The Tutorial Book

Have fun with PIC microcontrollers, Jal v2 and Jallib

Step by step tutorials, covering basic features of PIC microcontrollers, using jalv2 compiler and jallib libraries. (version 0.3)

Contents

Chapter 1

Back to basics...

Topics:

This chapter is about exploring basic tutorials. As a beginner, these are the very first steps you should experiment and fully understand before going further. As an advanced user, these tutorials are also here to help you testing new chips, or... when things go wrong and you can't figure out why, going back to basics

Don't worry, everything is gonna be alright...

Installation

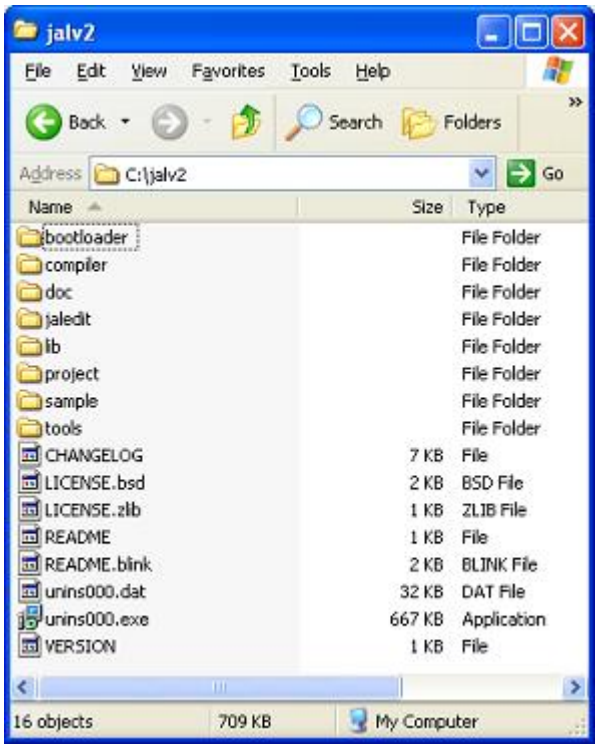
Jallib Group
Jallib Group

JALv2 & Jallib installation guide

Windows Install:

1. Download jalpack (installer executable) from *http://jaledit.googlecode.com/files/JALPack_2.4_0.4_0.6.1.0.exe*, This will install JALv2 + JalEdit
2. Update your installation (very important) - Download jallib-pack or jallib-pack-bee from *http://code.google.com/p/jallib/downloads/list*, copy the .zip contents into your Jallib installation directory.
3. Run the setup file
4. Run JalEdit.exe from the "jaledit" directory
5. (optional) Click Tools Menu -> Environment Options -> Programmer, Then Set the Programmer Executable Path

You should get something like this on windows:

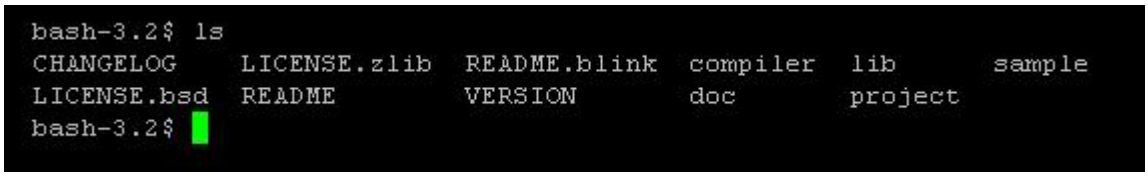


Linux Install

1. Go to *http://code.google.com/p/jallib/downloads/list*, get the link location of the jallib-pack (.tar.gz file)
2. Go to the directory you wish to install JALv2
3. Download the package with: `$ wget [link location of the jallib-pack]` or simply use your favorite browser to download archive in the appropriate directory.
4. unzip the package with: `$ tar xzf [filename.tar.gz]`

Note: Jaledit runs under Wine on Linux

You should get something like this on linux:



Getting Started

Matthew Schinkel
Jallib Group

Guide to getting started with PIC microcontrollers JALv2 & Jallib

So, you've heard all the hype about PIC microcontrollers & JALv2 and want to hear more?

Why use PIC microcontrollers, JALv2, and this book?

Simple usage:

Yes, that's right, microcontrollers are simple to use with the help of this open source language JAL. Not only are microcontrollers simple to use, but many other complex external hardware is made easy such as: USB, Analog to digital conversion (ADC), serial communication, Hard Disks, SD Cards, LCD displays, sensors and many more.

All you will need is a small amount of knowledge about general electronics. We will teach you the rest you need to know!

Circuit Simplicity:

Would you like to reduce the size of your circuits? What are you currently using to build your digital circuits?

When I got started, I liked to use things like the 74LS series, simple CMOS gate chips, 555 timers etc. You can build just about anything with these simple chips, but how many will you need to complete your project? One of the projects I built some time ago used five 74ls chips. With a microcontroller, I can now reduce my circuit to 1 microcontroller.

Bigger Projects:

When I say bigger, I mean cooler projects! There is no limit to what you can build! Choose from our small projects to build a large project of your own. What functionality do you need for your project? Check out our tutorial section for a complete list of compatible features you can use for your circuit.

What do I need to get started?

You will need the following:

1. PIC microcontroller chip
2. PIC programmer
3. Programming language (JALv2) + Libraries (JALLIB) + Editor, see our installation guide.
4. Computer (preferably one with a serial port)
5. PIC programming / burning software
6. Regular electronic stuff such as breadboard, resistors, wire, multimeter etc.
7. Oscilloscope is not required but suggested for some advanced projects.

Follow our Installation Guide for free programming language, libraries & text editor

How much will it cost?

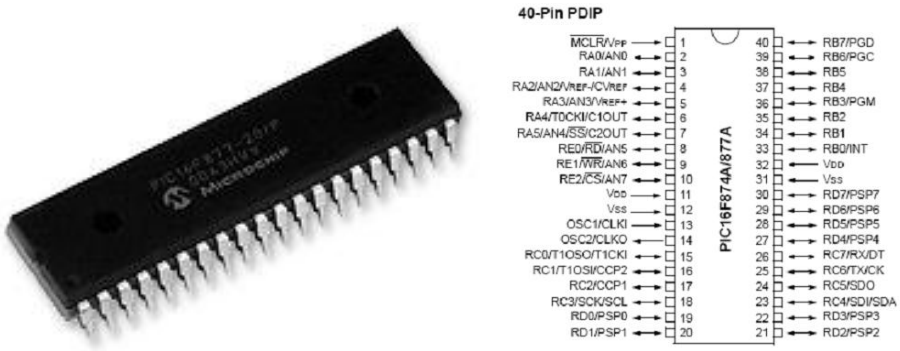
Yes, getting started with microcontrollers has it's price. A microcontroller can cost you anywhere between $1 to $10 USD, and a programmer will cost $20 to $50. But you can't put a price on FUN!

The programming language JALv2 is FREE, other languages will cost you somewhere between $200 and $2000.

When you compare this price to the price you are currently spending on those many IC's you currently require to build your circuits, this may be cheaper. You will not need many of your smaller IC's, and some specialty chips can be replaced. Of course you're going to save time and breadboard space as well!

As an example... Instead of buying a UART chip for serial communication, you can now use the microcontroller's internal UART for communication to your PC or other projects.

What PIC microcontroller should I buy?



PIC16F877 or PIC16F877A seem to be the most popular mid-range PIC at the moment (in the image above). You should be able to find them at your local electronics store for around $10. This microcontroller has many features and a good amount of memory. It will be sufficient for most of your projects. We will build our first project on this chip. I warn you however, you may eventually want to move to an 18F PIC. I only suggest 16f877A because it will be easy to find at a store.

There are many low-end PIC's to choose from, PIC16F84, PIC16F88 are smaller chips for around $5. There are also very low end 8 pin PIC's such as 12F675 for $1.

If you're looking for speed, functionality, and a whole lot of memory space, you can go with a PIC18Fxxx chip. Some of these have USB capability. I would suggest one of the following: 18f452, 18F4620, 18F4550. These PIC's will also work in our getting started "blink a led" tutorial with the same circuit diagram. If you can, get a 18F PIC. My current favorite is the 40 pin 18f4620.

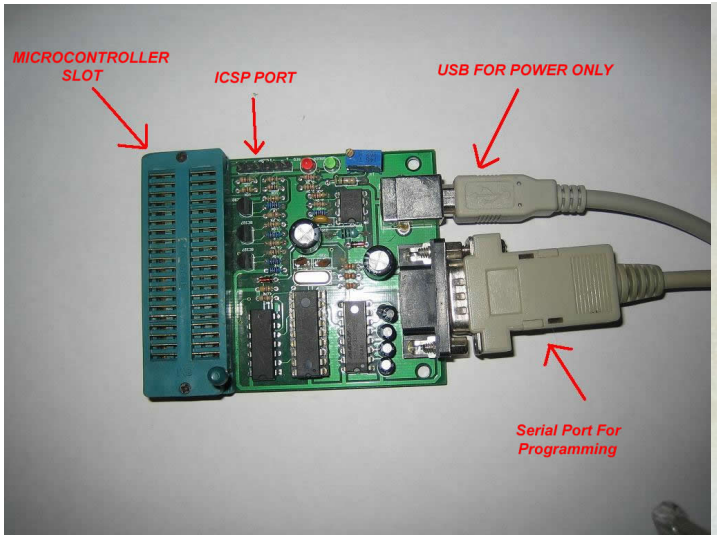
Here's a price chart from the manufacturer's sales website:

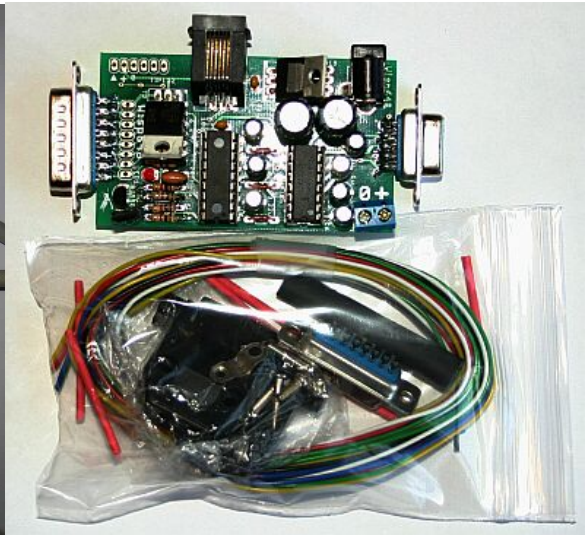
PIC	Price USD
16F877	$5.92
16F877A	$5.20
18F4550	$4.47
16F84	$5.01
12F88	$1.83
18F675	$1.01
18F452	$4.14
18F4550	$4.47
18F2550	$4.51
18F4620	$4.27

What programmer should I buy?

Any pic programmer will do. The only suggestions I have is to make sure it can program a wide variety of PIC's such as the ones listed above, and make sure it has a ICSP port for future use. ICSP is for in-circuit programming.

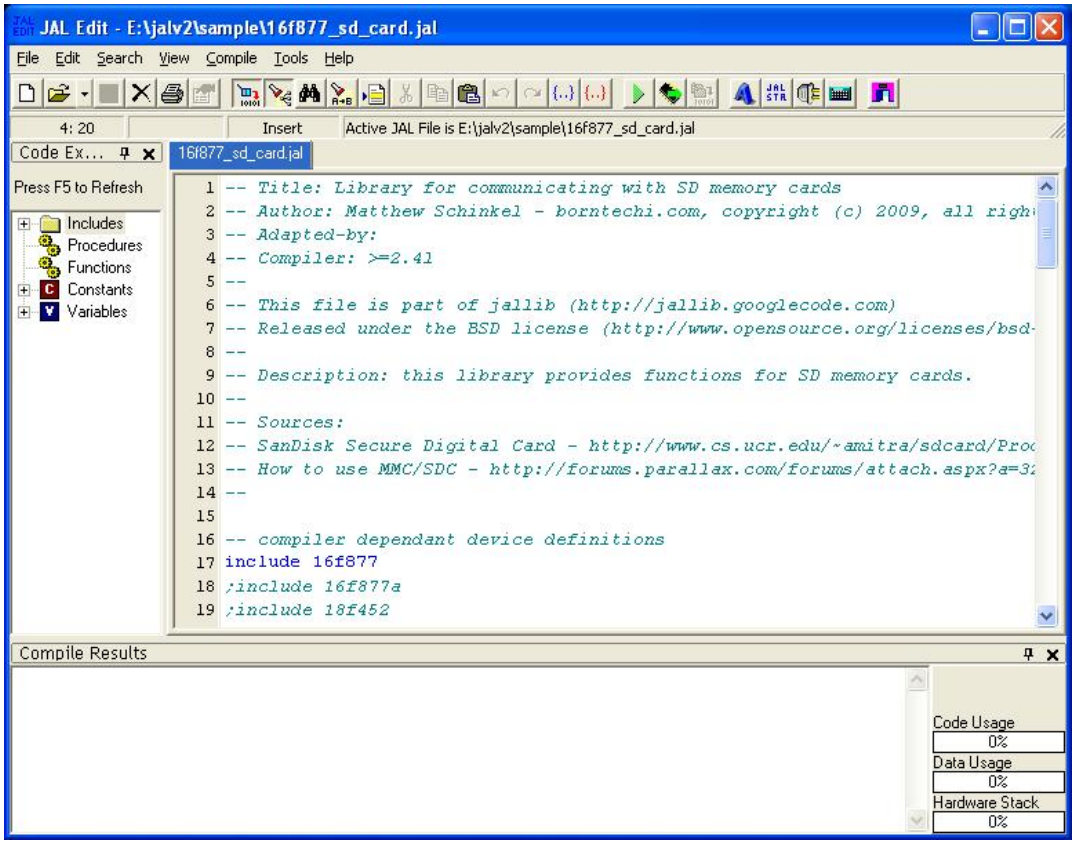
Here are some images of programmers we use:





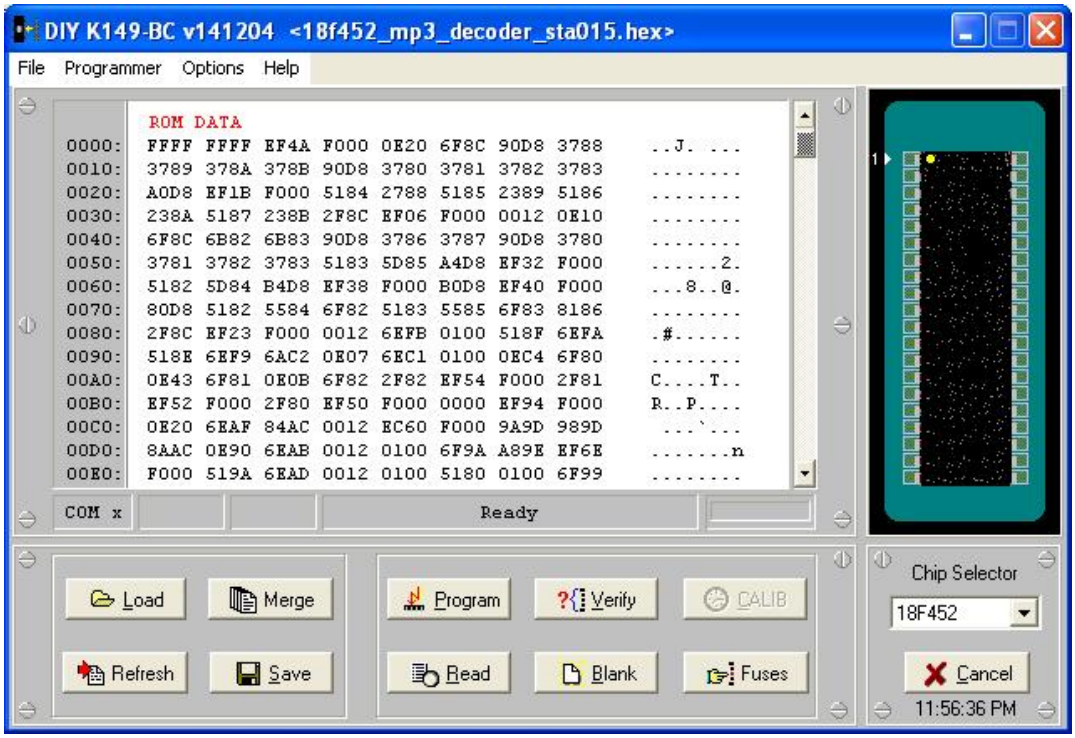
What editor should I use?

Any text editor is fine, but if you are on a windows machine. We suggest the free editor "JAL Edit" which will highlight & color important text as well as compile your JAL program to a hex file for burning to your microcontroller. If you followed our installation guide, you will already have this editor.



What programming/burning software should I use?

Did your programmer come with software? There are many to choose from so use whatever you prefer. I use "Micropro" from *http://www.ozitronics.com/micropro.html*. It's a free, open source software for programming a wide range of PIC's. However, it will most likely not support your programmer. I suggest you use the software that came with your programmer. You may see this programmer in other tutorials for demonstration only.



OK, enough of this boring stuff, lets build something! Start with the *Blink A Led (Your First Project)*.

Blink A Led (Your First Project)

Matthew Schinkel
Jallib Group

In this tutorial we are going to learn how to connect our first circuit and blink our first led.

Where to we start?

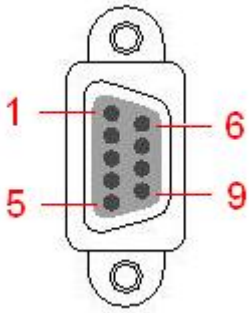
Let's make a led blink on and off, how fun is that!

So, you've followed the installation guide and now have a Programming language (JALv2) + Libraries (JALLIB) + Editor. We will be using JALEdIt for our first example.

Setup your workspace

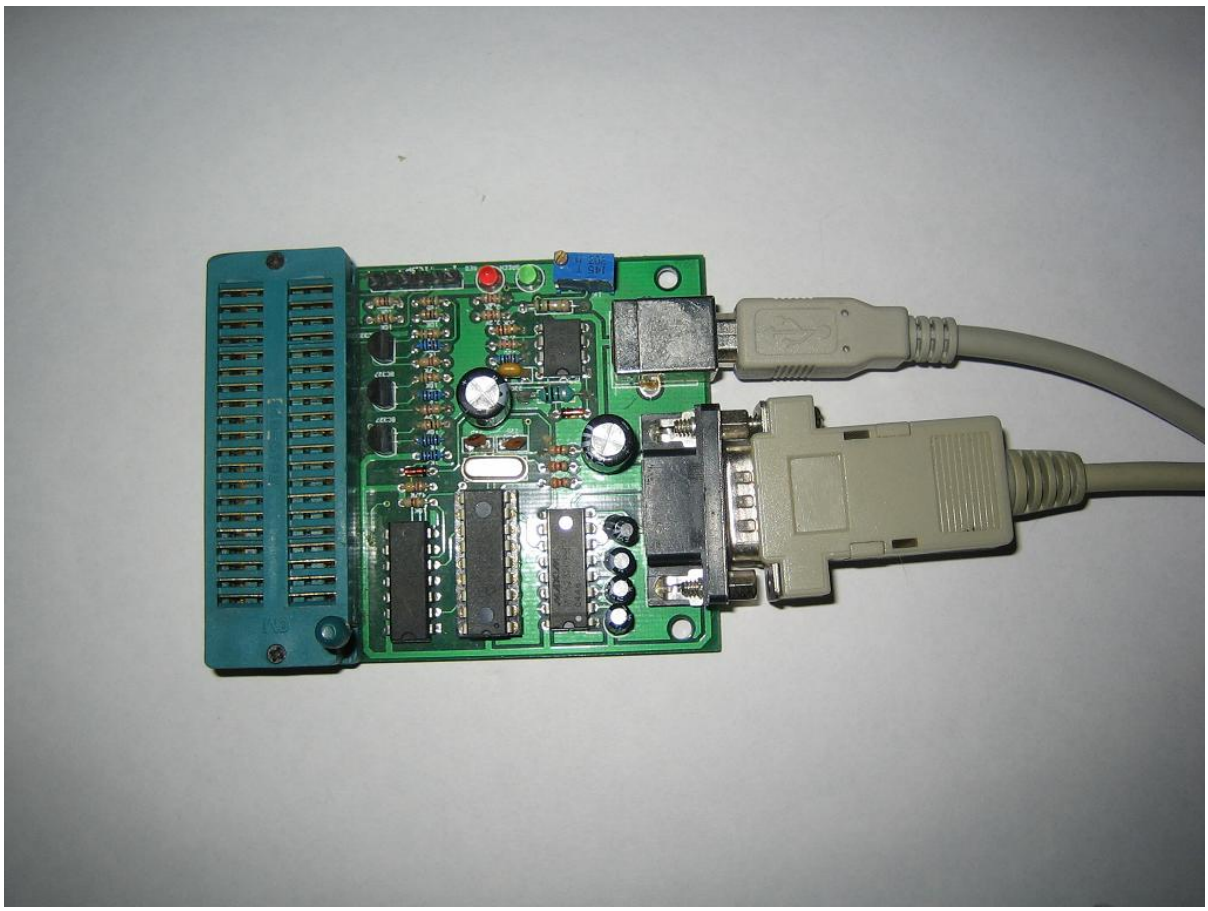
Start by getting out your programmer and connect it to your PC. Some connect by serial port, some connect via USB. I actually use a serial port programmer attached to a USB-to-Serial adapter to free up my serial port for other projects.

If you are using a serial port programmer you need to check that you have a regular serial cable and is not a null modem cable. Using your multimeter, check that each pin of your serial cable matches, if pins 7 & 8 are crossed, it is a null modem cable.

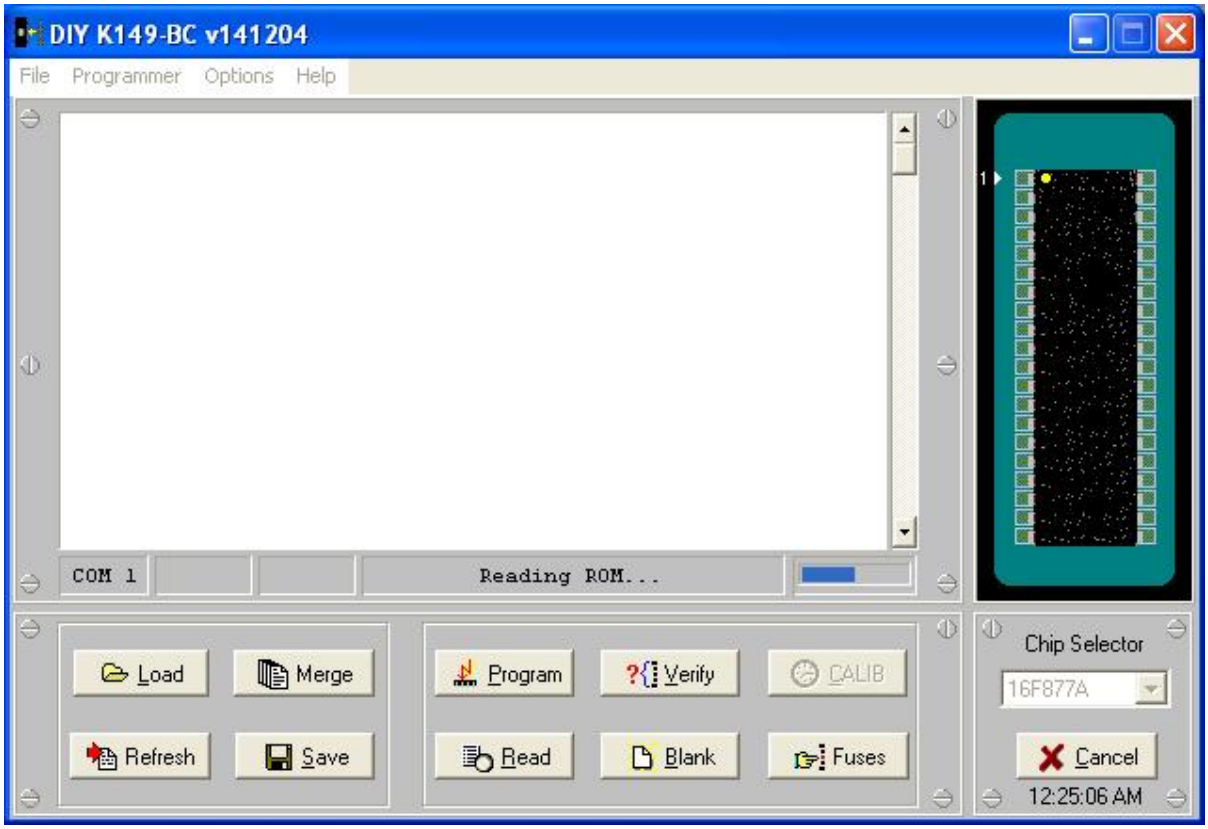


Get out your PIC microcontroller (we will now refer to it as a PIC). You can use PIC's 16f877, 16f877A, 18F2550 , 18F452 or 18F4550 for this project since the port pin outs are the same for all of them. I will use 16f877A for this blink a led project.

Now check PC connectivity to your programmer. Open your programming software on your PC, check the settings within your software to change the serial port number and programmer type (if available). Your programmer software may tell you that your board is connected, if not, put your PIC in your programmer and do some basic tests such as "read chip", "blank / erase chip"

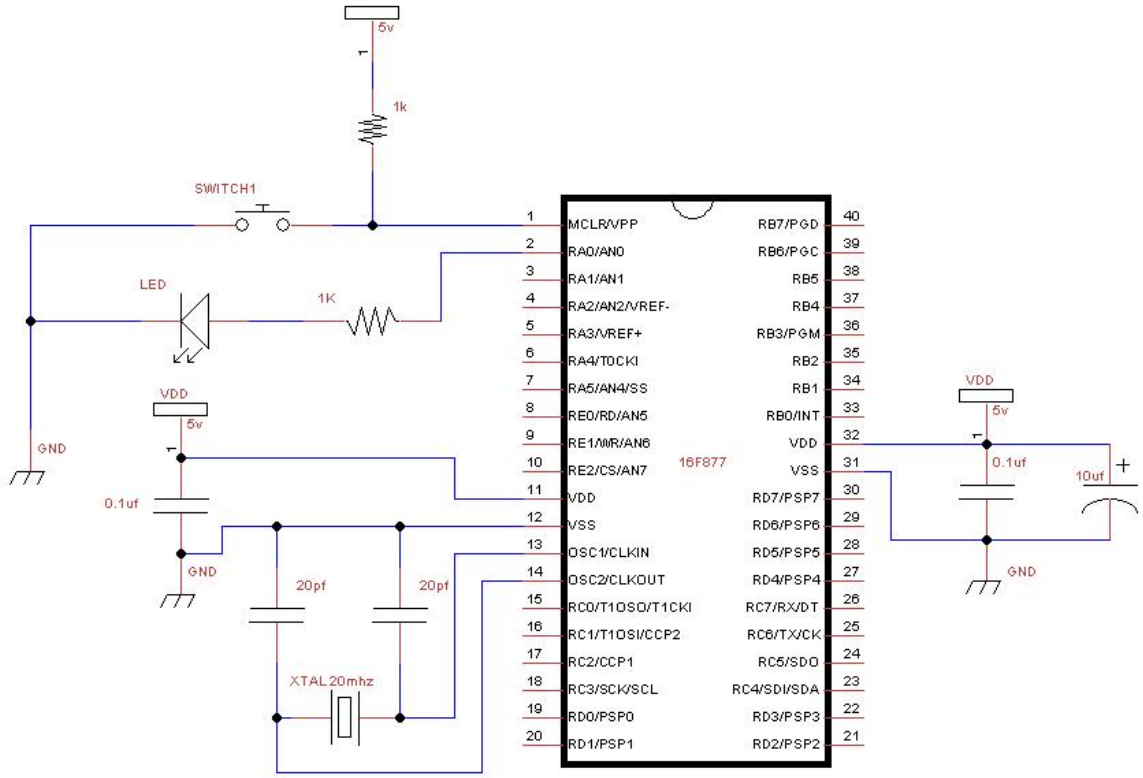


If you are using Micropro, click on “file” -> “port”, and “file” -> “programmer” -> (your programmer type). If you do not know the programmer type, you will have to guess until Micropro says something like “K149-BC board connected”, Put your PIC in your programmer and choose your PIC type from the “Chip Selector” text box. Now do some basic read/erase tests.

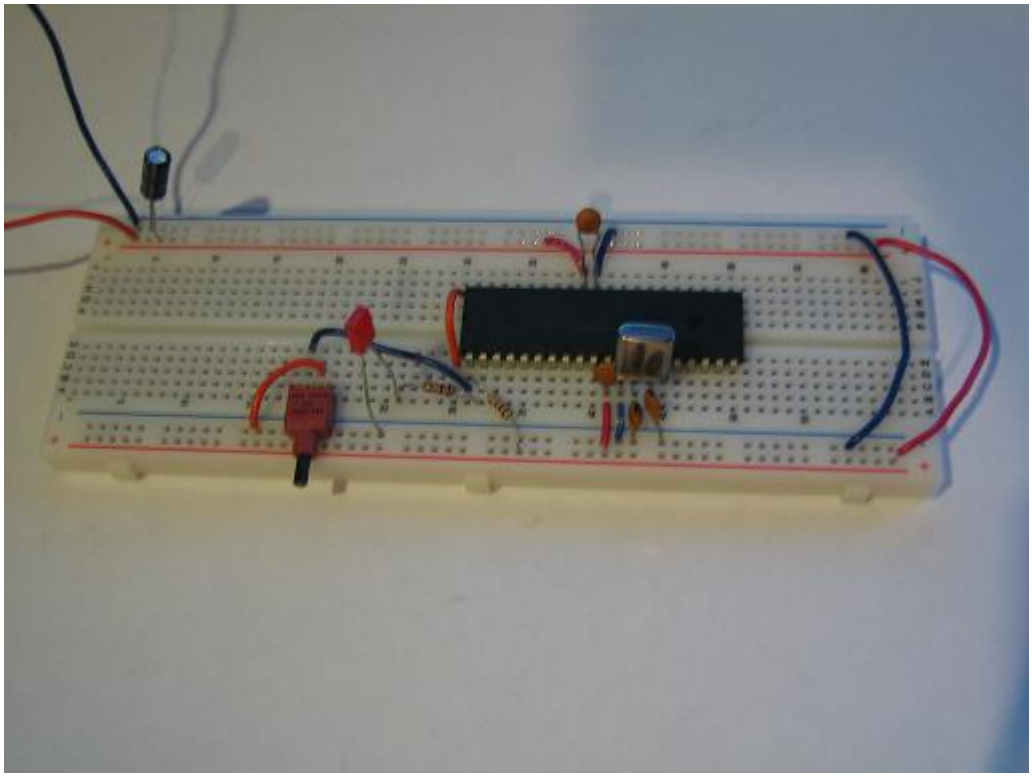


Build your circuit

Well, it looks like we're all set to go, so grab your breadboard and other components, put together the following circuit:



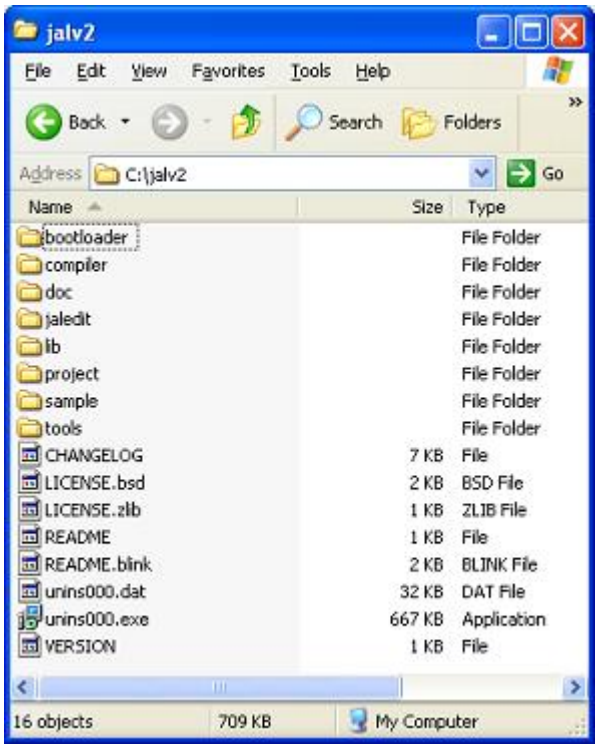
And here's what it looks like. Notice the additional orange wire to the left of my PIC, this ensures that I always put my PIC in the correct position after programming. Do not connect your power 5v supply till your circuit is complete and checked over at least twice. You will burn your PIC if power is on while building your circuit. You will want an on/off switch for your power supply.



Your circuit is done, and it looks pretty, but it doesn't do anything :o(..

Understand the jalv2 directory structure

First take a look at your jalv2 installation directory on your PC, wherever you installed it.



compiler – holds the jalv2.exe compiler program to convert your JAL code to microcontroller hex code

JALEdIt – JAL text editor where you will write your code

lib – A set of libraries to make things work

sample – Working examples.

Create yourself a folder called workspace, and in that folder create a folder called blink_a_led (eg. C:\jalv2\workspace\blink_a_led\)

Setup your editor & .jal file

Open up your favorite text editor. I will use JALEdIt. Run jaledit.exe from the JALEdIt directory. Start a new document, and save it in jalv2\workspace\blink_a_led\ and name it blink_a_led.jal (eg: C:\jalv2\workspace\blink_a_led\blink_a_led.jal)

Let's write some code

So now we're going to write the code that will make our led blink. All code will be in highlighted text. You can read more about JAL language usage here: *http://www.casadeyork.com/jalv2/language.html*

Title & Author Block

Start out by writing a nice title block so everyone know's who created it. Here's an example Title block from Rob Hamerling's working 16f877a_blink.jal blink a led example in the sample directory. Every PIC has at least one working sample. You can see that two dashes "-" declare a comment so your notes get ignored by the compiler. The

character ";" can also be used for comments. We will comment our code as we go along so it is easier for us to read our own code.

```
-- -------------------------------------------------------
-- Title: Blink-a-led of the Microchip pic16f877a
--
-- Author: Rob Hamerling, Copyright (c) 2008..2009, all rights reserved.
--
-- Adapted-by:
--
-- Compiler: 2.4l
--
-- This file is part of jallib  (http://jallib.googlecode.com)
-- Released under the BSD license (http://www.opensource.org/licenses/bsd-
license.php)
--
-- Description:
-- Sample blink-a-led program for Microchip PIC16f877a.
--
-- Sources:
--
-- Notes:
--  - File creation date/time: 14 Oct 2009 20:24:20.
--
-- -------------------------------------------------------
```

Choose your PIC

Write the following code to choose the PIC you are using, change 16f877a to whatever PIC you have:

```
include 16f877a                     -- target PICmicro
```

Choose your crystal speed

Write the following code according to the speed of the crystal you are using in your circuit. I suggest 20mhz for 16f877. You can check your chip's datasheet for it's max speed. Higher speeds may not work the way you want them to on a temporary breadboard.

```
-- This program assumes a 20 MHz resonator or crystal
-- is connected to pins OSC1 and OSC2.
pragma target clock 20_000_000     -- oscillator frequency
```

Configure PIC Settings

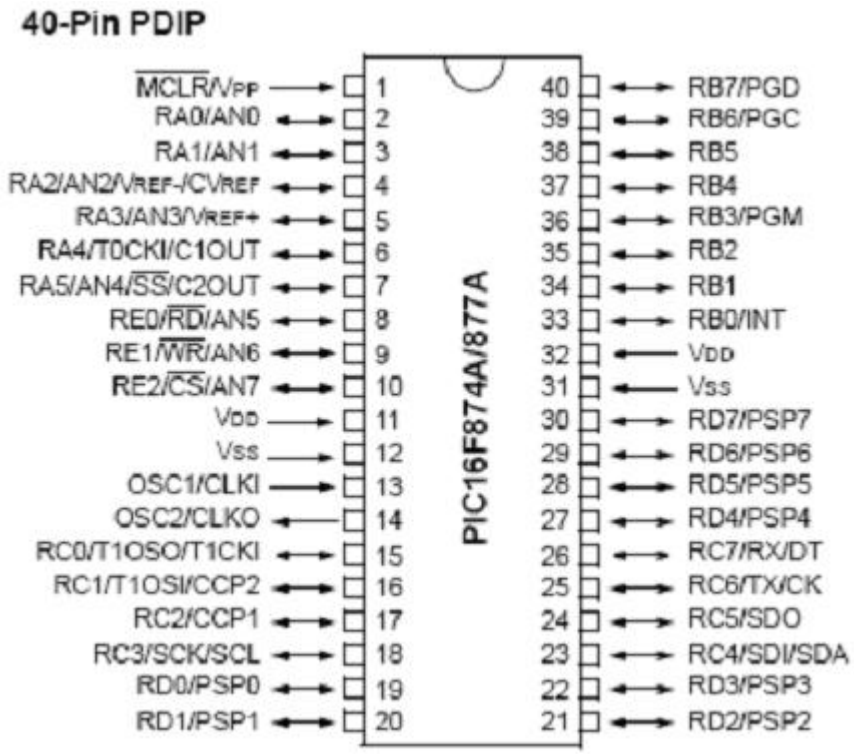
The following code sets some of the PIC's internal settings, called fuses. A OSC setting of HS tells the PIC there is an external clock or crystal oscillator source. You must disable analog pins with enable_digital_io() , you don't need to worry about the others.

```
-- configuration memory settings (fuses)
pragma target OSC  HS              -- HS crystal or resonator
pragma target WDT  disabled        -- no watchdog
pragma target LVP  disabled        -- no Low Voltage Programming
--
enable_digital_io()                -- disable analog I/O (if any)
--
```

Choose an output pin

Let's choose an output pin to control our led. As you can see from the circuit, our led is connected to pin #2. Let's check our datasheet to find the pin name from the pin out diagram.

The PDF datasheet for this PIC and for all others can be downloaded from the microchip website. Here is the datasheet for this PIC *http://ww1.microchip.com/downloads/en/DeviceDoc/30292c.pdf* , and here is the pin out diagram from the datasheet:



As you can see, we are using the pin RA0/ANO at pin #2. RA0 is the pin name we are looking for. AN0 is another name for this same pin (used in the analog to digital tutorial), but we can ignore it in this tutorial. In the JAL language RA0 is written as pin_A0

Now let's read the details of this pin in the datasheet on page 10. As you can see RA0 is a TTL Digital I/O pin. We are checking this to make sure it is not a open drain output. Open drain outputs (like pin RA4) require a pull-up resistor from the pin to V+

PIC16F87XA

TABLE 1-3: PIC16F874A/877A PINOUT DESCRIPTION

Pin Name	PDIP Pin#	PLCC Pin#	TQFP Pin#	QFN Pin#	I/O/P Type	Buffer Type	Description
RA0/AN0 RA0 AN0	2	3	19	19	 I/O I	TTL	PORTA is a bidirectional I/O port. Digital I/O. Analog input 0.

Legend: I = input O = output I/O = input/output P = power
— = Not used TTL = TTL input ST = Schmitt Trigger input

Note 1: This buffer is a Schmitt Trigger input when configured as the external interrupt.
2: This buffer is a Schmitt Trigger input when used in Serial Programming mode.
3: This buffer is a Schmitt Trigger input when configured in RC Oscillator mode and a CMOS input otherwise.

Now write code for pin A0. We are writing an “alias” only because in the future we can refer to pin 2 (A0) as “led”. This way we no longer need to remember the name of the pin (except for the directional register in the next line of code we will write).

```
--
-- You may want to change the selected pin:
alias   led      is pin_A0
```

Configure the pin as an input or output

Now we must tell the PIC if this is an input or an output pin. The directional setting is always named (pin_ + pinname_ + direction). Since we are writing data to the port, to turn the led on, it is an output.

```
pin_A0_direction =  output
```

We could make an alias for this as well: “alias led_direction is pin_A0_direction”, then write “led_direction = output”. This way, we can change it from output to input in the middle of the program without knowing the pin name. But in this case, we will only use pin_A0_direction once in our program so there is no need to make an alias.

