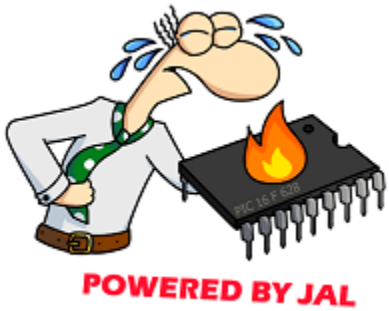
The Jallib Tutorial Book



Have fun with PIC microcontrollers, JAL v2 and Jallib

Contents

Chapter 1: Back to basics..... 5
Installation..... 5
Getting Started..... 6
Blink A Led (Your First Project)..... 13
Serial Port and RS-232 for communication..... 21

Chapter 2: PIC internals..... 33
ADC - Analog-to-Digital Conversion..... 33
I²C (Part 1) - Building an I²C slave + Theory..... 38
I²C (Part 2) - Setting up and checking an I²C bus..... 39
I²C (Part 3) - Implementing an I²C Slave..... 42
PWM Intro - Pulse Width Modulation..... 48
PWM (Part 1) - Dimming a led with PWM..... 49
PWM (Part 2) - Sound and Frequency with Piezo Buzzer..... 52
SPI Introduction..... 55
USB (Part 1) - Introduction..... 58
USB (Part 2) - The PIC as a serial port..... 59

Chapter 3: Experimenting with external parts..... 63
Hard Disks - IDE/PATA..... 63
IR Ranger with Sharp GP2D02..... 76
LCD Display - HD44780-compatible..... 82
Memory with 23k256 sram..... 89
RC Servo Control & RC Motor Speed Control..... 94
SD Memory Cards..... 101
DFPlayer Mini..... 110

Chapter 4: PIC software..... 115
Print & Format..... 115
FAT32 File System..... 118
Large Array..... 129

Appendix A: Appendix..... 131
Materials, tools and other additional how-tos..... 131
Building a serial port board with the max232 device..... 131
In Circuit Programming..... 135
Using Visual Studio Code with JAL..... 138
Making a Tutorial..... 145
Jallib Style Guide..... 146
Making a Library..... 153
Changing the contents of a Jallib release..... 155
Miscellaneous..... 156
Changelog..... 156
License..... 158

Chapter 1

Back to basics...

This is a brief introduction to exploring the basic tutorials. As a beginner, we recommend that you should experiment with and fully understand these first steps before going any further. If you're a more advanced user, these tutorials may help you with the testing of new chips, or... when things go wrong and you can't figure out why, by guiding you "back to basics".

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

Installation

JALv2 & Jallib installation guide

Getting Jallib

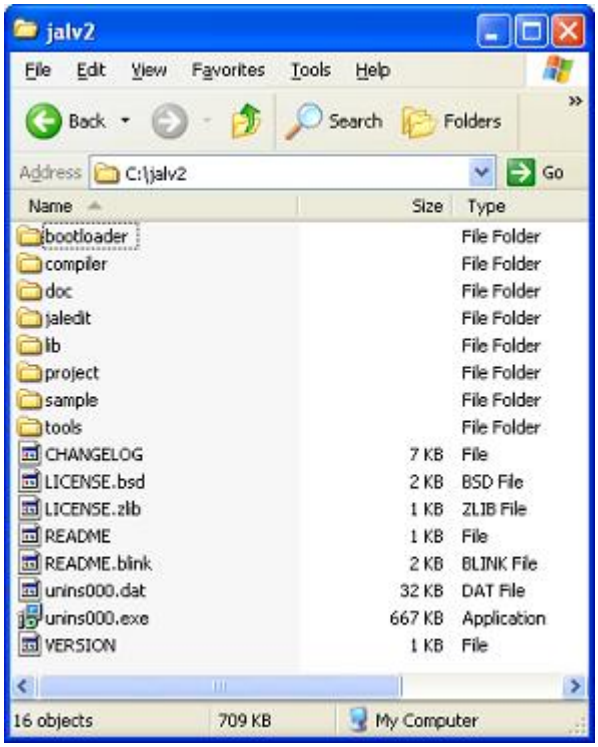
The Jallib repository is maintained at http://justanotherlanguage.org/downloads. It's safer to ignore the "*Automated weekly build...*" files in this download directory (these, as the prefix implies, are updates to the repository automatically compiled by the server and are probably untested). Look instead for the latest version of the release file which has the annotation "*Don't know which to choose? Take this one!*". These are the tested releases and, while we can't guarantee that there won't be any bugs at all, will provide a reasonably trouble-free installation.

The release files are all ZIP archives, which can be easily unpacked on most modern operating systems (Windows/Linux/MacOSX). Jallib releases also all come complete with a pre-compiled binary version of the JALv2 compiler for both Windows and Linux in the top-level *compiler* directory and a full set of documentation (including a copy of this tutorial) in the top-level *doc* directory, so there's nothing to stop you getting started straight away.

Windows Install:

1. Download the latest stable Jallib release installer executable from http://justanotherlanguage.org/downloads, This will install JALv2 + JalEdit
2. Update your installation (very important) - Download jallib-pack or jallib-pack-bee from http://justanotherlanguage.org/downloads, 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 see something like this under Windows:



Linux Install

Note: All instances of filenames or paths within square brackets below (eg:- `[filename.tar.gz]`) are for illustration only. You will need to change these examples to suit your specific installation. Do **not** just copy and paste the examples!

1. Go to http://justanotherlanguage.org/downloads and select the link for the latest, stable version of Jallib (see:- Getting Jallib, above)
2. Change directory ("`cd [/target/directory]`") to the location where you intend to install JALv2
3. Download the zip-file of the stable Jallib release.
4. unzip the package with: `$ unzip [filename.zip]`

You should see something like this under Linux:

```
bash-3.2$ ls
CHANGELOG    LICENSE.zlib  README.blink  compiler  lib      sample
LICENSE.bsd  README        VERSION       doc       project
bash-3.2$
```

Note: Jaledit also runs under Wine on Linux

Getting Started

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!

If you already know about how to setup and use microcontrollers, I suggest you start with the "Blink a led" tutorial, then read the "Jallib Starters Guide". The starters guide will give you more detailed information about the language JalV2 and Jallib. It also has some more advanced technical information and examples. It can be download from: http://justanotherlanguage.org/downloads

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!

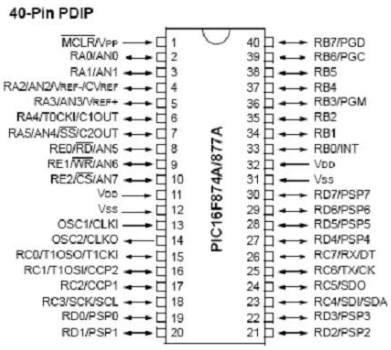
The programming language JALv2 is FREE, other languages will cost you somewhere between $200 and $2000.

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 for more memory, for example, you can run a SD Card, but you cannot use FAT32. 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. I 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.

You will notice that the better 18F series chips are actually cheaper then the outdated 16F chips.

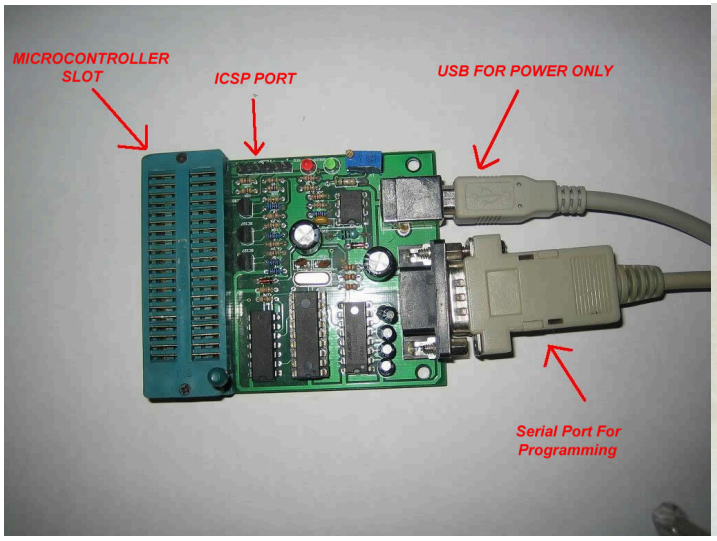
Here is a price chart from the manufacturers sales website (prices from July 2023):

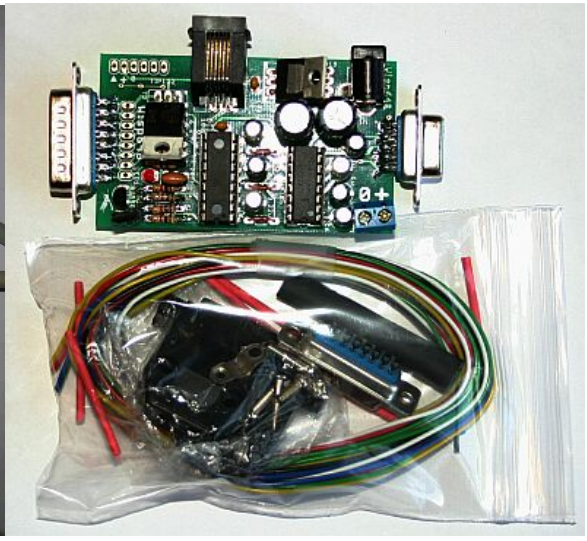
PIC	**Price USD**
12F675	$1.74
16F84	$746
16F877	$8.33
16F877A	$7.62
16F1455	$2.44
18F452	$7.60
18F2550	$788
18F4550	$8.19
18F4620	$9.56

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:

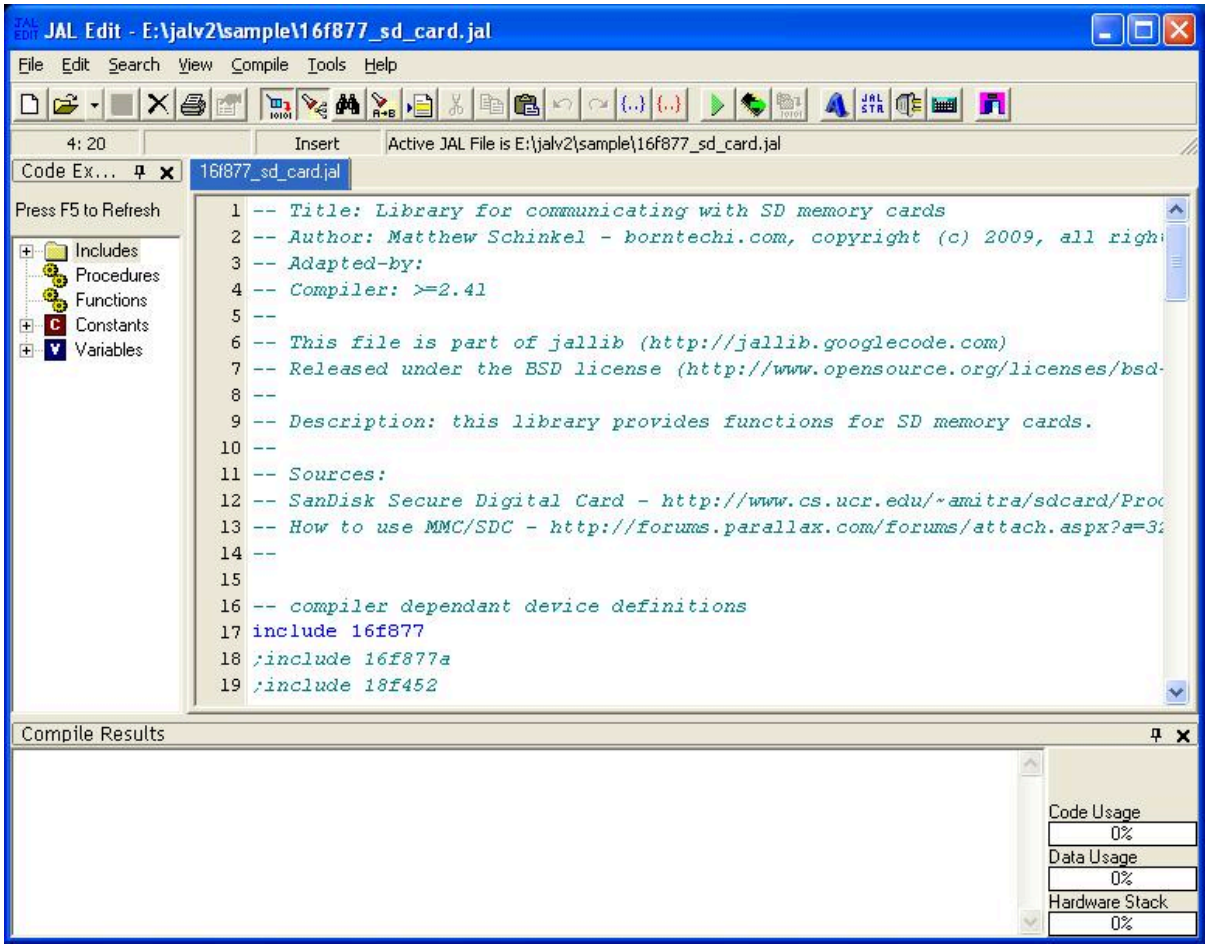






What editor should I use?

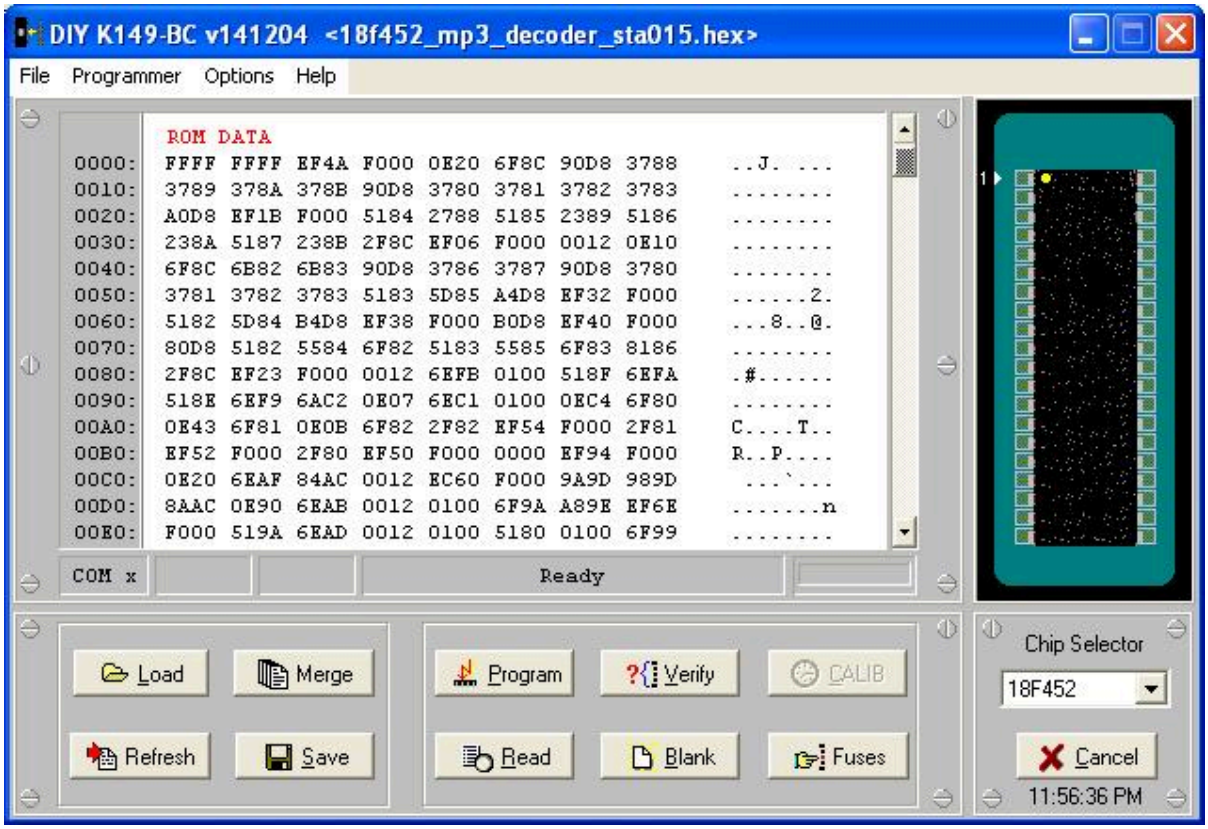
Any text editor is fine, but if you are on a windows machine. We suggest the free editor “JAL Edit” which will highlight & color important text as well as compile your JAL program to a hex file for burning to your microcontroller. If you followed our installation guide, you will already have this editor.



If you want to use another editor that supports the JAL syntax or you are using another operating system than Windows, you can make use of `Visual Studio Code` for your JAL development.

What programming/burning software should I use?

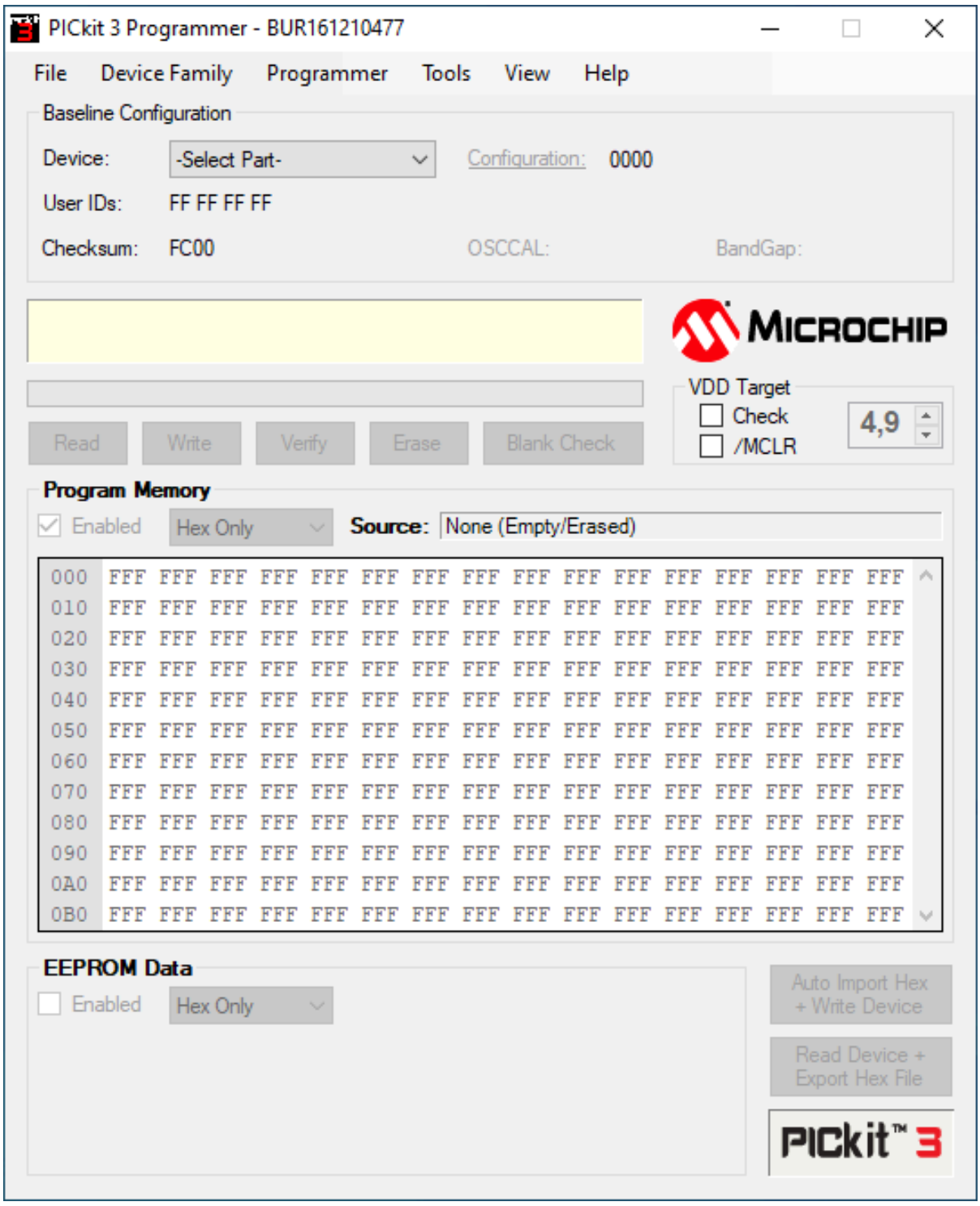
Did your programmer come with software? There are many to choose from so use whatever you prefer. I use "Micropro" from http://www.ozitronics.com/micropro.html. It's a free, open source software for programming a wide range of PIC's. However, it will most likely not support your programmer. I suggest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



