;  // 200 kHz
```

or:

```
TWBR = 72;  // 100 kHz (default)
```

or:

```
TWBR = 152;  // 50 kHz
```

or:

```
TWBR = 78;  // 25 kHz
TWSR |= bit (TWPS0);  // change prescaler
```

or:

```
TWBR = 158;  // 12.5 kHz
TWSR |= bit (TWPS0);  // change prescaler
```

Examples are for Atmega328P running at 16 MHz (eg. Arduino Uno, Duemilanove, etc.)

Master: send data to slave

You can send one or more bytes (up to **32**), like this:

```
  Wire.beginTransmission (SLAVE_ADDRESS);
  Wire.write (0x20);
  Wire.write (0x30);
  Wire.write (0x40);
  Wire.write (0x50);
  if (Wire.endTransmission () == 0)
    {
    // success!
    }
  else
    {
    // failure ... maybe slave wasn't ready or not connected
    }
```

Maximum of 32 bytes can be sent by the standard Wire library!

The actual writing occurs on the Wire.endTransmission function call. You should test on that line whether or not the send succeeded. If not there is no point in trying to get a response.

Slave: receive data from master

```
// set up receive handler  (in setup)
Wire.onReceive (receiveEvent);  // interrupt handler for incoming messages

...

volatile byte buf [32];

// called by interrupt service routine when incoming data arrives
void receiveEvent (int howMany)
  {
  for (byte i = 0; i < howMany; i++)
    {
    buf [i] = Wire.read ();
    }  // end of for loop
  }  // end of receiveEvent
```

Note that receiveEvent is called from an Interrupt Service Routine (ISR)!

You should **not**:

- Do serial prints
- Use "delay"
- Do anything lengthy

- Do anything that requires interrupts to be active

Master: request information from slave

```
byte buf [10];

if (Wire.requestFrom (SLAVE_ADDRESS, 10))  // if request succeeded
  {
  for (byte i = 0; i < 10; i++)
    buf [i] = Wire.read ();
  }
else
  {
  // failure ... maybe slave wasn't ready or not connected
  }
```

Important! You do not need to test for Wire.available here. The Wire.requestFrom function **does not return** until either the requested data is fully available, or an error occurred. Building in a wait for Wire.available simply makes it possible for the code to hang forever if the data is not available.

~~The correct method is to test whether the number of bytes you wanted was returned from Wire.requestFrom, as in the example above (10 bytes in this case).~~

See Reply #10 below. Wire.requestFrom() will either return zero, or the number of bytes requested. If there is doubt as to whether all of the requested bytes were received do a data integrity check on the received bytes.

You can use I2C_Anything (see below) to receive multiple bytes.

Slave: respond to request for information

```
// set up request handler  (in setup)
Wire.onRequest (requestEvent);  // interrupt handler for when data is wanted

...

// called by interrupt service routine when response is wanted
void requestEvent ()
  {
  Wire.write (0x42);  // send response
  }  // end of requestEvent
```

Note that requestEvent is called from an Interrupt Service Routine (ISR)!

You should **not**:

- Do serial prints
- Use "delay"
- Do anything lengthy
- Do anything that requires interrupts to be active

You can only do **one** Wire.write in a requestEvent! You do **not** do a Wire.beginTransmission or a Wire.endTransmission. There is a limit of 32 bytes that can be returned.

You can use I2C_Anything (see below) to return multiple bytes.

Example of returning multiple bytes:

```
void requestEvent ()
  {
  byte buf [4] = { 1, 2, 3, 4 };
  Wire.write (buf, sizeof buf);   // send response
  }  // end of requestEvent
```

Send/receive any data type

If you want to send or receive things other than simple bytes (eg. int, float, structures) you can use the I2C_Anything which simplifies this process.

The I2C_Anything library can be downloaded from:

http://gammon.com.au/Arduino/I2C_Anything.zip

Just unzip the downloaded file, and put the resulting folder "I2C_Anything" into your Arduino "libraries" folder. Then restart the IDE and you are ready to use it.

Example of sending with I2C_Anything:

```
#include <I2C_Anything.h>

...

  long foo = 42;
  float bar = 123.456;

  Wire.beginTransmission (SLAVE_ADDRESS);
  I2C_writeAnything (foo);
  I2C_writeAnything (bar);
  if (Wire.endTransmission () == 0)
      {
      // success!
      }
    else
      {
      // failure ... maybe slave wasn't ready or not connected
      }
```

Example of receiving with I2C_Anything:

```
#include <I2C_Anything.h>

...

volatile boolean haveData = false;
volatile long foo;
volatile float bar;

// called by interrupt service routine when incoming data arrives
void receiveEvent (int howMany)
 {
 if (howMany >= (sizeof foo) + (sizeof bar))
   {
   I2C_readAnything (foo);
   I2C_readAnything (bar);
   haveData = true;
   }  // end if have enough data
 }  // end of receiveEvent
```

Note that variables set within an ISR have to be declared volatile. Note the use of a flag to signal the main loop that we have received something.

Pull-up resistors

For reliable operation both SDA and SCL should be "pulled up" by connecting them via a resistor each to Vcc (+5V). Typically 4.7K resistors would be suitable. Note that some devices may already have such resistors installed, in which case you don't need another set.

Connections

As shown earlier in this thread, SDA on the Master is connected to SDA on the Slave, and SCL on the Master is connected to SCL on the Slave. Ground should also be connected together.

Some of the more modern Arduinos (eg, Uno R3) have dedicated header pins for SDA and SCL.

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

Posted by Nick Gammon Australia (22,538 posts) bio *Forum Administrator*

Date Reply #10 on Sun 22 Feb 2015 08:30 PM (UTC)

Amended on Sun 22 Feb 2015 08:31 PM (UTC) by Nick Gammon

Message

Wire.requestFrom correction

This page previously asserted (as did the Arduino documentation) that Wire.requestFrom returns the number of bytes sent by the slave.

I now believe this to be incorrect. Wire.requestFrom() either returns **zero** (if the slave could not be connected to) or the **number of bytes requested**, regardless of how many bytes the slave actually sent.

The reason is that, for the slave to terminate the communication early (ie. after sending less than the requested number of bytes) it would have to be able to raise a "stop condition". Only the master can do that, as the master controls the I2C clock.

Thus, if the slave stops sending before the requested number of bytes have been sent, the pull-up resistors pull SDA high, and the master simply receives one or more 0xFF bytes as the response.

This has been raised on GitHub:

https://github.com/arduino/Arduino/issues/2616

It does not help to use Wire.available() because that too will return the number of bytes requested.

What does this mean?

It means you can't rely on the slave necessarily sending the number of bytes you requested. If you request 10 bytes, Wire.requestFrom() will return 10, even if only 5 have been sent.

If you need to return a variable number of bytes (and if 0xFF is a possible value) then you could send a "length" byte as the first byte, followed by the number of "real" bytes. Then the master can use the length byte to determine the size of the response.

Also, data integrity can be affected because, if the slave stops sending early, you might get 0xFF and mistake it for real data. A solution in this case is to append a CRC check (cyclic redundancy check) to your data stream.

An example of calculating a CRC is here:

```
// calculate 8-bit CRC
byte crc8 (const byte *addr, byte len)
{
  byte crc = 0;
  while (len--)
    {
    byte inbyte = *addr++;
    for (byte i = 8; i; i--)
      {
      byte mix = (crc ^ inbyte) & 0x01;
      crc >>= 1;
      if (mix)
        crc ^= 0x8C;
      inbyte >>= 1;
      }  // end of for
    }  // end of while
  return crc;
}  // end of crc8
```

You supply to the above function a pointer to some bytes, and the length, and it returns the CRC value. If the sending end does that before sending, and the receiving end after receiving, and the CRC value agrees, you can be reasonably confident that the data is not corrupted.

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

Posted by **Nick Gammon** Australia (22,538 posts) bio *Forum Administrator*

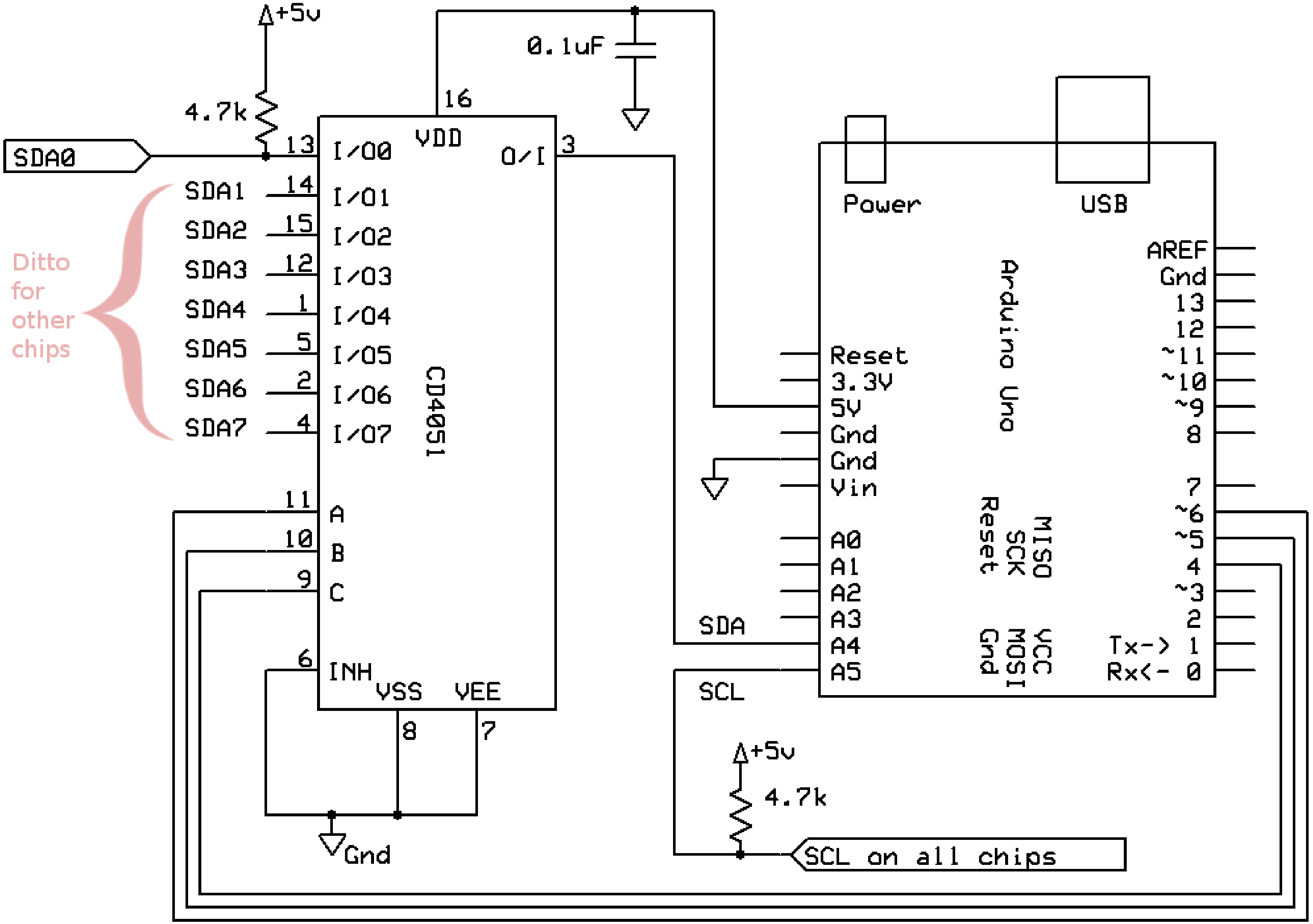
Date Reply #11 on Wed 03 Feb 2016 01:14 AM (UTC)

Amended on Wed 03 Feb 2016 01:15 AM (UTC) by Nick Gammon

Message

Multiplexing I2C

There have been various queries about how to deal with multiple target (slave) chips which are manufactured with the same I2C address. Since only device with one address can be present at once, these chips pose a challenge. You can buy dedicated I2C multiplexers, but for a simple situation with a fixed master, a single CD4051 analog multiplexer can handle it.


+5v
4.7k
SDA0
13
I/O0
VDD
16
0.1uF
O/I
3
SDA1
14
I/O1
SDA2
15
I/O2
SDA3
12
I/O3
SDA4
1
I/O4
SDA5
5
I/O5
SDA6
2
I/O6
SDA7
4
I/O7
Ditto for other chips
CD4051
11
A
10
B
9
C
6
INH
VSS
VEE
8
7
Gnd
Power
USB
Arduino Uno
AREF
Gnd
13
12
~11
~10
~9
8
7
~6
~5
4
~3
2
Tx-> 1
Rx<- 0
Reset
3.3V
5V
Gnd
Gnd
Vin
A0
A1
A2
A3
A4
A5
Reset
SCK
MISO
Gnd
MOSI
VCC
SDA
SCL
+5v
4.7k
SCL on all chips

Since the master generates the clock (SCL) we can just connect all the clock lines together (as usual).

We need to multiplex the data (SDA), and the CD4051 can multiplex up to 8 signals into one (at a time).

See http://www.gammon.com.au/forum/?id=11976 for more details about the CD4051. There is also a 16-channel multiplexer: 74HC4067

As a demonstration, this code looks for 3 x Arduino boards with the slave code loaded on it (from earlier in this thread) that turns on LEDs in sequence.