Write your program

So, now that we have the led under our control, let’s tell it what to do.

We will want our led to continue doing whatever we want it to do forever, so we’ll make a loop

```
forever loop
```

It is good practice to indent before each line within the loop for readability. 3 spaces before each line is the standard for Jallib.

In this loop, we will tell the led to turn on.

```
   led = ON
```

now have some delay (250ms) a quarter of a second so we can see the led on.

```
   _usec_delay(250000)
```

turn the led off again

```
   led = OFF
```

and have another delay before turning it back on again

```
   _usec_delay(250000)
```

close our loop, when the PIC gets to this location, it will go back to the beginning of the loop

```
end loop
--
```

And that’s it for our code. Save your file, It should look something like this:

```
-- -------------------------------------------------------
-- Title: Blink-a-led of the Microchip pic16f877a
--
-- Author: Rob Hamerling, Copyright (c) 2008..2009, all rights reserved.
--
-- Adapted-by:
--
-- Compiler: 2.4l
--
-- This file is part of jallib  (http://jallib.googlecode.com)
-- Released under the BSD license (http://www.opensource.org/licenses/bsd-
license.php)
--
-- Description:
```

```
-- Sample blink-a-led program for Microchip PIC16f877a.
--
-- Sources:
--
-- Notes:
--  - File creation date/time: 14 Oct 2009 20:24:20.
--
-- -----------------------------------------------------
--
include 16f877a                    -- target PICmicro
--
-- This program assumes a 20 MHz resonator or crystal
-- is connected to pins OSC1 and OSC2.
pragma target clock 20_000_000     -- oscillator frequency
-- configuration memory settings (fuses)
pragma target OSC  HS              -- HS crystal or resonator
pragma target WDT  disabled        -- no watchdog
pragma target LVP  disabled        -- no Low Voltage Programming
--
enable_digital_io()                -- disable analog I/O (if any)
--
-- You may want to change the selected pin:
alias   led      is pin_A0
pin_A0_direction =  output
--
forever loop
   led = on
   _usec_delay(250000)
   led = off
   _usec_delay(250000)
end loop
--
```

Compile your code to .hex

Now let’s get this beautiful code onto our PIC. Your PIC cannot understand JAL, but it does understand hex, this is what the compiler is for. The compiler takes people readable code and converts it to code your PIC can understand.

If you are using JALEdIt, click the compile menu at the top and choose compile.

If you are using your own text editor in windows, you will need to open windows command prompt. Click start -> run and type cmd and press OK. Now type (path to compiler) + (path to your .jal file) + (-s) + (path to JALLIB libraries) + (options) Here’s an example:

C:\jalv2\compiler\jalv2.exe "C:\jalv2\workspace\blink_a_led\blink_a_led.jal" -s "C:\jalv2\lib" -no-variable-reuse

The option -no-variable-reuse will use more PIC memory, but will compile faster.

If all this went ok, you will now have a blink_a_led.hex located in the same directory as your blink_a_led.jal, If there where errors or warnings, the compiler will tell you.

A error means the code has an problem and could not generate any .hex file. If there is a warning, the hex file was generated ok and may run on your PIC but the code should be fixed.

Write the hex file to your PIC

Take your PIC out of your circuit and put it in your programmer. With your programming software, open the blink_a_led.hex file. You should see that hex data loaded in your software. Now click the Write button. Your software will tell you when it is done.

Let's Try It

Put your PIC back into your circuit, double check your circuit if you haven’t already, and make sure your PIC is facing the correct direction. Apply power to your circuit.

It’s alive! You should see your led blinking! Congratulations on your first JALv2 + JALLIB circuit!

Here's a youtube video of the result: *http://www.youtube.com/watch?v=PYuPZO7isoo*

I strongly suggest you do this tutorial next: *Serial Port and RS-232 for communication*.

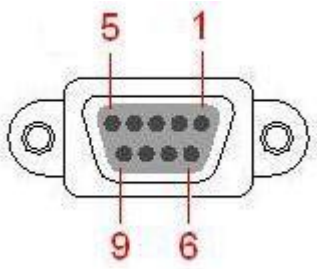
Serial Port and RS-232 for communication

Matthew Schinkel
Jallib Group

In this tutorial we are going to learn how use TX & RX pins for serial communication to your PC, and also learn communicate with another PIC or external device via RS-232.

What is a serial port?

You may have forgotten about this important part of history "The serial port". You have forgotten because you have been too up-to-date on all the new technologies such as USB and Bluetooth, but you have left the good old technologies in the past. Well, now it's time to put that funny looking port on the back of your PC to some good use! If you don't have a serial port on your PC, you can get a USB to serial converter/adapter.



At one time, there was a wide range of devices that used the serial port such as a mouse, keyboard, old GPS, modems and other networking.

In our case, we will use a serial port to send data to our PC, or to send data a second PIC. I find it most useful for troubleshooting my code, and for sending other readable information to my PC without the use of additional hardware such as a LCD. LCDs & displays can be an expensive addition to your circuit.

What is RS-232?

RS-232 is the data transfer standard used on serial ports. Basically this is composed of one start bit, some data bits, parity bit, and one or two stop bits. The transfer speed as well as the number of start, stop and data bits must match for both the transmitter and receiver. We will not need to cover the way in which it is transferred since the PIC does it for us. We will only need to know the following:

1. The number of start bits (always 1)

2. The Parity (usually no parity)

3. The number of data bits (usually 8)

4. The number of stop bits (1 or 2)

5. The data transmission speed

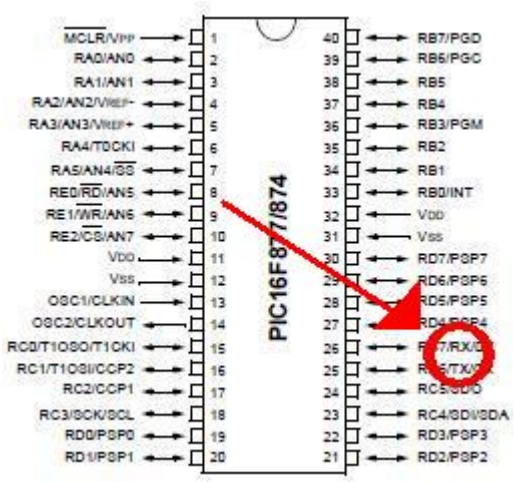
6. The port number on your PC

You will be able to choose the transmission speed yourself. The Jallib library we will be using will use 1 start bit, 8 data bits, no parity, and 1 stop bit. Your other device, such as your PC will also need to know this information.

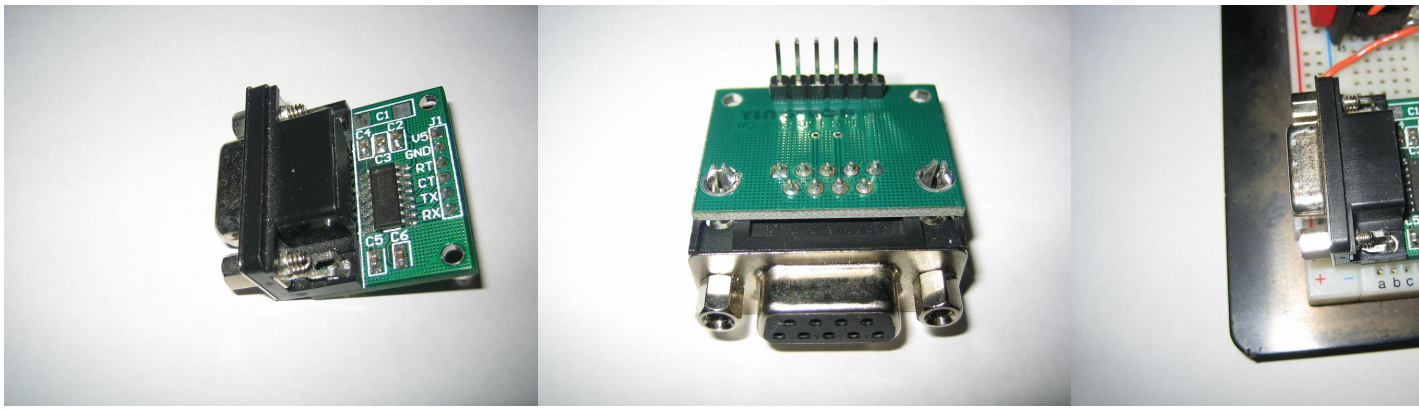
What do I need?

In the first part of this tutorial I will show you how to hook your serial port up to your PC. I will show you how to connect it to another PIC later on in this tutorial. I feel that connectivity to your PC is quite important. You will need:

1. A PIC that has TX and RX Pin names. Most PIC's have them. Check your pinout diagram in the PIC's datasheet.



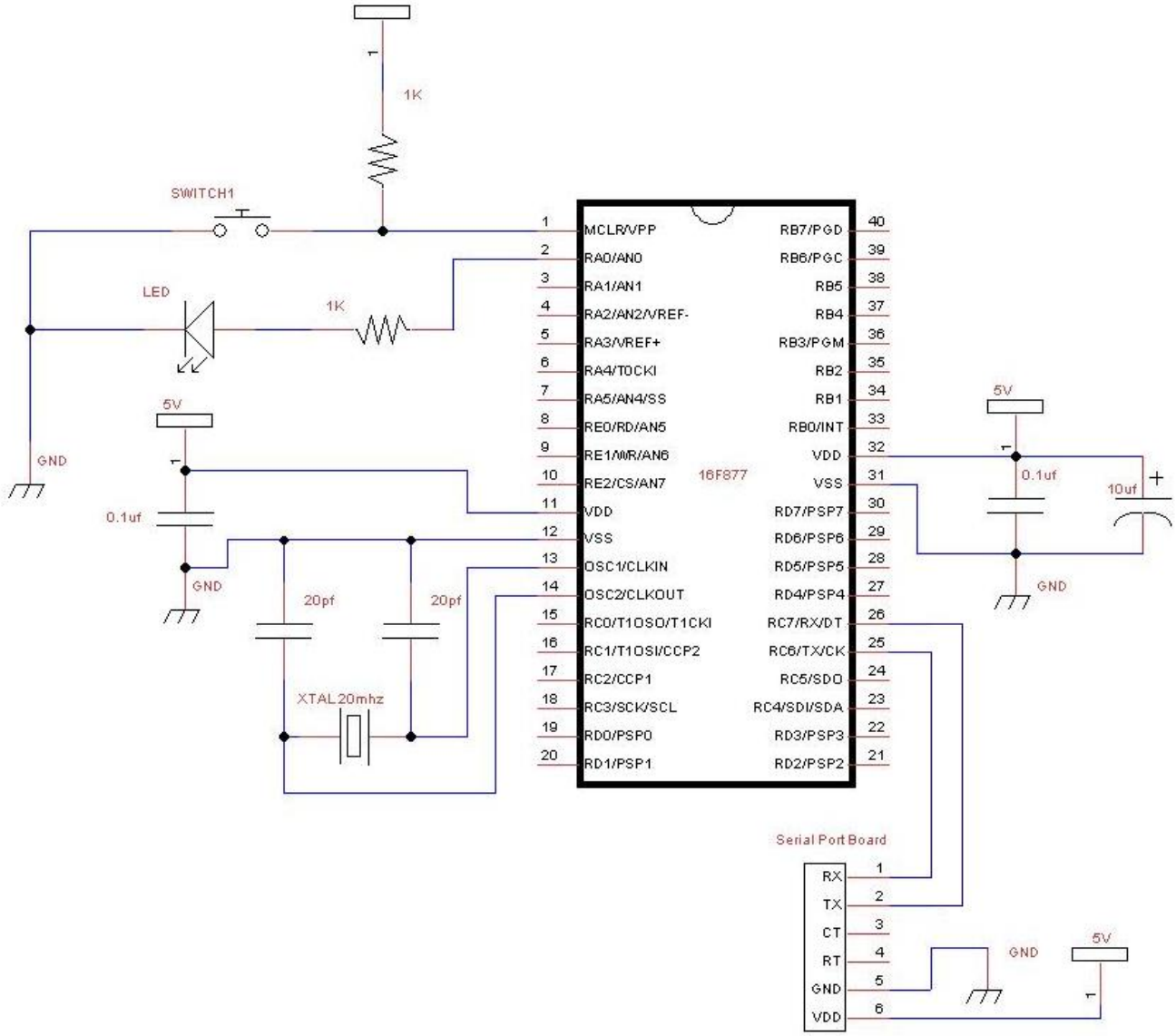
2. A serial port board. You can buy one on the net, or build your own. *Building a serial port borad with the max232 device* for more information. A serial port board is needed for voltage conversion. Serial ports output voltages up to 12v and go lower than 0v.



3. A regular RS-232 Cable (make sure it is not a null modem cable, they look the same). You can check what type of cable you have with your multimeter. Put your multimeter on each pin of your cable starting with pin 1. Check for a zero ohm reading. This will check that the pins are the same at both ends. Null modem cables have some pins crossed.

Build your circuit

The circuit will be quite simple, you can take your blink a led circuit, and attach your serial port board. Here's a schematic with 16F877. We will be using the TX and RX pins:



Test your circuit

Before you write your own code, you should make sure your circuit actually works.

Go into the sample directory within your jalv2/jallib installation. Find your pic, and look for a serial hardware sample such as 16f877a_serial_hardware.jal. Then compile it and burn it to your PIC. Don't turn on your circuit yet, we are not ready.

On your PC, you will have to install some serial communications program such as RealTerm. RealTerm is free and open source. I will use RealTerm for this tutorial. You can download it here:

http://realterm.sourceforge.net/

Open RealTerm and click on the "Port" tab, we need to select the port & speed, etc to the following values:

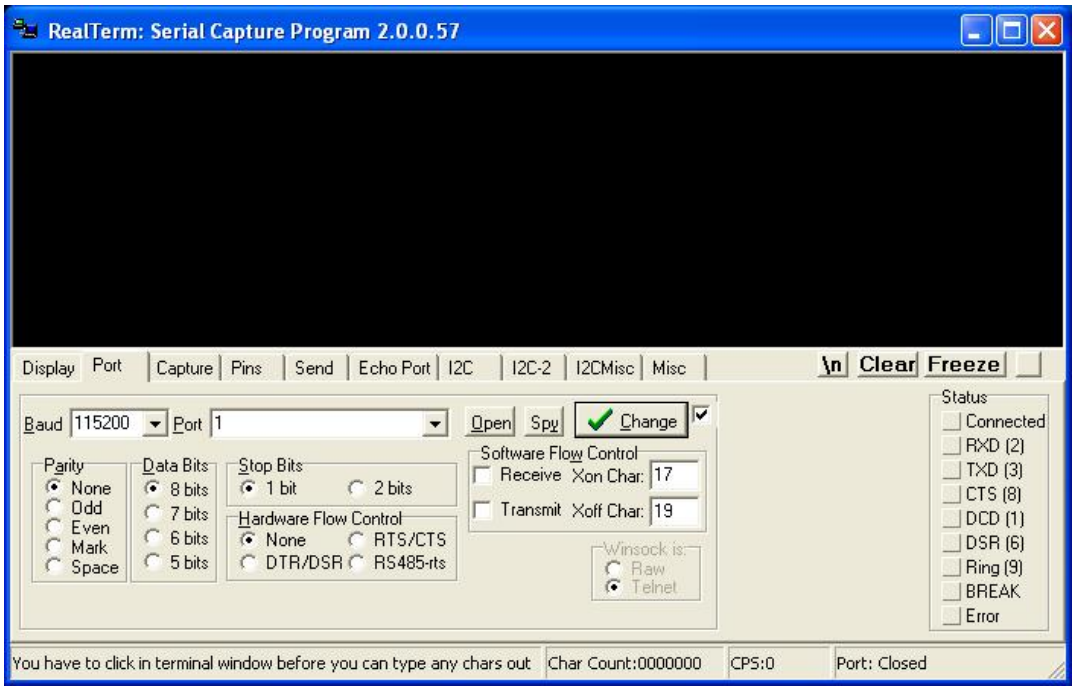
1. The Parity = no parity

2. The number of data bits = 8

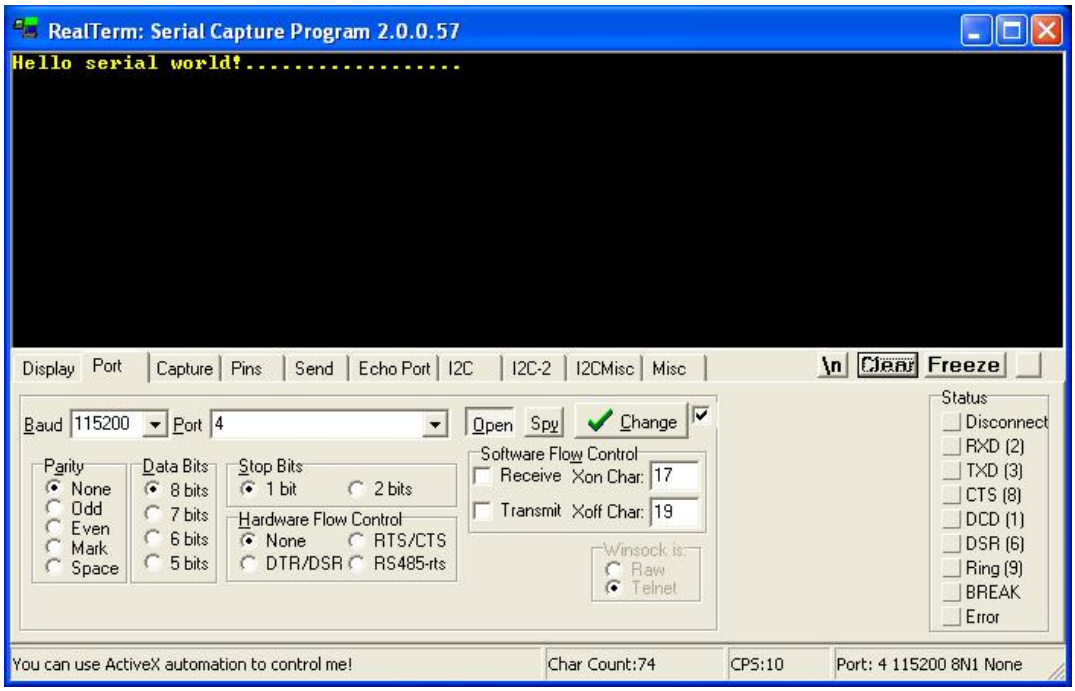
3. The number of stop bits = 1

4. The data transmission speed = 115200

5. The port number on your PC

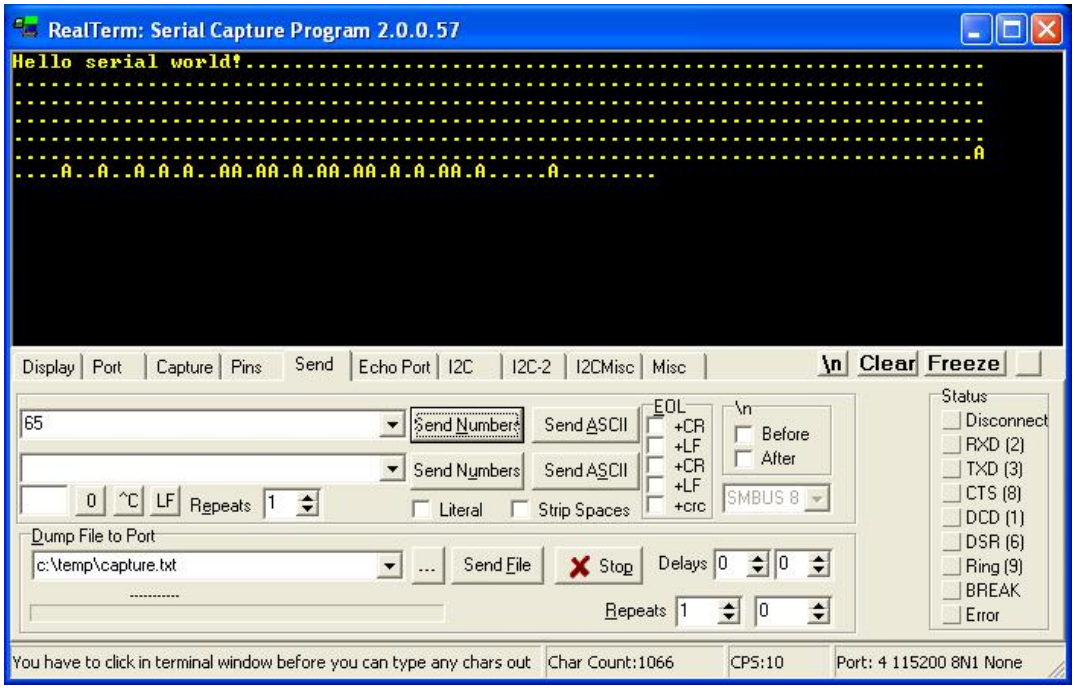


Now press "Open" in RealTerm and turn on your circuit. If you now see "Hello serial world....." showing in within RealTerm on your PC, you are able to receive data.



If your circuit doesn't work, your serial port board may have TX and RX switched (you can try switching your TX/RX wires around), or you may have selected the wrong port number, some PCs have more than one serial port.

Now click on RealTerm's "send" tab, type in the number "65" in the first box and press "Send Numbers". If it sent ok, the PIC will echo this value back to you. You will see the ASCII character "A", which is the same as decimal 65. You can see a full ASCII chart at *asciitable.com*.

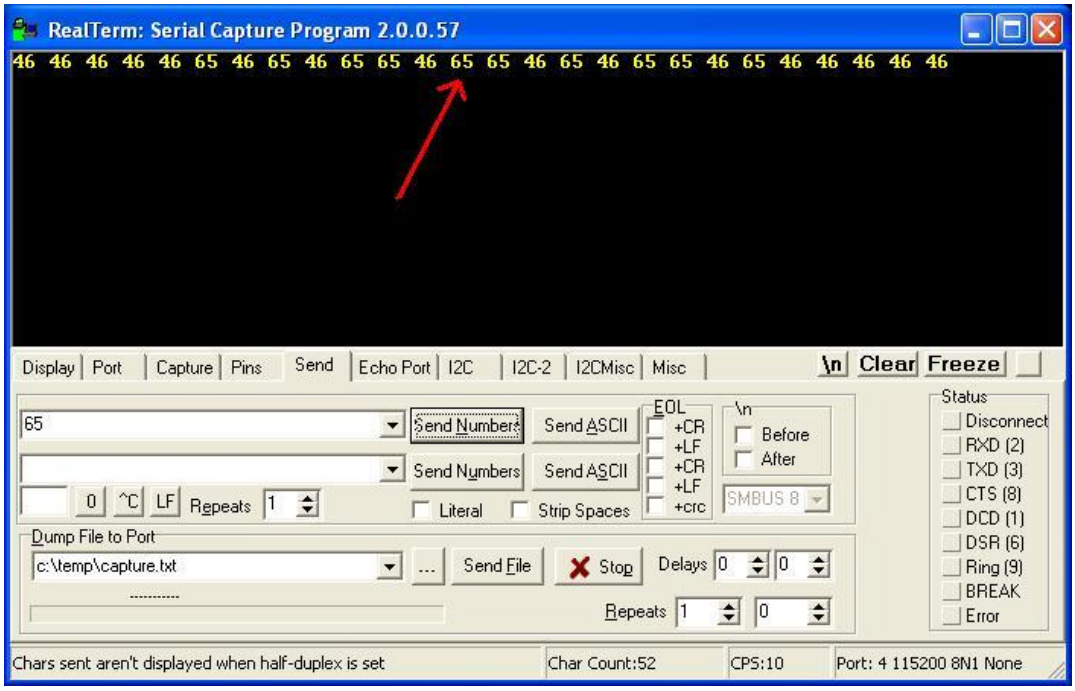


Now please change your RealTerm settings to receive decimal numbers by clicking on the "Display" tab, and choose "int8" under "Display As" at the left side. You will now continuously see the number "46" come in, and try sending the number "65" again. You will get the same number back on your screen.

int8 - shows integer numbers in RealTerm

Ascii - shows ascii text

Hex[space] - shows hex numbers with a space between each



Write code to send data from PIC to PC

Since this is one of the first circuits you will be building, I will try to give you detailed information so you can get some programming experience. We will continue with your code from "Blink a led". We will modify it to send data to your PC, Here's your original code:

```
include 16f877a                    -- target PICmicro
--
-- This program assumes a 20 MHz resonator or crystal
-- is connected to pins OSC1 and OSC2.
pragma target clock 20_000_000     -- oscillator frequency
-- configuration memory settings (fuses)
pragma target OSC  HS              -- HS crystal or resonator
pragma target WDT  disabled        -- no watchdog
pragma target LVP  disabled        -- no Low Voltage Programming
--
enable_digital_io()                -- disable analog I/O (if any)
--
-- You may want to change the selected pin:
alias   led      is pin_A0
pin_A0_direction =  output
--
forever loop
   led = on
   _usec_delay(250000)
   led = off
   _usec_delay(250000)
end loop
--
```

First we need to add serial functionality, I got the following code from 16f877a_serial_hardware.jal

```
-- ok, now setup serial;
const serial_hw_baudrate = 115_200
include serial_hardware
serial_hw_init()
```

So, now copy and past this into your code, I would put it somewhere after the line "enable_digital_io", and somewhere before your main program which starts at "forever loop".

This code will set your baudrate (speed), it will include the correct library file "serial_hardware", and it will initialize the library with "serial_hw_init()". You can change the speed if you wish, but you must change the speed in RealTerm as well.

Now we can put some code that will send data to your PC. If you want to send the number 65 to your PC, you must use this code:

```
serial_hw_data = 65
```

This code works because it is a procedure/function within serial_hardware.jal, and you have already included the serial_hardware library. serial_haredware.jal can be found in the "lib" folder of your jallib installation. You can open that file and read notes within it for more information and for other usable variables, functions and procedures.

Let's make your code send the number 65 when the led turns on, and send the number 66 when your led turns off. Just place your code after your "led = on", and after "led = off"

```
forever loop
   led = on
   serial_hw_data = 65 -- send 65 via serial port
   _usec_delay(250000)
   led = off
   serial_hw_data = 66 -- send 66 via serial port
   _usec_delay(250000)
end loop
```

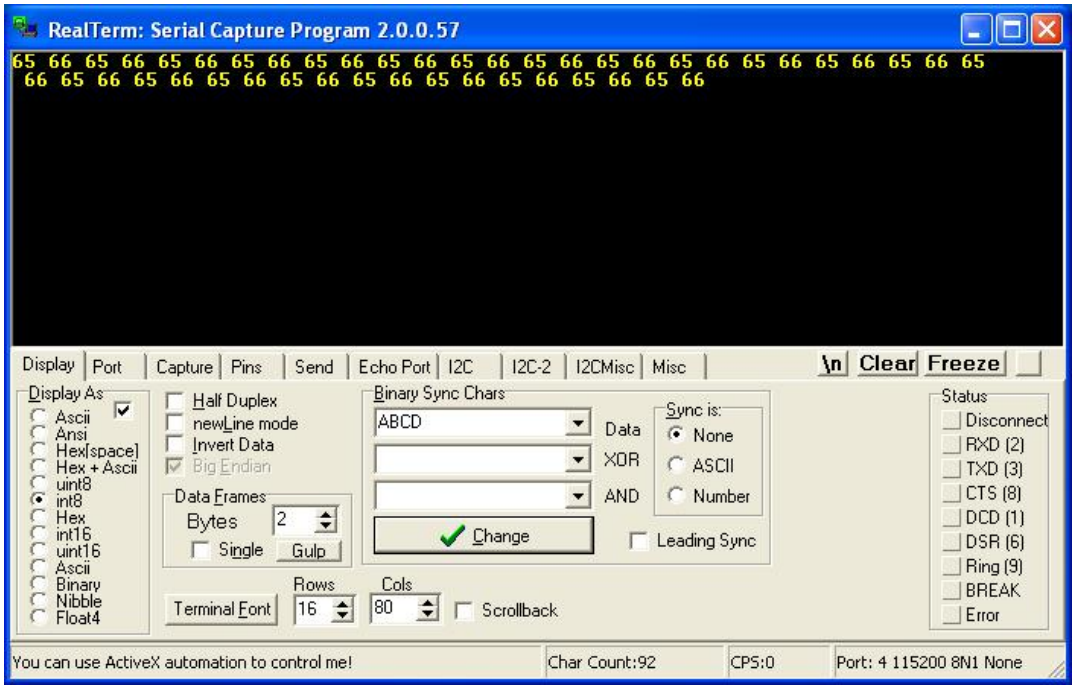
Or, if you wish to send Ascii letters to your PC instead, you could use the following:

```
forever loop
   led = on
   serial_hw_data = "A" -- send letter A via serial port
   _usec_delay(250000)
   led = off
   serial_hw_data = "B" -- send letter B via serial port
   _usec_delay(250000)
end loop
```

Both of the above loops will continuously send the decimal number's 65 and 66 via your serial port each time your led turns on or off. Your completed code should look like this:

```
include 16f877a                    -- target PICmicro
--
-- This program assumes a 20 MHz resonator or crystal
-- is connected to pins OSC1 and OSC2.
pragma target clock 20_000_000     -- oscillator frequency
-- configuration memory settings (fuses)
pragma target OSC  HS              -- HS crystal or resonator
pragma target WDT  disabled        -- no watchdog
pragma target LVP  disabled        -- no Low Voltage Programming

enable_digital_io()                -- disable analog I/O (if any)

-- ok, now setup serial;@jallib section serial
const serial_hw_baudrate = 115_200
include serial_hardware
serial_hw_init()

-- You may want to change the selected pin:
alias   led      is pin_A0
pin_A0_direction =  output

forever loop
   led = on
   serial_hw_data = 65 -- send 65 via serial port
   _usec_delay(250000)
   led = off
   serial_hw_data = 66 -- send 66 via serial port
   _usec_delay(250000)
end loop
```

Compile and burn your code to your PIC, then turn on your circuit. You should get this while your led is blinking:



Awesome, now that you can send data to your pc! This was an important step since it will greatly help you with your troubleshooting by sending you readable information such as text, numbers and other types of data.

If you feel your programming skills are not as good as they should be, practice practice practice! Continue using the language reference at *http://www.casadeyork.com/jalv2/jalv2/index.html*

Write code to send data from PC to PIC

In the beginning, you may not have a use for sending data from your PC to your circuit, so you may skip this and go onto other things.

Here we are going to get the PIC to receive data from the PC. We will write some code that will only start blinking a led when you send data to the PIC. Also we will tell the PIC to send the number 65 to the PC once per second.

We will now learn to use the following variables from serial_hardware.jal:

serial_hw_data_available - If the PIC received data, this variable will equal TRUE, otherwise FALSE

serial_hw_data - If data is available, this variable will contain the data

So let's modify your current loop:

```
forever loop
   led = on
   serial_hw_data = 65 -- send 65 via serial port
   _usec_delay(250000)
   led = off
   serial_hw_data = 66 -- send 66 via serial port
   _usec_delay(250000)
end loop
```

First change it so it will send the number 65 to your PC every one second:

```
forever loop
   _usec_delay(1_000_000) -- one second delay
   serial_hw_data = 65    -- send 65 via serial port
end loop
```

We now can add an if statement to find out if there is serial data available:

```
forever loop
```

```
   _usec_delay(1_000_000) -- one second delay
   serial_hw_data = 65 -- send 65 via serial port

   if serial_hw_data_available then -- check if there is data available
   end if
end loop
```

You will need to create a variable "x" before your "forever loop", this variable will hold the data when you want to receive it:

```
var byte x
```

Now you have a place to store the data, so you may now write a line within your "if" statement to get the data:

```
x = serial_hw_data
```

Then build a for loop after that to blink the led "x" number of times

```
for x loop   -- loop (x number of times) using data received on serial port

   led = on
   _usec_delay(250000)
   led = off
   _usec_delay(250000)
end loop
```

Here is your completed code:

```
include 16f877a                    -- target PICmicro
--
-- This program assumes a 20 MHz resonator or crystal
-- is connected to pins OSC1 and OSC2.
pragma target clock 20_000_000     -- oscillator frequency
-- configuration memory settings (fuses)
pragma target OSC  HS              -- HS crystal or resonator
pragma target WDT  disabled        -- no watchdog
pragma target LVP  disabled        -- no Low Voltage Programming

enable_digital_io()                -- disable analog I/O (if any)

-- ok, now setup serial;@jallib section serial
const serial_hw_baudrate = 115_200
include serial_hardware
serial_hw_init()

-- You may want to change the selected pin:
alias   led      is pin_A0
pin_A0_direction =  output

var byte x

forever loop              -- continue forever
   _usec_delay(1_000_000) -- one second delay
   serial_hw_data = 65    -- send 65 via serial port

   if serial_hw_data_available then -- check if data is ready for us
      x = serial_hw_data            -- get the data
      for x loop                    -- loop "x" number of times
         led = on                   -- turn the led on
         _usec_delay(250000)        -- 250ms delay
         led = off                  -- turn the led off
         _usec_delay(250000)        -- 250ms delay
      end loop                      -- loop
   end if                           -- end the if statement

end loop
```

As you can see, this code will do the following:

1. delay 1 second

2. send the number 65 via serial port

3. see if there is data waiting for us, if so, get it and blink the led (the number of times of the data received)

4. loop back to the start

So, turn it on, you will start getting decimal numbers: "65 65 65 65 65" or ascii: "AAAAAA" in RealTerm. Now send your PIC the number 5, you will see your led blink 5 times. Now isn't that awesome!

PIC to PIC ommunication via serial port

Sending data to your PC is not the only use. If you have an extra PIC laying around, we can get two PIC's to talk to each other. And it's quite easy too!

I think you can do this on your own by now, you know how to make one PIC send data, and how to make a PIC receive data, so all you have to do is write some sending code on one PIC and receiving code on the other.

Build another circuit the same as your current one, then do the following:

1. connect the TX pin from PIC # 1 to the RX pin of PIC # 2

2. connect the RX pin from PIC # 1 to the TX pin of PIC # 2

On one of your PIC's, make it send data every one second, like we did before at *Write code to send data from PIC to PC*.

On the other PIC, make it loop continuously. Put an if statement in the loop that will see if there is data available, and if there is, make the led blink once, like we did at *Write code to send data from PC to PIC*.

You should then see your led blinking on your second PIC.

Wow, that was a lot, now I think you know your stuff!

Chapter 2

PIC internals

Topics:

This chapter covers main and widely used PIC microcontroller internals (also referred as *PIC peripherals* in datasheets), like PWM, ADC, etc... For each section, you'll find some basic theory explaining how things works, then a real-life example.

Pulse Width Modulation (PWM)

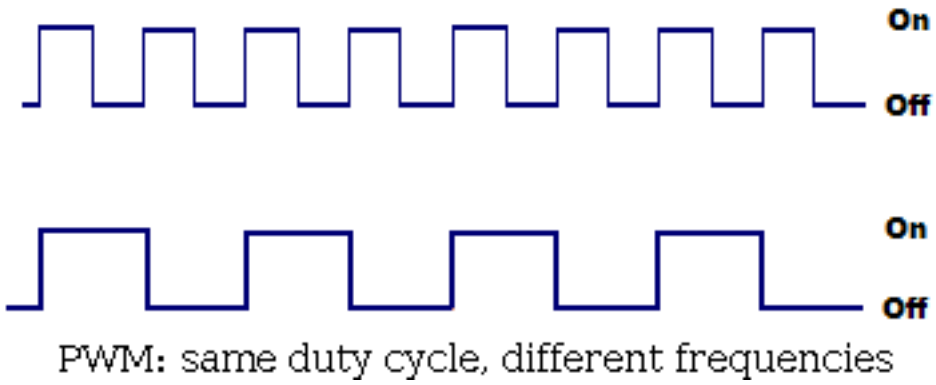
Sébastien Lelong
Jallib Group

In the following tutorials, we're going to (try to) have some fun with PWM. PWM stands for *Pulse Width Modulation*, and is quite weird when you first face this (this was at least my first feeling). So here's a brief explanation of what it is about.

How does PWM look like ?...

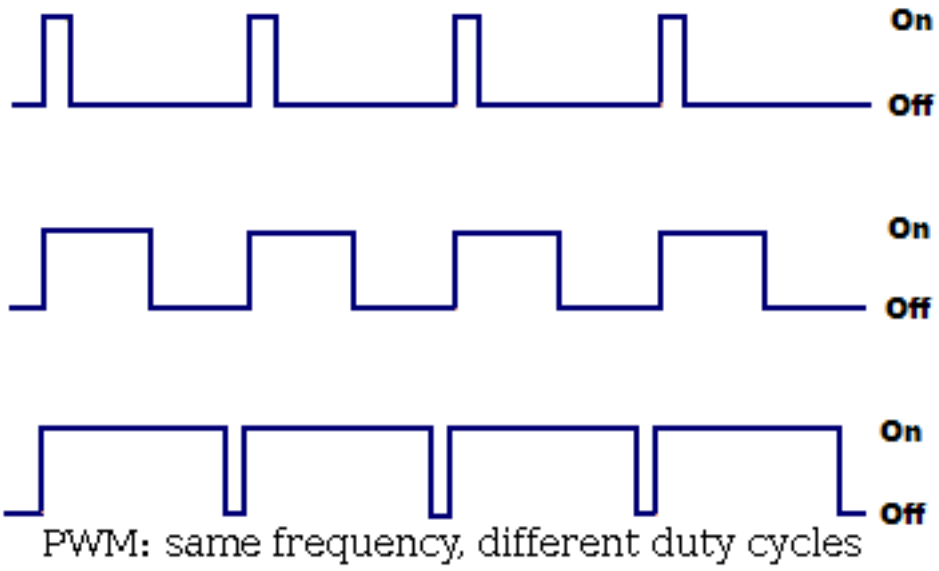
PWM is about switching one pin (or more) high and low, at different frequencies and duty cycles. This is a on/off process. You can either vary:

- the **frequency**,
- or the **duty cycle**, that is the proportion where the pin will be high



Both have a 50% duty cycle (50% on, 50% off), but the upper one's frequency is twice the bottom

Figure 1: PWM: same duty cycle, different frequencies.



Three different duty cycle (10%, 50% and 90%), all at the same frequency

Figure 2: PWM: same frequency, different duty cycles

But what is PWM for ? What can we do with it ? Many things, like:

- producing variable voltage (to control DC motor speed, for instance)
- playing sounds: duty cycle is constant, frequency is variable
- playing PCM wave file (PCM is Pulse Code Modulation)
- ...

That said, we're now goind to experiment these two major properties.

Dimming a LED with PWM

Sébastien Lelong
Jallib Group

One PWM channel + one LED = fun

For now, and for this first part, we're going to see how to *control the brightness of a LED*. If simply connected to a pin, it will light at its max brightness, because the pin is "just" high (5V).

Now, if we connect this LED on a PWM pin, maybe we'll be able to control the brightness: as previously said, *PWM can be used to produce variable voltages*. If we provide half the value (2.5V), maybe the LED will be half its brightness (though I guess the relation between voltage and brightness is not linear...). Half the value of 5V. How to do this ? Simply **configure the duty cycle to be 50% high, 50% low**.

But we also said *PWM is just about switching a pin on/off*. That is, either the pin will be 0V, or 5V. So how will we be able to produce 2.5V ? Technically speaking, we won't be able to produce a real 2.5V, but if PWM frequency is high enough, then, on the average, and from the LED's context, it's as though the pin outputs 2.5V.

Building the whole

Enough theory, let's get our hands dirty. Connecting a LED to a PWM pin on a 16f88 is quite easy. This PIC has quite a nice feature about PWM, it's possible to select which pin, between RB0 and RB3, will carry the PWM signals. Since I use *tinybootloader* to upload my programs, and since tiny's fuses are configured to select the RB0 pin, I'll keep using this one (if you wonder why tinybootloader interferes here, *read this post*).