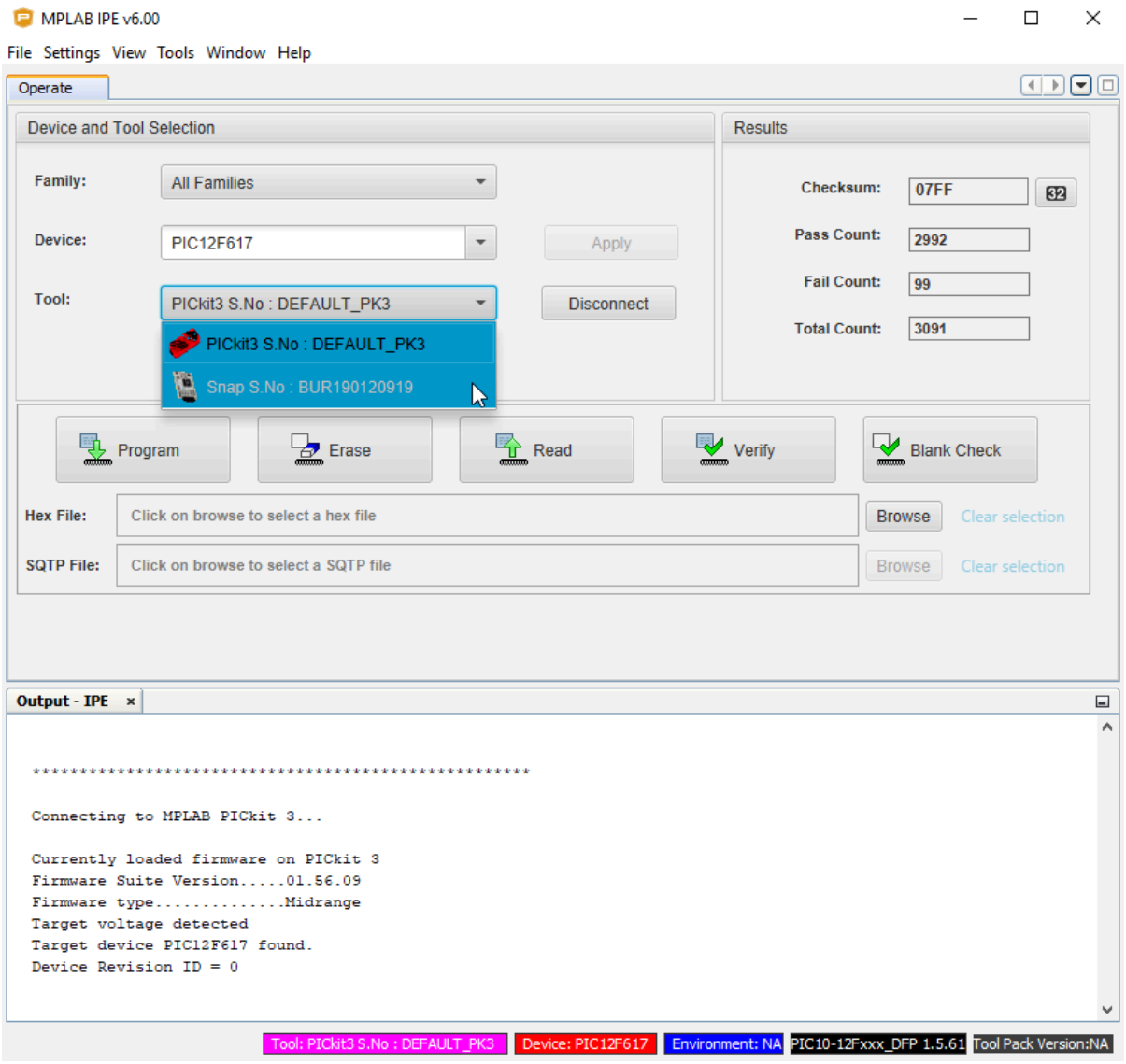
When using a PICkit2 or a PICKit3, you can use the standalone PICKit2 or PICKit3 software from Microchip. This software can be downloaded from the `Microchip archive site` (scroll down to the bottom of the site) but this software is no longer supported.

Luckily there are still people who spend effort in supporting newer PICs for this software. In order to upgrade your PICKit2 or PICKit3 software so it supports more PICs, you can download the PICkitMinus software. This software supports Windows, with the same UI as the PICkit software from Microchip, and for Windows and Linux you can use the command line version. The software supports newer PICs and is open source. You can download it from the GitHub.

The interface of this standalone software for the PICKit3 looks like this.



An alternative when using a PICkit2, a PICKit3, a PICKit4 or a SNAP programmer from Microchip, is to use the Integrated Programming Environment (IPE) from Microchip. This environment is part of `MPLABX` and can be downloaded for free. Note that not all programmers are capable of programming all PICs except for the PICKit4. The PICKitMinus software solves that problem. The SNAP cannot program all older PICs since the SNAP programmer only supports Low Voltage Programming (LVP). All these programmers make use of the `In Circuit Programming` feature of the PIC. The following image shows a screenshot of the IPE environment where two programmers are connected to the computer, a PICKit3 and a SNAP.



OK, enough of this boring stuff, lets build something! Start with the Blink A Led Tutorial.

Blink A Led (Your First Project)

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.

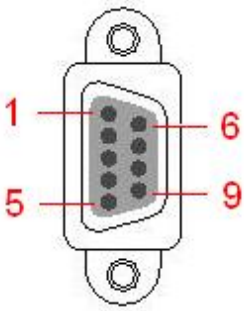
Setup your workspace

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.

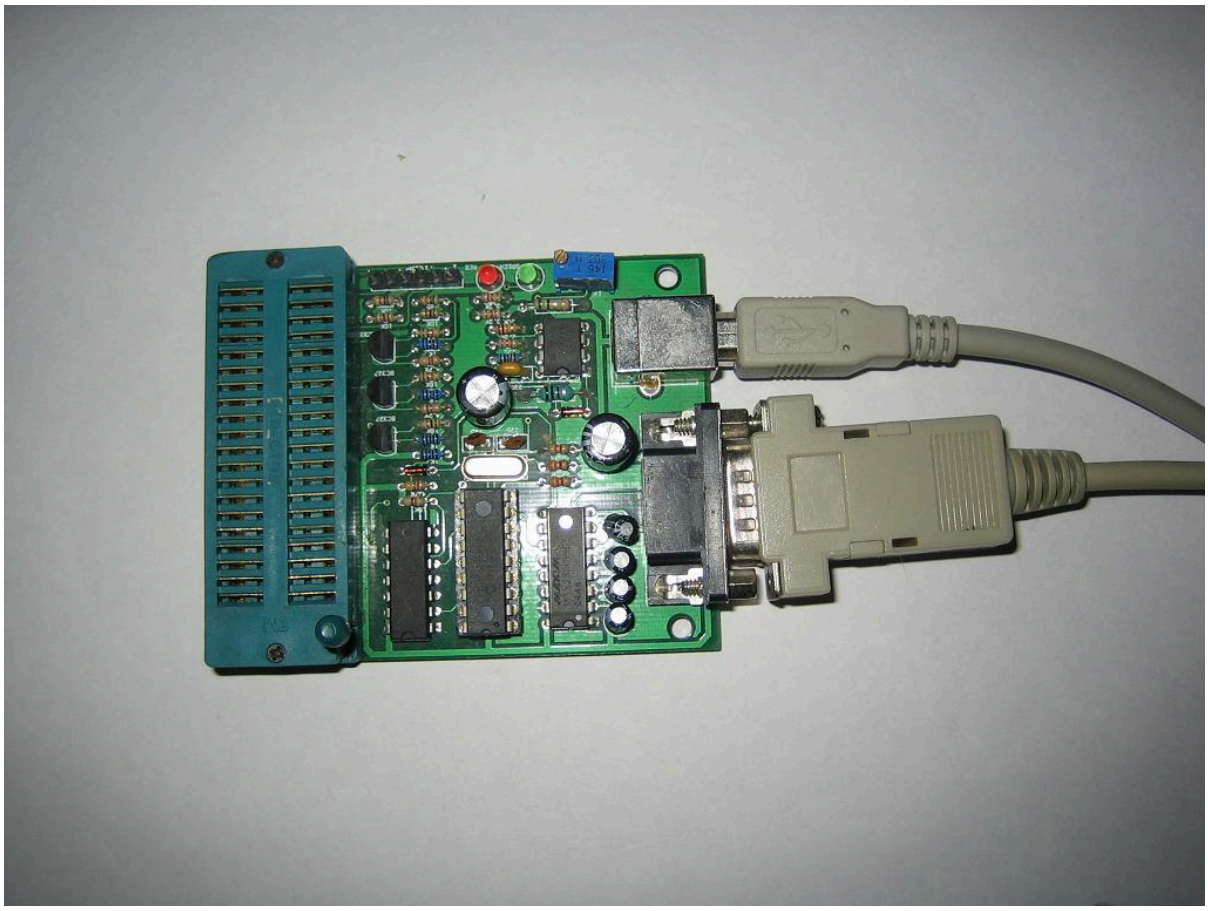




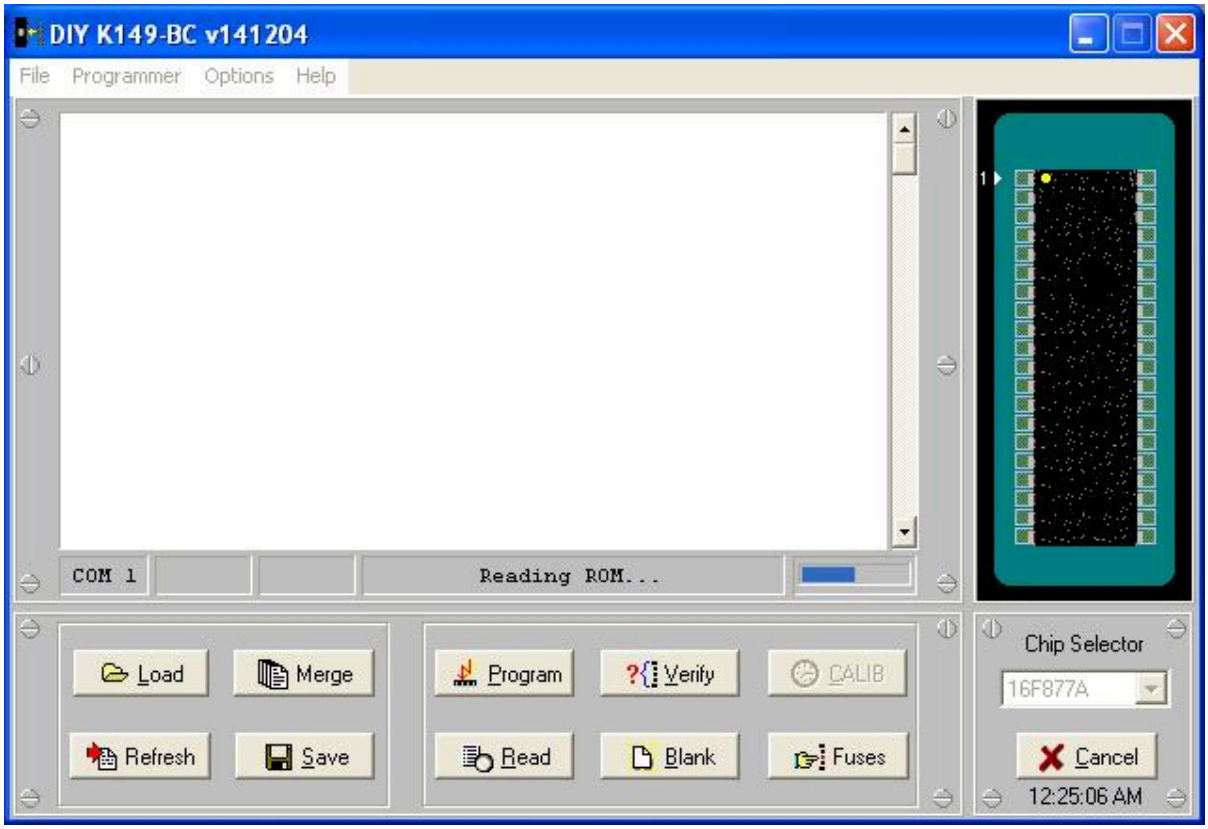


Get out your PIC microcontroller (we will now refer to it as a PIC). You can use PIC's 16f877, 16f877A, 18F2550 , 18F452 or 18F4550 for this project since the port pin outs are the same for all of them. I will use 16f877A for this blink a led project.