```
#include <Wire.h>

const byte SLAVE_ADDRESS = 42;
const byte LED = 13;

const byte slaveCount = 3;  // how many slaves

// the multiplexer address select lines (A/B/C)
const byte addressA = 6; // low-order bit
const byte addressB = 5;
const byte addressC = 4; // high-order bit


void setup ()
  {
  Wire.begin ();
  pinMode (LED, OUTPUT);
  pinMode (addressA, OUTPUT);
  pinMode (addressB, OUTPUT);
  pinMode (addressC, OUTPUT);

  }  // end of setup

int selectTarget (const byte which)
  {
  // select correct MUX channel
  digitalWrite (addressA, (which & 1) ? HIGH : LOW);  // low-order bit
  digitalWrite (addressB, (which & 2) ? HIGH : LOW);
  digitalWrite (addressC, (which & 4) ? HIGH : LOW);  // high-order bit
  }  // end of readSensor


void loop ()
  {
  for (byte target = 0; target < slaveCount; target++)
    {
```

```
    selectTarget (target);
    for (byte x = 2; x <= 7; x++)
      {
      Wire.beginTransmission (SLAVE_ADDRESS);
      Wire.write (x);
      if (Wire.endTransmission () == 0)
        digitalWrite (LED, HIGH);
      else
        digitalWrite (LED, LOW);

      delay (200);
      }  // end of for loop
    }
  }  // end of loop
```

The code above calls **selectTarget** to cause the multiplexer to switch the appropriate SDA line to the master, and then in turn transmits the number 2 to 7, instructing the slave to toggle the requested pin.

Note the pull-up resistor on SCL, and the pull-up resistor on **each** SDA line (on the slave side of the multiplexer) to ensure that SDA is kept high if that slave is not selected.

- Nick Gammon

www.gammon.com.au, www.mushclient.com

top

The dates and times for posts above are shown in Universal Co-ordinated Time (UTC).

To show them in your local time you can join the forum, and then set the 'time correction' field in your profile to the number of hours difference between your location and UTC time.

466,455 views.

Postings by administrators only.

Refresh page

Go to topic: (Choose topic) Go Search the forum

top

Quick links: **MUSHclient**. MUSHclient **help**. Forum **shortcuts**. Posting **templates**. Lua **modules**. Lua **documentation**.

Home

Designed & written by Nick Gammon

Nick Gammon
42k ●10 ●108 ●273

Comments to: Gammon Software support

XML Forum RSS feed (https://gammon.com.au/rss/forum.xml)