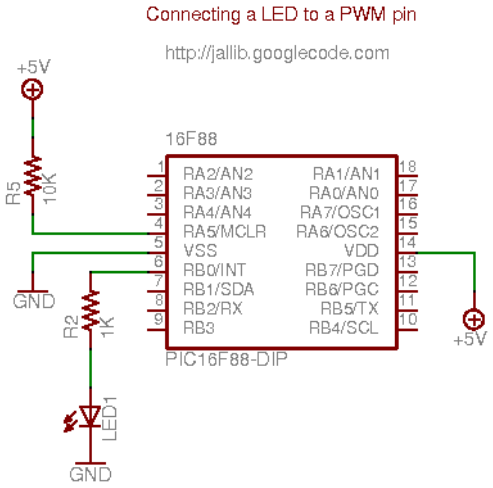
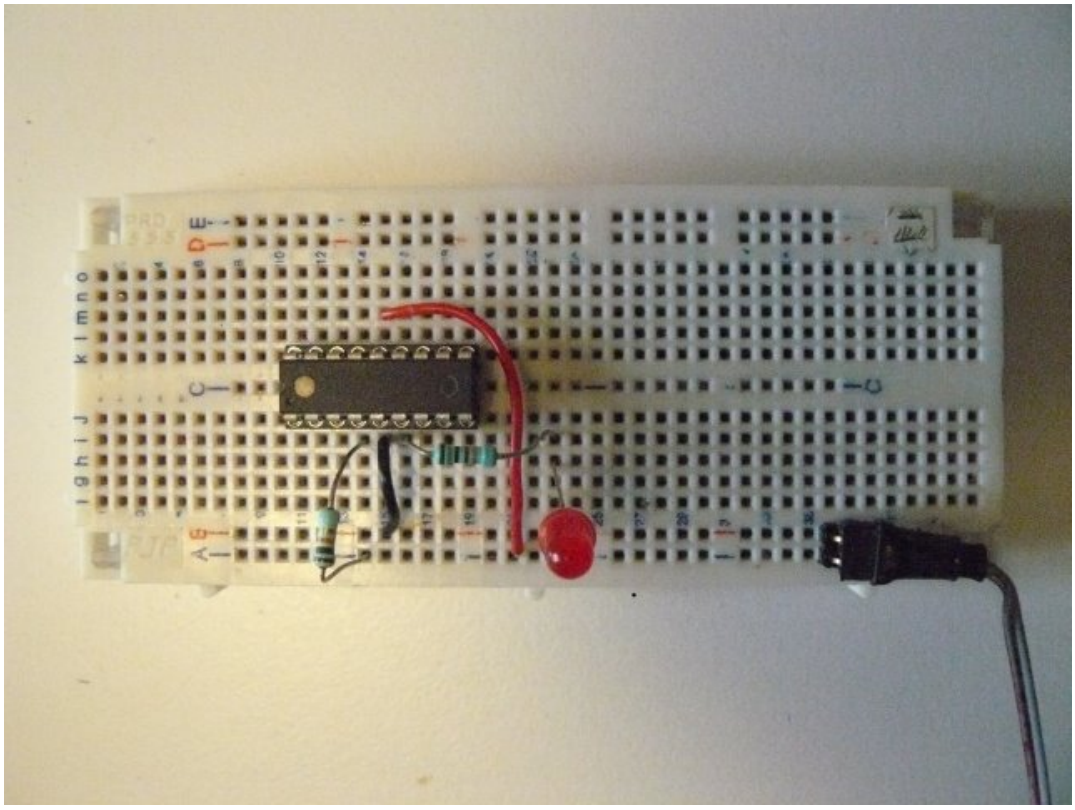
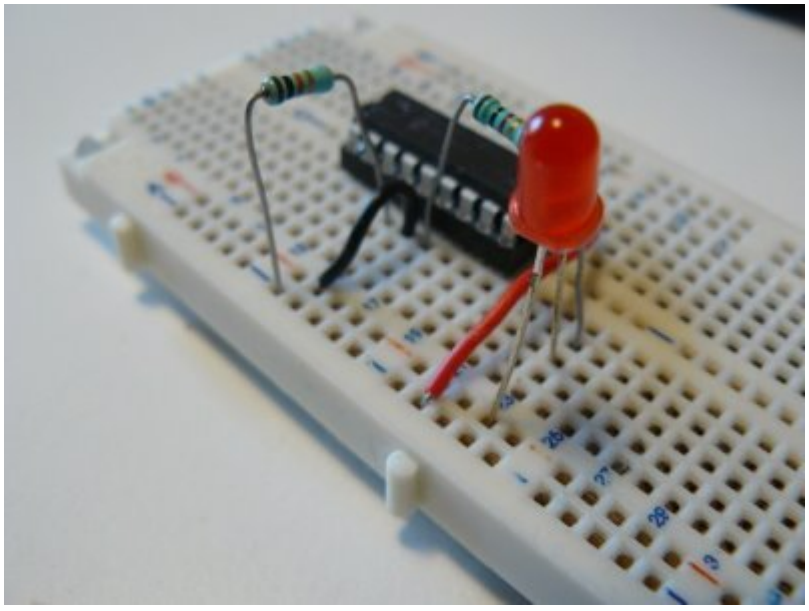


Figure 3: Connecting a LED to a PWM pin

On a breadboard, this looks like this:



The connector brings +5V on the two bottom lines (+5V on line A, ground on line B).



LED is connected to RB0

Writing the software

For this example, I took one of the 16f88's sample included in jallib distribution (*16f88_pwm_led.jal*), and just adapt it so it runs at 8MHz, using internal clock. It also select RB0 as the PWM pin.

So, step by step... First, as we said, we must select which pin will carry the PWM signals...

```
pragma target CCP1MUX      RB0     -- ccp1 pin on B0
```

and configure it as output

```
var volatile bit pin_ccp1_direction is pin_b0_direction
pin_ccp1_direction = output
-- (simply "pin_b0_direction = output" would do the trick too)
```

Then we include the PWM library.

```
include pwm_hardware
```

Few words here... This library is able to handle **up to 10 PWM channels** (PIC using CCP1, CCP2, CCP3, CCP4, ... CCP10 registers). Using conditional compilation, it **automatically selects the appropriate underlying PWM libraries**, for the selected target PIC.

Since 16f88 has only one PWM channel, it just includes "pwm_ccp1" library. If we'd used a 16f877, which has two PWM channels, it would include "pwm_ccp1" *and* "pwm_ccp2" libraries. What is important is it's transparent to users (you).

OK, let's continue. We now need to configure the **resolution**. What's the resolution ? Given a frequency, the **number of values you can have for the duty cycle** can vary (you could have, say, 100 different values at one frequency, and 255 at another frequency). Have a look at the datasheet for more.

What we want here is to have the **max number of values we can for the duty cycle**, so we can select the exact brightness we want. We also want to **have the max frequency** we can have (ie. no pre-scaler).

```
pwm_max_resolution(1)
```

If you read the *jalapi documentation* for this, you'll see that the frequency will be 7.81kHz (we run at 8MHz).

PWM channels can be turned on/off independently, now we want to activate our channel:

```
pwm1_on()
```

Before we dive into the forever loop, I forgot to mention PWM can be used in **low or high resolution**. On *low resolution*, duty cycles values range from *0 to 255* (8 bits). On *high resolution*, values range from *0 to 1024* (10 bits). In this example, we'll use low resolution PWM. For high resolution, you can have a look at the other sample, *16f88_pwm_led_highres.jal*. As you'll see, there are very few differences.

Now let's dive into the loop...

```
forever loop
   var byte i
   i = 0
   -- loop up and down, to produce different duty cycle
   while i < 250 loop
      pwm1_set_dutycycle(i)
      _usec_delay(10000)
      i = i + 1
   end loop
   while i > 0 loop
      pwm1_set_dutycycle(i)
      _usec_delay(10000)
      i = i - 1
   end loop
   -- turning off, the LED lights at max.
   _usec_delay(500000)
   pwm1_off()
   _usec_delay(500000)
   pwm1_on()

end loop
```

Quite easy right ? There are *two main waves*: one will light up the LED progressively (0 to 250), another will turn it off progressively (250 to 0). On each value, we set the duty cycle with `pwm1_set_dutycycle(i)` and wait a little so we, humans, can see the result.

About the result, how does this look like ? See this video: *http://www.youtube.com/watch?v=r9_TfEmUSf0*

"I wanna try, where are the files ?"

To run this sample, you'll need the *last jallib pack* (at least 0.2 version). You'll also find the exact code we used *here*.

Producing sounds with PWM and a piezo buzzer

Sébastien Lelong
Jallib Group

In *Dimming a LED with PWM*, we had fun by controlling the brightness of a LED, using PWM. This time, we're going to have even more fun with a *piezo buzzer*, or a small *speaker*.

If you *Pulse Width Modulation (PWM)*, with PWM, you can either vary the **duty cycle** or the **frequency**. Controlling the brightness of a LED, ie. produce a variable voltage on the average, can be done by having a *constant frequency* (high enough) and *vary the duty cycle*. This time, this will be the opposite: we'll have a constant duty cycle, and vary the frequency.

What is a piezo buzzer ?

It's a "component" with a material having *piezoelectric* ability. *Piezoelectricity* is the ability for a material to produce voltage when it get distorted. The reverse is also true: *when you produce a voltage, the material gets distorted*. When you stop producing a voltage, it gets back to its original shape. If you're fast enough with this on/off voltage setting, then *the piezo will start to oscillate*, and will **produce sound**. How sweet...

Constant duty cycle ? Why ?

So we now know why we need to vary the frequency. This will make the piezo oscillates more and less, and produces sounds at different levels. If you produce a 440Hz frequency, you'll get a nice *A3*.

But why having a *constant duty cycle* ? What is the role of the duty cycle in this case ? Remember: when making a piezo oscillate, it's not the amount of volts which is important, it's how you turn the voltage on/off[1]:

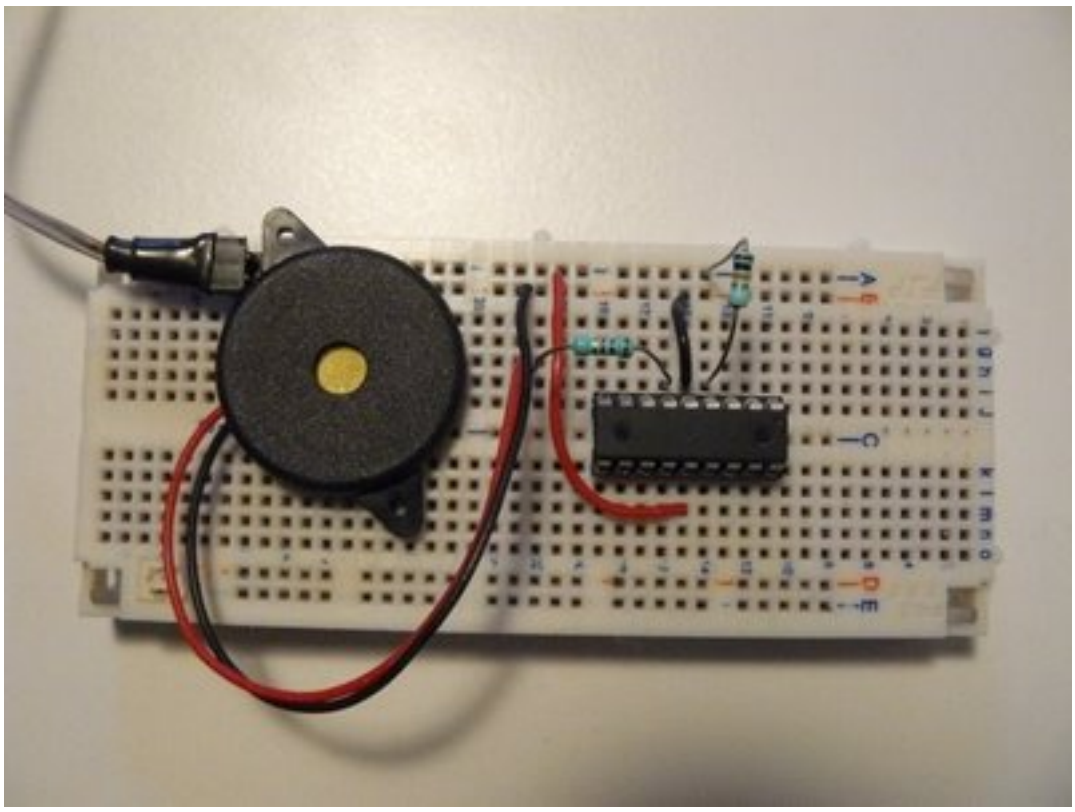
- **when setting the duty cycle to 10%**: during a period, piezo will get distorted 10% on the time, and remain inactive 90% on the time. *The oscillation proportion is low*.
- **when setting the duty cycle to 50%**: the piezo is half distorted, half inactive. The *oscillation proportion is high*, because the piezo oscillates at the its maximal amplitude, it's half and equally distorted and inactive.
- **when setting the duty cycle to 90%**: the piezo will get distorted during 90% of a period, then nothing. The *oscillation proportion is low again*, because the proportion between distortion and inactivity is not equal.

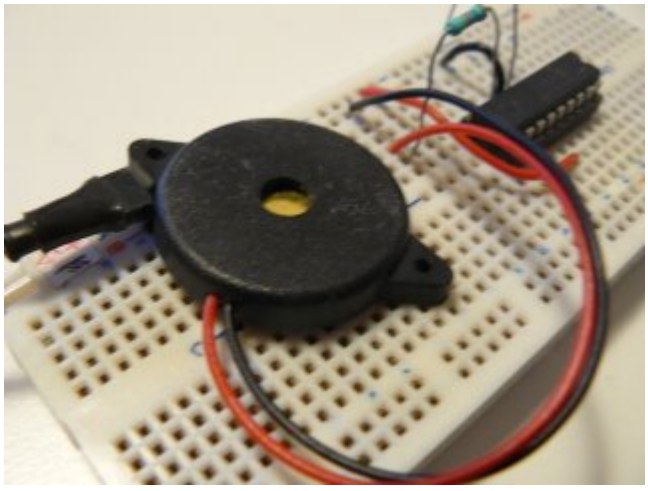
So, to summary, what is the purpose of the duty cycle in our case ? The **volume** ! You can vary the volume of the sound by modifying the duty cycle. 0% will produce no sounds, 50% will be the max volume. Between 50% and 100% is the same as between 0% and 50%. So, when I say when need a constant duty cycle, it's not that true, it's just that we'll set it at 50%, so the chances we hear something are high :)

Let's produce sounds !

The schematics will use is exactly the same as on the previous post with the LED, except the LED is replaced with a piezo buzzer, like this:

[1] I guess this is about energy or something like that. One guru could explain the maths here...





By the way, how to observe the "duty cycle effect" on the volume ? Just program your PIC with the previous experiment one, which control the brightness of a LED, and power on the circuit. I wanted to show a video with sounds, but the frequency is too high, my camera can't record it...

Anyway, that's a little bit boring, we do want sounds...

Writing the software

The software part has a lot of similarities with the *Dimming a LED with PWM*. The initialization is the same, I let you have a look. Only the `forever loop` has changed:

```
var dword counter = 0
forever loop
```

```
  for 100_000 using counter loop
     pwm_set_frequency(counter)
     -- Setting @50% gives max volume
     -- must be re-computed each time the frequency
     -- changes, because it depends on PR2 value
     pwm1_set_percent_dutycycle(50)
  end loop

end loop
```

Quite straightforward:

- we "explore" frequencies between 0 and 100 000 Hz, using a `counter`
- we use `pwm_set_frequency(counter)` to set the frequency, in Hertz. It takes a dword as parameter (ie. you can explore a lot of frequencies...)
- finally, as we want a 50% duty cycle, and since its value is different for each frequency setting, we need to re-compute it on each loop.

Note: jallib's PWM libraries are coming from a "heavy refactoring" of Guru Stef Mientki's PWM library. While integrating it to jallib, we've modified the library so frequencies can be set and changed during program execution. This wasn't the case before, because the frequency was set as a constant.

So, how does this look like ? Hope you'll like the sweet melody :)

http://www.youtube.com/watch?v=xZ9OhQUKGtQ

"Where can I download the files ?"

As usual, you'll need the *last jallib pack* (at least 0.2 version). You'll also find the exact code we used *here*.

Analog-to-Digital Conversion (ADC)

Sébastien Lelong
Jallib Group

Analog-to-Digital Conversion is yet another nice feature you can get with a PIC. It's basically used to convert a voltage as an analog source (continuous) into a digital number (discrete).

ADC with water...

To better understand ADC, imagine you have some water going out of a pipe, and you'd like to know how many water it goes outside. One approach would be to collect all the water in a bucket, and then measure what you've collected. But what if water flow never ends ? And, more important, what if water flow isn't constant and you want to measure the flow in real-time ?

The answer is ADC. With ADC, you're going to extract samples of water. For instance, you're going to put a little glass for 1 second under the pipe, every ten seconds. Doing the math, you'll be able to know the mean rate of flow.

The faster you'll collect water, the more accurate the rate will be. That is, if you're able to collect 10 glasses of water each second, you'll have a better overview of the rate of water than if you collect 1 glass each ten seconds. This is the process of making a continous flow a discrete, finite value. And this is about **resolution**, one important property of ADC (and this is also about clock speed...). The higher the resolution, the more accurate the results.

Now, what if the water flow is so high that your glass gets filled before the end of the sample time ? You could use a bigger glass, but let's assume you can't (scenario need...). This means you can't measure any water flow, this one has to be scaled according to your glass. On the contrary, the water flow may be so low samples you extract may not be relevant related to the glass size (only few drops). Fortunately, you can use a smaller glass (yes, scenario need) to scale down your sample. That is about **voltage reference**, another important property.

Leaving our glass of water, many PICs provide several **ADC channels**: pins that can do this process, measuring voltage as input. In order to use this peripheral, you'll first have to configure how many ADC channels you want. Then you'll need to specify the **resolution**, usually using 8 bits (0 to 255), 10 bits (0 to 1024) or even 12 bits (0 to 4096). Finally, you'll have to setup **voltage references** depending on the voltage spread you plan to measure.

ADC with jallib...

As usual, Microchip PICs offers a wide choice configuring ADC:

- N**ot all PICs** have ADC module (...)
- Analog pins are **dispatched differently** amongst PICs, still for user's sake, they have to be automatically configured as input. We thus need to know, for each PIC, where analog pins are...
- Some PICs have their **analog pins dependent** from each other, and some are **independent** (more on this later)
- **Clock configuration** can be different
- As previously stated, some PICs have **8-bits low resolution** ADC module, some have **10-bits high resolution** ADC module[2]
- Some PICs can have **two voltage references** (VRef+ and VRef-), only **one voltage reference** (Vref+) and some **can't handle voltage references at all**
- (and probably other differences I can't remember :)...

Luckily most of these differences are transparent to users...

Dependent and independent analog pins

OK, let's write some code ! But before this, you have to understand one very important point: some PICs have their analog pins *dependent* from each other, some PICs have their analog pins *independent* from each other. "What is this suppose to mean ?" I can hear...

Let's consider two famous PICs: 16F877 and 16F88. 16F877 datasheet explains how to configure the number of analog pins, and vref, setting **PCFG bits**:

PCFG3: PCFG0	AN7(1) RE2	AN6(1) RE1	AN5(1) RE0	AN4 RA5	AN3 RA3	AN2 RA2	AN1 RA1	AN0 RA0	VREF+	VREF-	CHAN/ Refs(2)
0000	A	A	A	A	A	A	A	A	VDD	Vss	8/0
0001	A	A	A	A	VREF+	A	A	A	RA3	Vss	7/1
0010	D	D	D	A	A	A	A	A	VDD	Vss	5/0
0011	D	D	D	A	VREF+	A	A	A	RA3	Vss	4/1
0100	D	D	D	D	A	D	A	A	VDD	Vss	3/0
0101	D	D	D	D	VREF+	D	A	A	RA3	Vss	2/1
011x	D	D	D	D	D	D	D	D	VDD	Vss	0/0
1000	A	A	A	A	VREF+	VREF-	A	A	RA3	RA2	6/2
1001	D	D	A	A	A	A	A	A	VDD	Vss	6/0
1010	D	D	A	A	VREF+	A	A	A	RA3	Vss	5/1
1011	D	D	A	A	VREF+	VREF-	A	A	RA3	RA2	4/2
1100	D	D	D	A	VREF+	VREF-	A	A	RA3	RA2	3/2
1101	D	D	D	D	VREF+	VREF-	A	A	RA3	RA2	2/2
1110	D	D	D	D	D	D	D	A	VDD	Vss	1/0
1111	D	D	D	D	VREF+	VREF-	D	A	RA3	RA2	1/2

A = Analog input D = Digital I/O

Figure 4: 16F877 ADC channels are controlled by PCFG bits

Want 6 analog pins, no Vref ? Then PCFG bits must be set to 0b1001. What will then be the analog pins ? RA0, RA1, RA2, RA3, RA5 and RE0. "*What if I want 7 analog pins, no Vref ?*" You can't because you'll get a Vref pin, no choice. "*What if I want 2 analog pins being RE1 and RE2 ?*" You can't, because there's no such combination. So, for this PIC, **analog pins are dependent from each other**, driven by a combination. In this case, you'll have to specify:

- the **number of ADC channels** you want,
- and *amongst* them, the **number of Vref channels**

Now, let's consider 16F88. In this case, there's no such table:

bit 6-0 **ANS<6:0>:** Analog Input Select bits
Bits select input function on corresponding AN<6:0> pins.
1 = Analog I/O(1,2)
0 = Digital I/O

Note 1: Setting a pin to an analog input disables the digital input buffer. The corresponding TRIS bit should be set to input mode when using pins as analog inputs. Only AN2 is an analog I/O, all other ANx pins are analog inputs.

2: See the block diagrams for the analog I/O pins to see how ANSEL interacts with the CHS bits of the ADCON0 register.

Figure 5: 16F88 ADC channels are controlled by ANS bits

Mmmh... OK, there are **ANS bits**, one for each analog pins. Setting an ANS bit to 1 sets the corresponding pin to analog. This means I can set whatever pin I want to be analog. "*I can have 3 analog pins, configured on RA0, RA4 and RB6. Freedom !*"

Analog pins are independent from each other in this case, you can do what you want. As a consequence, since it's not driven by a combination, you won't be able to specify the number of ADC channels here. Instead, you'll use `set_analog_pin()` procedure, and if needed, the reverse `set_digital_pin()` procedure. These procedures takes a analog pin number as argument. Say analog pin AN5 is on pin RB6. To turn this pin as analog, you just have to write `set_analog_pin(5)`, because this is about analog pin AN**5**, and not RB**6**.

Remember: as a consequence, these procedures don't exist when analog pins are dependent as in our first case.

Caution: it's not because there are PCFG bits that PICs have dependent analog pins. Some have PCFG bits which act exactly the same as ANS bits (like some of recent 18F)

Tip: how to know if your PIC has dependent or independent pins ? First have a look at its datasheet, if you can a table like the one for 16F877, there are dependent. Also, if you configure a PIC with dependent pins as if it was one with independent pins (and vice-versa), you'll get an error. Finally, if you get an error like: "*Unable to configure ADC channels. Configuration is supposed to be done using ANS bits but it seems there's no ANS bits for this PIC. Maybe your PIC isn't supported, please report !*", or the like, well, this is not a normal situation, so as stated, please report !

Once configured, using ADC is easy. You'll find `adc_read()` and **adc_read_low_res()** functions, for respectively read ADC in high and low resolution. Because low resolution is coded on 8-bits, `adc_read()` returns a `byte` as the result. **adc_read_low_res()** returns a `word`.

Example with 16F877, dependent analog pins

The following examples briefly explains how to setup ADC module when analog pins are dependent from each other, using PIC 16F877.

The following diagram is here to help knowing where **analog pins** (blue) are and where **Vref pins** (red) are:

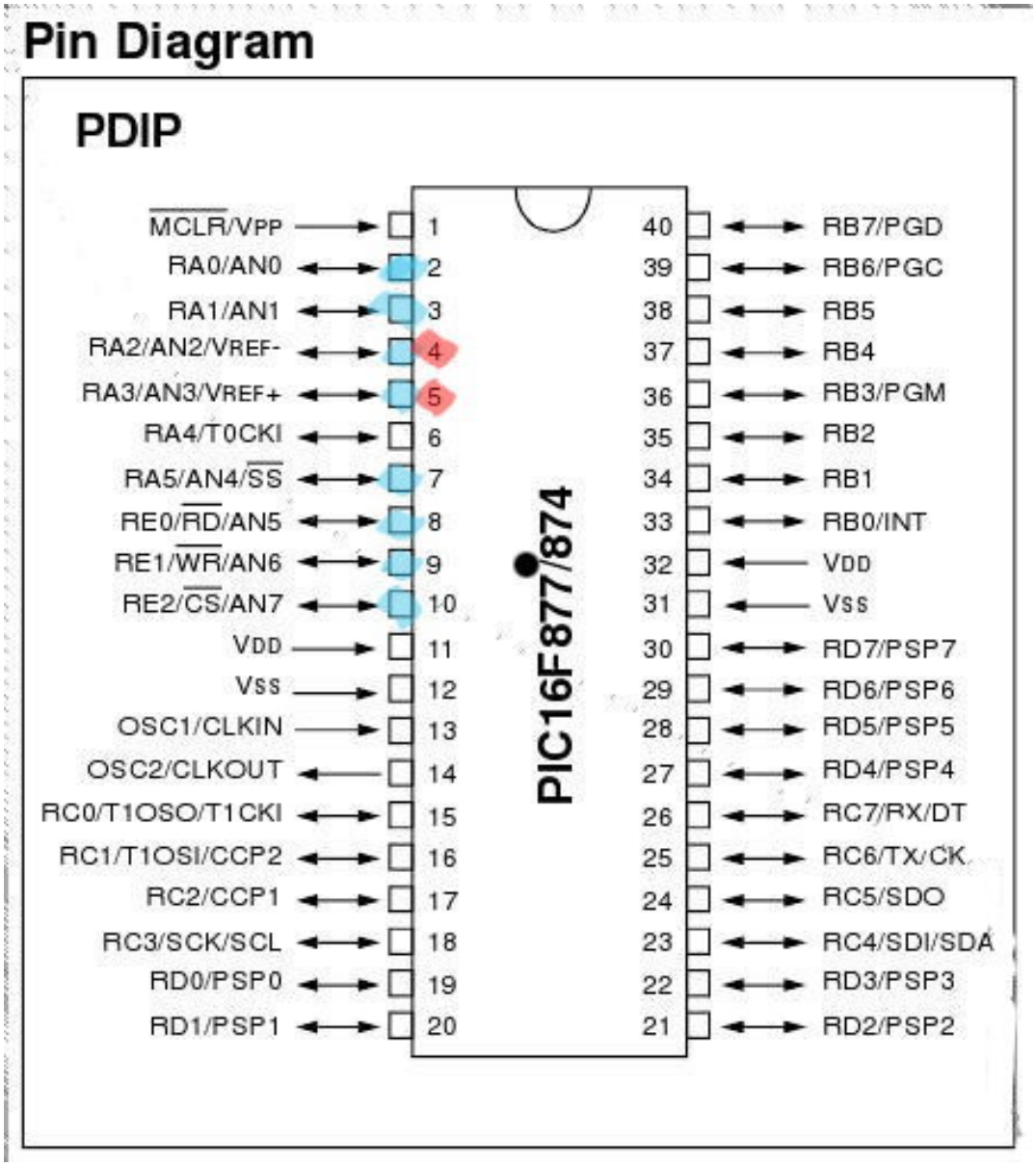


Figure 6: Analog pins and Vref pins on 16F877

Example 1: *16F877, with only one analog pin, no voltage reference*

```
-- beginning is about configuring the chip
-- this is the same for all examples for about 18F877
include 16f877
-- setup clock running @20MHz
pragma target OSC HS
pragma target clock 20_000_000
-- no watchdog
pragma target WDT  disabled
pragma target LVP  disabled
enable_digital_io()
include delay

-- ok, now setup serial, we'll use this
-- to get ADC measures
const serial_hw_baudrate = 19_200
include serial_hardware
serial_hw_init()


-- ok, now let's configure ADC
-- we want to measure using low resolution
-- (that's our choice, we could use high resolution as well)
const bit ADC_HIGH_RESOLUTION = false
-- we said we want 1 analog channel...
const byte ADC_NCHANNEL = 1
-- and no voltage reference
const byte ADC_NVREF = 0
-- now we can include the library
-- note it's now named "adc", not "adc_hardware" anymore
include adc
-- and run the initialization step
adc_init()


-- will periodically send those chars
var byte measure
forever loop
   -- get ADC result, on channel 0
   -- this means we're currently reading on pin RA0/AN0 !
   measure = adc_read_low_res(0)
   -- send it back through serial
   serial_hw_write(measure)

   -- and sleep a litte to prevent flooding serial...
   delay_1ms(200)
end loop
```

Example 2: *16F877, with 5 analog pins, 1 voltage reference, that is, Vref+*

This is almost the same as before, except we now want 5 (analog pins) + 1 (Vref) = **6 ADC channels** (yes, I consider Vref+ pin as an ADC channel).

The beginning is the same, here's just the part about ADC configuration and readings:

```
const bit ADC_HIGH_RESOLUTION = false
-- our 6 ADC channel
const byte ADC_NCHANNEL = 6
-- and one Vref pin
const byte ADC_NVREF = 1
-- the two parameters could be read as:
-- "I want 6 ADC channels, amongst which 1 will be
-- reserved for Vref, and the 5 remaining ones will be
-- analog pins"
```

```
include adc
adc_init()

-- will periodically send those chars
var byte measure
forever loop
   -- get ADC result, on channel 0
   -- this means we're currently reading on pin RA0/AN0 !
   measure = adc_read_low_res(0)
   -- send it back through serial
   serial_hw_write(measure)

   -- same for pin RA1/AN1
   measure = adc_read_low_res(1)
   serial_hw_write(measure)

   -- same for pin RA2/AN2
   measure = adc_read_low_res(2)
   serial_hw_write(measure)

   -- pin RA3/AN3 can't be read, since it's Vref+

   -- same for pin RA5/AN4
   -- 4 is from from "AN4" !
   measure = adc_read_low_res(4)
   serial_hw_write(measure)

   -- same for pin RE10/AN5
   measure = adc_read_low_res(5)
   serial_hw_write(measure)

   -- and sleep a litte to prevent flooding serial...
   delay_1ms(200)
end loop
```

Example with 16F88, independent analog pins

The following example is about setting up ADC module with PIC 16F88, where analog pins are independent from each other.

The following diagram is here to help knowing where **analog pins** (blue) are and where **Vref pins** (red) are:

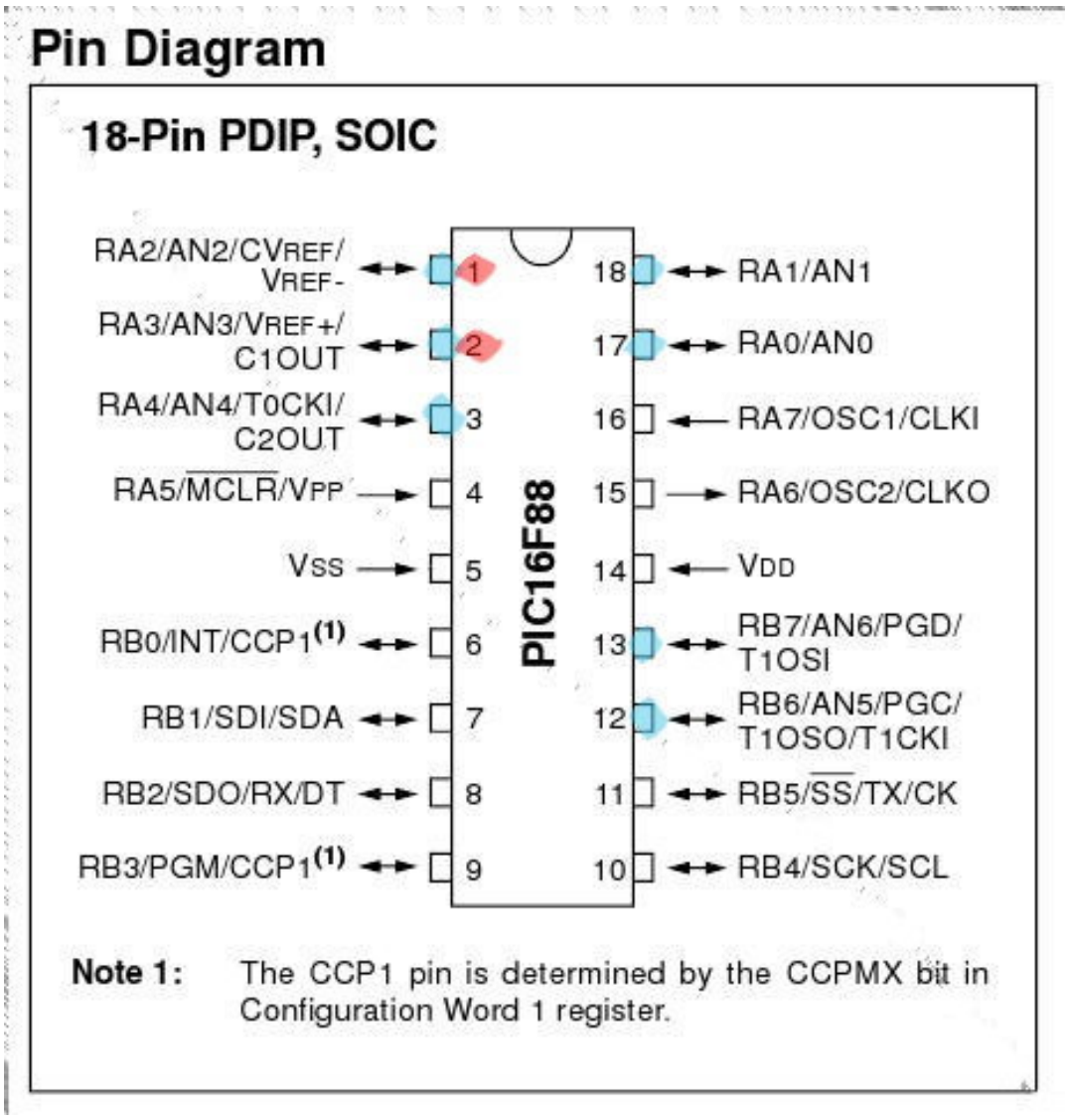


Figure 7: Analog pins and Vref pins on 16F88

Example 1: *16F88, analog pins on RA0/AN0, RA4/AN4 and RB6/AN5. No voltage reference.*

```
-- beginning is about configuring the chip
include 16f88
-- We'll use internal oscillator. It work @ 8MHz
pragma target CLOCK     8_000_000
pragma target OSC       INTOSC_NOCLKOUT
OSCCON_IRCF = 0b_111
pragma target WDT       disabled
enable_digital_io()

-- ok, now setup serial, we'll use this
-- to get ADC measures
const serial_hw_baudrate = 19_200
include serial_hardware
serial_hw_init()

-- now configure ADC
const bit ADC_HIGH_RESOLUTION = false
const byte ADC_NVREF = 0
-- we can't specify a number of ADC channel here,
-- or we'll get an error !
```

```
include adc
adc_init()
-- now we declare the pin we want as analog !
set_analog_pin(0)  -- RA0/AN0
set_analog_pin(4)  -- RA4/AN4
set_analog_pin(5)  -- RB6/AN5

-- reading is then the same
var byte measure
forever loop

   measure = adc_read_low_res(0)
   serial_hw_write(measure)

   measure = adc_read_low_res(4)
   serial_hw_write(measure)

   measure = adc_read_low_res(5)
   serial_hw_write(measure)

end loop
```

Whether you would want to turn RB6/AN5 into a digital pin again, you'd just call:

```
set_digital_pin(5)
```

I²C

Building an I²C slave, some theory (part 1)

Sébastien Lelong
Jallib Group

i2c is a nice protocol: it is quite fast, reliable, and most importantly, it's addressable. This means that on a single 2-wire bus, you'll be able to plug up to 128 devices using 7bits addresses, and even 1024 using 10bits address. Far enough for most usage... I won't cover i2c in depth, as there are *plenty resources* on the Web (and I personally like *this page*).

A few words before getting our hands dirty...

i2c is found in many chips and many modules. Most of the time, you create a master, like when accessing an EEPROM chip. This time, in this three parts tutorial, we're going to build a slave, which will thus respond to master's requests.

The *slave* side is somewhat more difficult (as you may have guess from the name...) because, as it does not initiate the talk, it has to listen to "events", and be as responsive as possible. You've guessed, we'll use *interrupts*. I'll only cover i2c hardware slave, that is using *SSP peripheral*[3]. Implementing an i2c software slave may be very difficult (and I even wonder if it's reasonable...).

There are different way implementing an i2c slave, but one seems to be quite common: defining a *finite state machine*. This implementation is well described in Microchip AppNote *AN734*. It is highly recommended that you read this appnote, and the i2c sections of your favorite PIC datasheet as well (I swear it's quite easy to read, and well explained).

Basically, during an i2c communication, there can be **5 distinct states**:

1. **Master writes, and last byte was an address**: to sum up, master wants to talk to a specific slave, identified by the address, it wants to send data (write)
2. **Master writes, and last byte was data**: this time, master sends data to the slave
3. **Master read, and last byte was an address**: almost the same as 1., but this time, master wants to read something from the salve
4. **Master read, and last byte was data**: just the continuation of state 3., master has started to read data, and still wants to read more data
5. **Master sends a NACK**: basically, master doesn't want to talk to the slave anymore, it hangs up...

Note: in the i2c protocol, one slave has actually two distinct addresses. One is for read operations, and it ends with bit 1. Another is for write operations, and it ends with bit 0.

Example: consider the following address (8-bits long, last bit is for operation type)

```
0x5C => 0b_0101_1100 => write operation
```

The same address for read operation will be:

```
0x93 => 0b_0101_1101 => read operation
```

Note: jallib currently supports up to 128 devices on a i2c bus, using 7-bits long addresses (without the 8th R/W bits). There's currently no support for 10-bits addresses, which would give 1024 devices on the same bus. If you need it, please let us know, we'll modify libraries as needed !

OK, enough for now. Next time, we'll see how two PICs must be connected for i2c communication, and we'll check the i2c bus is fully working, before diving into the implementation.

[3] some PICs have MSSP, this means they can also be used as i2c hardware Master

Setting up and checking an I²C bus (part 2)

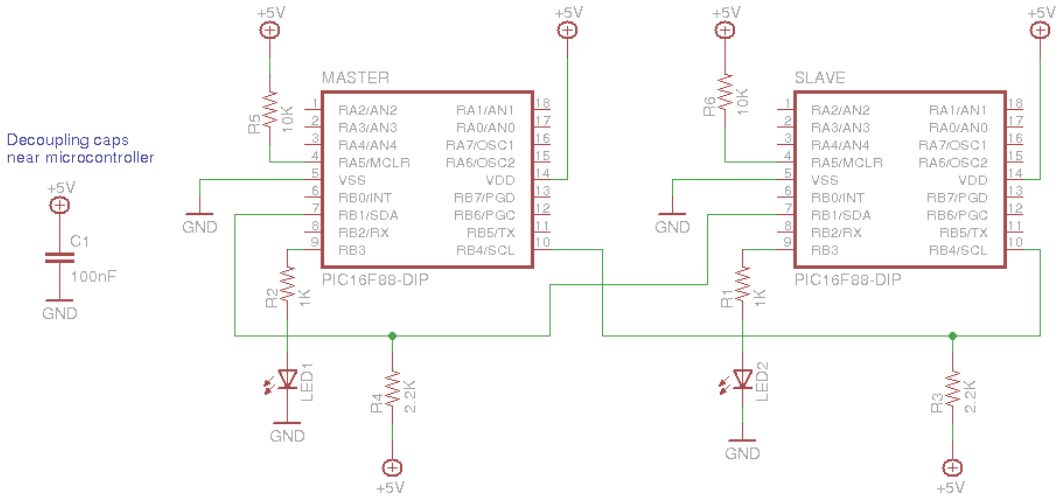
Sébastien Lelong
Jallib Group

In *Building an I²C slave, some theory (part 1)*, we saw a basic overview of how to implement an i2c slave, using a finite state machine implementation. This time, we're going to get our hands a little dirty, and starts connecting our master/slave together.