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"

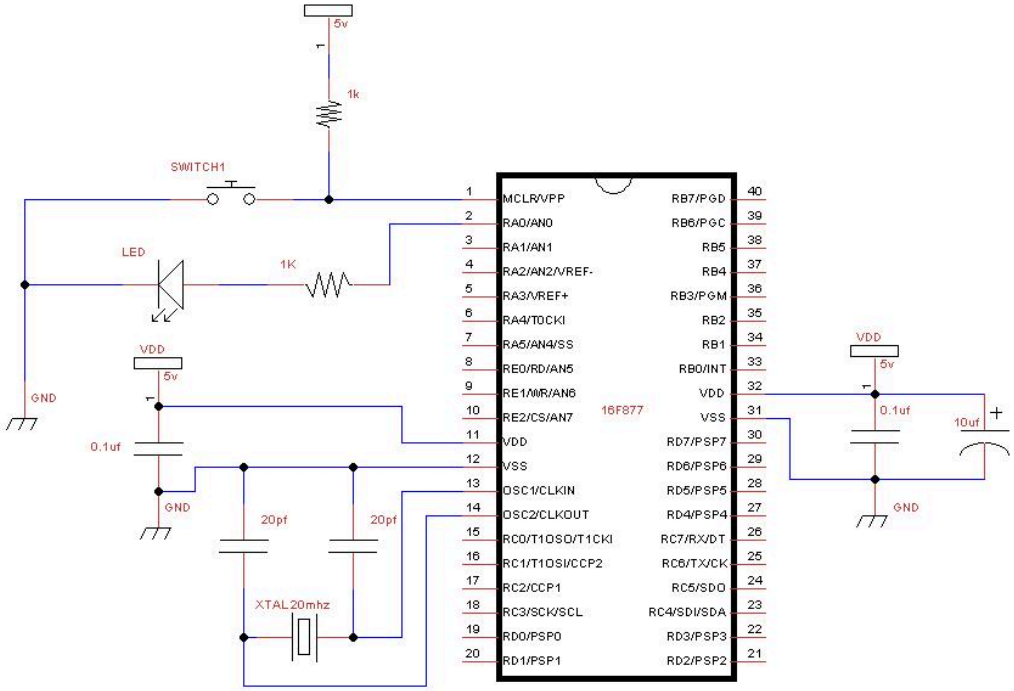


If you are using Micropro, click on "file" -> "port", and "file" -> "programmer" -> (your programmer type). If you do not know the programmer type, you will have to guess until Micropro says something like "K149-BC board connected", Put your PIC in your programmer and choose your PIC type from the "Chip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

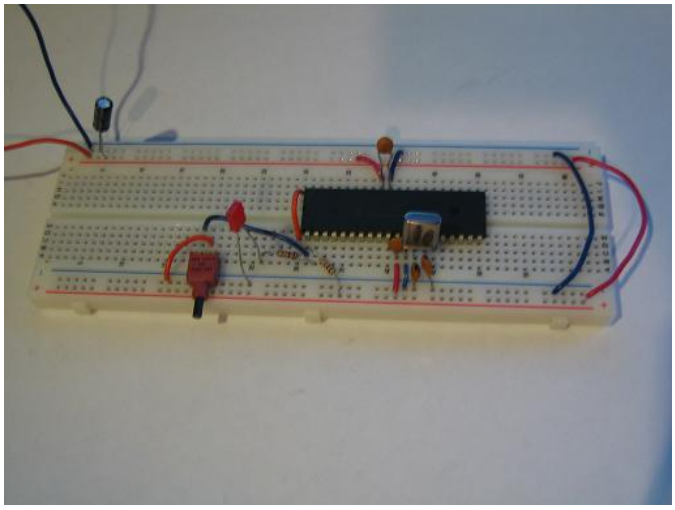


Build your circuit

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



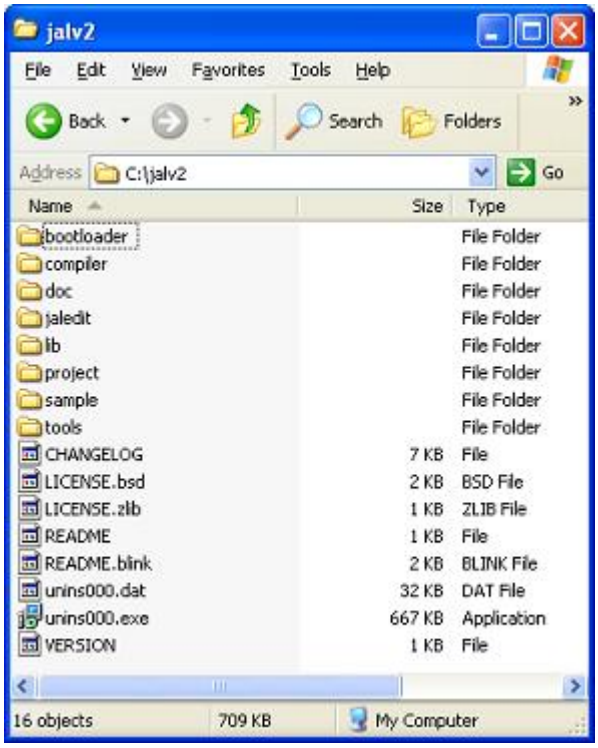
And here's what it looks like. Notice the additional orange wire to the left of my PIC, this ensures that I always put my PIC in the correct position after programming. Do not connect your power 5v supply till your circuit is complete and checked over at least twice. You will burn your PIC if power is 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 for Windows and Linux to convert your JAL code to microcontroller hex code
- **JALEdIt** – JAL text editor where you will write your code. Note that you can also use Visual Studio Code for writing your code.
- **lib** – Device files and libraries to make things work

- **sample** – Working examples. For each PIC for which there is a device file there is also a blink-a-led sample file in the **blink** sub folder
- **project** – Some JALproject to play with
- **doc** JAL documentation.

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 in the jalv2.pdf file in directory 'compiler' of your Jallib installation.

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 and Rob Jansen'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, Rob Jansen, Copyright (c) 2008..2022 all rights reserved.
-- Adapted-by:
-- Compiler:2.5r5
--
-- This file is part of jallib (https://github.com/jallib/jallib)
-- Released under the ZLIB license (http://www.opensource.org/licenses/zlib-license.html)
--
-- Description:
--    Simple blink-a-led program for Microchip pic16f877a
--    using an external crystal or resonator.
--
-- Sources:
--
-- Notes:
--  - Creation date/time: Sat Jan  8 17:03:04 2022
--  - This file is generated by 'blink-a-led.py' script! Do not change!
--
-- -------------------------------------------------------
```

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

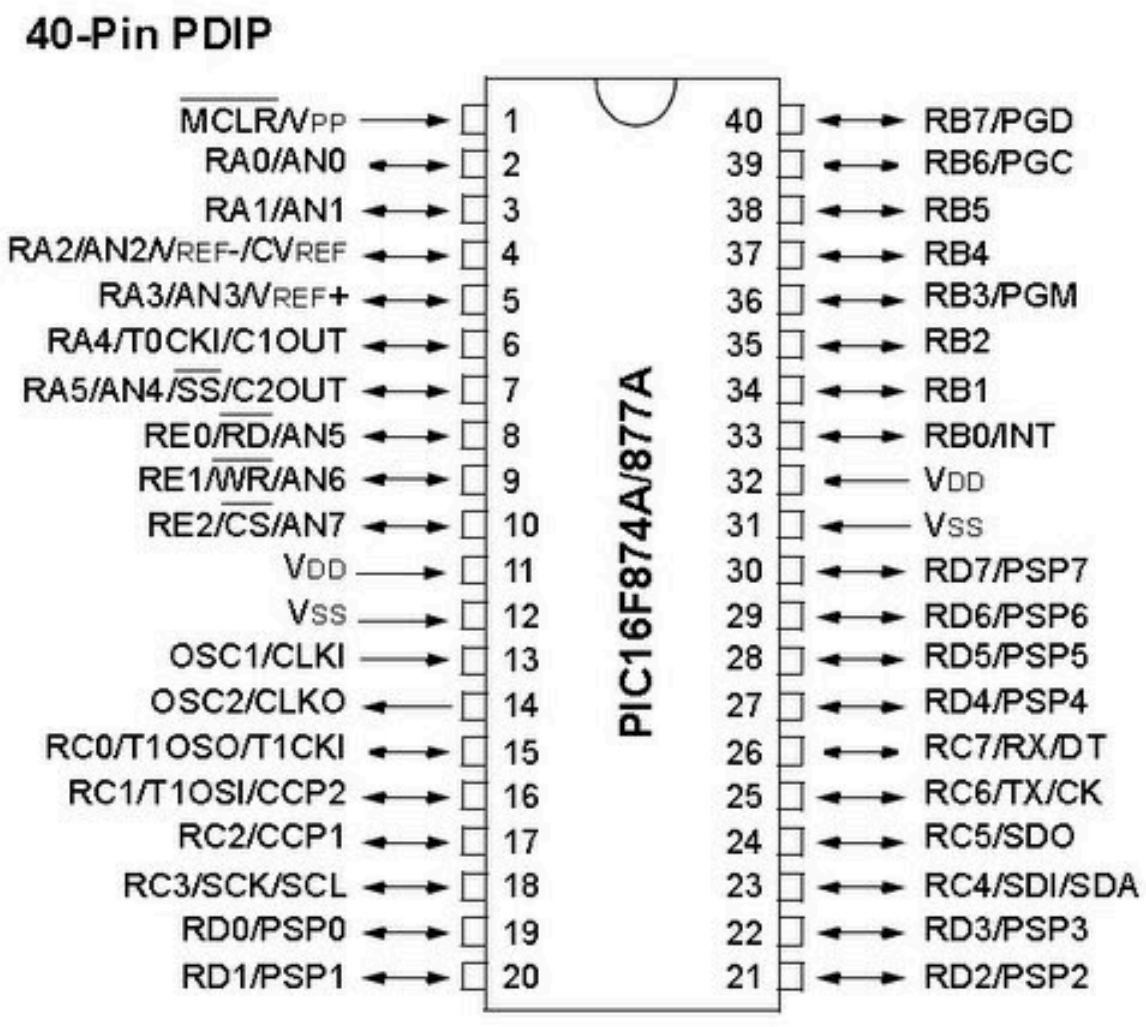
The following code sets some of the PIC's internal settings, called fuses. A OSC setting of HS tells the PIC there is an external clock or crystal oscillator source. You must disable analog pins with enable_digital_io() , you don't need to worry about the others.

```
pragma target OSC      HS                       -- crystal or resonator
pragma target WDT      DISABLED                 -- watchdog
pragma target DEBUG    DISABLED                 -- no debugging
pragma target BROWNOUT DISABLED                 -- no brownout reset
pragma target LVP      DISABLED                  -- no low voltage programming
--
-- The configuration bit settings above are only a selection, sufficient
-- for this program. Other programs may need more or different settings.
--
enable_digital_io()                -- make all pins digital I/O
```

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 https://www.microchip.com/en-us/product/PIC16F877A , 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	2	3	19	19		TTL	PORTA is a bidirectional I/O port.
RA0					I/O		Digital I/O.
AN0					I		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(250_000)
```

turn the led off again

```
   led = OFF
```

and have another delay before turning it back on again

```
   _usec_delay(250_000)
```

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, Rob Jansen, Copyright (c) 2008..2022 all rights reserved.
-- Adapted-by:
--
-- Compiler:2.5r5
--
-- This file is part of jallib (https://github.com/jallib/jallib)
-- Released under the ZLIB license (http://www.opensource.org/licenses/zlib-license.html)
--
-- Description:
--    Simple blink-a-led program for Microchip pic16f877a
--    using an external crystal or resonator.
--
-- Sources:
--
-- Notes:
--  - Creation date/time: Sat Jan  8 17:03:04 2022
--
-- -----------------------------------------------------
--
include 16f877a                     -- target PICmicro
--
-- This program assumes that a 20 MHz resonator or crystal
-- is connected to pins OSC1 and OSC2.
pragma target clock 20_000_000      -- oscillator frequency
--
pragma target OSC      HS                       -- crystal or resonator
pragma target WDT      DISABLED                 -- watchdog
pragma target DEBUG    DISABLED                 -- no debugging
pragma target BROWNOUT DISABLED                 -- no brownout reset
pragma target LVP      DISBLED                  -- no low voltage programming
--
-- The configuration bit settings above are only a selection, sufficient
-- for this program. Other programs may need more or different settings.
--
enable_digital_io()                 -- make all pins digital I/O
--
-- A low current (2 mA) led with 2.2K series resistor is recommended
-- since the chosen pin may not be able to drive an ordinary 20mA led.
--
alias  led       is pin_A0          -- alias for pin with LED
--
pin_A0_direction = OUTPUT
--
forever loop
   led = ON
   _usec_delay(250_000)
   led = OFF
```

```
   _usec_delay(250_000)
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 Visual Studio Code you can use ctrl-shift-b to compile your code.

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 for Windows and for Linux assuming you are running this from the directory where you have your JAL file:

- Windows: C:\jalv2\compiler\jalv2.exe blink_a_led.jal -s C:\jalv2\lib -no-variable-reuse
- Linux: ~/jalv2/compiler/bin/jalv2-x86-64 blink_a_led.jal -s ~/jalv2/lib -no-variable-reuse

The option -no-variable-reuse will use more PIC memory, but will compile faster but since computer are quite fast you and PIC memory is scarce you can leave out this option. In the documentation of the JAL compiler you find more compiler options that you can use.

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. When using a PICKit programmer using the PICkitMinus software you can leave your PIC in your circuit, import the blink_a_led.hex file and program your PIC while it is in the circuit.

Let's Try It

If you removed your PIC for programming 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

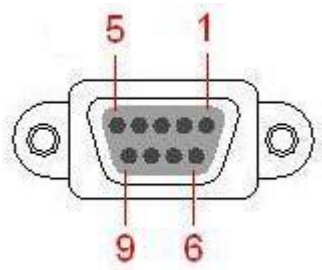
I strongly suggest you do this tutorial next: Serial Port & RS-232 for communication.

Serial Port and RS-232 for communication

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

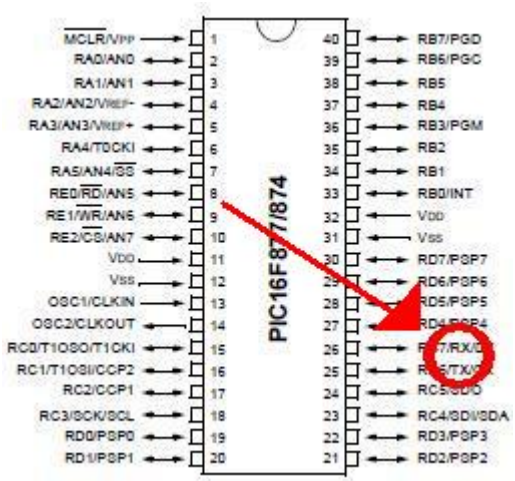
6. The port number on your PC

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.

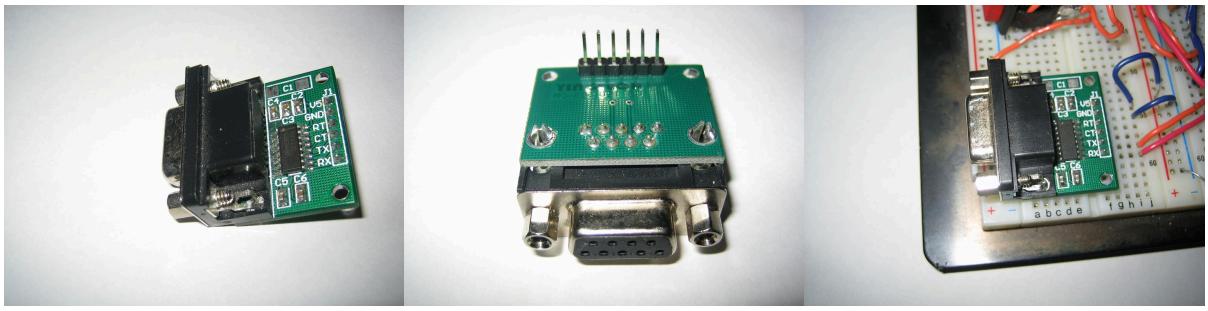
What do I need?

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. See Here for more information. A serial port board is needed for voltage conversion. Serial ports output voltages up to 12v and go lower than 0v.

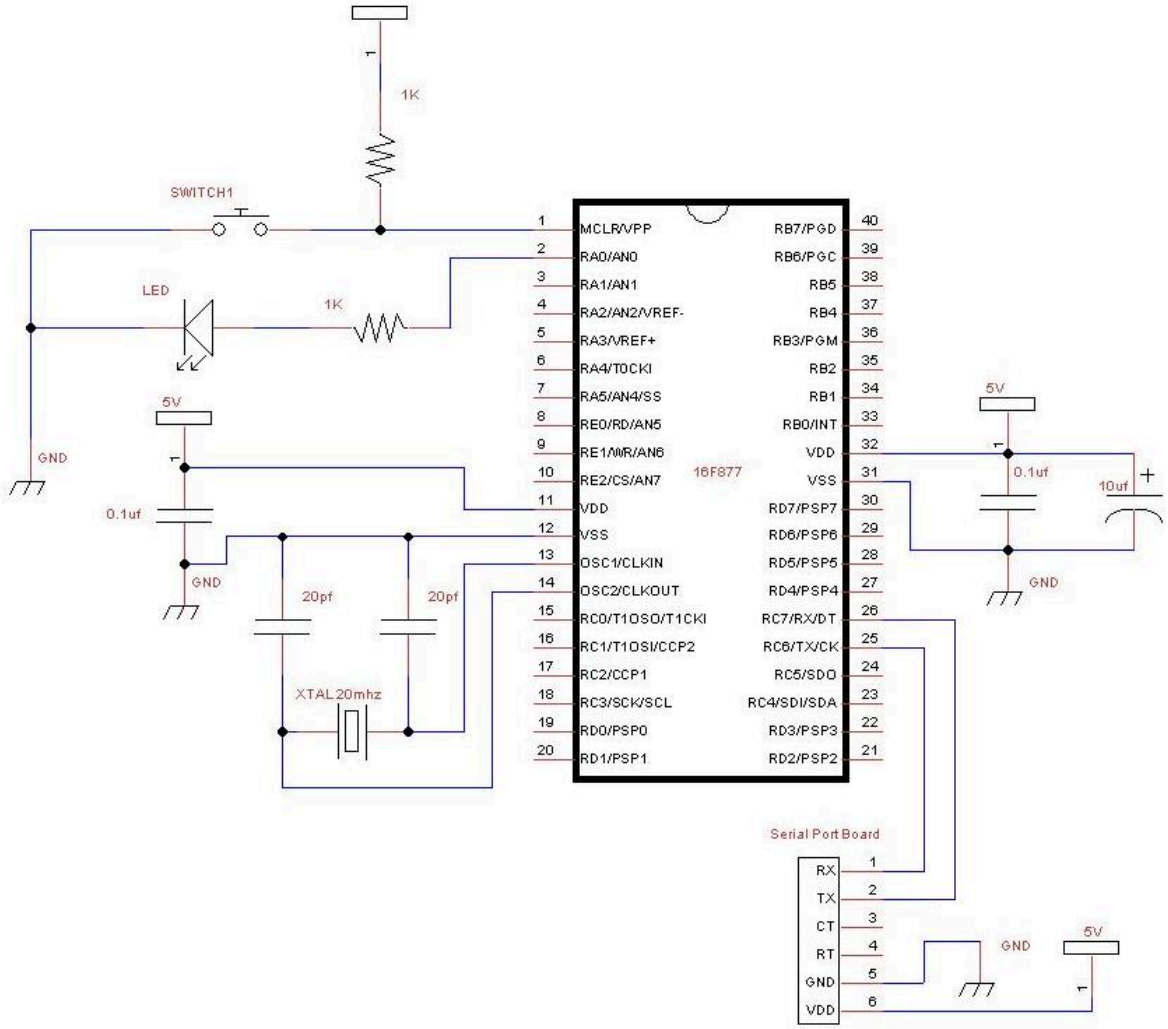


3. A regular RS-232 Cable (make sure it is not a null modem cable, they look the same). You can check what type of cable you have with your multimeter. Put your multimeter on each pin of your cable starting with pin 1. Check for a zero ohm reading. This will check that the pins are the same at both ends.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