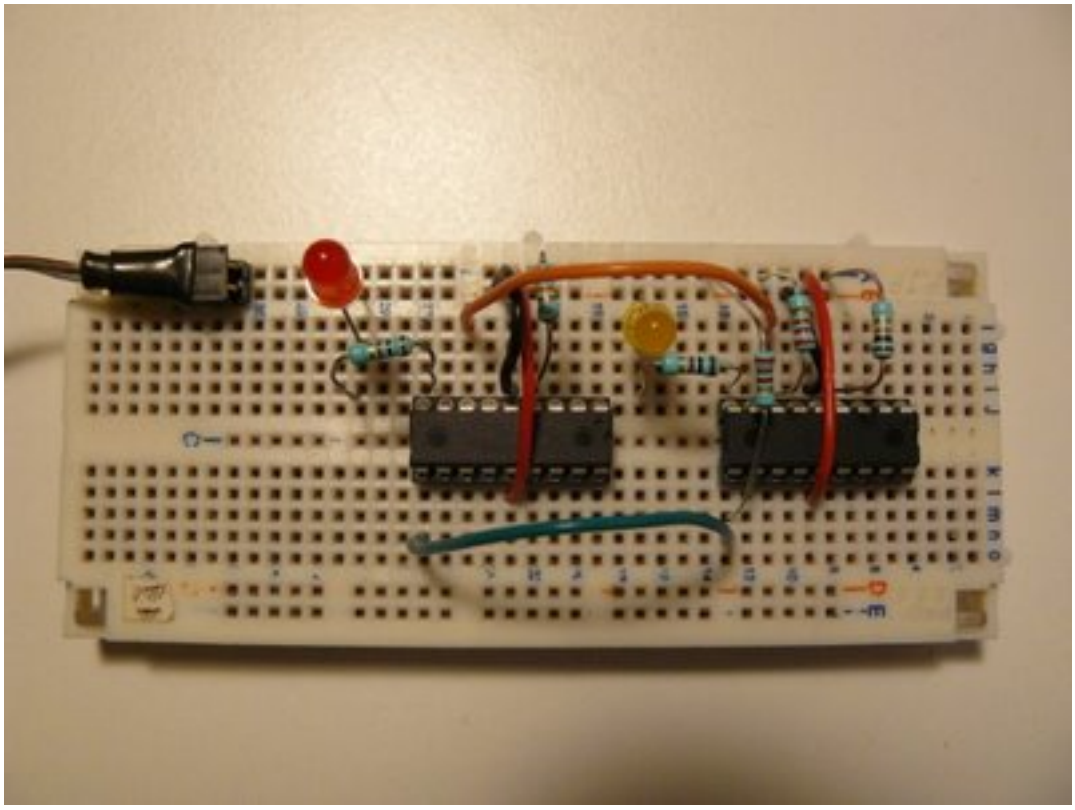
Checking the hardware and the i2c bus...

First of all, i2c is quite hard to debug, especially if you don't own an oscilloscope (like me). So you have to be accurate and rigorous. That's why, in this second part of this tutorial, we're going to setup the hardware, and just make sure the i2c bus is properly operational.

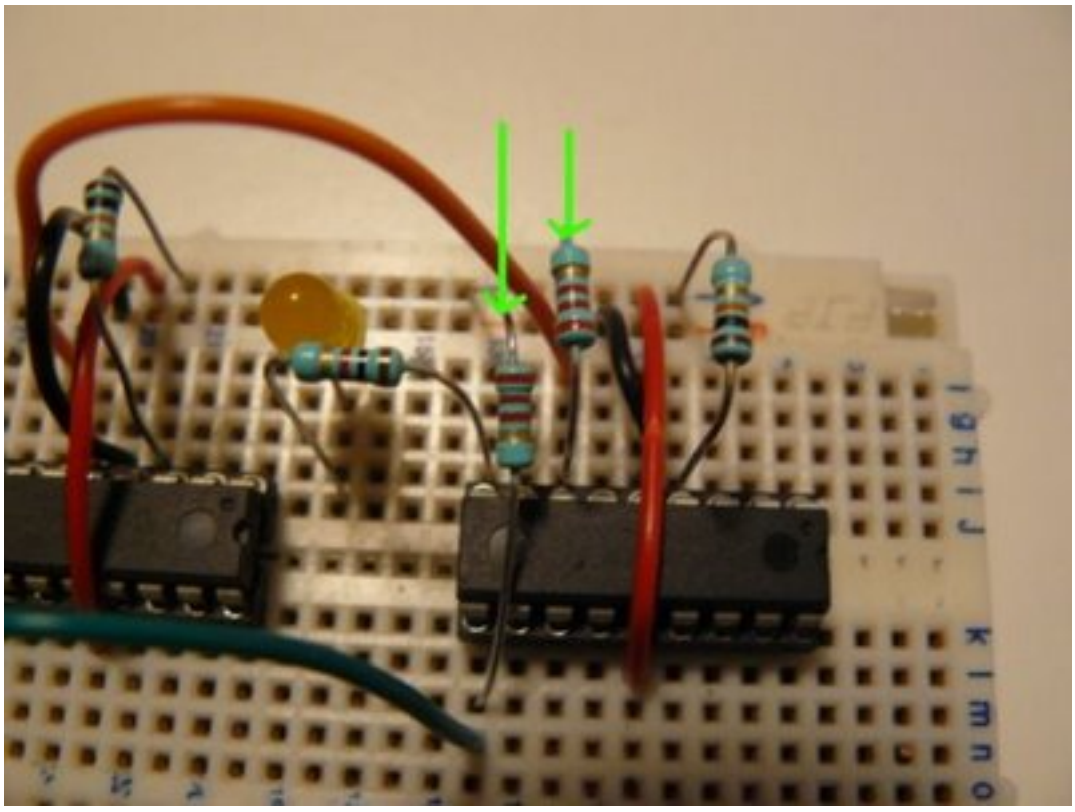
Connecting two PIC together through i2c is quite easy from a hardware point of view. Just connect SDA and SCL together, and **don't forget pull-ups resistors**. There are many differents values for these resistors, depending on *how long the bus is*, or the *speed you want to reach*. Most people use 2.2K resistors, so let's do the same ! The following schematics is here to help:



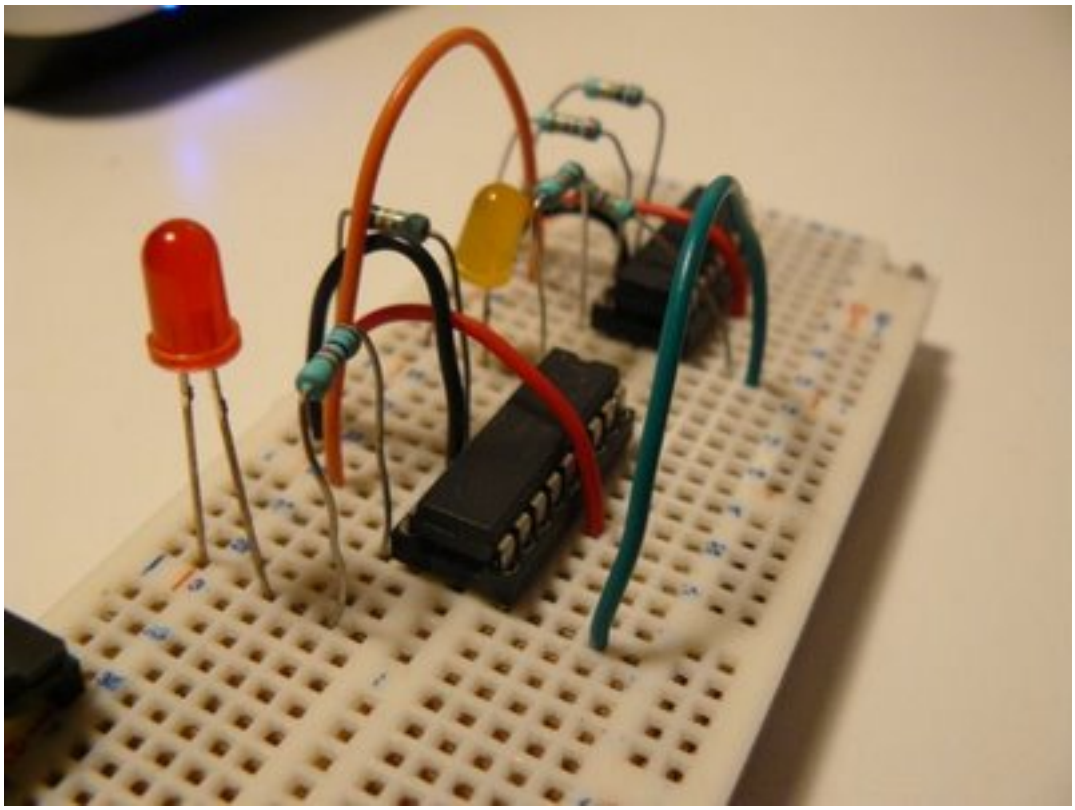
In this circuit, both PIC have a LED connected, which will help us understand what's going on. On a breadboard, this looks like that:



The master is on the right side, the slave on the left. I've put the two pull-ups resistors near the master:



Green and orange wires connect the two PICs together through SDA and SCL lines:



The goal of this test is simple: check if the i2c bus is properly built and operational. How ? PIC 16F88 and its SSP peripheral is able to be configured so it triggers an interrupts when a Start or Stop signal is detected. Read *this page* (part of an nice article on i2c, from previous tutorial's recommandations).

How are we gonna test this ? The idea of this test is simple:

1. On power, master will blink a LED a little, just to inform you it's alive
2. On the same time, slave is doing the same
3. Once master has done blinking, it sends a i2c frame through the bus
4. If the bus is properly built and configured, slave will infinitely blink its LED, at high speed

Note master will send its i2c frame to a specific address, which don't necessarily need to be the same as the slave one (and I recommand to use different addresses, just to make sure you understand what's going on).

What about the sources ? *Download* last jallib pack, and check the following files (either in `lib` or `sample` directories):

- *i2c_hw_slave.jal*: main i2c library
- *16f88_i2c_sw_master_check_bus.jal*: code for master
- *16f88_i2c_hw_slave_check_bus.jal*: code for slave

The main part of the slave code is the way the initialization is done. A constant is declared, telling the library to enable Start/Stop interrupts.

```
const SLAVE_ADDRESS = 0x23 -- whatever, it's not important, and can be
                           -- different from the address the master wants
                           -- to talk to
-- with Start/Stop interrupts
const bit i2c_enable_start_stop_interrupts = true
-- this init automatically sets global/peripherals interrupts
i2c_hw_slave_init(SLAVE_ADDRESS)
```

And, of course, the Interrupt Service Routine (ISR):

```
procedure i2c_isr() is
   pragma interrupt
   if ! PIR1_SSPIF then
      return
   end if
   -- reset flag
   PIR1_SSPIF = false
   -- tmp store SSPSTAT
   var byte tmpstat
   tmpstat = SSPSTAT
   -- check start signals
   if (tmpstat == 0b_1000) then
      -- If we get there, this means this is an SSP/I2C interrupts
      -- and this means i2c bus is properly operational !!!
      while true loop
         led = on
         _usec_delay(100000)
         led = off
         _usec_delay(100000)
      end loop
   end if
end procedure
```

The important thing is to:

- check if interrupt is currently a SSP interrupts (I2C)
- reset the interrupt flag,
- analyze SSPSTAT to see if Start bit is detected
- if so, blinks 'til the end of time (or your battery)

Now, go compile both samples, and program two PICs with them. With a correct i2c bus setting, you should see the following:

http://www.youtube.com/watch?v=NalAkRhFP-s

On this next video, I've removed the pull-ups resistors, and it doesn't work anymore (slave doesn't high speed blink its LED).

http://www.youtube.com/watch?v=cNK_cCgWctY

Next time (and last time on this topic), we'll see how to implement the state machine using jallib, defining callback for each states.

Implementing an I²C slave with jallib (part 3)

Sébastien Lelong
Jallib Group

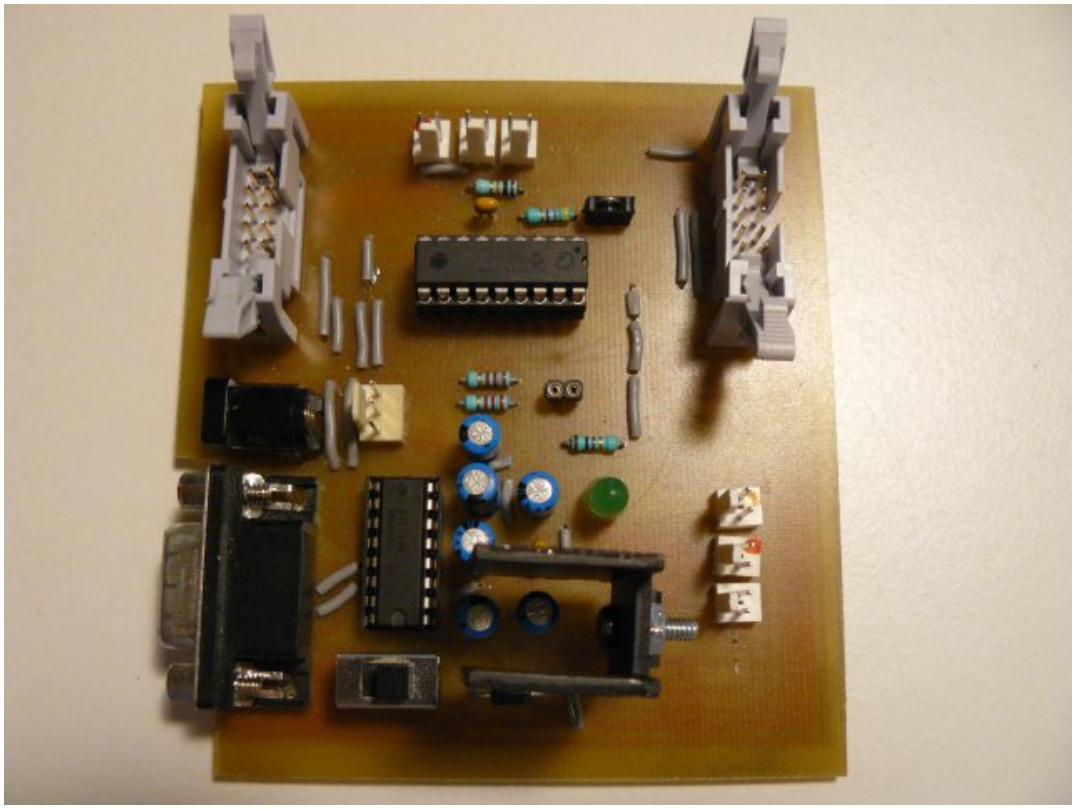
In previous parts of this tutorial, we've seen a little of theory, we've also seen how to check if the i2c bus is operational, now the time has come to finally build our i2c slave. But what will slave will do ? For this example, slave is going to do something amazing: it'll echo received chars. Oh, I'm thinking about something more exciting: it will "almost" echo chars:

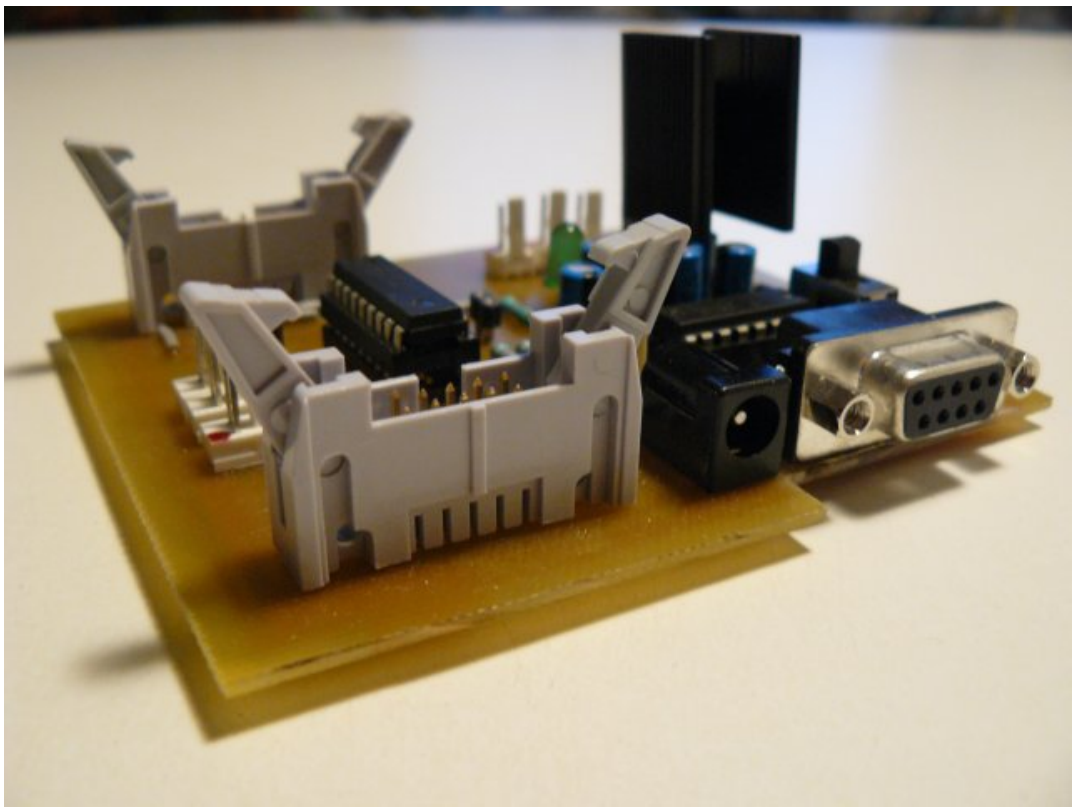
- if you send "a", it sends "b"
- if you send "b", it sends "c"
- if you send "z", it sends "{"[4]

Building the i2c master

Let's start with the easy part. What will master do ? Just collect characters from a serial link, and convert them to i2c commands. So you'll need a PIC to which you can send data via serial. I mean you'll need a board with serial com. capabilities. I mean we won't do this on a breadboard... There are plenty out there on the Internet, pick your choice. If you're interested, you can *find one* on my *SirBot* site: dedicated to 16f88, serial com. available, and i2c ready (pull-ups resistors).

It looks like this:





Two connectors are used for earch port, *PORTA* and *PORTB*, to plug daughter boards, or a breadboard in our case.

The i2c initialization part is quite straight forward. SCL and SDA pins are declared, we'll use a standard speed, 400KHz:

```
-- I2C io definition
var volatile bit i2c_scl            is pin_b4
var volatile bit i2c_scl_direction  is pin_b4_direction
var volatile bit i2c_sda            is pin_b1
var volatile bit i2c_sda_direction  is pin_b1_direction
-- i2c setup
const word _i2c_bus_speed = 4 ; 400kHz
const bit _i2c_level = true   ; i2c levels (not SMB)
include i2c_software
i2c_initialize()
```

We'll also use the level 1 i2c library. The principle is easy: you declare two buffers, one for receiving and one for sending bytes, and then you call procedure specifying how many bytes you want to send, and how many are expected to be returned. Joep has written *a nice post about this*, if you want to read more about this. We'll send one byte at a time, and receive one byte at a time, so buffers should be one byte long.

```
const single_byte_tx_buffer = 1 -- only needed when length is 1
var byte i2c_tx_buffer[1]
var byte i2c_rx_buffer[1]
include i2c_level1
```

What's next ? Well, master also has to read chars from a serial line. Again, easy:

```
const usart_hw_serial = true
const serial_hw_baudrate = 57_600
include serial_hardware
serial_hw_init()
-- Tell the world we're ready !
serial_hw_write("!")
```

So when the master is up, it should at least send the "!" char.

Then we need to specify the slave's address. This is a 8-bits long address, the 8th bits being the bit specifying if operation is a read or write one (see *Building an I²C slave, some theory (part 1)* for more). We then need to collect those chars coming from the PC and sends them to the slave.

The following should do the trick (believe me, it does :))

```
var byte icaddress = 0x5C   -- slave address

forever loop
   if serial_hw_read(pc_char)
   then
      serial_hw_write(pc_char)  -- echo
      -- transmit to slave
      -- we want to send 1 byte, and receive 1 from the slave
      i2c_tx_buffer[0] = pc_char
      var bit _trash = i2c_send_receive(icaddress, 1, 1)
      -- receive buffer should contain our result
      ic_char = i2c_rx_buffer[0]
      serial_hw_write(ic_char)
   end if
end loop
```

The whole program is available on jallib SVN repository *here*.

Building the i2c slave

So this is the main part ! As exposed on *Building an I²C slave, some theory (part 1)*, we're going to implement a *finite state machine*. jallib comes with a library where all the logic is already coded, in a ISR. You just have to define what to do for each state encountered during the program execution. To do this, we'll have to **define several callbacks**, that is procedures that will be called on appropriate state.

Before this, we need to **setup and initialize our slave**. i2c address should exactly be the same as the one defined in the master section. This time, we won't use interrrupts on Start/Stop signals; we'll just let the SSP module triggers an interrupts when the i2c address is recognized (no interrupts means address issue, or hardware problems, or...). Finally, since slave is expected to receive a char, and send char + 1, we need a global variable to store the results. This gives:

```
include i2c_hw_slave

const byte SLAVE_ADDRESS = 0x5C
i2c_hw_slave_init(SLAVE_ADDRESS)

-- will store what to send back to master
-- so if we get "a", we need to store "a" + 1
var byte data
```

Before this, let's try to understand how master will talk to the slave (*italic*) and what the slave should do (underlined), according to each state (with code following):

- **state 1**: *master initiates a write operation* (but does not send data yet). Since no data is sent, slave should just do... nothing (slave just knows someone wants to send data).

  ```
  procedure i2c_hw_slave_on_state_1(byte in _trash) is
     pragma inline
     -- _trash is read from master, but it's a dummy data
     -- usually (always ?) ignored
  end procedure
  ```

- **state 2**: *master actually sends data, that is one character*. Slave should get this char, and process it (char + 1) for further sending.

  ```
  procedure i2c_hw_slave_on_state_2(byte in rcv) is
     pragma inline
     -- ultimate data processing... :)
     data = rcv + 1
  ```

```
end procedure
```

- **state 3**: *master initiates a read operation, it wants to get the echo back*. Slave should send its processed char.

```
procedure i2c_hw_slave_on_state_3() is
   pragma inline
   i2c_hw_slave_write_i2c(data)
end procedure
```

- **state 4**: *master still wants to read some information*. This should never occur, since one char is sent and read at a time. Slave should thus produce an error.

```
procedure i2c_hw_slave_on_state_4() is
   pragma inline
   -- This shouldn't occur in our i2c echo example
   i2c_hw_slave_on_error()
end procedure
```

- **state 5**: *master hangs up the connection*. Slave should reset its state.

```
procedure i2c_hw_slave_on_state_5() is
   pragma inline
   data = 0
end procedure
```

Finally, we need to define a callback in case of error. You could do anything, like resetting the PIC, and sending log/debug data, etc... In our example, we'll blink forever:

```
procedure i2c_hw_slave_on_error() is
   pragma inline
   -- Just tell user user something's got wrong
   forever loop
      led = on
      _usec_delay(200000)
      led = off
      _usec_delay(200000)
   end loop
end procedure
```

Once callbacks are defined, we can include the famous ISR library.

```
include i2c_hw_slave_isr
```

So the sequence is:

1. **include i2c_hw_slave**, and setup your slave
2. define your callbacks,
3. include the ISR

The full code is available from jallib's SVN repository:

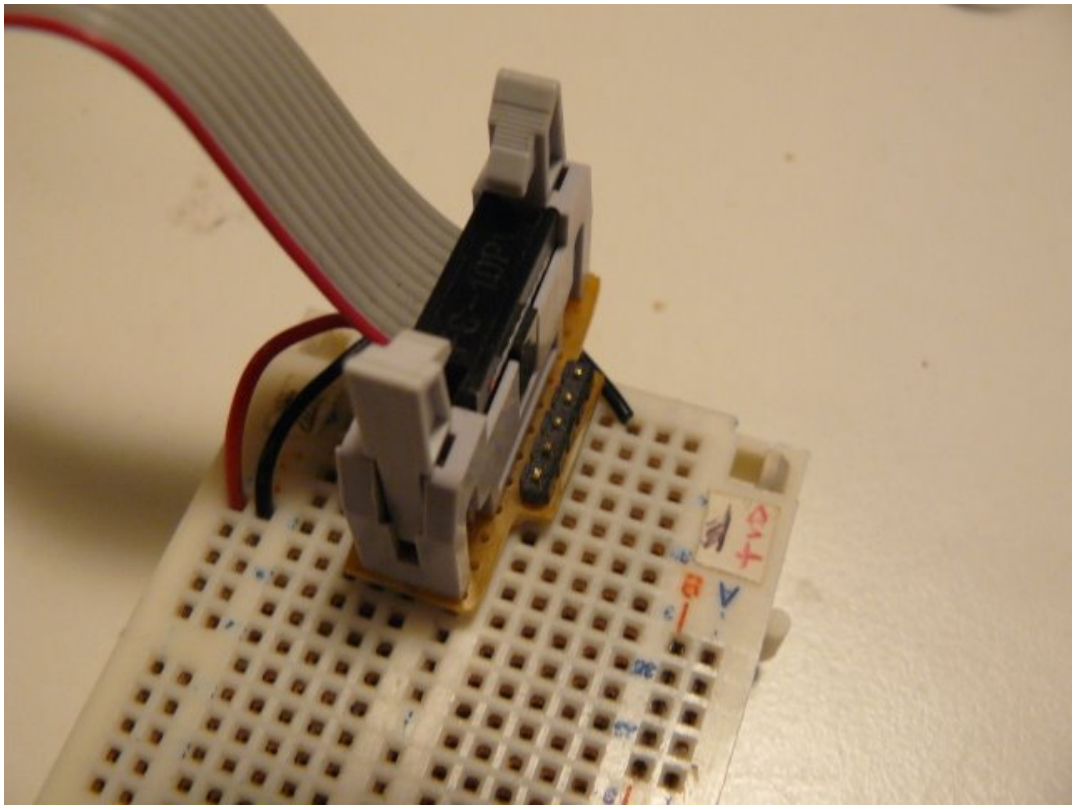
- *i2c_hw_slave.jal*
- *i2c_hw_slave_isr.jal*
- *16f88_i2c_sw_master_echo.jal*
- *16f88_i2c_hw_slave_echo.jal*

All those files and other dependencies are also available in last jallib-pack (see jallib *downloads*)

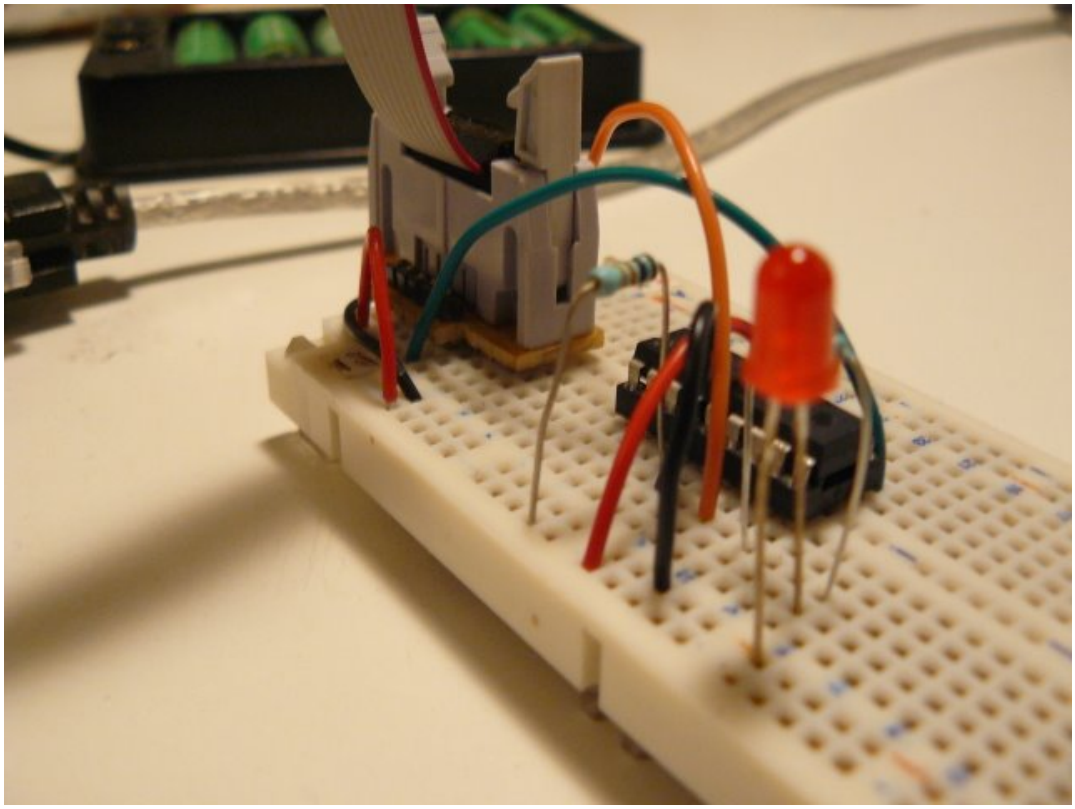
Connecting and testing the whole thing...

As previously said, the board I use is ready to be used with a serial link. It's also i2c ready, I've put the two pull-ups resistors. If your board doesn't have those resistors, you'll have to add them on the breadboard, or it won't work (read *Setting up and checking an I²C bus (part 2)* to know and see why...).

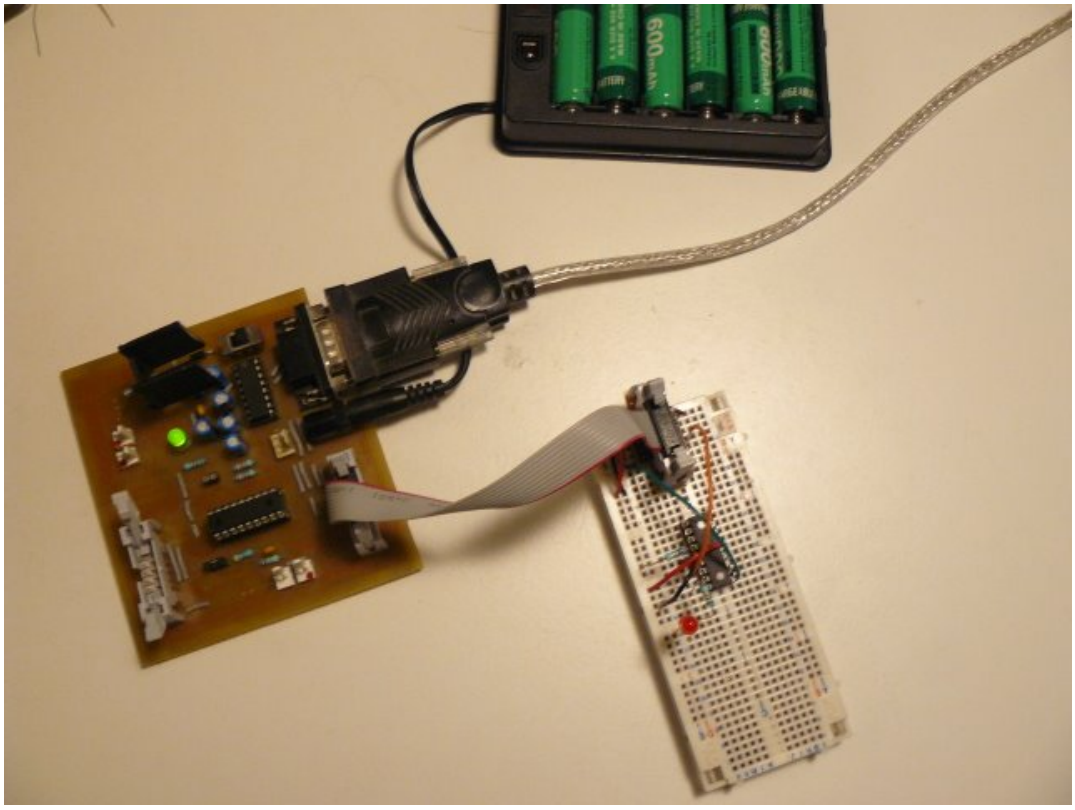
I use a connector adapted with a PCB to connect my main board with my breadboard. Connector's wires provide power supply, 5V-regulated, so no other powered wires it required.



Connector, with power wires



Everything is ready...



Crime scene: main board, breadboard and battery pack

Once connected, power the whole and use a terminal to test it. When pressing "a", you'll get a "a" as an echo from the master, then "b" as result from the slave.

```
sirloon@storm ~
sirloon@storm ~ cu -l /dev/ttyUSB0 -s 57600
Connected.
abbccddeefxyyzz{01122334455667788 99:
```

What now ?

We've seen how to implement a simple i2c hardware slave. The ISR library provides all the logic about the finite state machine. *You just have to define callbacks, according to your need*.

i2c is a widely used protocol. Most of the time, you access i2c devices, acting as a master. We've seen how to be on the other side, on the slave side. Being on the slave side means you can build modular boards, accessible with a standard protocol. For instance, I've built a *DC motor controller* daughter board using this. It's a module, a unit on its own, just plug, and send/receive data, with just two wires.

SPI Introduction

Matthew Schinkel
Jallib Group

Introduction to SPI - Serial Peripheral interface

What is SPI?

SPI is a protocol is simply a way to send data from device to device in a serial fashion (bit by bit). This protocol is used for things like SD memory cards, MP3 decoders, memory devices and other high speed applications.

We can compare SPI to other data transfer protocols:

Table 1: Protocol Comparison Chart

	SPI	RS-232	I2C
PINS	3 + 1 per device	2	2
Number Of Devices	unlimited	2	1024
Bits in one data byte transfer	8	10 (8 bytes + 1 start bit + 1 stop bit)	9 (8 bytes + 1 ack bit)
Must send one device address byte before transmission	No	No	Yes
Clock Type	Master clock only	Both device clocks must match	Master Clock that slave can influence
Data can transfer in two directions at the same time (full-duplex)	Yes	Yes	No

As you can see SPI sends the least bit's per data byte transfer byte and does not need to send a device address before transmission. This makes SPI the fastest out of the three we compared.

Although SPI allows "unlimited" devices, and I2C allows for 1024 devices, the number of devices that can be connected to each of these protocol's are still limited by your hardware setup. This tutorial does not go into detail about connecting a large number of devices on the same bus. When connecting more devices, unrevealed problems may appear.

How does SPI work?

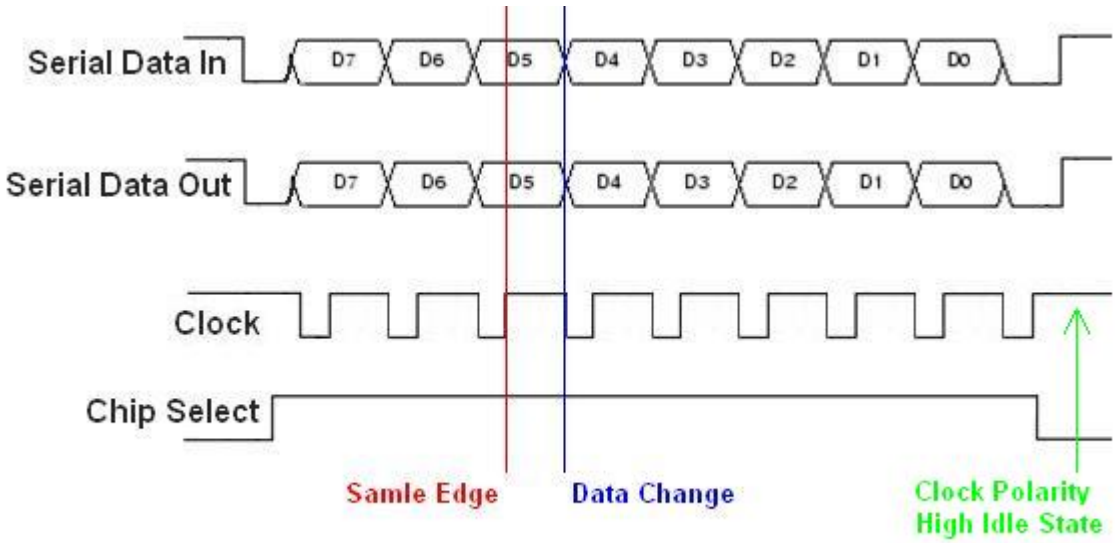
Firstly, SPI works in a master/slave setup. The master is the one that sends the clock pulses. At each pulse, data will be sent and received.

SPI has a chip select pin. Every device will share the "SDI", "SDO" and "Clock" pins, but each device will have it's own chip select pin (also known as slave select). This means we can have a virtually unlimited number of devices on the same SPI bus. You should also note that the chip select pin can be active high or active low depending on the device.

For some devices, the chip select pin must stay enabled throughout the transmission, and others require a change in the chip select line before the next transmission.

SPI is Dual-Duplex. This means data can be sent and received at the same time. If you wish to send data and not receive any, the PIC will receive data anyways. You may ignore the return byte.

Here's a diagram showing the way in which SPI sends & receives data:



SPI Modes

If you are using a device that does not yet have a Jallib library, you will need to get the devices SPI mode. Some device datasheets tell you the SPI mode, and some don't. Your device should tell you the clock idle state and sample edge, with this information, you can find the SPI mode. SPI devices can be set to run in 4 different modes depending on the clock's idle state polarity & data sample rising or falling edge.

The image above is SPI mode 1,1. See if you can understand why.

Clock Polarity (CKP) - Determines if the clock is normally high or normally low during it's idle state.

If CKP = 1 - the clock line will be high during idle.

If CKP = 0 - the clock will be low during idle.

Data Clock Edge (CKE) - The edge that the data is sampled on (rising edge or falling edge)

If CKP = 0, CKE = 0 - Data is read on the clocks rising edge (idle to active clock state)

If CKP = 0, CKE = 1 - Data is read on the clocks falling edge (active to idle clock state)

If CKP =1, CKE = 0 - Data is read on the clocks falling edge (idle to active clock state)

If CKP = 1, CKE = 1 - Data is read on the clocks rising edge (active to idle clock state)

We can put this in a chart to name the modes:

Table 2: SPI MODE NAMES

MODE NAME	CKP	CKE
0,0	0	1
0,1	0	0
1,0	1	1
1,1	1	0

Note: I noticed the mode numbers & mode table on Wikipedia is different then the table in the Microchip PDF. I am going by the Microchip PDF, as well as the tested and working PIC Jallib library + samples. Wikipedia also names these registers CPOL/CPHA instead of CKP/CKE.

Using The Jallib Library

At the moment, there is only a SPI master hardware library, therefore any device you wish to control must be connected to the PIC's SDI, SDO, SCK pins. The chip select pin can be any digital output pin.

The library requires you to set the pin directions of the SDI, SDO, SCK lines as follows:

```
-- setup SPI
include spi_master_hw         -- first include the library

-- define SPI inputs/outputs
pin_sdi_direction = input    -- spi data input
pin_sdo_direction = output   -- spi data output
pin_sck_direction = output   -- spi data clock
```

You only need to set the pin direction of the chip select pin, the PIC will set the direction of the SDI, SDO & SCK for you. You will Alias this chip select pin as required by the device's jallib library.

If you are using more then one device in your circuit, you will need to declare your chip select pin near the beginning of your program. If you do not do this at the beginning of your program, some of your devices may receive data because their chip select pin could be enabled during init procedures of other devices on the SPI bus.

```
-- choose your SPI chip select pin
-- pin_SS is the PIC's slave select (or chip select) pin.
ALIAS device_chip_select_direction   is pin_SS_direction
ALIAS device_chip_select             is pin_SS
device_chip_select_direction = output    -- chip select/slave select pin
device_chip_select = low                -- disable the device
```

Now the last step in setting up the SPI library is to use the init procedure.

Use the SPI mode name chart to get your SPI mode. The modes can be any of the following:

SPI_MODE_00

SPI_MODE_01

SPI_MODE_10

SPI_MODE_11

You will also need to set the spi bus speed. Here is a list of the speeds you may choose from:

SPI_RATE_FOSC_4 -- oscillator / 4

SPI_RATE_FOSC_16 -- oscillator / 16

SPI_RATE_FOSC_64 -- oscillator / 64

SPI_RATE_TMR -- PIC's internal timer

You will use the following init procedure with your custom values entered:

```
spi_init(SPI_MODE_11,SPI_RATE_FOSC_16) -- choose spi mode and speed
```

Now your ready to use the procedures to send and receive data. First you must enable the device with the chip select line:

```
device_chip_select = high -- enable the device
```

You can use the pseudo variable spi_master_hw to send and receive data as follows:

```
-- send decimal 50 to spi bus
spi_master_hw = 50
```

Or receive data like this:

```
-- receive data from the spi port into byte x
var byte x
x = spi_master_hw
```

You can also send and receive data at the same time with the spi_master_hw_exchange procedure. here's an example:

```
-- send decimal byte 50 and receive data into byte x
var byte x
x = spi_master_hw_exchange (50)
```

When your done transmitting & receiving data, don't forget to disable your device

```
device_chip_select = low -- enable the device
```

Alright, now you should be able to implement SPI into any of your own devices. If you need assistance, contact us at the *Jallist Support Group* or at *Jallib Group*.

References

The Jallib spi_master_hw library - Written by William Welch

Microchip Technology SPI Overview - *http://ww1.microchip.com/downloads/en/devicedoc/spi.pdf*

Wikipedia - *http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus*

Chapter 3

Experimenting external parts

Topics:

You now have learned enough and can start to interface your PIC with externals parts. Without being exhaustive, this chapter explains how to use a PIC with several widely used parts, like LCD screen.

SD Memory Cards

Matthew Schinkel
Jallib Group

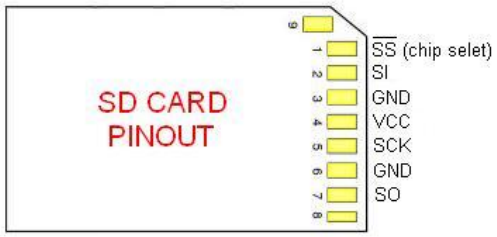
In this tutorial we will learn how to use an SD Card for mass data storage.

SD Card Introduction

SD Cards (Secure Digital Cards) are quite popular these days for things like digital camera's, video camera's, mp3 players and mobile phones. Now you will have one in your project! The main advantages are: small size, large data storage capability, speed, cost. It has flash storage that does not require power to hold data. The current version of the sd card library that we will be using in this tutorial works with "standard capacity" sd cards up 4gb in size. I hope to find time to add "high capacity" and "extended capacity" capability to the library.

SD Card have 2 data transfer types "SD Bus" and "SPI Bus". Most PIC's have an SPI port. The "SD Bus" is faster, however uses more pins. We will be using SPI in our circuit. For more info on SPI read the tutorial in this book: *SPI Introduction*. The SPI mode for SD Cards is 1,1.

We are not responsible for your data or SD card. Make sure you have nothing important on your SD card before you continue.

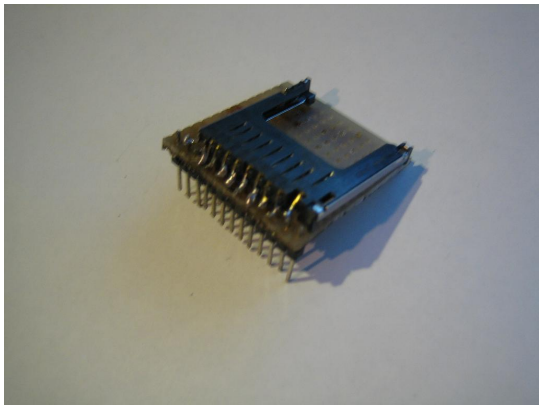
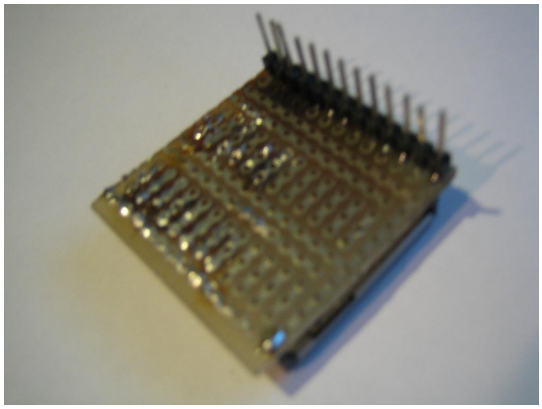
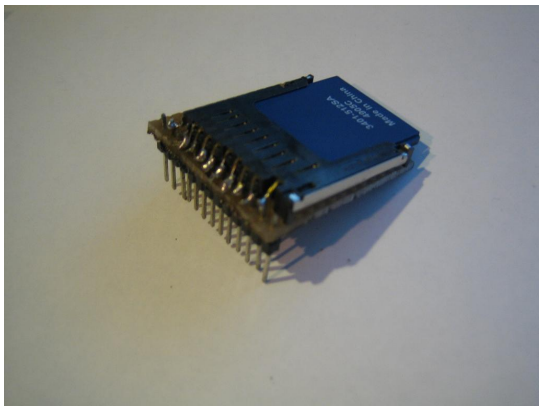


These SD Cards are 3.3v devices, therefore a 5v to 3v conversion is needed between the PIC and the sd card. We will use resistors to do the conversion, however there are many other methods. See *http://www.microchip.com/3v/* for more information. Another setup may be needed if you are putting more devices on the same SPI bus.

This circuit will use 16F877 If you are using a different PIC for your project, refer to the PIC's datasheet for pin output levels/voltage. For example, 18F452 has many pins that are 5v-input that give 3v-output. These pins show as "TTL / ST" - TTL compatible with CMOS level outputs in the datasheet and they will not require any voltage conversion resistors. If you are not sure, set a pin as an output, and make it go high then test with a volt meter.

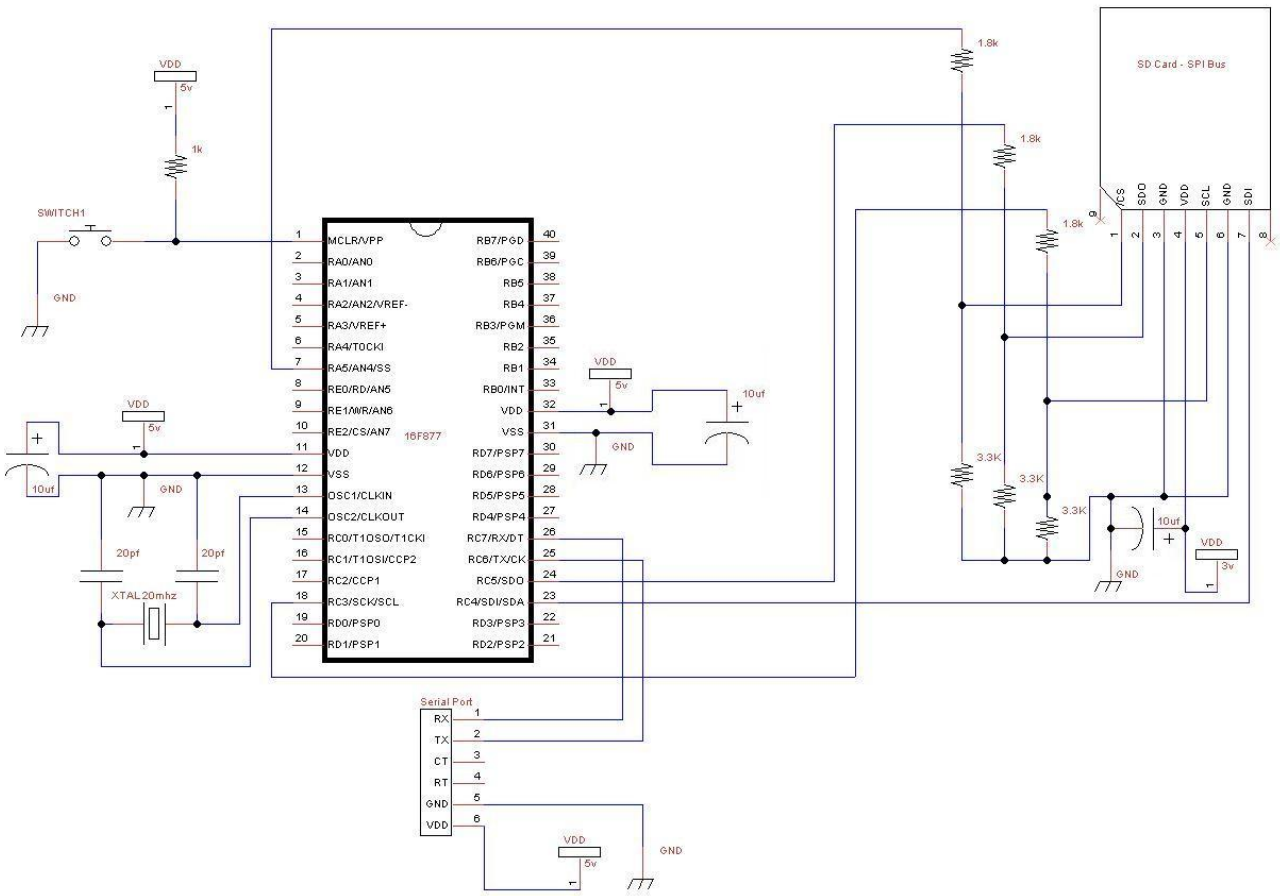
Build a SD Card Slot

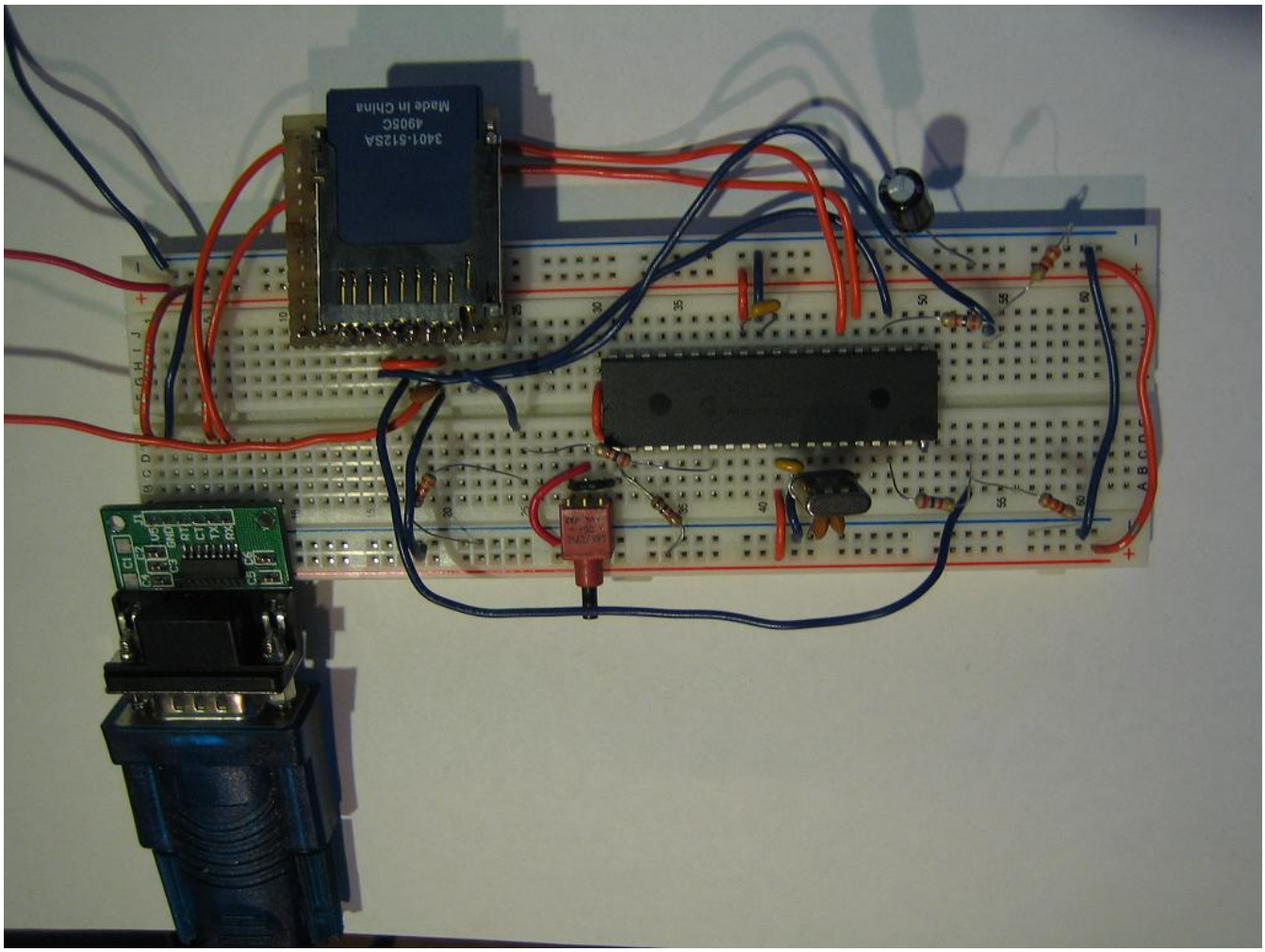
Before we can build our circuit, we will need to find ourselves an sd card slot that can plug into our breadboard. You can find pre-made sd card slots on ebay and other places around the net. It is quite easy to make your own anyways. I took one out of a broken digital camera and placed it on some blank breadboard and soldered on some pins. Here are some images of my sd card holder:

Build the circuit

Follow this schematic for 16f877, if you are using another PIC, check the pin-outs for the SPI bus. The pin-outs of your pic will show SDI, SDO, SCL and SS. The pin SS is the chip select pin, you can use any pin for it but the others must match.


SD Card - SPI Bus
CS
SDO
GND
VDD
SCL
GND
SDI
SWITCH1
VDD
5v
1k
GND
MCLR/VPP
RA0/AN0
RA1/AN1
RA2/AN2/VREF-
RA3/VREF+
RA4/T0CKI
RA5/AN4/SS
RE0/RD/AN5
RE1/WR/AN6
RE2/CS/AN7
VDD
VSS
OSC1/CLKIN
OSC2/CLKOUT
RC0/T1OSO/T1CKI
RC1/T1OSI/CCP2
RC2/CCP1
RC3/SCK/SCL
RD0/PSP0
RD1/PSP1
16F877
RB7/PGD
RB6/PGC
RB5
RB4
RB3/PGM
RB2
RB1
RB0/INT
VDD
VSS
RD7/PSP7
RD6/PSP6
RD5/PSP5
RD4/PSP4
RC7/RX/DT
RC6/TX/CK
RC5/SDO
RC4/SDI/SDA
RD3/PSP3
RD2/PSP2
10uf
20pf
XTAL 20mhz
1.8k
3.3K
Serial Port
RX
TX
CT
RT
GND
VDD
3v


Made in China

Compile and write the software to your PIC

With the use of the sd card lib (sd_card.jal) and a sample file 16f877_sd_card.jal, we can easily put one in our own circuit for mass data storage! You will find these files in the lib & sample directories of your jallib installation.

The most up to date version of the sample & library can be found at:

Sample file - *http://jallib.googlecode.com/svn/trunk/sample/16f877_sd_card.jal*

Library file - *http://jallib.googlecode.com/svn/trunk/include/external/storage/sd_card/sd_card.jal*

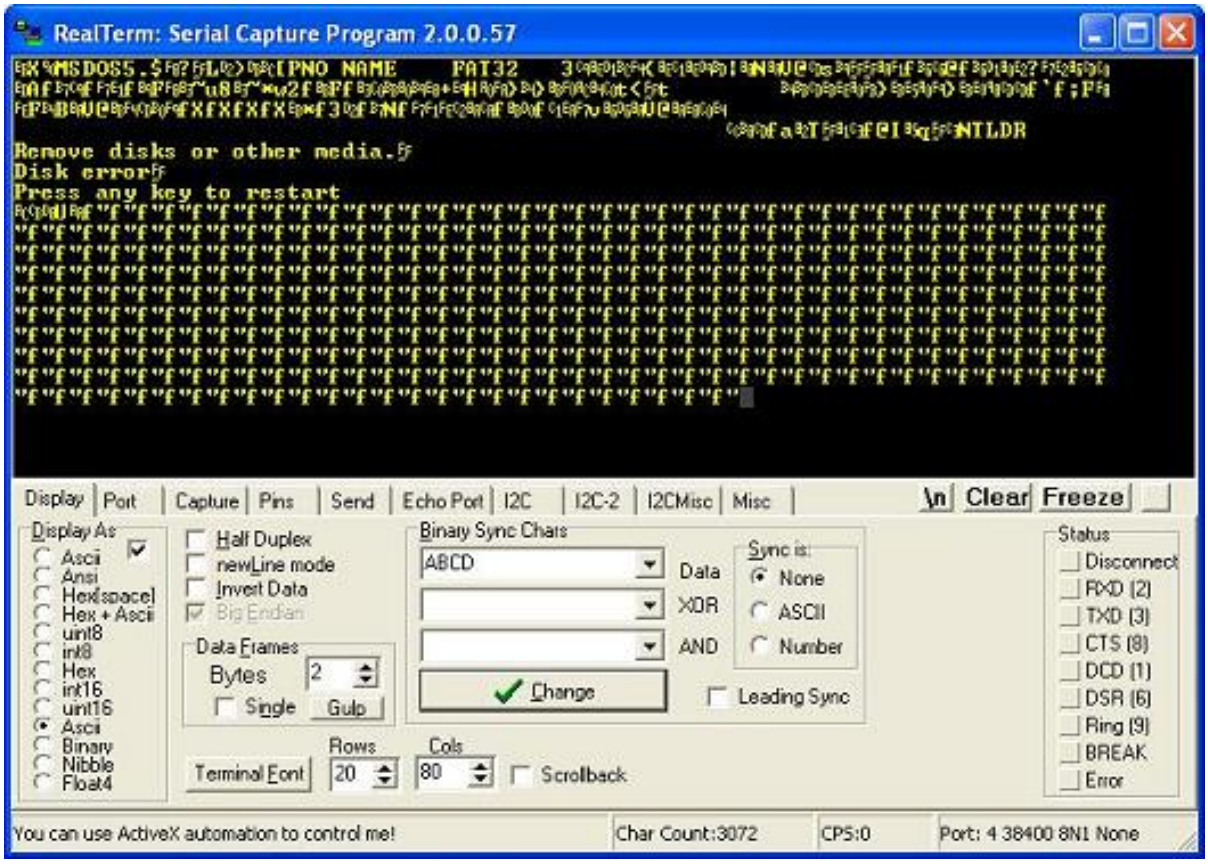
Now that our circuit is built, lets test it and make sure it works before we continue with more details. Compile and program your pic with 16f877_sd_card.jal from your jallib samples directory. If you are using another pic, change the "include 16f877" line in 16f877_sd_card.jal to specify your PIC before compiling.

Now that you have compiled it, burn the .hex file to your PIC

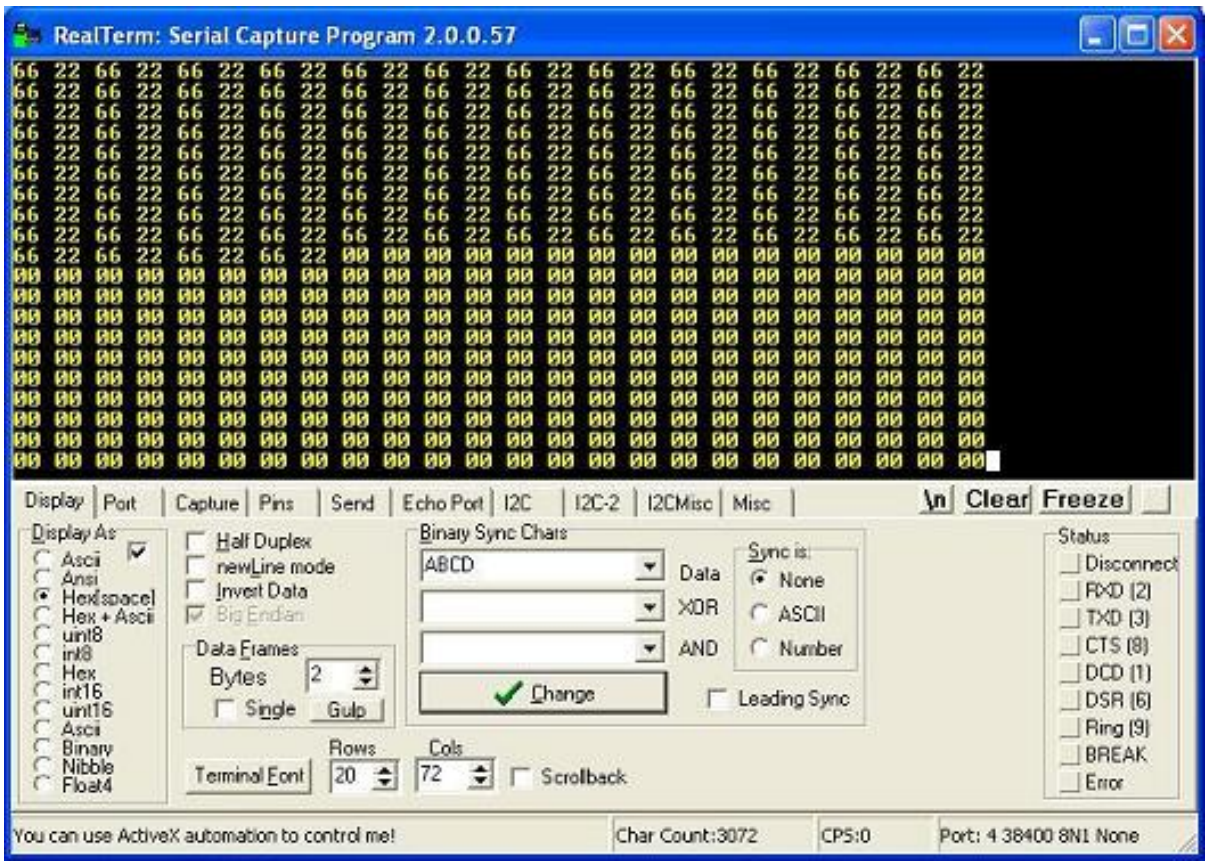
Power It Up

Plug your circuit into your PC for serial port communication at 38400 baud rate. Now turn it on. Press the reset button in your circuit, you should get a result similar to this:

ASCII output



Hex output



As you can see from the first image, we got some actual readable data off the sd card as well as a bunch of junk. The sample file reads the first sector (512 bytes) from the sd card. My sd card is formated with fat32, this is why I can read some of the data output.

In the second image (after clearing the output and resetting the circuit), there was too much data to show it all. It only shows the last bytes received. If you get the same hex output "66 22" followed by many "00", your circuit has successfully written data and read it back again. You now have a working sd card circuit!

Understand and modify the code

I'm just going to quickly go over some of the key points you need to know about sd cards. Open the sample file with an editor if you have not done so already.

The code in the sample file may change, therefore it may be different then what you see here. The sample file you have downloaded will always be tested and correct.

Include the chip

Specify the PIC you wish to use as well as your clock frequency

```
include 16f877
--
pragma target OSC HS                -- HS crystal or resonator
pragma target clock 20_000_000      -- oscillator frequency
--
pragma target WDT  disabled
pragma target LVP  disabled
```

Disable Analog Pins

```
enable_digital_io() -- disable all analog pins if any
```

Include other libraries required

```
-- include the delay library
include delay
```

Setup serial communication and port speed

```
-- setup uart for communication
const serial_hw_baudrate  = 38400    -- set the baudrate
include serial_hardware
serial_hw_init()
```

SPI Settings

Here you may change the chip select pin "pin_SS" and "pin_SS_direction" to another pin. SDI, SDO and SCK must stay the same for the SPI hardware library.

You may notice that we are not defining/aliasing pins sdi, sdo and sck. We do not need to define them with a line like "alias pin_sdo is pin_c5" becase they are set within the PIC and cannot be changed. If we use the SPI hardware library, we must use the spi hardware pins. We only need to define there direction like this "pin_sdo_direction = output".

You may also choose the SPI rate. According to the SPI hardware library, you can use SPI_RATE_FOSC_4 SPI_RATE_FOSC_16, SPI_RATE_FOSC_64 or SPI_RATE_TMR. The fastest is FOSC_4 (oscillator frequency / 4). You may require a breadboard for the fastest speed, keep your SD Card as close to the PIC as possible.

```
-- setup spi
include spi_master_hw         -- includes the spi library
-- define spi inputs/outputs
pin_sdi_direction = input    -- spi input
pin_sdo_direction = output   -- spi output
pin_sck_direction = output   -- spi clock
-- spi chip select pin
ALIAS sd_chip_select_direction   is pin_SS_direction
ALIAS sd_chip_select             is pin_SS
sd_chip_select_direction = output    -- chip select/slave select pin
sd_chip_select = high                -- disable the sd card
--
spi_init(SPI_MODE_11,SPI_RATE_FOSC_16) -- choose spi mode and speed
```

Include the SD card library

Select sd card settings & Include the library file, then initalize the sd card.

Some sd cards may require a 10ms delay every time you stop writing to the sd card, you can choose weater or not to have this delay. If you are doing many small writes and are worried about speed, you may set SD_DELAY_AFTER_WRITE to "FALSE".

```
-- setup sd card library
const bit SD_DELAY_AFTER_WRITE = TRUE
include sd_card -- include sd card library
sd_init()       -- initalize the sd card
```

Read the first sector from the SD card

Reading is easy, there are 3 procedures within the library that MUST be used.

sd_start_read(0) - start reading at specified sector (sector 0)

sd_read_data(byte1, byte2) - actually read data from the card (2 bytes at a time)

sd_stop_read() - stop the read process

You can also use the sd_read_pulse(number) procedure to skip past data. For every 1 value added, there will be 2 bytes skipped since this procedure simply reads data and ignores the input.

If you have more then one SPI device on the SPI bus, do not interrupt or switch devices until the complete read process has finished with sd_stop_read, do not allow the chip select pin to go high.

```
_usec_delay(100_000)                    -- wait for power to settle
var byte low_byte, high_byte            -- vars for sending and recieving data

-- read the boot sector (sector 0)
sd_start_read(0)                        -- get sd card ready for read at sector
 0
for 256 loop                            -- read 1 sector (256 words)
  sd_read_data (low_byte, high_byte)  -- read 2 bytes of data
  serial_hw_write (low_byte)            -- send byte via serial port
  serial_hw_write (high_byte)           -- send byte via serial port
end loop
sd_stop_read()                          -- tell sd card you are done reading
```

Write some data to your sd card

Writing is also easy, there are 3 procedures within the library that MUST be used.

sd_start_write(20) - start writing at specified sector (sector 20)

sd_read_data(byte1, byte2) - write to the card (2 bytes at a time)

sd_stop_write() - stop the read process

When writing to your SD card, you MUST write 512 bytes at a time. In this example, we are writing (256x2) = 512 bytes + (128x2) = 256 bytes for a total of 768 bytes. This means we have written one sector (512 bytes), as well as half of a sector (265 bytes). The half of a sector(256 bytes) that we have written, will not actually be written to the sd card until we finish the sector with data.

For this reason, you will need to use the sd_write_to_sector_end(value) procedure. This procedure will automatically finish the sector for you with the "value" data specified. In our case we are writing 0x00 till the end of the 512 bytes (end of the sector).

Just as we noted with reading data, you may not interrupt the SPI port until you have completed the write process with the sd_stop_write procedure.

Please note that we are writing to sector 20

```
-- write (0x66, 0x22) to sector 20 over and over.
low_byte = 0x66                         -- set low byte to write
high_byte = 0x22                        -- set high byte to write

sd_start_write(20)                      -- get sd card ready for write
for 256 + 128 loop                      -- write 1 sector + 1/2 sector
  sd_write_data(low_byte, high_byte)  -- write data to the card
end loop
sd_write_to_sector_end(0x00)            -- 2nd sector is not done, so finish it
                                        -- sectors must be completed during
 write

sd_stop_write()                         -- tell sd card you are done writing
```

Read back the data we have written

Now read 2 sectors (1024 bytes) from sector 20 (where we had previously written data). You will get 512 + 256 bytes of 0x66 & 0x22 as well as 256 bytes of 0x00's

```
-- read the data back, should get (0x66, 0x22) over and over.
sd_start_read(20)                       -- get sd card ready for read at sector
 20
for 512 loop                            -- read 2 sectors (512 words)
  sd_read_data (low_byte, high_byte)  -- read 2 bytes of data
  serial_hw_write (low_byte)            -- send byte via serial port
```

```
  serial_hw_write (high_byte)          -- send byte via serial port
end loop
sd_stop_read()                         -- tell sd card you are done reading
```

Now you can put whatever you want on your SD card, or possibly read lost data off of it.

If you want to read files stored on the card by your PC, there wil soon be a FAT32 library and tutorial so you can easily browse, read and write to files and folders stored on your card.

What are you waiting for, go build something cool!

Sources

The Jallib SD Card Library - Written by Matthew Schinkel

SanDisk Secure Digital Card - *http://www.cs.ucr.edu/~amitra/sdcard/ProdManualSDCardv1.9.pdf*

How to use MMC/SDC - *http://forums.parallax.com/forums/attach.aspx?a=32012*

Hard Disks - IDE/PATA

Matthew Schinkel
Jallib Group

IDE Paralel ATA hard disk drive tutorial

Introduction to hard disks drives

If your are like me, you have too many old hard disks laying around. I have gathered quite a collection of drives from PC's I have had in the past. Now you can dust off your drives and put them in your circuit. I have extra drives ranging in size from 171MB to 120GB.

Before you start, make sure you use a drive you do not care about. We are not responsible for your drive of the data that is on it.

You can find more general info at *http://en.wikipedia.org/wiki/Parallel_ATA*, and you can find more detailed technical info at *http://www.gaby.de/gide/IDE-TCJ.txt*

Drive Types - PATA vs SATA

There are two types of hard disks PATA (parallel ata) and SATA (serial ata). In this tutorial we will use PATA, these drives use a 40 pin IDE connector. The newer type of drive SATA has only 7 pins but there is no Jallib library for these drives at the moment. Both types of hard disks are available with massive amounts of data space.

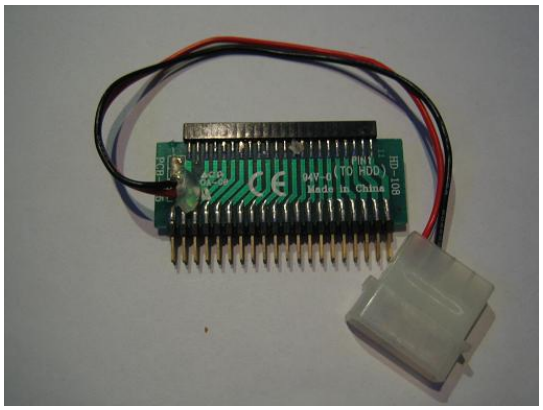
Drive Data Size

The current jallib library will accept drives up to 128GB. The 128GB limit is due to and addressing limitation, this is the 28 bit addressing limitation.The max address you will be able to reach is hex 0xFFFFFFF. If you multiply this address by 512 bytes (1 sector) you get a max size of 137,438,952,960 bytes, yes this does equal 128GB. Eventually I may upgrade the library for 48bit addressing which will allow up to a max drive size hex 0xFFFFFFFFFFFF * 512 = 128P Petabytes. But now that I think about it, 128 GB should be enough!

Actual Size

The most common drive sizes today are 3.5" and 2.5". The 3.5 inch drives are commonly used in desktop computers, 2.5" drives are used in laptops. The 2.5" drives are nice for your circuit because they do not require a 12v supply voltage, and they use much less power.

If you wish to use a 2.5" laptop hard drive, you may need a 2.5" to 3.5" IDE adapter like this one:

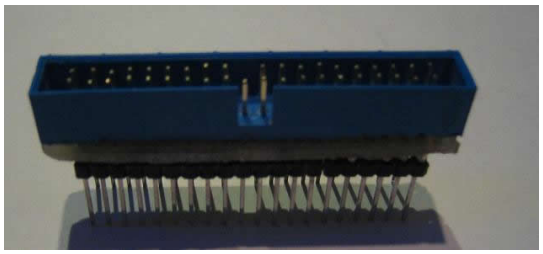
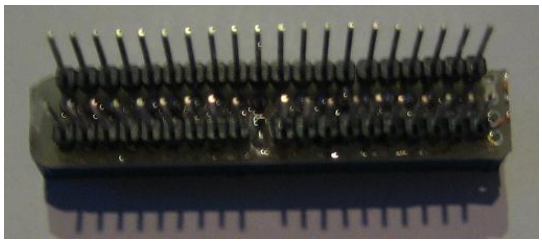
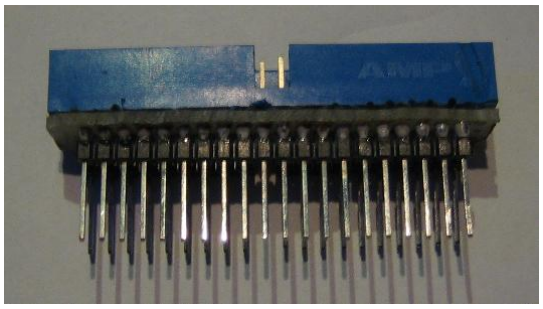


Build a breadboard connector

Now, if your going to put one of these into your circuit, you'll need to plug the drive into your breadboard. I took a 40pin IDE connector off an old motherboard. The easiest way to get large components of a board is to use a heat gun on the bottom side of the board to melt the solder on all pins at once.

Now take this connector and stick it into some blank breadboard and add some pins. The blank breadboard I cut is 4 holes wide by 20 long. Put the connector in the middle and connect the pins on the outside, join each pin with each pin of the connector.

Of course you will also need a 40pin IDE cable, I like the ones with the notch so you don't plug it in backwards. Here's the one I made:

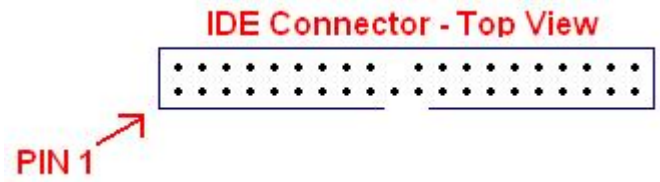




Circuit Power

It is very important that you have enough power to drive your circuit. Hard drives need a lot of amps to run, especially the 3.5" drives, so make sure you have a decent 5v and 12v power supply. I suggest that you DO NOT use your PC's power supply to drive your circuit. You can easily short circuit your power supply and blow up your PC. If you really insist on doing this, you better put a fuse on both 5v and 12v between your PC and your circuit. Just remember that I told you not to!