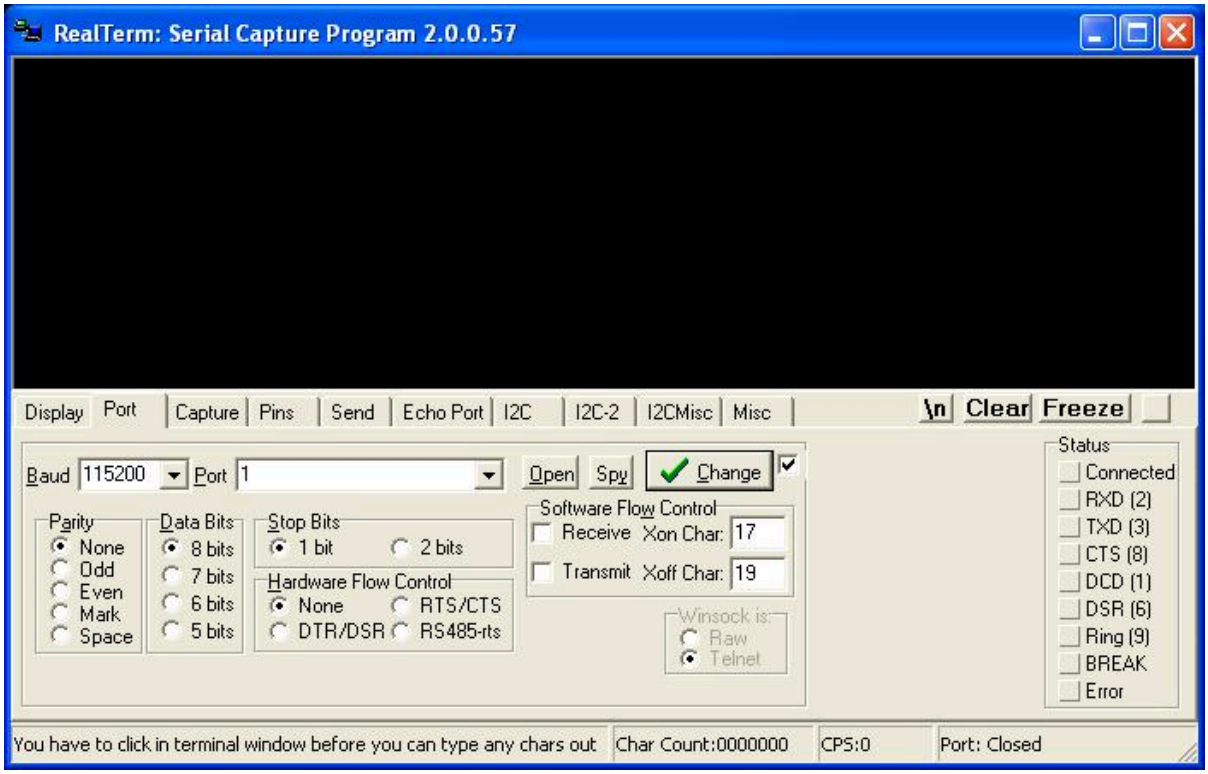
1. The Parity = no parity

2. The number of data bits = 8

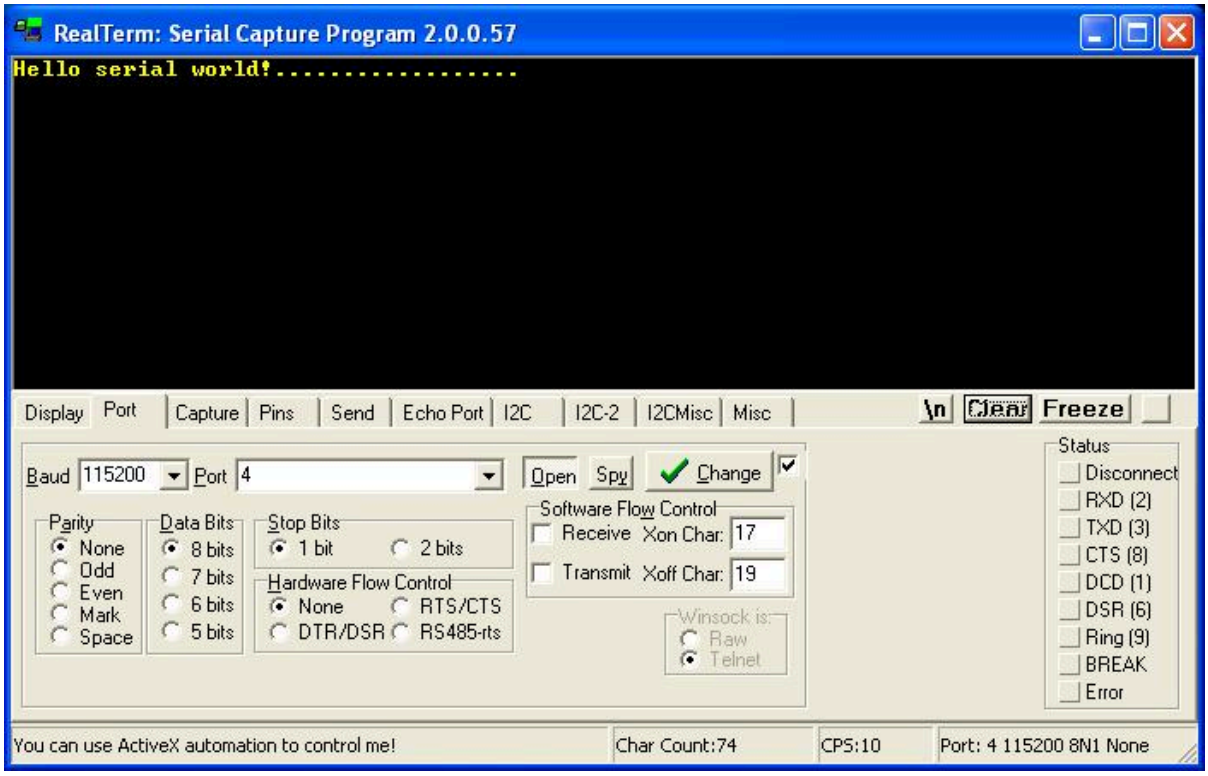
3. The number of stop bits = 1

4. The data transmission speed = 115200

5. The port number on your PC

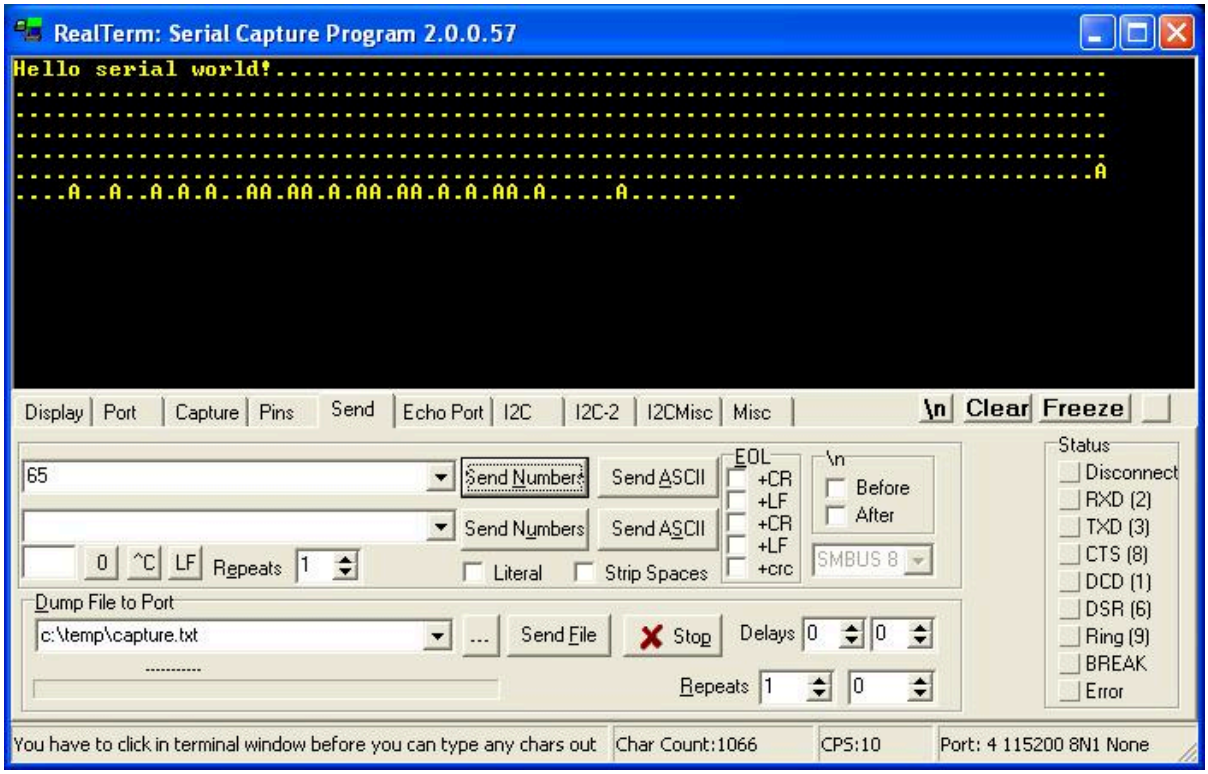


Now press "Open" in RealTerm and turn on your circuit. If you now see "Hello serial world....." showing in within RealTerm on your PC, you are able to receive data.



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.

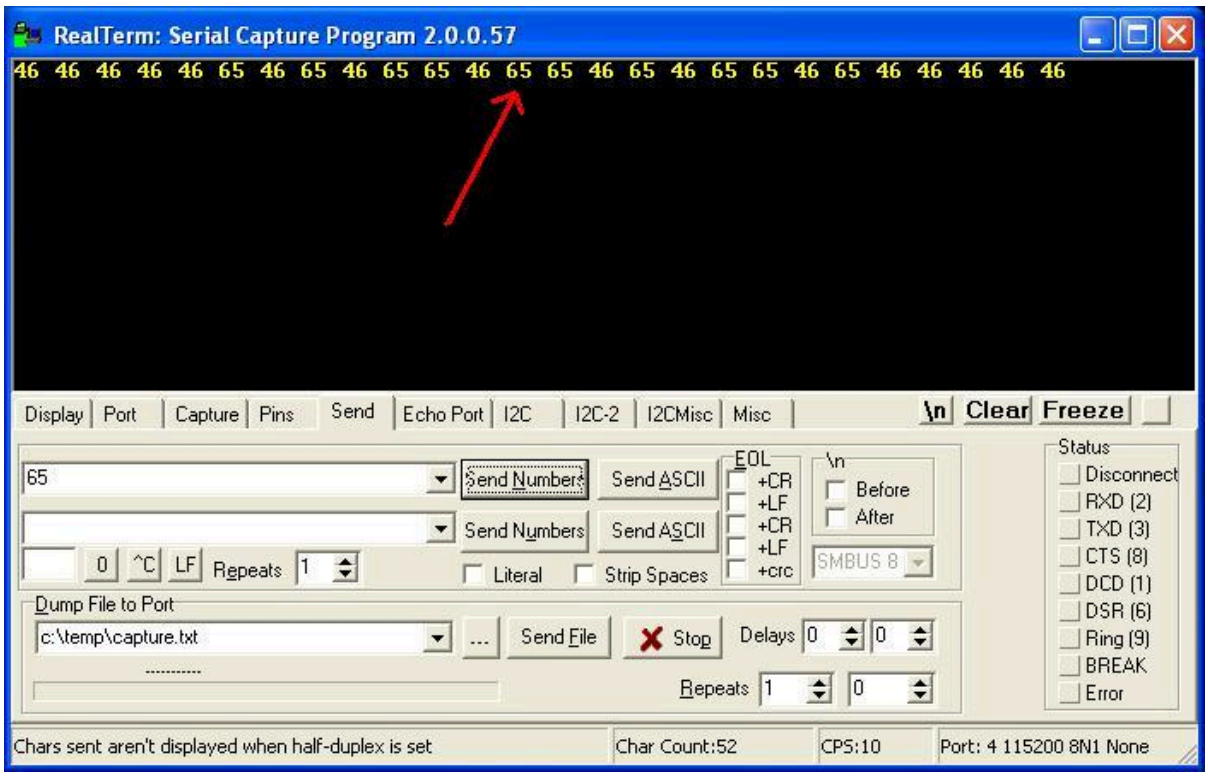


Now please change your RealTerm settings to receive decimal numbers by clicking on the "Display" tab, and choose "int8" under "Display As" at the left side. You will now continuously see the number "46" come in, and try sending the number "65" again. You will get the same number back on your screen.