IDE Connector Pin-out

Pin 1 on the IDE cable is the red stripe. Here the pin out for the male connector I took off a motherboard:

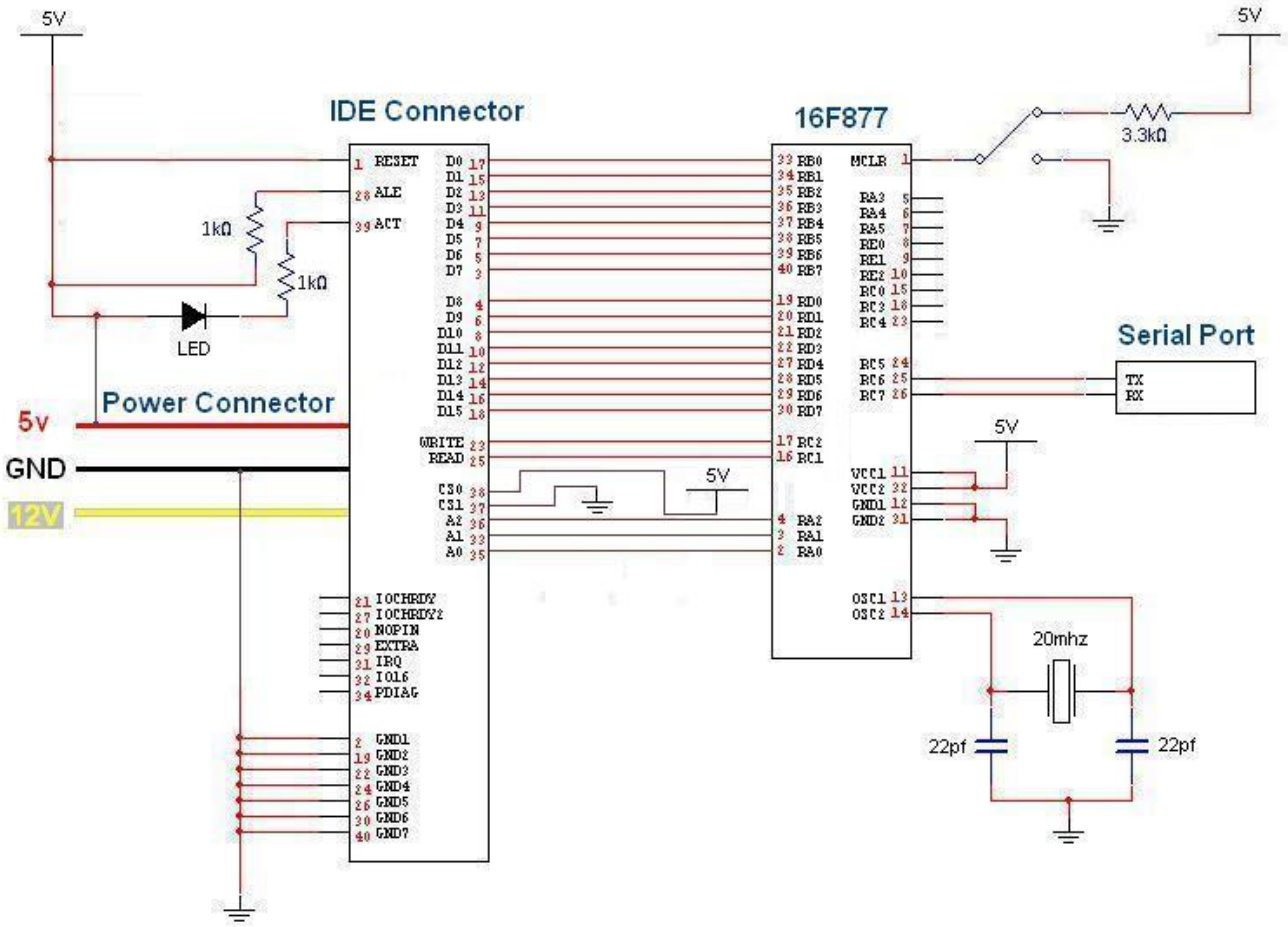


PIN	FUNCTION	PIN	FUNCTION
1	/RESET	2	GND
3	D7	4	D8
5	D6	6	D9
7	D5	8	D10

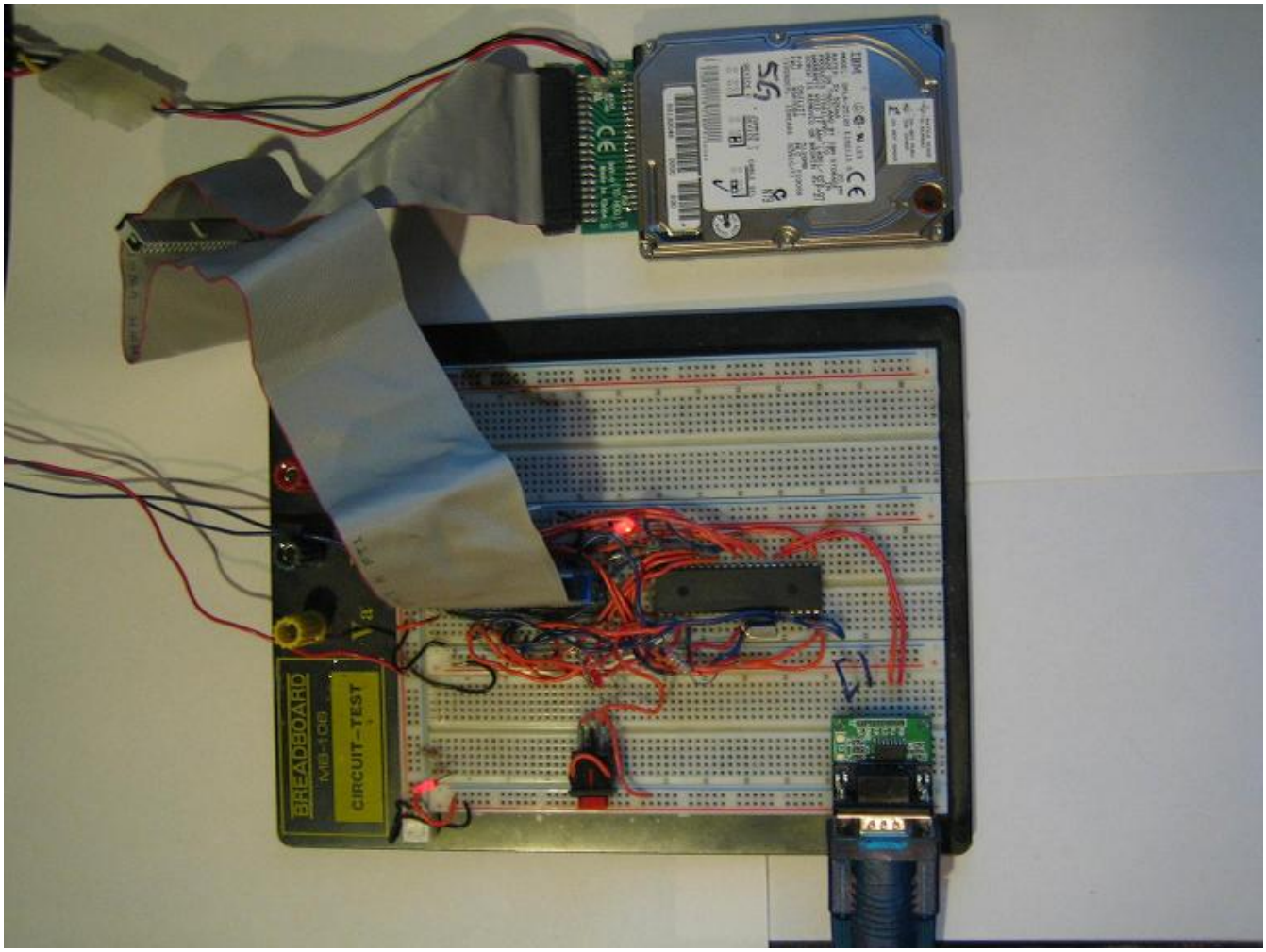
PIN	FUNCTION	PIN	FUNCTION
9	D4	10	D11
11	D3	12	D12
13	D2	14	D13
15	D1	16	D14
17	D0	18	D15
19	GND	20	NO PIN
21		22	GND
23	/IOWR - READ Pin	24	GND
25	/IORD - Write Pin	26	GND
27		28	ALE - 1K resistor to 5v
29		30	GND
31		32	
33	A1	34	
35	A0	36	A2
37	/CS0 (to 5v)	38	/CS1 (to GND)
39	ACT - BUSY LED	40	GND

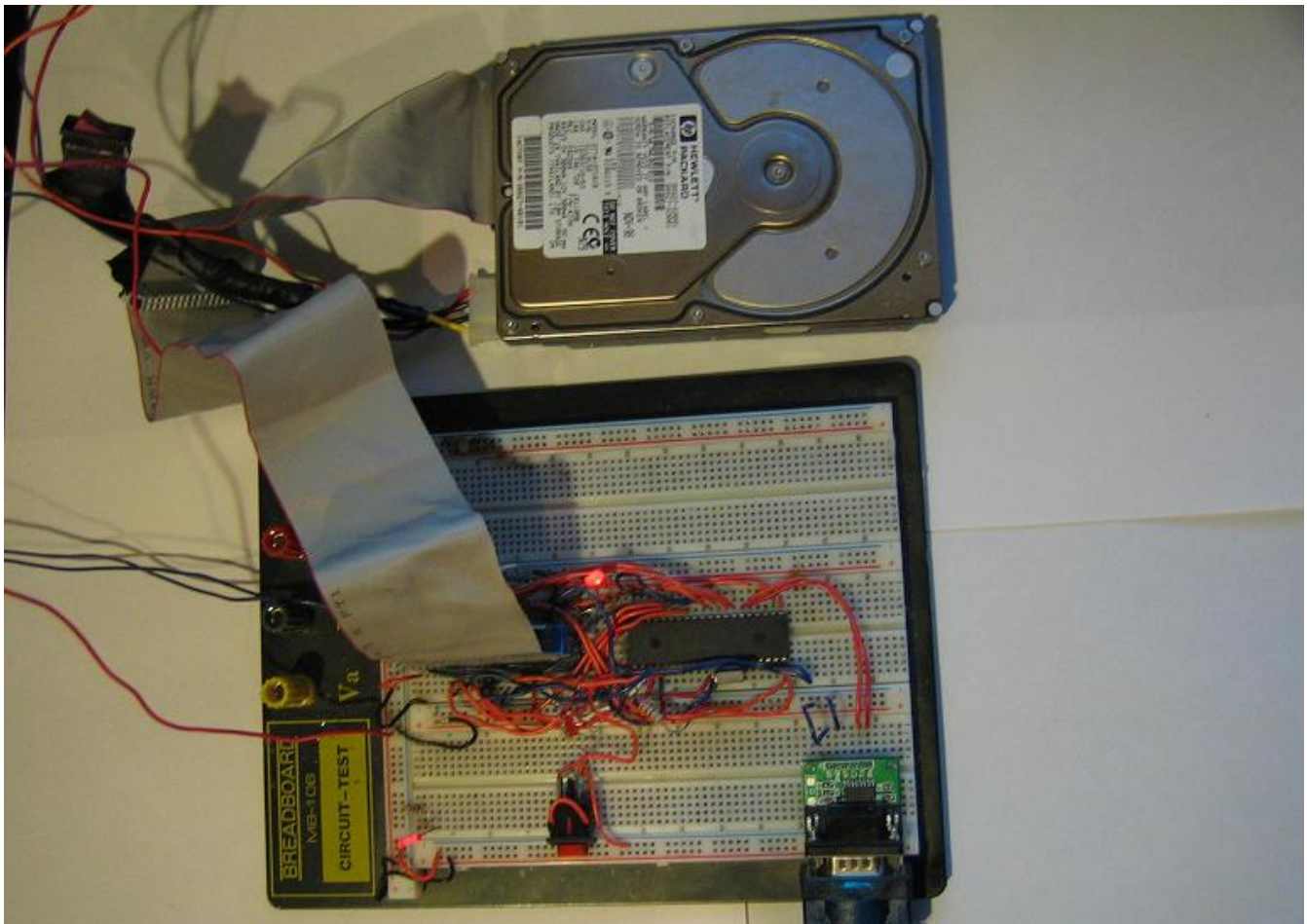
Build the circuit

Build the circuit below. As you can see it is quite simple. As you can see, it only requires 3 resistors, a led and a bunch of wire. You can put a reset button on the IDE connector if you like, but I have found no use for it so I connect it direct to 5v.



Here's what the completed circuit should look like (don't turn on the power yet):


BREADBOARD
MB-102
CIRCUIT-TEST


HEWLETT
PACKARD
BREADBOARD
MB-102
CIRCUIT-TEST

Compile and write the software to your PIC

The hard disk lib (pata_hard_disk.jal) and a sample file (16f877_pata_hard_disk.jal) will be needed for this project. You will find these files in the lib & sample directories of your jallib installation.

The most up to date version of the sample & library can be found at:

Sample file - *http://jallib.googlecode.com/svn/trunk/sample/16f877_pata_hard_disk.jal*

Library file - *http://jallib.googlecode.com/svn/trunk/include/external/storage/harddisk/pata_hard_disk.jal*

Now lets test it and make sure it works. Compile and program your pic with 16f877_sd_card.jal from your jallib samples directory. If you are using another pic, change the "include 16f877" line in 16f877_sd_card.jal to specify your PIC before compiling.

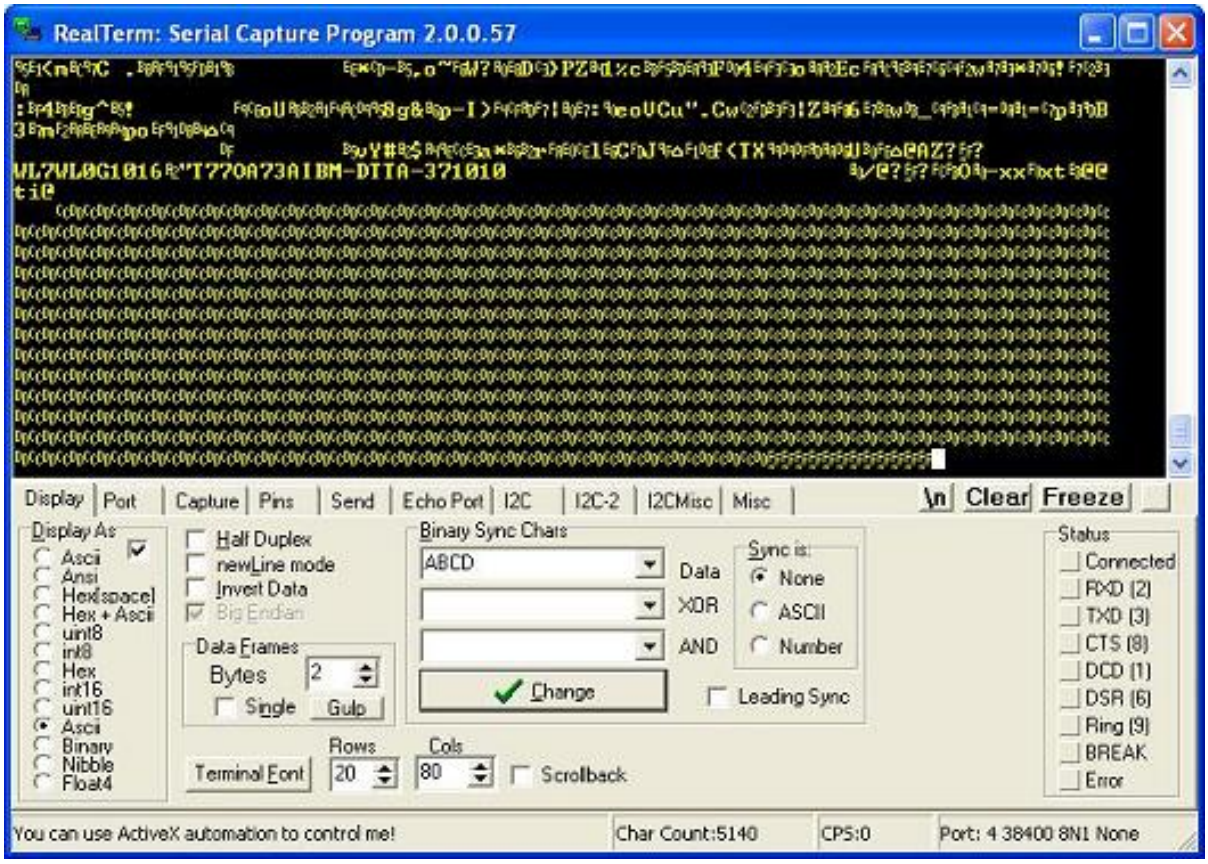
Now that you have compiled it, burn the .hex file to your PIC with your programmer

Power It Up

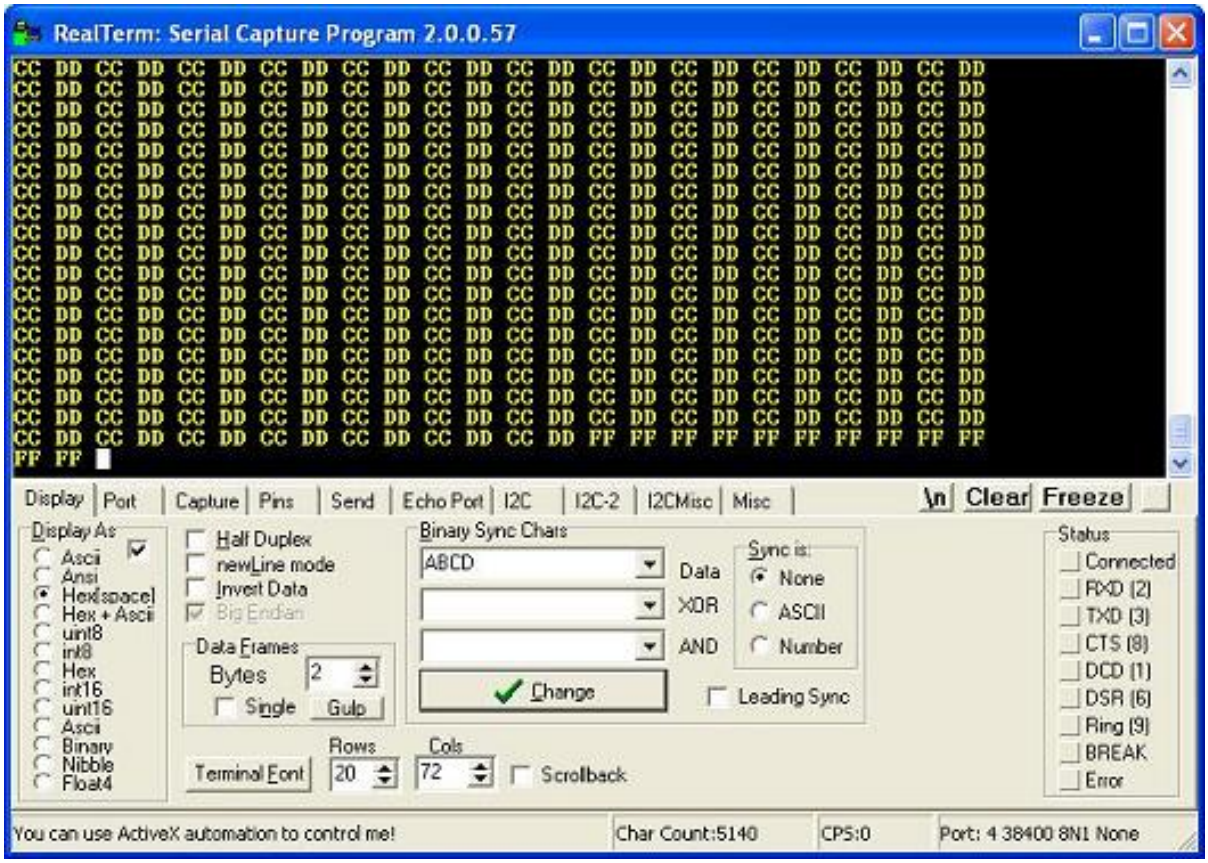
Plug your circuit into your PC for serial port communication at 38400 baud rate. Now turn it on. It should do the following in this order:

1. Drive will power up with the led on, after power up the led will go off.
2. The led will blink once quickly and the drive will "spin down".
3. The led will turn on while the drive now "spins up"
4. The led will blink once quickly and the drive will "spin down" again.
5. The led will turn on while the drive now "spins up" again.
6. The led will turn on and off a few times and send some data to your PC's serial port.
7. The PIC now "spin down" the drive at the end of the program.

ASCII output



Hex output



In the first image, If your disk is formatted with fat32 you may be able to see some readable data as well as some junk. There is too much data for me to show it all in the image, on my drive formatted with fat32 I can read "Invalid partition t ableError loading operating system..."

In the second image (after clearing the output and resetting the circuit), there was too much data to show it all again. It only shows the last bytes received. If you get the same hex output "CC DD" followed by many "FF", your circuit has successfully written data and read it back again. You now have a working hard disk circuit!

Understand and modify the code

I will go over some of the key points you need to know about hard disk coding. Open the sample file with an editor if you have not done so already. The code in the sample file may change, therefore it may be different then what you see here. The sample file you have downloaded will always be tested and correct.

Include the chip

Select the PIC you wish to use and your clock frequency

```
-- include chip
include 16f877                  -- target picmicro
;include 16f877a                -- target picmicro

-- this program assumes a 20 mhz resonator or crystal
-- is connected to pins osc1 and osc2.
pragma target osc hs            -- hs crystal or resonator
pragma target clock 20_000_000  -- oscillator frequency
--
pragma target wdt  disabled
pragma target lvp  disabled
--
```

Disable all analog pins

```
enable_digital_io() -- disable all analog pins if any
```

Include required libraries

```
include delay       -- include the delay library
```

Setup serial port and choose baud rate 38400

```
-- setup uart for communication
const serial_hw_baudrate  = 38400   -- set the baudrate
include serial_hardware
serial_hw_init()
```

Setup the hard disk library constants/settings

The registers Alternate Status, Digital Output, and Drive Address registers will only be used by advanced users, so keep the default PATA_HD_USE_CS0_CS1_PINS = FALSE

The pins /iowr, /iord, /cs0, /cs1 are active low pins that are supposed to require an inverter. If you leave PATA_HD_NO_INVERTER = TRUE, the PIC will do the inversion for you. You will most likely want to keep the default "TRUE".

```
-- setup hard disk library

-- set true if you will use Alternate Status,
-- Digital Output or Drive Address registers
const byte PATA_HD_USE_CS0_CS1_PINS = FALSE

-- if true, an external inverter chip is not
-- needed on /iowr, /iord, /cs0, /cs1 pins
const bit PATA_HD_NO_INVERTER = TRUE
```

Setup pin assignments

Yes, pata hard disks have a lot of pins. You will need two full 8pin port's (port B and port D of 16F877) for data transfer, three register select pins, one read pulse pin and one write pulse pin. A total of 19 io pins. I am able to commented out cs1/cs0 and save pins because of the constant we set.

```
-- pin assignments
var volatile byte    pata_hd_data_low             is portb   -- data port
 (low bits)
alias    pata_hd_data_low_direction    is portb_direction
alias    pata_hd_data_high             is portd   -- data port (high bits)
alias    pata_hd_data_high_direction   is portd_direction

alias     pata_hd_a0                    is pin_a0
alias     pata_hd_a0_direction          is pin_a0_direction
alias     pata_hd_a1                    is pin_a1
alias     pata_hd_a1_direction          is pin_a1_direction
alias     pata_hd_a2                    is pin_a2
alias     pata_hd_a2_direction          is pin_a2_direction

alias     pata_hd_iowr                  is pin_c2
alias     pata_hd_iowr_direction        is pin_c2_direction
alias     pata_hd_iord                  is pin_c1
alias     pata_hd_iord_direction        is pin_c1_direction

alias     pata_hd_cs1                   is pin_a3
alias     pata_hd_cs1_direction         is pin_a3_direction
alias     pata_hd_cs0                   is pin_a4
alias     pata_hd_cs0_direction         is pin_a4_direction

pata_hd_a0_direction = output    -- register select pin
pata_hd_a1_direction = output    -- register select pin
```

```
pata_hd_a2_direction = output    -- register select pin

pata_hd_iowr_direction = output  -- used for write pulse
pata_hd_iord_direction = output  -- used for read pulse

;pata_hd_cs1_direction = output   -- register select pin
;pata_hd_cs0_direction = output   -- register select pin
```

Now include the library

```
include pata_hard_disk           -- include the parallel ata ide hard disk
 library
pata_hd_init()                   -- initialize startup settings
```

Add user's procedure and variables

Hard disks send data 2 bytes at a time since there are two 8 pin data ports, so I made a small serial port procedure to send 2 bytes via the serial port:

```
-- Function for sending hard disk data via serial
-- port, data is read 2 bytes at a time.
procedure send_word(byte in lowbit, byte in highbit) is
serial_hw_write(lowbit)  -- send 1st serial data byte
serial_hw_write(highbit) -- send 2nd serial data byte
end procedure
```

Now declare variables for recieved data

```
-- Declare variables for this example.
var byte in_a
var byte in_b
```

Wait for power to stabilize then send "START" to the serial port to notify the user (YOU) that the program has started ok

```
_usec_delay (1_000_000) -- wait for power to stabilize

-- send "start" to pc / test uart communication
send_word("S", "T")
send_word("A", "R")
send_word("T", 0x20)
send_word(13, 10)
send_word(13, 10)
```

Spin Up/Spin Down test

It is important to know if we have some basic communication to the drive. We will try to spin up (turn on the drive's motor) and spin down (turn off the drive's motor). This will simply send the "spin up" command to the command register then "spin down", then it will do the same once more. This shows that we have communication from your PIC to the hard drive.

```
for 2 loop
   pata_hd_register_write(PATA_HD_COMMAND_REG,PATA_HD_SPIN_UP)   -- turn on
 motor
   _usec_delay(5_000_000) -- 5 sec delay
   pata_hd_register_write(PATA_HD_COMMAND_REG,PATA_HD_SPIN_DOWN) -- turn off
 motor
   _usec_delay(5_000_000) -- 5 sec delay
end loop

pata_hd_register_write(PATA_HD_COMMAND_REG,PATA_HD_SPIN_UP)       -- turn on
 motor
```

Wait 10 seconds before next example

```
_usec_delay(10_000_000) -- wait 10 seconds before next example
```

Read the first and second sector from the hard drive

Now that we know we are able to write to the registers, we can try to read some data. One sector is 512 bytes. Since data is transfered 2 bytes at a time, we will loop 256 times to read one full sector while sending the data via serial port.

Reading is easy, there are 3 procedures within the library that MUST be used. You will notice this process is similar to the SD card tutorial.

pata_hd_start_read(0) - start reading at specified sector (sector 0)

pata_hd_read_data(byte1, byte2) - actually read data from the card (2 bytes at a time)

pata_hd_stop_read() - stop the read process

You can also use the pata_hd_read_pulse(number) procedure to skip past data. For every 1 value added, there will be 2 bytes skipped since this procedure simply reads data and ignores the input.

```
-- Read one sector
for 256 loop                            -- 256 words, 512 bytes per sector
   pata_hd_read_data(in_b, in_a)        -- read data
   send_word(in_b, in_a)                -- send data via serial port
end loop

-- You will see hard disk LED on during this delay
-- because you did not finnish reading.
_usec_delay(2_000_000) -- 2 second delay

-- Read 2nd sector.
for 256 loop                            -- 256 words, 512 bytes per sector
   pata_hd_read_data (in_b, in_a)        -- read data
   send_word(in_b, in_a)            -- send data via serial port
end loop

pata_hd_stop_read()       -- tell drive you are done reading
-- hard disk led will turn off at this point.

_usec_delay(10_000_000) -- wait 10 seconds before next example
```

Identify drive command

The identify drive command loads 512 bytes of data for you that contains information about your drive. You can retrieve info like drive serial number, model number, drive size, number of cylinders, heads, sectors per track and a bunch of other data required by your PC. Of course you can read more info on this at the links I have given you.

On the sticker of some older drives, you will see "CYL", "HEADS", "SEC/T" (this can also be found with the Identify command). You can calculate drive's addressable sectors with (cylinders * heads * sectors per track), and multipy that by 512) for the size of the drive.

On newer drives, you will see on the front sticker the number of LBA's, this is the number of addressable sectors. If you multiply this value by 512, you will get the size of the drive in bytes. For example, one of my drive says 60058656 LBA's. With this drive, you can send a pata_start_read command with a addresses from 0 to (60058656 - 1). The size of this drive is 60058656 * 512 = 30GB

Let's try it out, first we send the command:

```
-- send the identify drive command
pata_hd_register_write(PATA_HD_COMMAND_REG,PATA_HD_IDENTIFY_DRIVE)
```

Now we must wait till the drive is ready and has data for us:

```
-- check if drive is ready reading and set data ports as inputs
-- this MUST be used before reading since we did not use pata_hd_start_read
pata_hd_data_request(PATA_HD_WAIT_READ)
```

The drive is now has data for us, so let's read it. Notice that the input data bytes (in_b & in_a) are backwards for identify drive (don't ask me why).

```
-- Read 512 bytes
for 256 loop                      -- 256 words, 512 bytes per sector
   pata_hd_read_data(in_b, in_a) -- read data
   send_word(in_a, in_b )         -- send data via serial port
end loop                          -- drive info high/low bytes are in reverse
 order
```

Wait 10 seconds before the next example

```
_usec_delay(10_000_000) -- wait 10 seconds before next example
```

Write data to the drive

Just like reading, there are 3 procedures that MUST be used.

pata_hd_start_write(20) - start writing at specified sector (sector 20)

pata_hd_read_data(byte1, byte2) - write to the card (2 bytes at a time)

pata_hd_stop_write() - stop the read process

When writing to your hard drive, you MUST write 512 bytes at a time. In this example, we are writing (256x2) = 512 bytes + (250x2) = 500 bytes for a total of 1012 bytes. This means we have written one sector (512 bytes), as well as 500 bytes of the next sector. The second sector (500 bytes) that we have written, will not actually be written to the hard drive until we finish the sector with data.

For this reason, you will need to use the pata_hd_write_to_sector_end(value) procedure. This procedure will automatically finish the sector for you with the "value" data specified. In our case we are writing 0xFF till the end of the 512 bytes (end of the sector).

Here's an example write, Please note that we are starting to write at sector 200

```
pata_hd_start_write(200) -- tell hd to get ready for reading

-- now write 1 sector + most of 2nd sector, data will not
-- be written unless 512 bytes are sent
for 256 + 250 loop
   pata_hd_write_data(0xCC, 0xDD) -- write data 0xCC, 0xDD over and over
end loop
-- first sector has been written to the disk since 512 bytes where sent,
-- but 2nd sector is not finnished, only 500 bytes sent,
-- so lets finnish the sector with 6 more write pulses (0xFF's as data)
pata_hd_write_to_sector_end(0xFF)

pata_hd_stop_write()     -- tell hd we are done writing
```

Now read back the data the 1012 bytes have been written

```
-- Now read the 1st sector you just wrote, should get
-- 0xCC, 0xDD over and over
pata_hd_start_read(200)                -- get drive ready for reading
for 256 + 250 loop                     -- read 512 bytes + 500 bytes
   pata_hd_read_data(in_b, in_a)       -- read data
   send_word(in_b, in_a)           -- send data via serial port
end loop

-- if you want, you can read back the last 6 bytes that are 0xFF
for 6 loop
   pata_hd_read_data(in_b, in_a)       -- read data
   send_word(in_b, in_a)           -- send data via serial port
end loop

pata_hd_stop_read()                    -- tell drive we are done reading
```

If you want, you can turn off the hard drive motor at the end of the program

```
-- We're done, lets turn off the hd motor
pata_hd_register_write(PATA_HD_COMMAND_REG,PATA_HD_SPIN_DOWN)
```

Your Done!

That's it, Now you can read & write to all those hard drives you have laying around. You can read raw data from drives and possibly even get back some lost data.

Alright, go build that hard disk thingy you where dreaming about!

Interfacing a Sharp GP2D02 IR ranger

Sébastien Lelong
Jallib Group

Sharp IR rangers are widely used out there. There are many different references, depending on the beam pattern, the minimal and maximal distance you want to be able to get, etc... The way you got results also make a difference: either **analog** (you'll get a voltage proportional to the distance), or **digital** (you'll directly get a digital value). This nice article will explain these details (and now I know GP2D02 seems to be discontinued...)

Overview of GP2D02 IR ranger

GP2D02 IR ranger is able to measure distances between approx. 10cm and 1m. Results are available as digital values you can access through a dedicated protocol. One pin, Vin, will be used to act on the ranger. Another pin, Vout, will be read to determine the distance. Basically, getting a distance involves the following:

1. First you wake up the ranger and tell it to perform a distance measure
2. Then, for each bit, you read Vout in order to reconstitute the whole byte, that is, the distance
3. finally, you switch off the ranger

The following timing chart taken from the datasheet will explain this better.

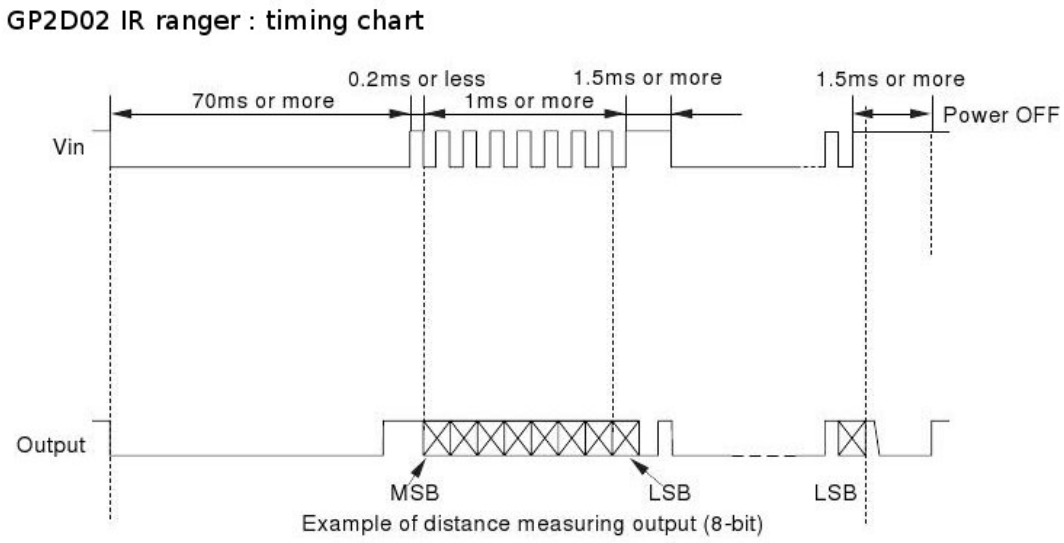
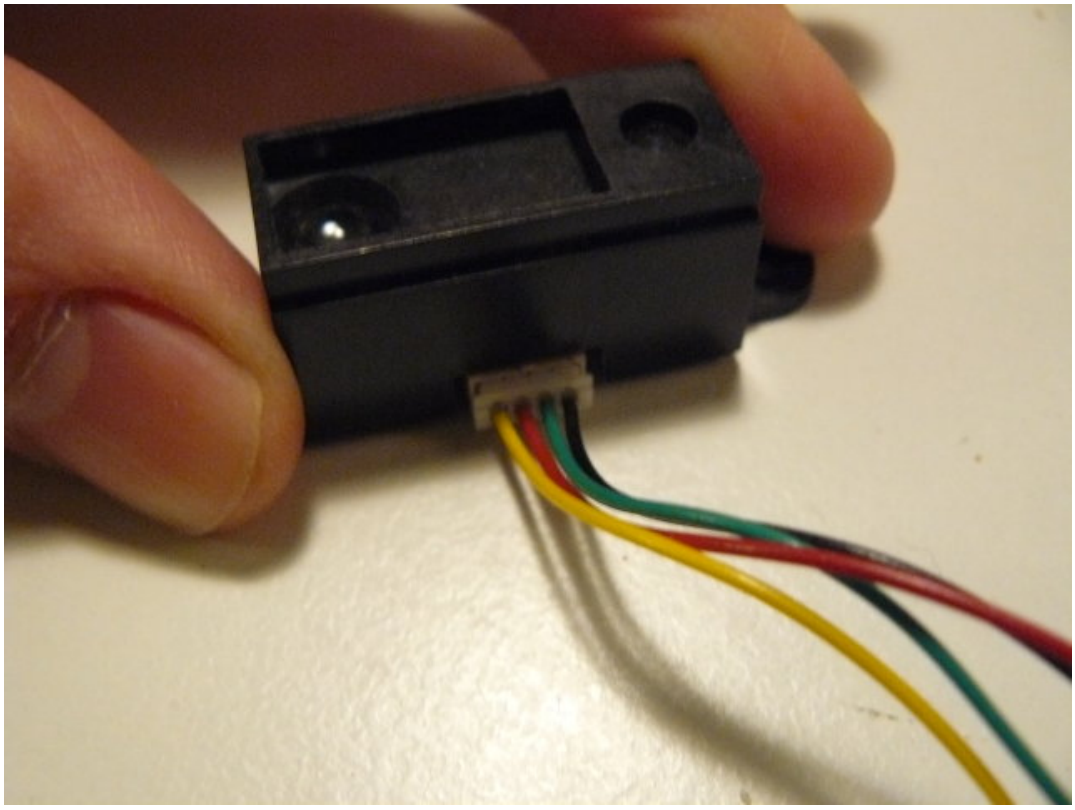


Figure 8: GD2D02 IR ranger : timing chart

Note: the distances obtained from the ranger aren't linear, you'll need some computation to make them so.

Sharp GP2D02 IR ranger looks like this:



- *Red* wire is for +5V
- *Black* wire ground
- *Green* wire is for Vin pin, used to control the sensor
- *Yellow* wire is for Vout pin, from which 8-bits results read

(make a mental note of this...)

Interfacing the Sharp GP2D02 IR ranger

Interfacing such a sensor is quite straight forward. The only critical point is **Vin** ranger pin can't handle high logic level of the PIC's output, *this level mustn't exceed 3.3 volts*. A **zener diode** can be used to limit this level.

Note: be careful while connecting this diode. Don't forget it, and don't put it in the wrong side. You may damage your sensor. And I'm not responsible for ! You've been warned... That's said, I already forgot it, put it in the wrong side, and thought I'd killed my GP2D02, but this one always got back to life. Anyway, be cautious !

Here's the whole schematic. The goal here is to collect data from the sensor, and light up a LED, more or less according to the read distance. That's why we'll use a LED driven by PWM.

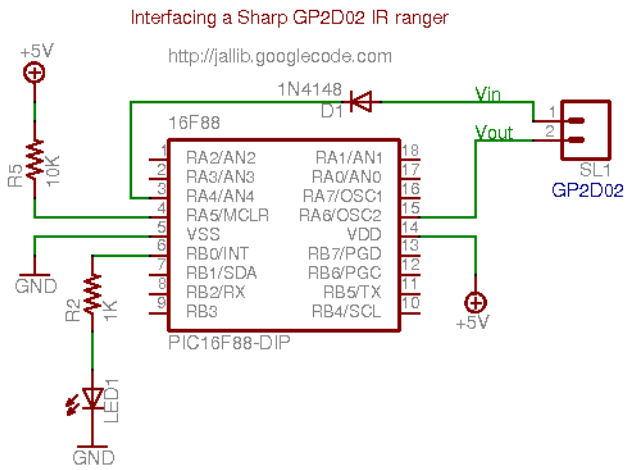
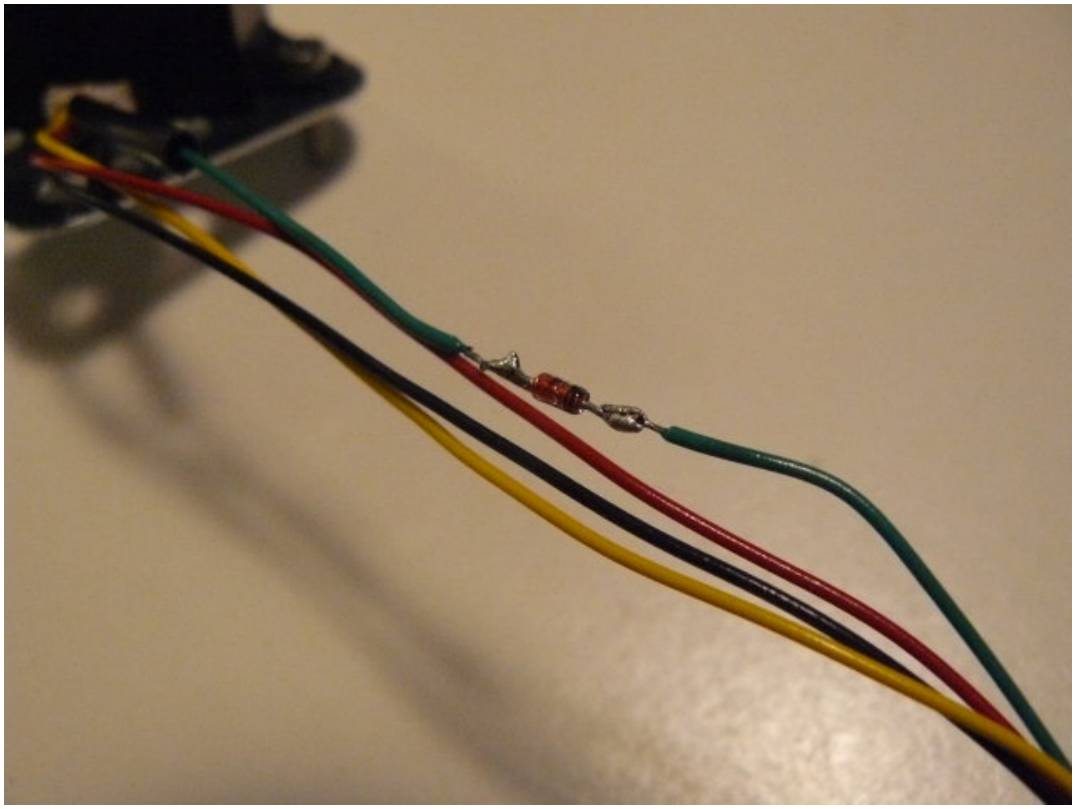
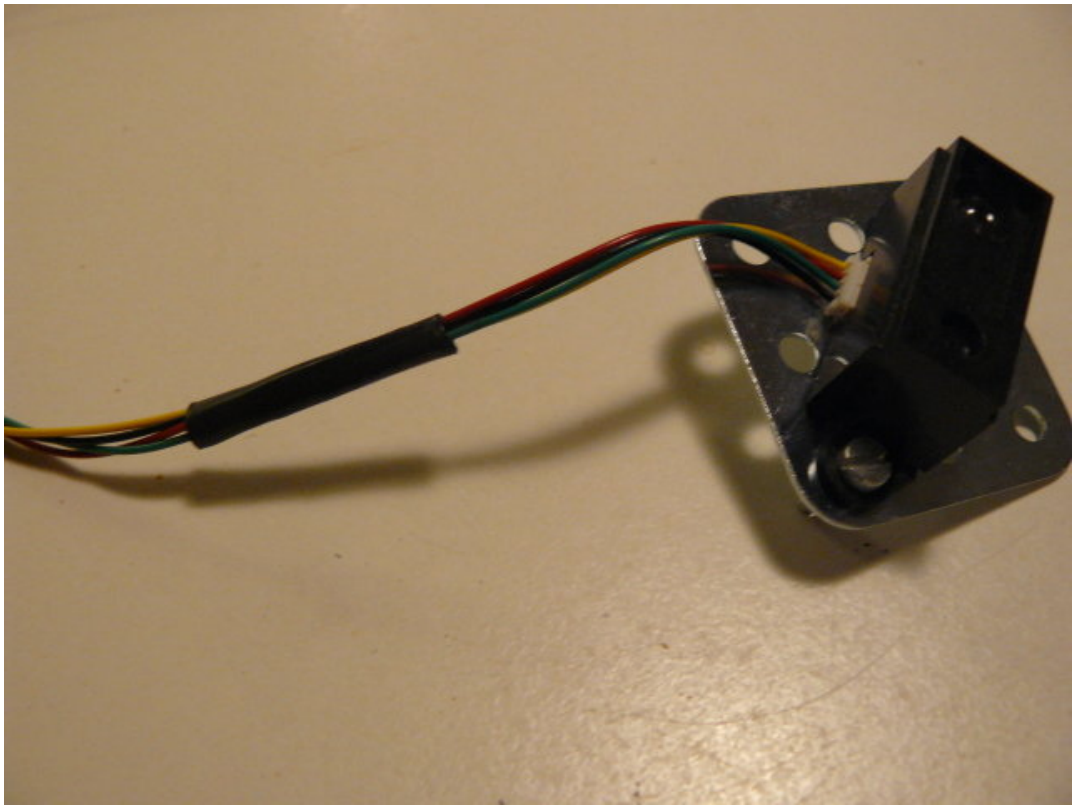


Figure 9: Interfacing Sharp GP2D02 IR range : schematic

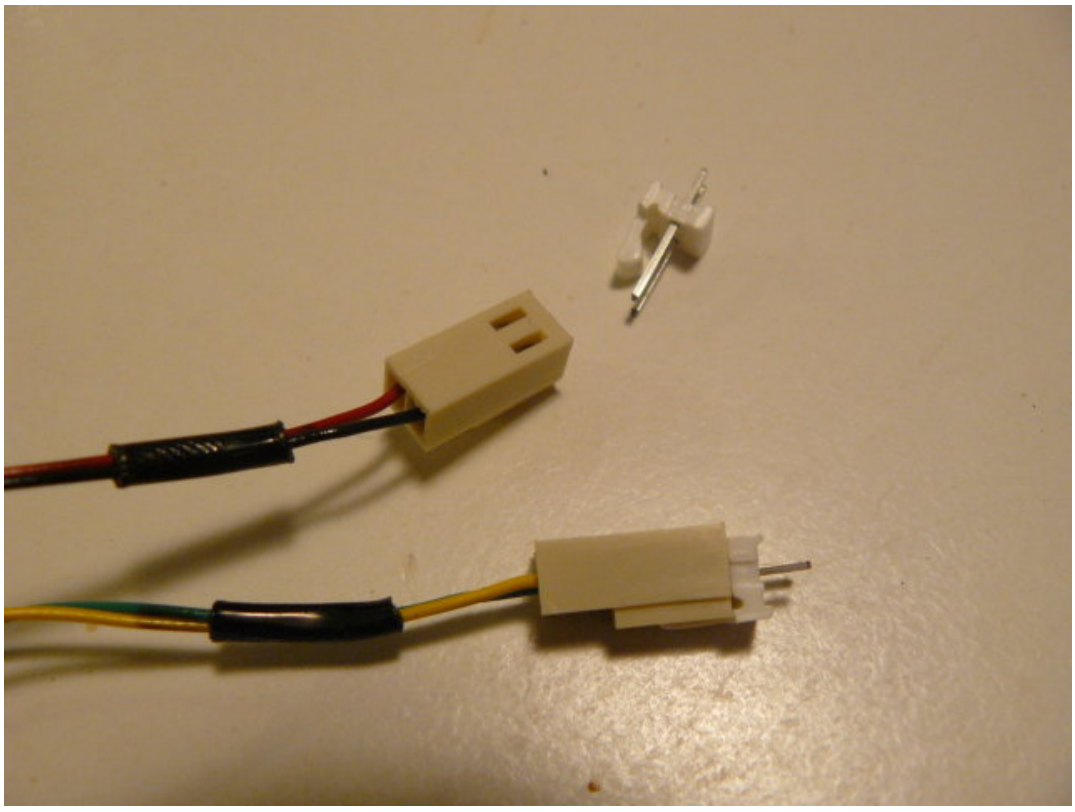
Here's the ranger with the diode soldered on the green wire (which is Vin pin, using your previously created mental note...):



I've also added thermoplastic rubber tubes, to cleanly join all the wires:



Finally, in order to easily plug/unplug the sensor, I've soldered nice polarized connectors:



Writing the program

jallib >=0.3 contains a library, *ir_ranger_gp2d02.jal*, used to handle this kind of rangers. The setup is quite straight forward: just declare your Vin and Vout pins, and pass them to the `gp2d02_read_pins()`. This function returns the distance as a raw value. Directly passing pins allows you to have multiple rangers of this type (many robots have many of them arranged in the front and back sides, to detect and avoid obstacles).

Using PWM libs, we can easily make our LED more or less bright. In the mean time, we'll also transmit the results through a serial link.

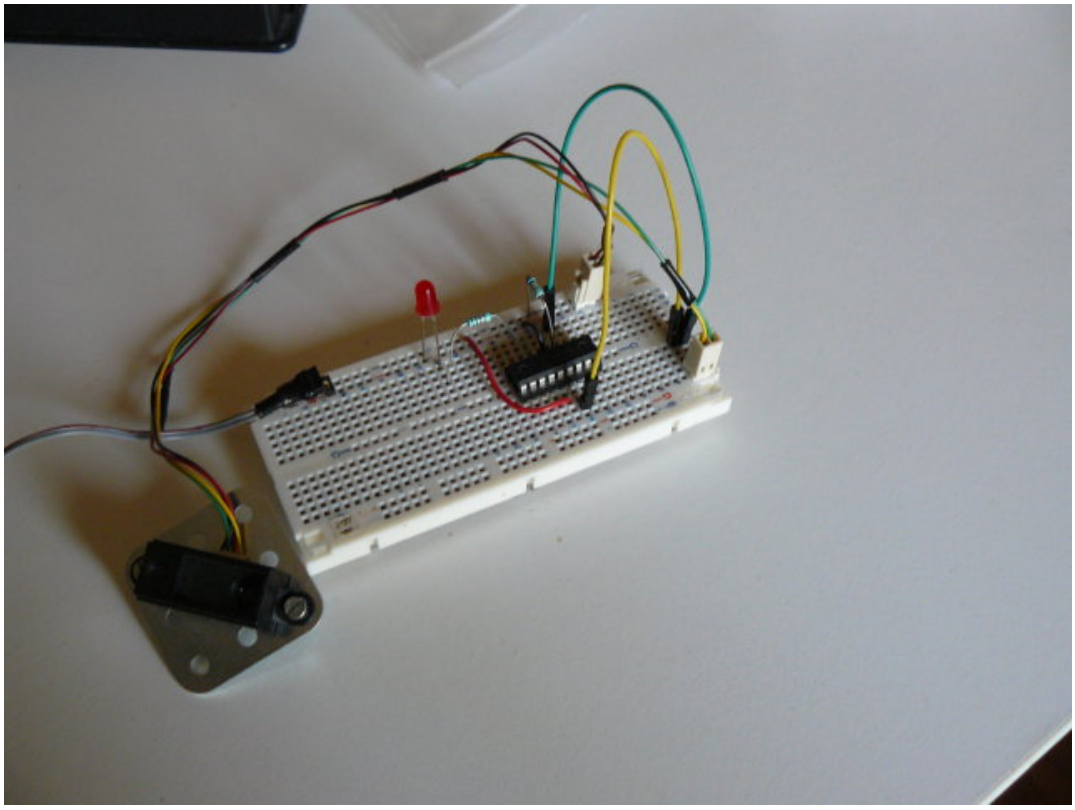
```
var volatile bit gp2d02_vin is pin_a4
var volatile bit gp2d02_vout is pin_a6
var bit gp2d02_vin_direction is pin_a4_direction
var bit gp2d02_vout_direction is pin_a6_direction
include ir_ranger_gp2d02
-- set pin direction (careful: "vin" is the GP2D02 pin's name,
-- it's an input for GP2D02, but an output for PIC !)
gp2d02_vin_direction = output
gp2d02_vout_direction = input

var byte measure
forever loop
   -- read distance from ranger num. 0
   measure = gp2d02_read_pins(gp2d02_vin,gp2d02_vout)
   -- results via serial
   serial_hw_write(measure)
   -- now blink more or less
   pwm1_set_dutycycle(measure)
end loop
```

Note: I could directly pass `pin_A4` and `pin_A6`, but to avoid confusion, I prefer using *aliases*.

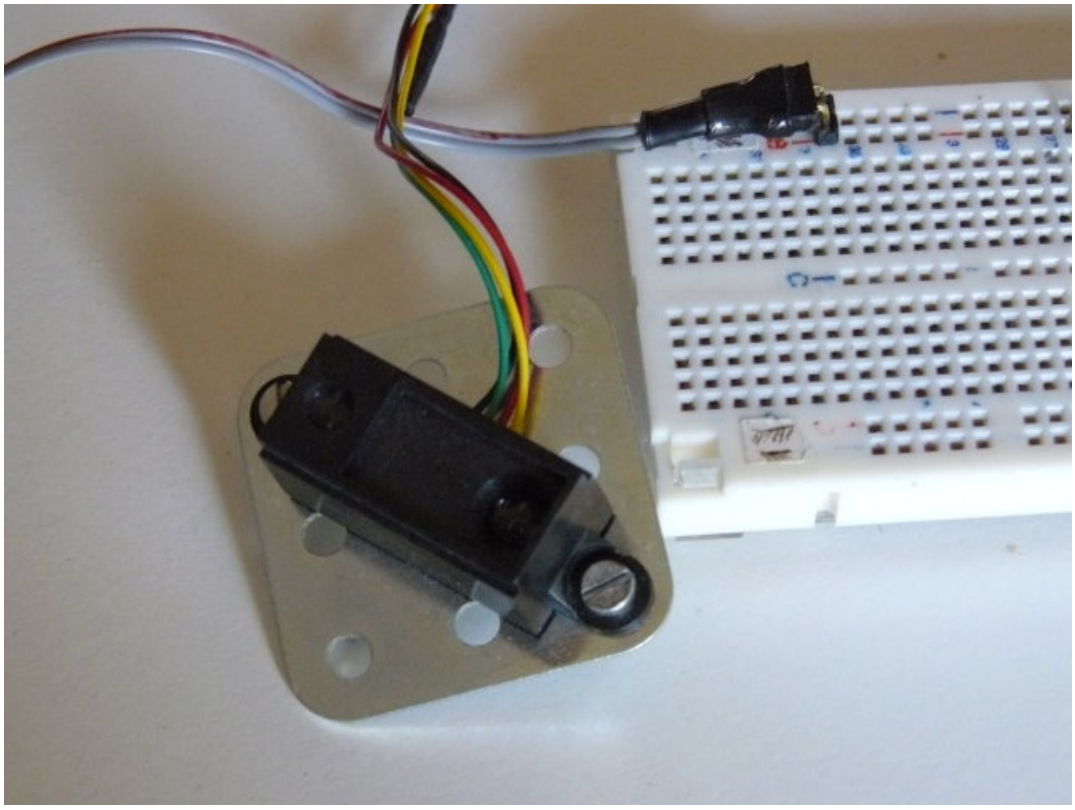
A sample, *16f88_ir_ranger_gp2d02.jal*, is available in *jallib SVN repository*jallib released packages, and also in , starting from version 0.3. You can access downloads *here*.

Building the whole on a breadboard

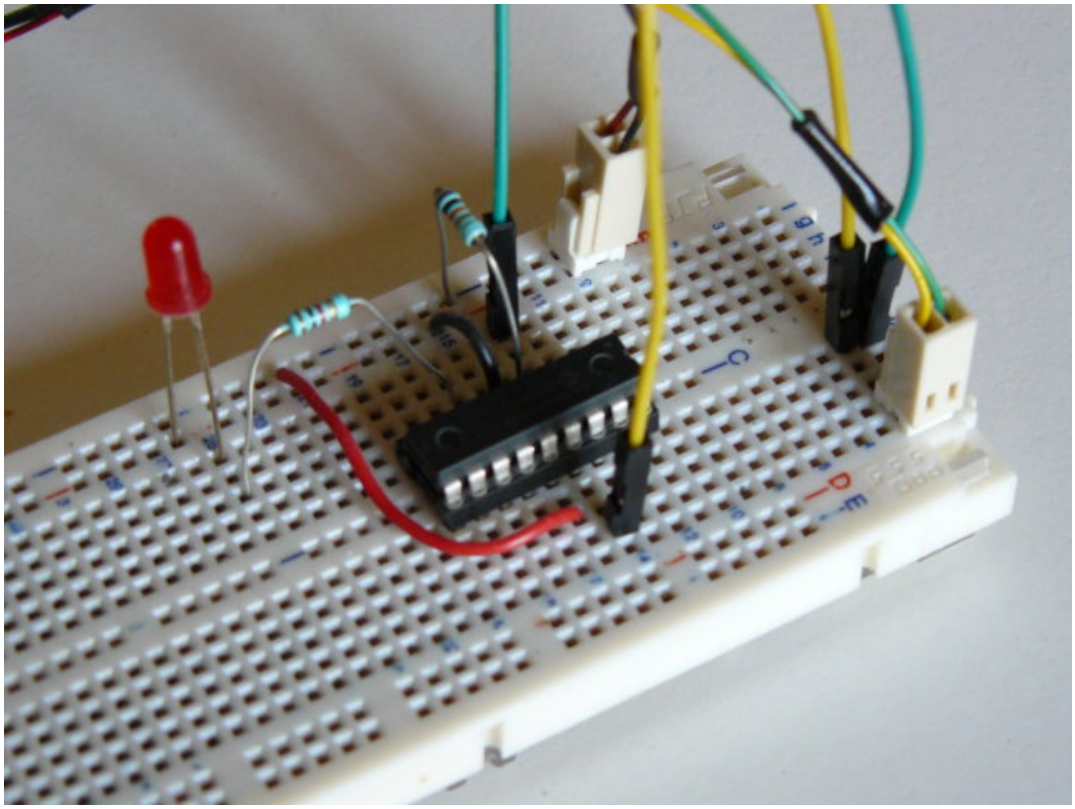
Building the whole on a breadboard



I usually power two tracks on the side, used for the PIC and for the ranger:



Using the same previously created mental note, I connected the yellow Vout pin using a yellow wire, and the green Vin pin using a green wire...



Testing (and the video)

Time to test this nice circuit ! Power the whole, and check no smoke is coming from the PIC or (and) the ranger. Now get an object, like you hand, more or less closed to the ranger and observe the LED, or the serial output... Sweet !

http://www.youtube.com/watch?v=l5AZwv7LzyM

Interfacing a HD44780-compatible LCD display

Sébastien Lelong
Jallib Group

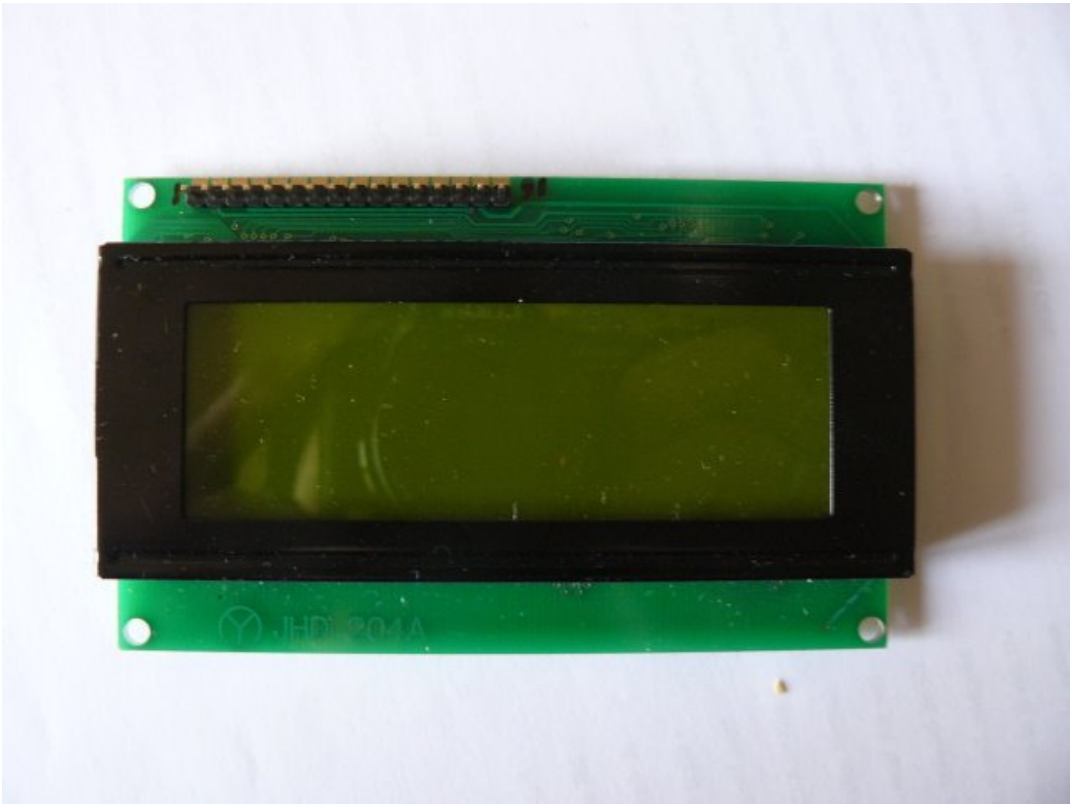
In this "Step by Step" tutorial, we're going to (hopefully) have some fun with a LCD display. Particularly one compatible with HD44780 specifications, which seems to be most widely used.

Setting up the hardware

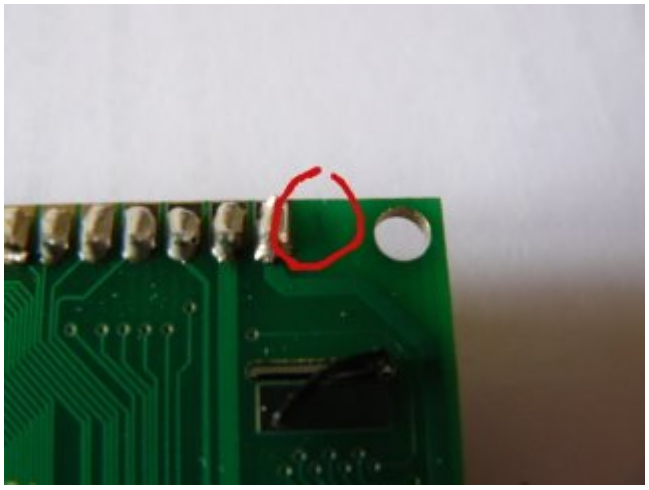
As usual, there are *plenty resources* on the web. I found *this one* quite nice, covering lots of thing. Basically, LCDs can be accessed with two distinct interfaces: *4-bit or 8-bit interfaces*. *4-bit interface requires less pins* (4 pins), but is somewhat slow, *8-bit interface requires more pins* (8 pins) but is faster. jallib comes with the two flavors, it's up to you deciding which is most important, but usually, pins are more precious than speed, particularly when using a 16F88 which only has 16 I/O pins (at best). Since 4-bit interface seems to be most used, and we'll use this one here...

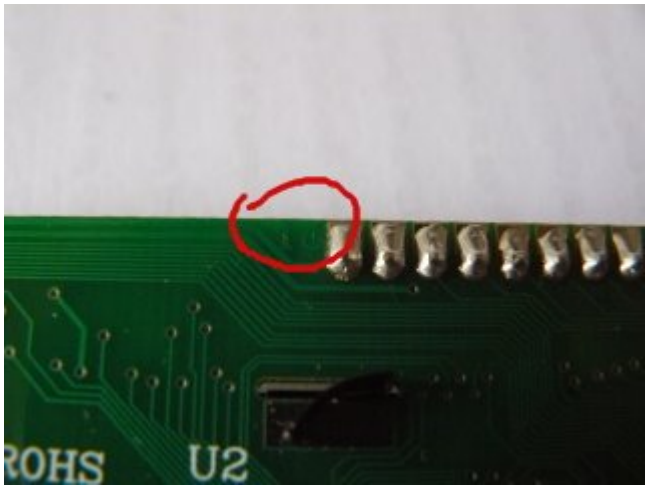
So, I've never used LCD, to be honest. Most guys consider it as an absolute minimum thing to have, since it can help a lot when debugging, by printing messages. I tend to use serial for this... Anyway, I've been given a LCD, so I can't resist anymore :)

The LCD I have seems to be quite a nice beast ! It has 4 lines, is 20 characters long.



Looking closer, "*JHD 204A*" seems to be the reference. Near the connector, only a "1" and a "16". No pin's name.





Googling the whole points to a forum, and at least a link to the *datasheet*. A section describes the "Pin Assignement". Now I'm sure about how to connect this LCD.

- **Pin assignment**

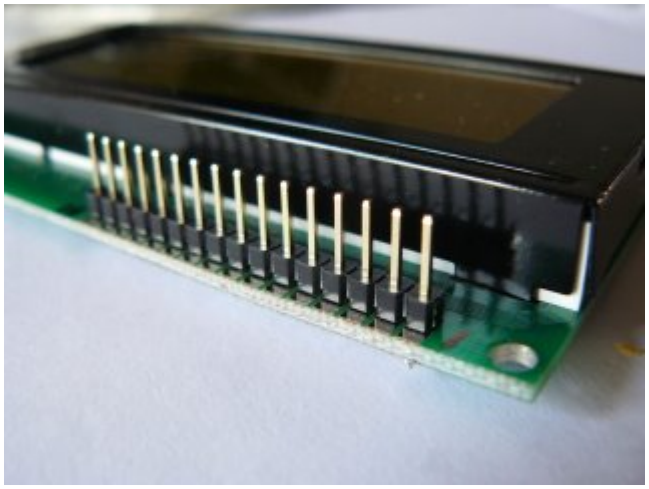
Pin NO.	Symbol	Function		Remark
1	GND	Power supply	0V	
2	Vdd		+5V	
3	V5		For LCD	Variable
4	RS	Register Select(H=Data,L=Instruction)		
5	R/W	Read/Write L=MPU to LCM,H=LCM to MPU		
6	E	Enable		
7	DB0	Data bus bit 0		
8	DB1	Data bus bit 1		
9	DB2	Data bus bit 2		
10	DB3	Data bus bit 3		
11	DB4	Data bus bit 4		
12	DB5	Data bus bit 5		
13	DB6	Data bus bit 6		
14	DB7	Data bus bit 7		
15	A	Anode of LED Unit		
16	K	Cathode of LED Unit		

For this tutorial, we're going to keep it simple:

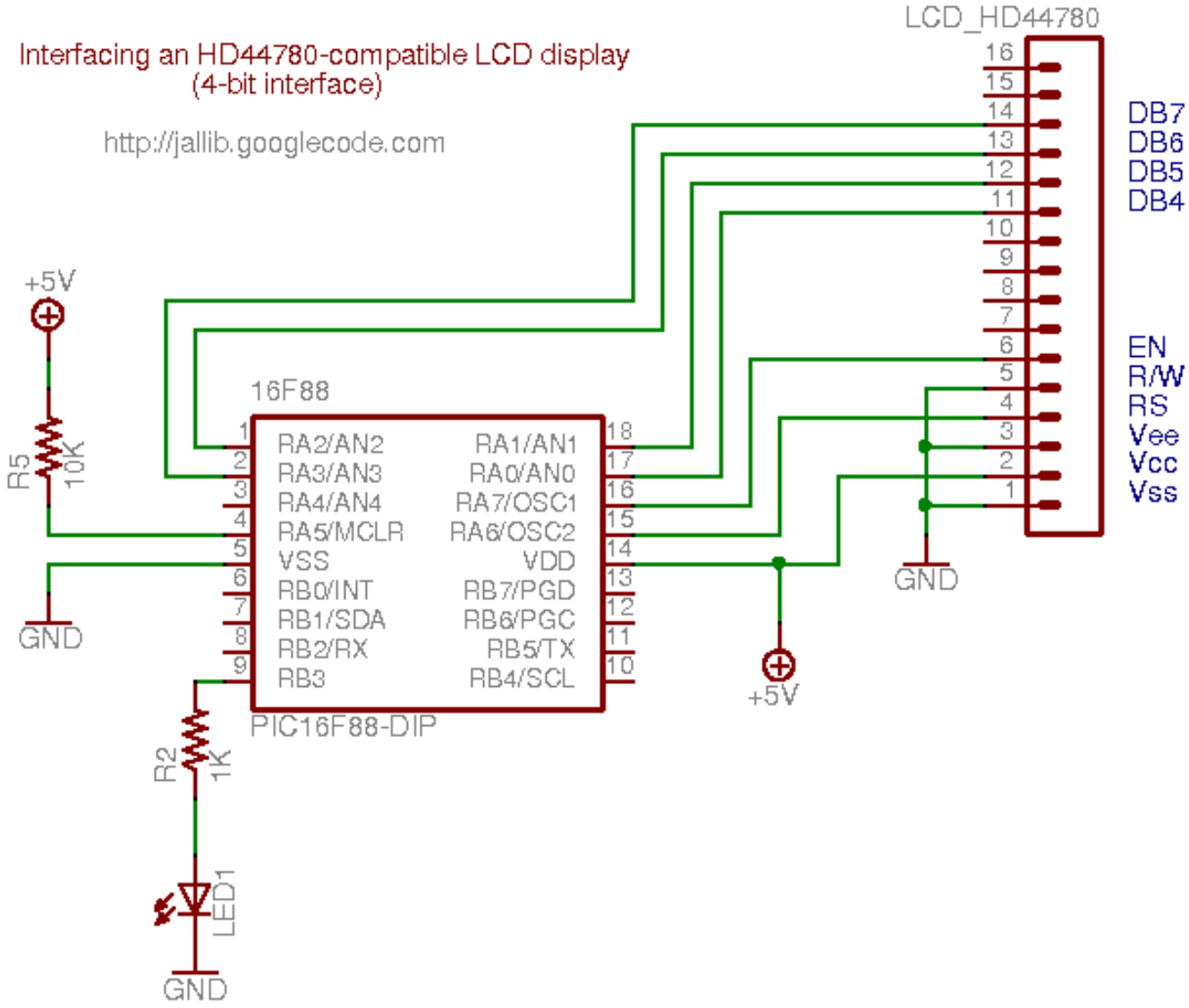
- as previously said, we'll use 4-bit interface. This means we'll use DB4, DB5, DB6 and DB7 pins (respectively pin 11, 12, 13 and 14).
- we won't read from LCD, so R/W pin must be grounded (pin 5)
- we won't use contrast as well, V5 pin (or Vee) must be grounded (pin 3)

Including pins for power, we'll use 10 pins out of the 16 available, 6 being connected to the PIC (RS, EN and 4 data lines).

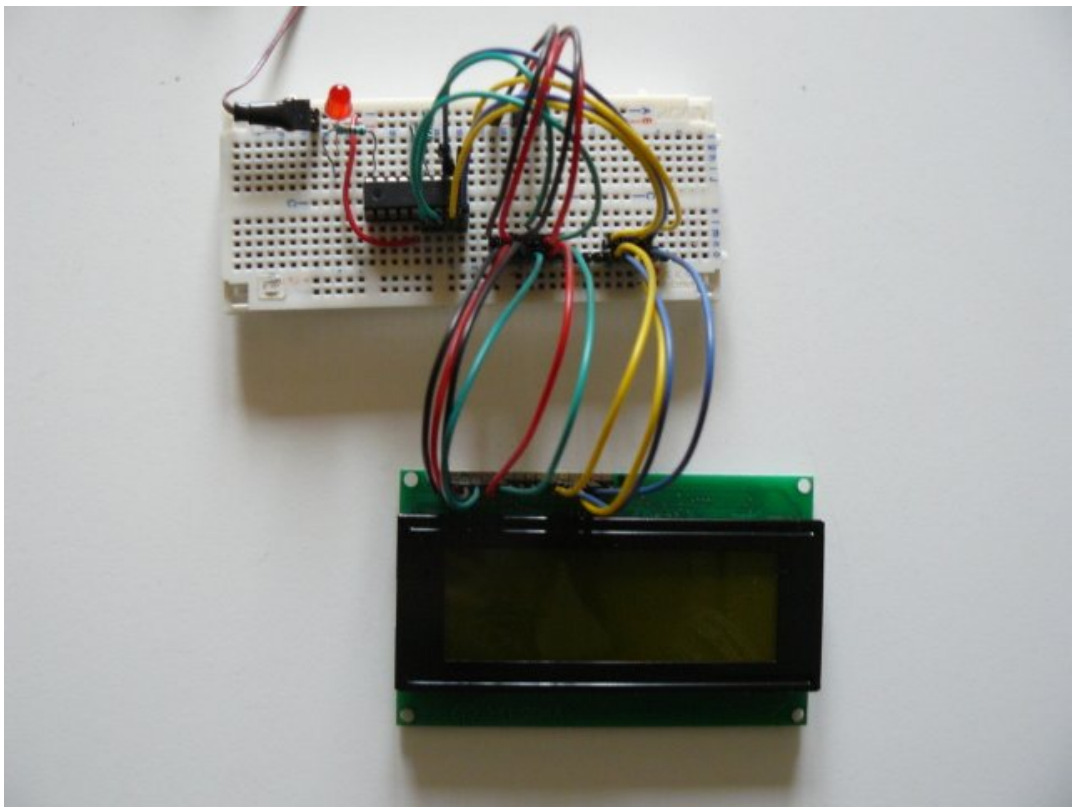
For convenience, I soldered a male connector on the LCD. This will help when building the whole on a breadboard.

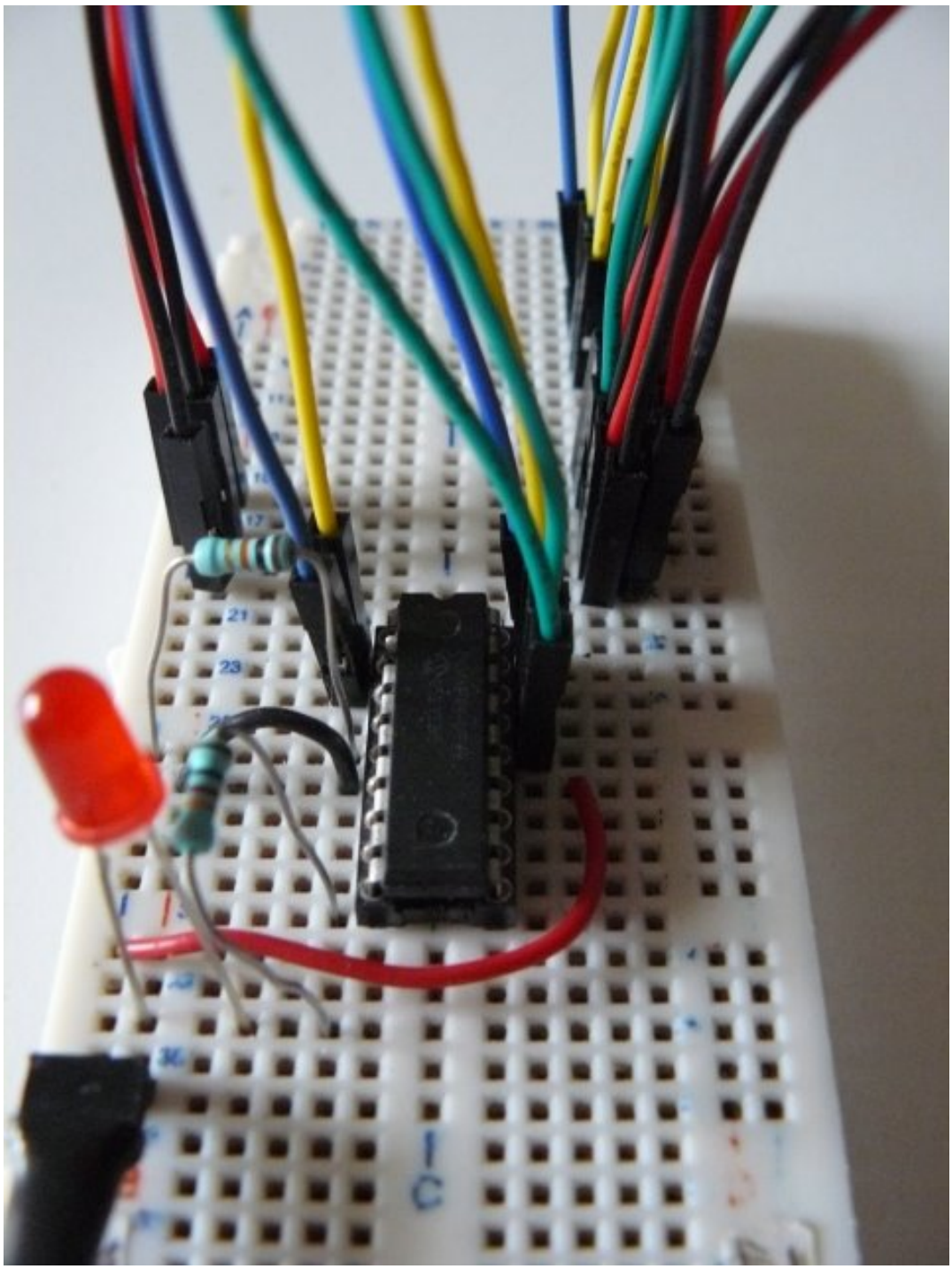


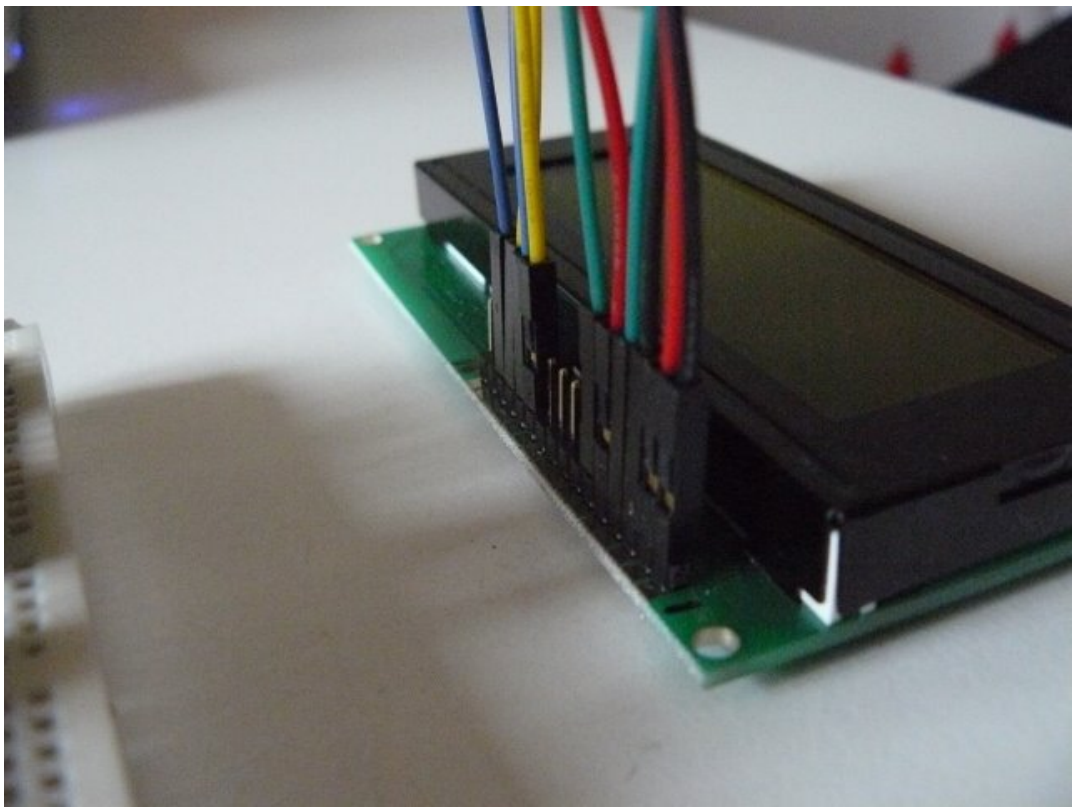
So we now have everything to build the circuit. Here's the schematic. It also includes a LED, it will help us checking everything is ok while powering up the board.



Using a breadboard, it looks like this:







Writing the software

For this tutorial, we'll use one of the available samples from *jallib repository*. I took *one* for 16f88, and adapt it to my board (specifically, I wanted to use PORTA when connecting the LCD, and let PORTB's pins as is).

As usual, writing a program with jallib starts with configuring and declaring some parameters. So we first have to declare which pins will be connected:

```
 -- LCD IO definition
var bit lcd_rs           is pin_a6              -- LCD command/data select.
var bit lcd_rs_direction is pin_a6_direction
var bit lcd_en           is pin_a7              -- LCD data trigger
var bit lcd_en_direction is pin_a7_direction

var byte lcd_dataport is porta_low              -- LCD data  port
var byte lcd_dataport_direction is porta_low_direction

-- set direction
lcd_rs_direction = output
lcd_en_direction = output
lcd_dataport_direction = output
```

This is, pin by pin, the translation of the schematics. Maybe except `porta_low`. This represents pin A0 to A3, that is pins for our 4 lines interface. `porta_high` represents pin A4 to A7, and `porta` reprensents the whole port, A0 to A7. These are just "shorcuts".

We also have to declare the **LCD geometry**:

```
const byte LCD_ROWS     = 4      -- 4 lines
const byte LCD_CHARS    = 20     -- 20 chars per line
```

Once declared, we can then include the library and initialize it:

```
include lcd_hd44780_4   -- LCD library with 4 data lines
```

```
lcd_init()              -- initialize LCD
```

For this example, we'll also use the `print.jal` library, which provides nice helpers when printing variables.

```
include print
```

Now the main part... How to write things on the LCD.

- You can either use a procedure call: `lcd_write_char("a")`
- or you can use the pseudo-variable : `lcd = "a"`
- `lcd_cursor_position(x,y)` will set the cursor position. x is the line, y is the row, starting from 0
- finally, `lcd_clear_screen()` will, well... clear the screen !

Full API documentation is available on *jalapi*.

So, for this example, we'll write some chars on each line, and print an increasing (and incredible) counter:

```
const byte str1[] = "Hello world!"      -- define strings
const byte str2[] = "third line"
const byte str3[] = "fourth line"

print_string(lcd, str1)                 -- show hello world!
lcd_cursor_position(2,0)                   -- to 3rd line
print_string(lcd, str2)
lcd_cursor_position(3,0)                   -- to 4th line
print_string(lcd, str3)

var byte counter = 0

forever loop                            -- loop forever

   counter = counter + 1                -- update counter
   lcd_cursor_position(1,0)                -- second line
   print_byte_hex(lcd, counter)         -- output in hex format
   delay_100ms(3)                       -- wait a little

   if counter == 255 then               -- counter wrap
      lcd_cursor_position(1,1)             -- 2nd line, 2nd char
      lcd = " "                         -- clear 2nd char
      lcd = " "                         -- clear 3rd char
   end if

end loop
```

The full and ready-to-compile code is available on jallib repository:

- *blog_16f88_sl_lcd_hd44780_4.jal*

You'll need last jallib-pack, available on jallib's *download section*.

How does this look when running ?

Here's the video !

http://www.youtube.com/watch?v=hIVMuaz8OS8

Memory with 23k256 sram

Matthew Schinkel
Jallib Group

Learn how to use Microchip's cheap 256kbit (32KB) sram for temporary data storage

What is the 23k256 sram and why use it?

So, you need some data storage? Put your data on a 23k256!

If speed is your thing, this one is for you! This is FAST. According to Microchip's datasheet, data can be clocked in at 20mhz. The disadvantage to this memory however is that it will not hold it's memory when power is off since it is a type of RAM (Random Access memory).

If you wish to hold memory while power is off, you will have to go with EEPROM but it is much slower. EEPROM requires a 1ms delay between writes. In the time that I could write 1 byte to an EEPROM (1ms), I could write 2500 bytes to the 23k256 (if I can get my PIC to run fast enough).

Yet another advantage, is that it is only 8 pins (as you can see from the image). Other RAM memories have 10 or so address lines + 8 data lines. If you haven't guessed yet, we are sending serial data for reads & writes. We will be using SPI (Serial Peripheral Interface Bus).

I suggest you start by reading the *SPI Introduction* within this book first.

You can read more about the 23k256 here:

http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en539039

What will I learn?