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

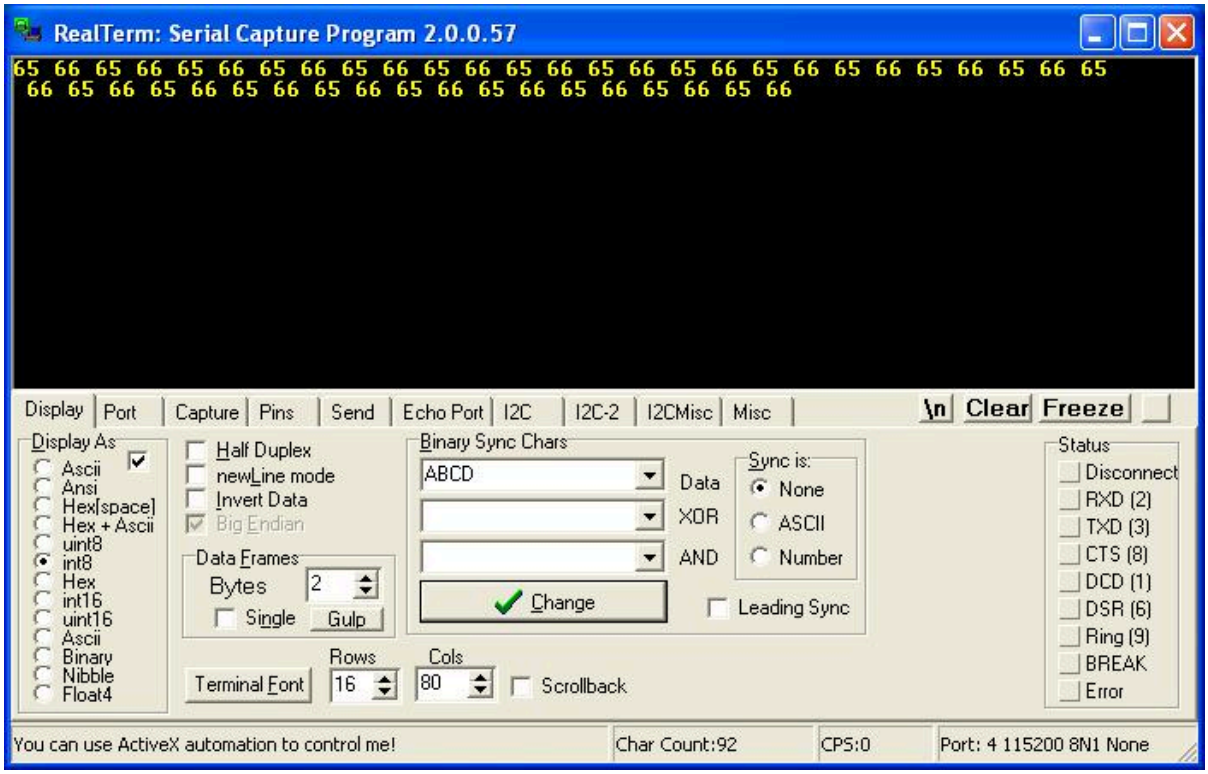
enable_digital_io()                -- disable analog I/O (if any)

-- 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 described in the document jalv2.pdf in the 'compiler' directory of your Jallib installation.

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 communication 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!

Your Next Step

Now that you know how serial works, I suggest you take a look at the print and format libraries which will help you format numbers and strings in an easy & readable. Check out this tutorial:

Print & Format on page 115

Chapter 2

PIC internals

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.

ADC - Analog-to-Digital Conversion

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 continuous 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[1]
- Some PICs can have **two external voltage references** (VRef+ and VRef-), only **one voltage reference** (Vref+ or Vref-) and some **can't handle external 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

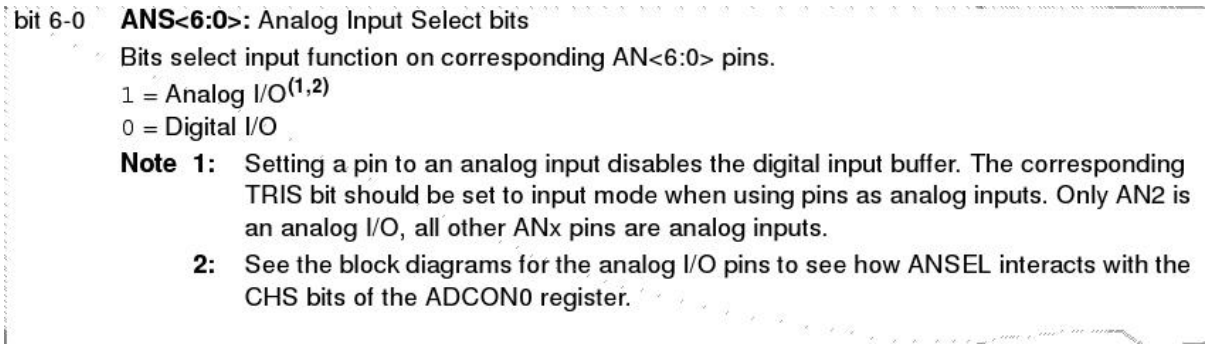
Figure 1: 16F877 ADC channels are controlled by PCFG bits

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:

[1] and some have 12-bits, those aren't currently handled by jallib ADC libraries, That's a restriction.



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 2: 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_high_res()` 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_low_res()` returns a `byte` as the result. **adc_read_high_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:

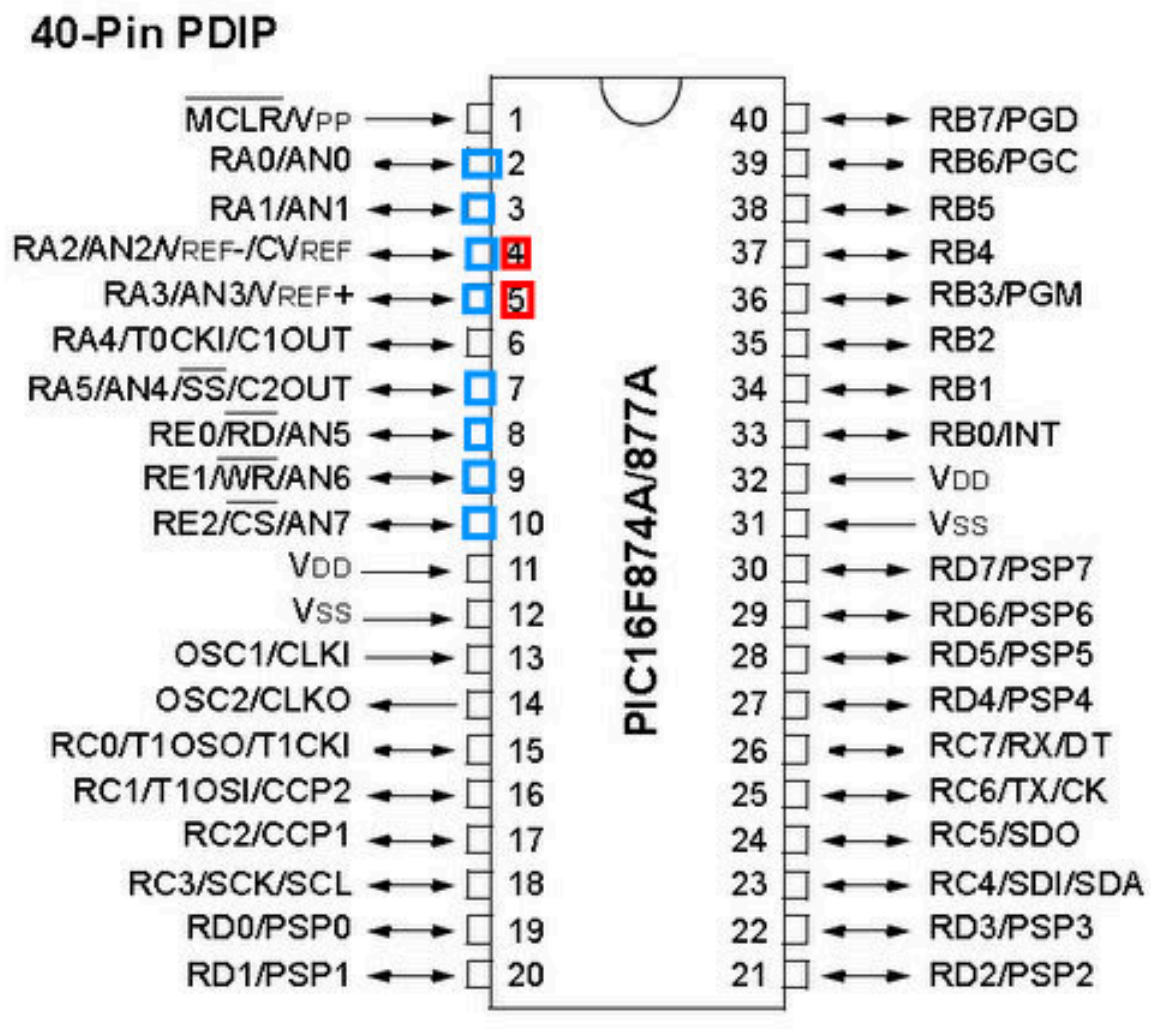


Figure 3: Analog pins and Vref pins on 16F877

Example 1: *16F877, with only one analog pin, no external 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, no LVP
pragma target WDT  disabled
pragma target LVP  enabled

-- We'll start to set all pins as digital
-- then, using ADC, we'll configure needed
-- ones as analog.
enable_digital_io()

include print
include delay
const serial_hw_baudrate = 115_200
include serial_hardware
serial_hw_init()

-- Step 1: ADC input pin setup we wil use channel 0
pin_AN0_direction = input
-- Step 2: Set A0 analog input and VDD as Vref
ADCON1_PCFG = 0b0000
-- Step 3: Use Frc as ADC clock
ADCON0_ADCS = 0b11
```

```
-- Now we can include the library
include adc
-- And initialize the whole with our parameters
adc_init()

-- will periodically send those chars
var word wmeasure
var byte bmeasure
const byte wprefix[] = "Result in high resolution: "
const byte bprefix[] = "Result in low  resolution: "

forever loop

   -- get ADC result, on channel 0
   -- this means we're currently reading on pin AN0 !