We will be using the jallib sram_23k256 library & sample written by myself. With this library, we will be able to do the following:

1. Initialization settings for 23k256.

2. Read settings from the 23k256.

3. Read & Write one byte to/from a specific address

4. Use the 23k256 as a large byte, word or dword array (32k bytes, 16k words, 8k dwords)

5. Fast read/write lots of data.

OK, lets get started

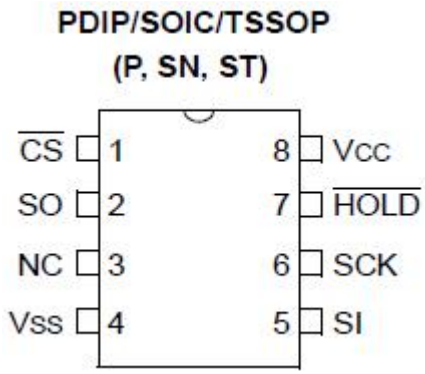
I suggest you start by compiling and writing the sample file to your PIC. We must make sure your circuit is working before we continue. As always, you will find the 23k256 sample file in the sample directory of your jallib installation "16f877_23k256.jal"

You will need to modify this sample file for your target PIC.

Note: Jallib version 0.5 had this following line in the library file, but in the next version (0.6) it will be removed from the library and you will have to add it to your sample file before the line include sram_23k256.

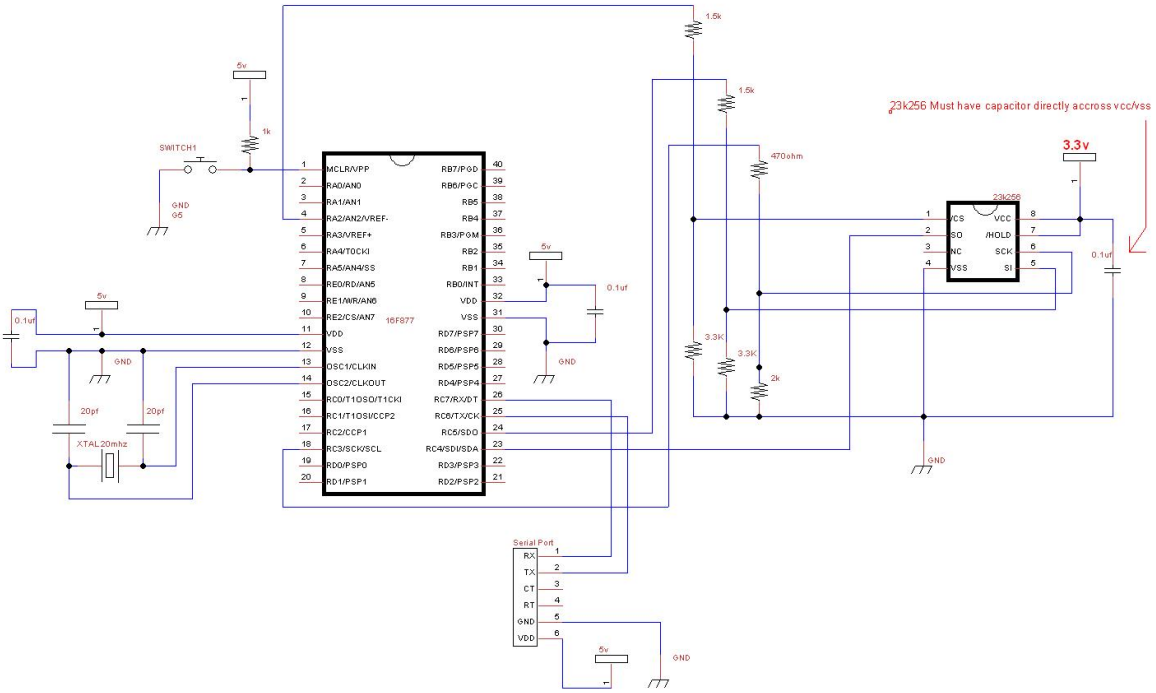
```
const bit SRAM_23K256_ALWAYS_SET_SPI_MODE = TRUE
```

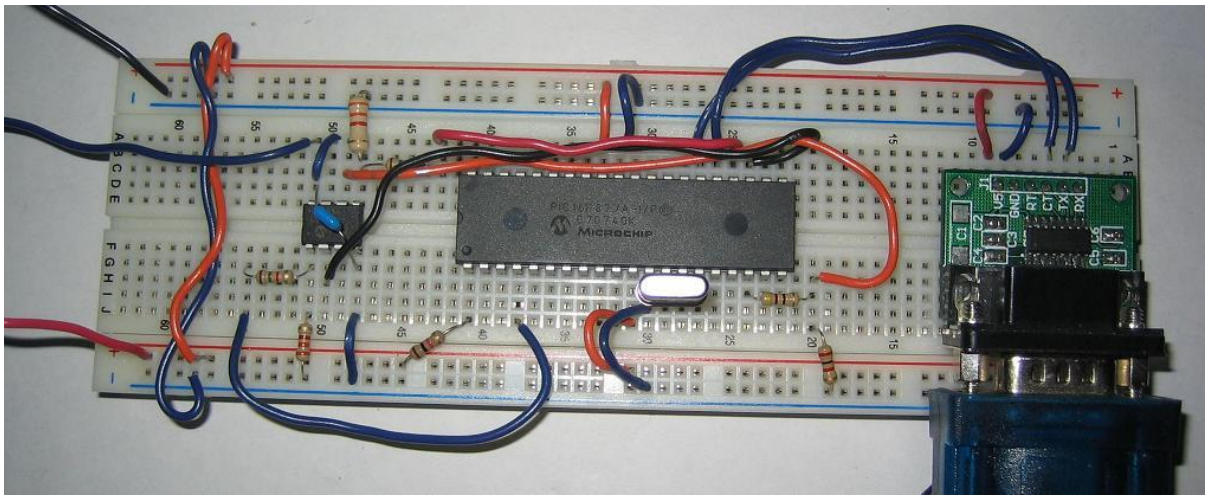
Here's the 23K256 pin-out diagram:



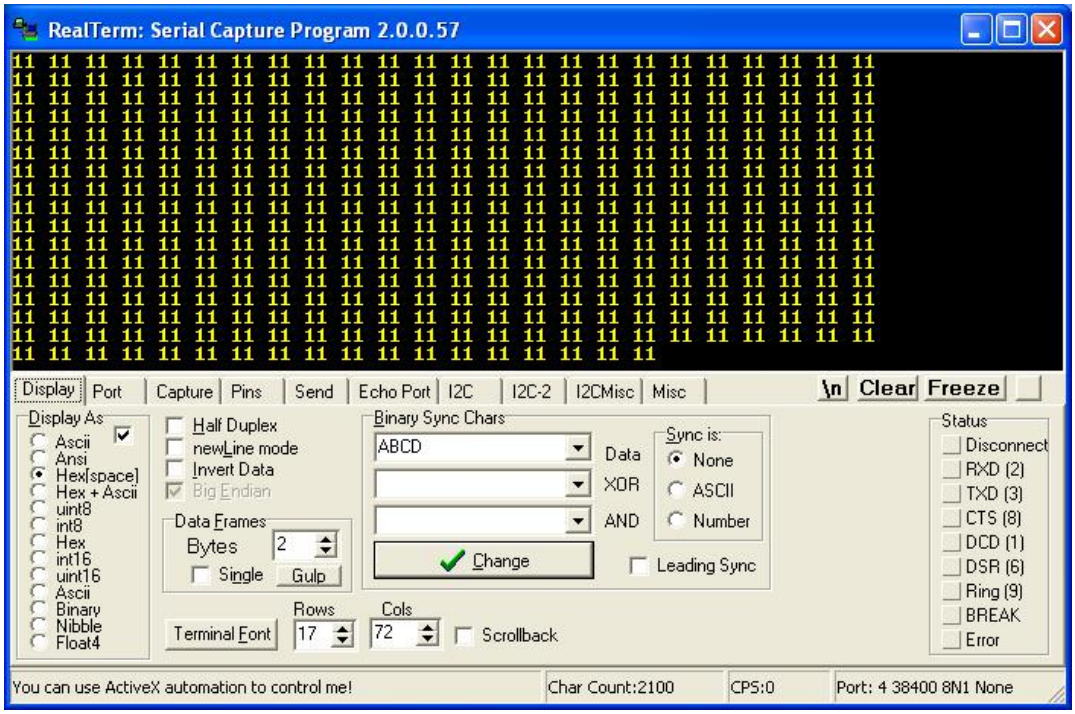
Now build this schematics. Notice the resistors for 5v to 3.3v conversion. Also notice that we are using the PIC's hardware SPI port pins. These pins are named (SDO, SDI, SCK) + One chip select pin of your choice.

Note: This is a 3.3V Device





Plug in your serial port and turn it on (serial baud rate 38400). If it is working, you will get mostly the hex value "11" to your PC's serial port. Here's an image from my serial port software:



If it is not working, lets do some troubleshooting. First start by checking over your schematic. If your schematic is correct, the most likely problem is voltage levels. Check over your PIC's datasheet to see what the PIN types are, if any of the pins have CMOS level outputs, you will not need voltage conversion resistors.

In the past, I have had issues with the voltage conversion resistors on the SCK line.

Setup the devices

Since the beginning initialization has already been written for you, and you already know how to include your PIC, you can skip this section and go down to the *23k256 Usage* section if you wish.

Take a look at the sample file you have. As you know, firstly, we will include your chip, disable all analog pins and setup serial communication.

```
include 16F877                    -- target PICmicro
--
-- This program assumes a 20 MHz resonator or crystal
-- is connected to pins OSC1 and OSC2.
pragma target clock 20_000_000     -- oscillator frequency
-- configure fuses
pragma target OSC  HS              -- HS crystal or resonator
pragma target WDT  disabled        -- no watchdog
pragma target LVP  disabled        -- no Low Voltage Programming

enable_digital_io()                -- disable analog I/O (if any)

-- setup uart for communication
const serial_hw_baudrate  = 38400   -- set the baudrate
include serial_hardware
serial_hw_init()
```

As stated before, the 23k256 MUST be connected to the pic's SPI port, so let's setup the SPI port as well as the SPI library. We do not need to alias SPI hardware pins to another name. First include the library, then set the pin directions for the 2 data lines and the clock line:

```
-- setup spi
include spi_master_hw         -- includes the spi library
-- define spi inputs/outputs
```

```
pin_sdi_direction = input    -- spi input
pin_sdo_direction = output   -- spi output
pin_sck_direction = output   -- spi clock
```

Now that SPI data/clock pins are setup, the only pin left to define is the 23k256 chip select pin. If you have more then one device on the SPI bus, this chip select pin setup should be done at the beginning of your program instead. This chip select pin can be any digital output pin you choose to use.

```
-- setup chip select pin
ALIAS sram_23k256_chip_select             is pin_a2
ALIAS sram_23k256_chip_select_direction   is pin_a2_direction
-- initial settings
sram_23k256_chip_select_direction = output    -- chip select/slave select pin
sram_23k256_chip_select = high                -- start chip slect high (chip
 disabled)
--
```

Choose SPI mode and rate. 23k256 uses SPI mode 1,1

We will start with peed SPI_RATE_FOSC_16. (oscillator/16). These are the speeds that are available:

SPI_RATE_FOSC_4 -- Fastest

SPI_RATE_FOSC_16 -- Mid speed

SPI_RATE_FOSC_64 -- Slower

SPI_RATE_TMR -- Use timer

```
spi_init(SPI_MODE_11,SPI_RATE_FOSC_16) -- init spi, choose mode and speed
```

This line tells the PIC to set the SPI mode before each read & write. If you have multiple devices on the SPI bus using different modes, you will need to set this to TRUE

```
const byte SRAM_23K256_ALWAYS_SET_SPI_MODE = TRUE
```

Now we can finally include the library file, and initialize the chip:

```
include sram_23k256 -- setup Microchip 23k256 sram
sram_23k256_init(SRAM_23K256_SEQUENTIAL_MODE, SRAM_23K256_HOLD_DISABLE)  --
 init 23k256 in sequential mode
```

23k256 Usage

I'm going to go over this quickly since the code is simple.

Read & Write Byte

Write hex "AA" to address 1:

```
sram_23k256_write(1,0xAA)   -- write byte
```

Now read it back:

```
var byte data
sram_23k256_read(1, data) -- read byte
```

Byte Array

You can use the 23k256 as a large byte, word or dword array like this:

```
-- Example using 23k256 as a 32KByte array (at array address 2)
var byte data1
sram_23k256_byte[2] = 0xBB   -- set array byte 2 to value 0xBB
data1 = sram_23k256_byte[2]  -- read array byte 2, data2 should = 0xBB

-- Example using 23k256 as a 16K word array
var word data2
sram_23k256_word[3] = 0xEEFF     -- set array word 3 to value 0xEEFF
```

```
data2 = sram_23k256_word[3]      -- read array word 3, data2 should = 0xEEFF

-- Example using 23k256 as a 8K dword array
var dword data3
sram_23k256_dword[3] = 0xCCDDEEFF -- set array dword 3 to value 0xCCDDEEFF
data3 = sram_23k256_dword[3]      -- read array dword 3, data2 should =
 0xCCDDEEFF
```

If you are looking for a quick way to write lots of data, you can use the start_write, do_write and stop_write procedures. You should not use any other SPI devices on the same SPI bus between start_write() and stop_write()

sram_23k256_start_write (word in address) -- sets the address to write to

sram_23k256_do_write (byte in data) -- send the data

sram_23k256_stop_write() -- stops the write process

Here's an example:

```
-- Example fast write lots of data
sram_23k256_start_write (10)
for 1024 loop
  sram_23k256_do_write (0x11)
end loop
sram_23k256_stop_write()
```

This works the same for the read procedures:

sram_23k256_start_read (word in address) -- sets the address to read from

sram_23k256_do_read (byte out data) -- get the data

sram_23k256_stop_read() -- stop the read process

```
-- Example fast read lots of data
sram_23k256_start_read (10)
for 1024 loop
  sram_23k256_do_read (data1)
  serial_hw_write (data1)
end loop
sram_23k256_stop_read()
```

Your done, enjoy!

License

We, *Jallib Group*, want this book to be as *open* and *free* as possible. We decided to release it under *Creative Common Attribution-Noncommercial-Share Alike 3.0* license.

Basically (and repeating what's on Creative Common website), you are free:

- **to Share** - to copy, distribute, and transmit the work
- **to Remix** - to adapt the work

Under the following conditions:

- **Attribution** - You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- **Noncommercial** - You may not use this work for commercial purposes.
- **Share Alike** - If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

Full license lecal code can be read at: *http://creativecommons.org/licenses/by-nc-sa/3.0/legalcode*

This license applies to the book content itself, not on codes, libraries, examples, etc... you may find, or when it's explicitely stated work is released under another license. For instance, most work on Jallib is released under BSD and ZLIB license, not under this Creative Common license. In doubt, please ask on Jallib Group (*http://groups.google.com/group/jallib*)

Appendix

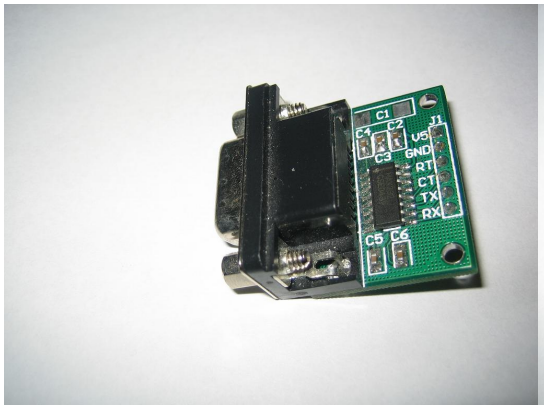
Materials, tools and other additional how-tos

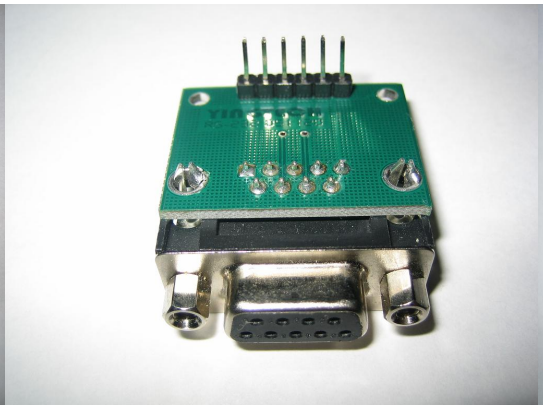
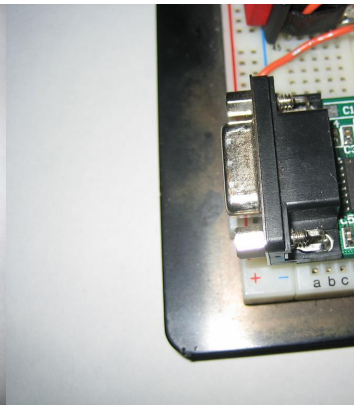
Building a serial port borad with the max232 device

Matthew Schinkel
Jallib Group

In this tutorial, we're going to build a serial port that can connect your PIC's TX and RX pins to your pc or other hardware using a max232 chip.

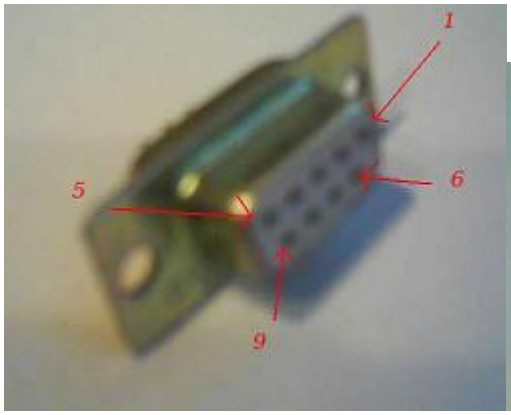
Many circuits will require some serial port communication, you may buy yourself a rs232 to TTL adapter off the net for as little as $10, or you can build one yourself. The max232 is a very popular chip. It converts your 5v circuit to the 12v required for serial communication to things like your PC. Many microcontrollers have RX and TX output pins. Here is a image of the max232 adapter I purchased. It has input pins for RX, TX, CT, RT as well as GND and 5v. The RX and TX pins can be directly connected to your PIC.



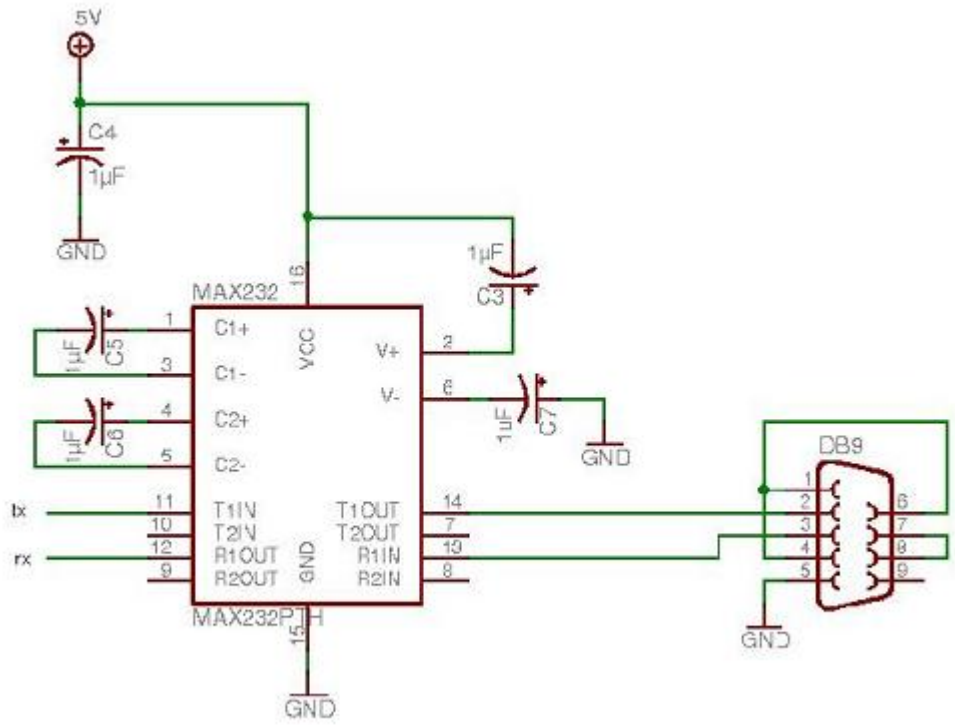
Now, lets build our own!

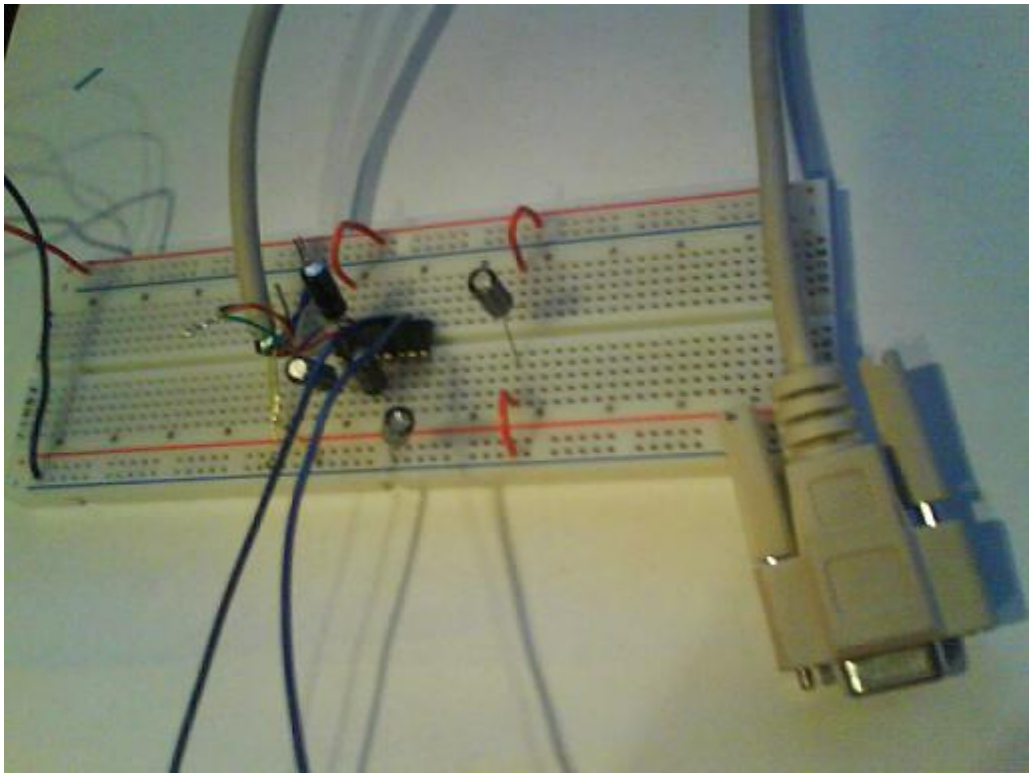
First get yourself a RS232 port, you can cut up one of your serial port cords, or buy a port from the store for a dollar or two.



I am going to use a cut serial port cord since it already has leads on it, and is long enough to reach my pc. Use your multi-meter to find the pin numbers, and touch up the wires with solder so they'll go into your breadboard easily.

Now build the circuit, As you can see, you will need the max232 chip from your local electronics store and a few 1uf capacitors.

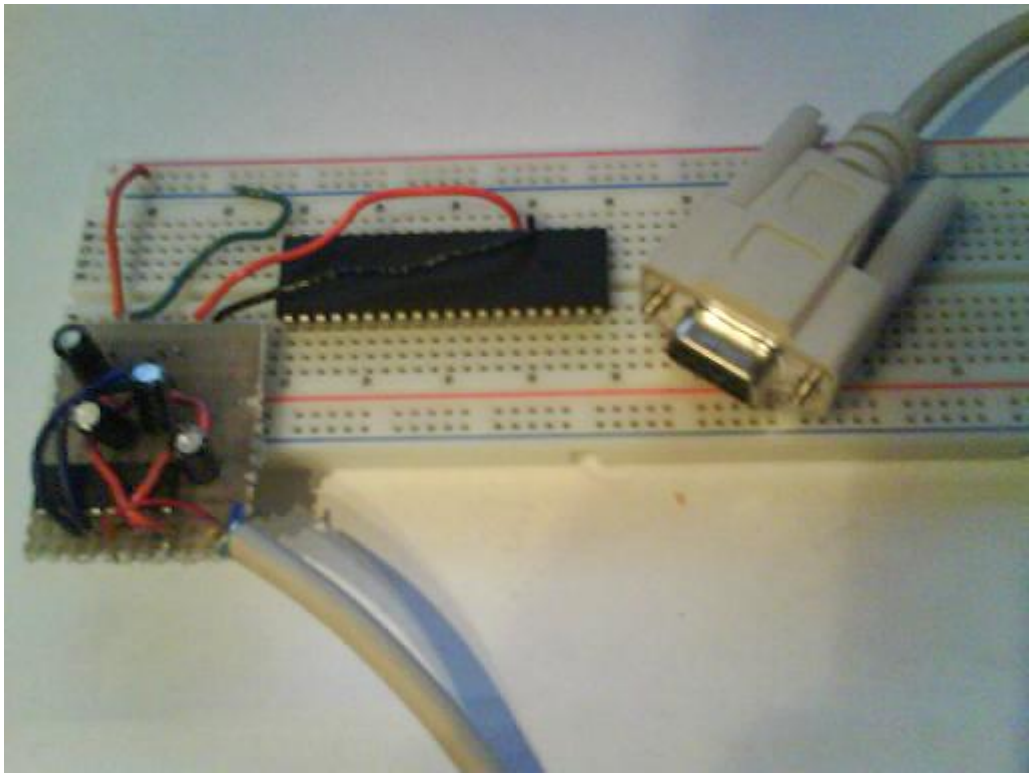




Great job, now connect the RX and TX pins to your circuit, and plug the rs232 port directly your pc, or to a usb-to-serial adapter, or even to a bluetooth-to-serial adapter for short range wireless.

I strongly suggest you make this on a PCB with pins that will plug to your breadboard. you'll use it a lot!

In this image, I did not complete my PIC circuit, but I think you get the idea:



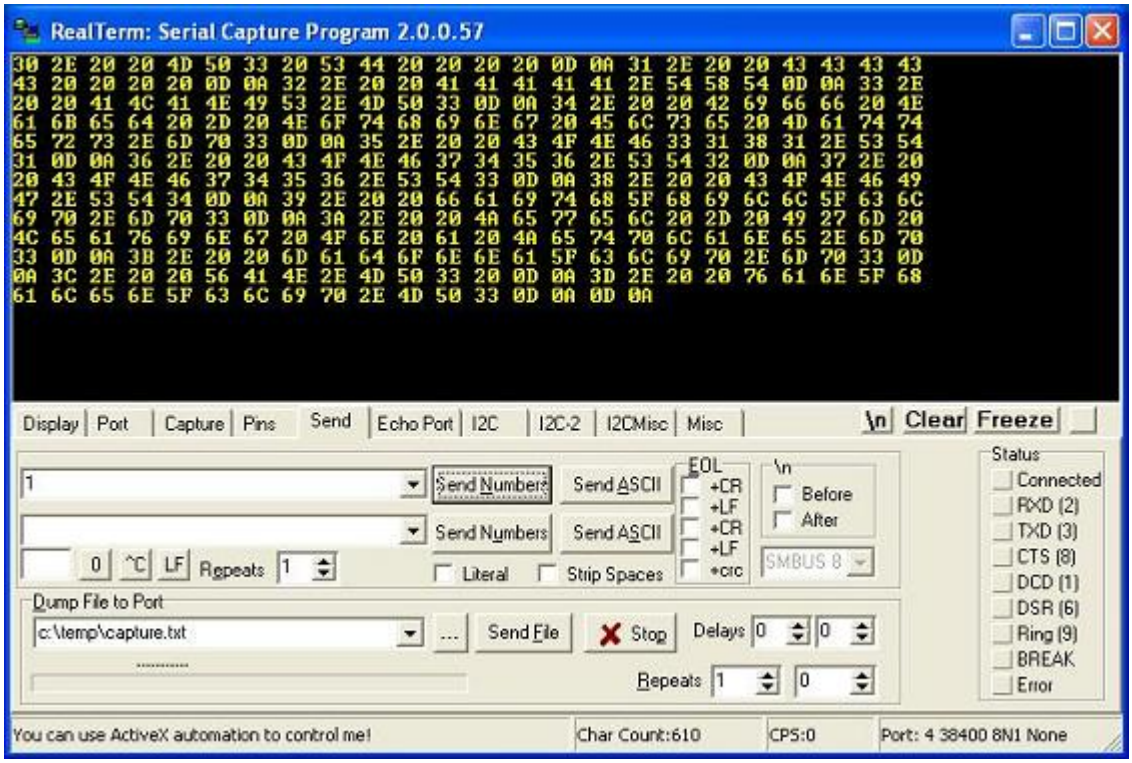
You can use serial_hardware lib or serial_software lib to transmit data to your pc, check for it in the other jallib projects. I suggest the software realterm for sending/receiving data to your PIC

Open Source REALTERM http://realterm.sourceforge.net/

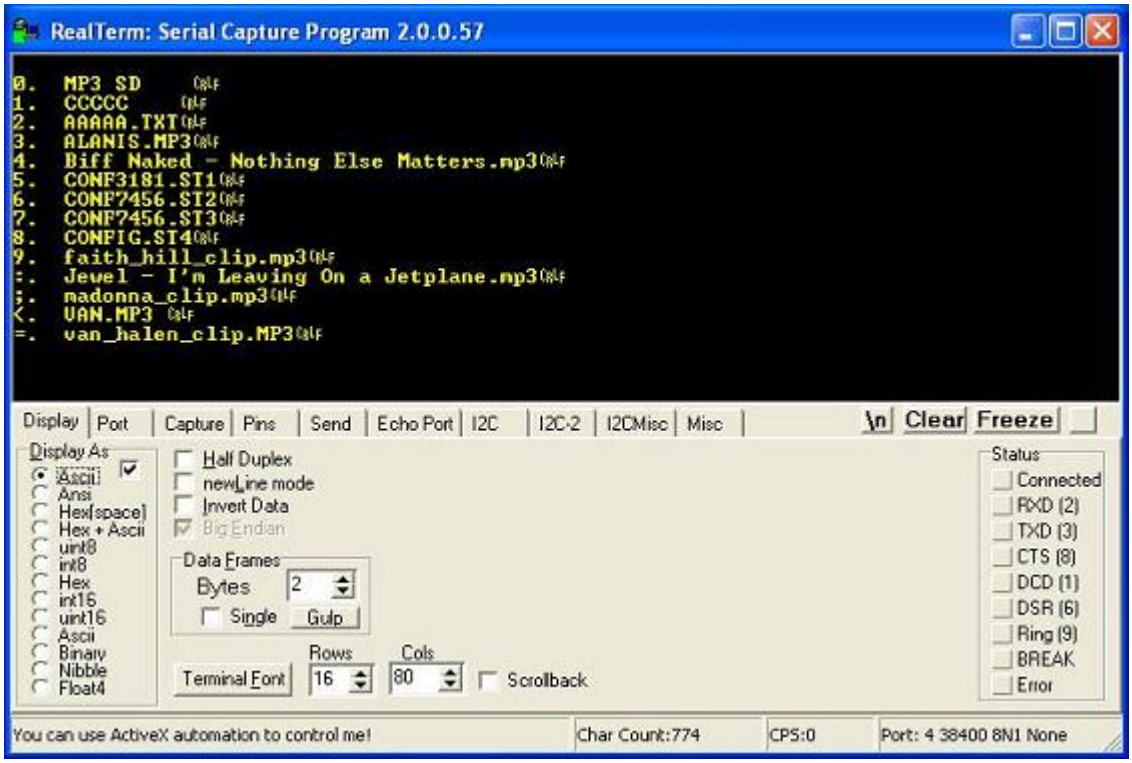
It can be downloaded for free from http://sourceforge.net/projects/realterm/files/

Open the software, click “Port”, choose your speed and port number and press “open”

Hex output



Ascii output



In Circuit Programming

Matthew Schinkel
Jallib Group

Intro to in-circuit programming & ICSP

What is ICSP?

ICSP stands for In-Circuit Serial Programming. More information can be found at *http://ww1.microchip.com/downloads/en/DeviceDoc/30277d.pdf*

Benefits of ICSP

1. You may program your PIC while it is in your breadboard circuit
2. You may program your PIC while it is on a soldered circuit board
3. You will save time programming so you can write more code faster
4. You can reset your circuit from your PC
5. You can program surface mount PIC's that are on soldered circuit board
6. You won't bend or break any pins
7. You won't damage your PIC by placing it in your breadboard wrong
8. With a remote desktop software like VNC, you can program your PIC from anywhere around the world.
9. I can program my PIC in my livingroom on my laptop while I watch tv with my wife! (I keep my mess in my office)

Intro to ICSP & in-circuit programming

When I got started in micro-controllers and JAL, I needed to choose a programmer. At the time, I did not know anything about choosing a programmer, so I just went on ebay and bought one that is able to program many different PIC's.

For years, I used this programmer by putting my 16f877 chip into it, programming it, and putting it into my circuit. I broke pins and wasted a lot of time. Little did I know, my programmer has an ICSP output for in-circuit programming. My programmer even says ICSP on it, but I did not know what ICSP is.

Eventually I got sick and tired of moving my micro-controller back and forth from the breadboard to the programmer, and I had herd some talk about ICSP. I found a ICSP circuit on the net, and I took a harder look at my programmer, it has 6 pins sticking up labeled ICSP. However, I did not know what pin was what, they where not marked well, and I could not find info about my programmer. One of the pins was marked pin 1 on the programmer. If you know your ICSP pinouts already, you may skip to the circuit diagram.

I searched for 6-pin ICSP in Google with no results, mostly I found 5 pin circuits. So, I took out my volt-meter and logic probe (and oscilloscope, although it is not needed) and measured the voltages off each pin while programming a chip and while not. I could see on the breadboard that pin 6 is connected to ground. Here's what I got:

PIN #	While Idle	While Programming
1	0v	12v
2	0v	0v
3	5v	Pulsing 0v to 5v (random)
3	0v	Pulsing 0v to 5v (square wave)
4	0v	5v
5	0v	0v

Get the pin names

The pin names for ICSP are VPP1, LOW, DATA, CLK, VCC, GND. So lets match them up:

0v pin 2 must be pin “GND”, I think this one is actually not connected

0v pin 6 must be pin “GND”

pin 1 & 5 seem to be programming enable pins, VPP1 and VCC

The two pulsing pins must be “CLK” and “DATA” (you may have to guess which is which if you don't have a oscilloscope.

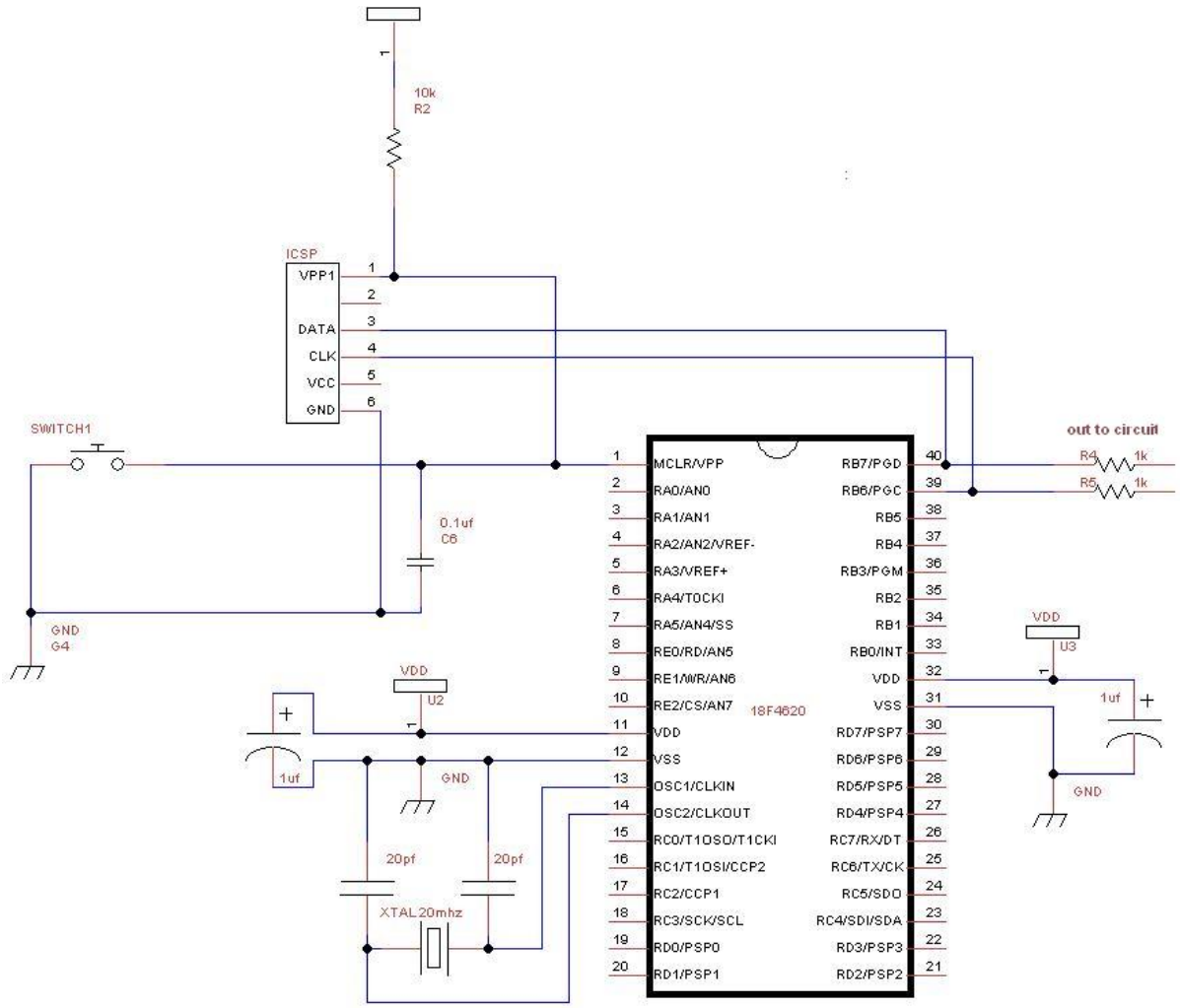
Lets make a new chart. I believe most ICSP ports have pins in this order:

PIN #	PIN NAME	While idle	While Programming
1	VPP1	0v	12v
2	Not Connected	0v	0v
3	DATA	5v	Pulsing 0v to 5v (random)
4	CLK	0v	Pulsing 0v to 5v (square wave)
5	VCC	0v	5v
6	GND	0v	0v

Build a circuit with ICSP

VCC can be connected to your PIC’s 5v supply for power-off programming. It does not work on my circuit because there is too much current drain. Do not connect both VCC directly to your power supply since there may be a voltage difference. In my circuit, I will not use the VCC pin, and I will program my chips while my circuit power supply is ON

GND must be connected to your circuits ground. Follow this circuit diagram:



Your done! Try to program your chip!

Changelog

Jallib Group
Jallib Group

Jallib Tutorial Book Changes & Updates

Table 3: Version 0.3 (Release Date: To Be Released)

Date	Comments
2010/01/27	Fixed I²C bus schematic and modified I²C titles
2010/01/21	Added ADC introduction, re-organized PWM tutorials and titles
2010/01/20	Better quality Images on Getting Started, serial board, blink a led tutorials.
2010/01/19	Added serial & rs-232 tutorial
2010/01/15	New ICSP Schematic

Table 4: Version 0.2 (Release date: 2009/12/30)

Date	Comments
2009/12/06	Added SD Card tutorial
2009/12/06	Added PATA Hard Disk tutorial
2009/12/06	Added ICSP tutorial

Table 5: Version 0.1 (Release date: 2009/11/22)

Date	Comments
2009/11/22	Initial Release