   -- access results in high resolution
   wmeasure = adc_read_high_res(0)
   -- wmeasure contains the result, as a word (byte*2)
   print_string(serial_hw_data,wprefix)
   print_word_dec(serial_hw_data,wmeasure)
   print_crlf(serial_hw_data)

   -- though we are in high resolution mode,
   -- we can still get a result as a byte, as though
   -- it were in low resolution.
   bmeasure = adc_read_low_res(0)
   print_string(serial_hw_data,bprefix)
   print_byte_dec(serial_hw_data,bmeasure)
   print_crlf(serial_hw_data)

   -- and sleep a little to prevent flooding serial...
   delay_1ms(200)

end loop
```

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

This is almost the same as before, except we now want 2 (analog pins A0 and A1) + 1 (Vref+ A3), so yes we are using 3 analog pins here.

The beginning is the same, here's just the part about ADC configuration and readings. We use a for loop to go over the 2 analog pins A0 and A1:

```
-- Step 1: ADC input pin setup we wil use channel 0 and 1 (2 channels)
pin_AN0_direction = input
pin_AN1_direction = input
-- Step 2: Set A0 and A1 analog input and A3 as Vref
ADCON1_PCFG = 0b0011
-- Step 3: Use Frc as ADC clock
ADCON0_ADCS = 0b11
-- Now we can include the library
include adc
-- And initialize the whole with our parameters
adc_init()

-- will periodically send those chars
var word measure
var byte lowmeasure, channel
const byte prefix[] = "Channel "
const byte highstr[] = " (high) "
const byte lowstr[] = " (low) "
const byte suffix[] = ": "

forever loop

   -- loop over all channels and read
   for 2 using channel loop

  -- get ADC result, high resolution
  measure = adc_read_high_res(channel)
  -- send it back through serial
  print_string(serial_hw_data,prefix)
  print_string(serial_hw_data,highstr)
  print_byte_dec(serial_hw_data,channel)
  print_string(serial_hw_data,suffix)
  print_word_dec(serial_hw_data,measure)
```

```
   print_crlf(serial_hw_data)
   -- and sleep a little...
   delay_1ms(100)

   -- Even if we set high resolution, we can still access results
   -- in low resolution (the 2 LSb will be removed)
   lowmeasure = adc_read_low_res(channel)
   print_string(serial_hw_data,prefix)
   print_string(serial_hw_data,lowstr)
   print_byte_dec(serial_hw_data,channel)
   print_string(serial_hw_data,suffix)
   print_byte_dec(serial_hw_data,lowmeasure)
   print_crlf(serial_hw_data)
   -- and sleep a little...
   delay_1ms(100)

    end loop
end loop
```

I²C (Part 1) - Building an I²C slave + Theory

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*[2]. 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:

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

```
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.

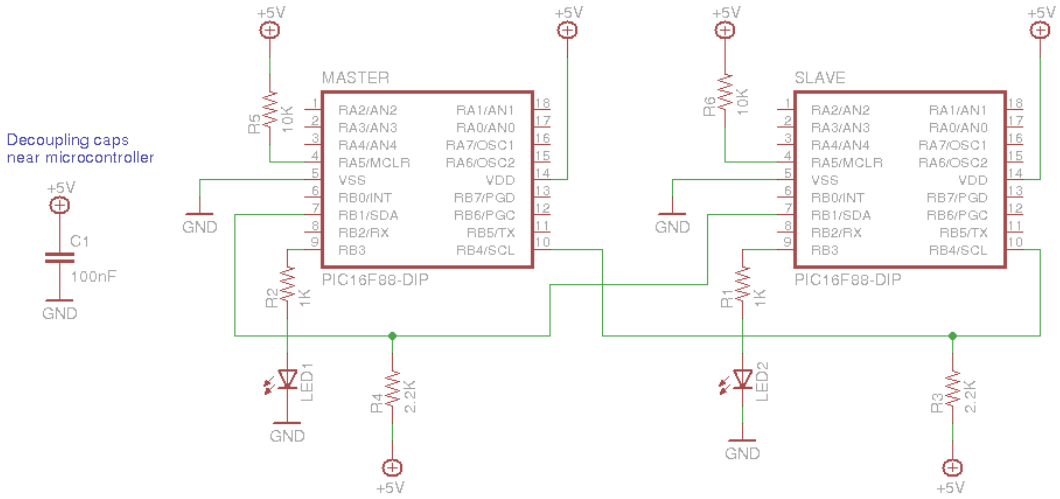
I²C (Part 2) - Setting up and checking an I²C bus

In previous tutorial, 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.

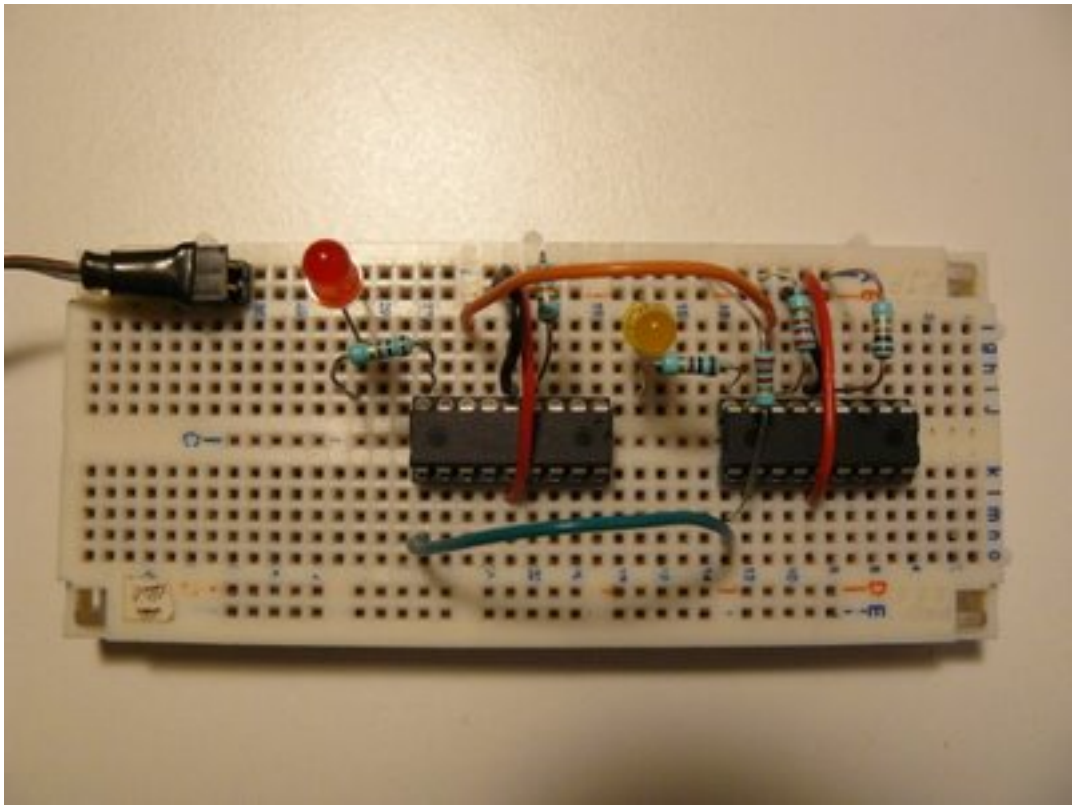
Checking the hardware and the i2c bus...

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.

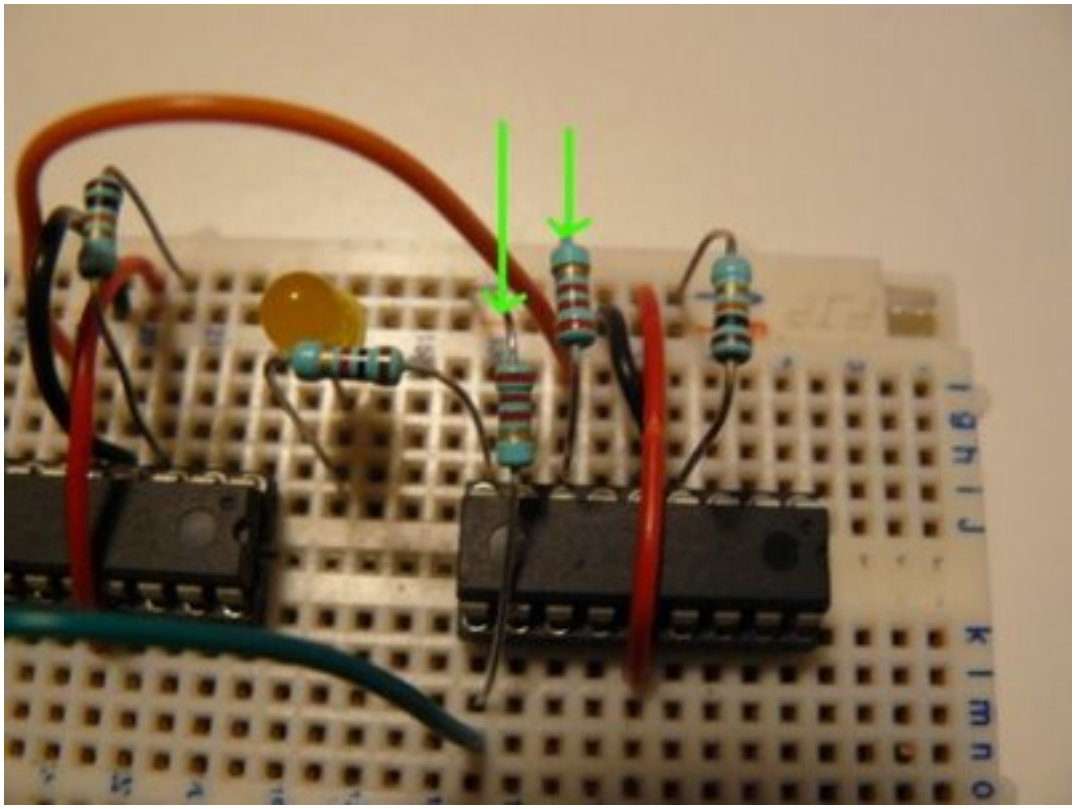
Connecting two PIC together through i2c is quite easy from a hardware point of view. Just connect SDA and SCL together, and **don't forget pull-ups resistors**. There are many differents values for these resistors, depending on *how long the bus is*, or the *speed you want to reach*. Most people use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



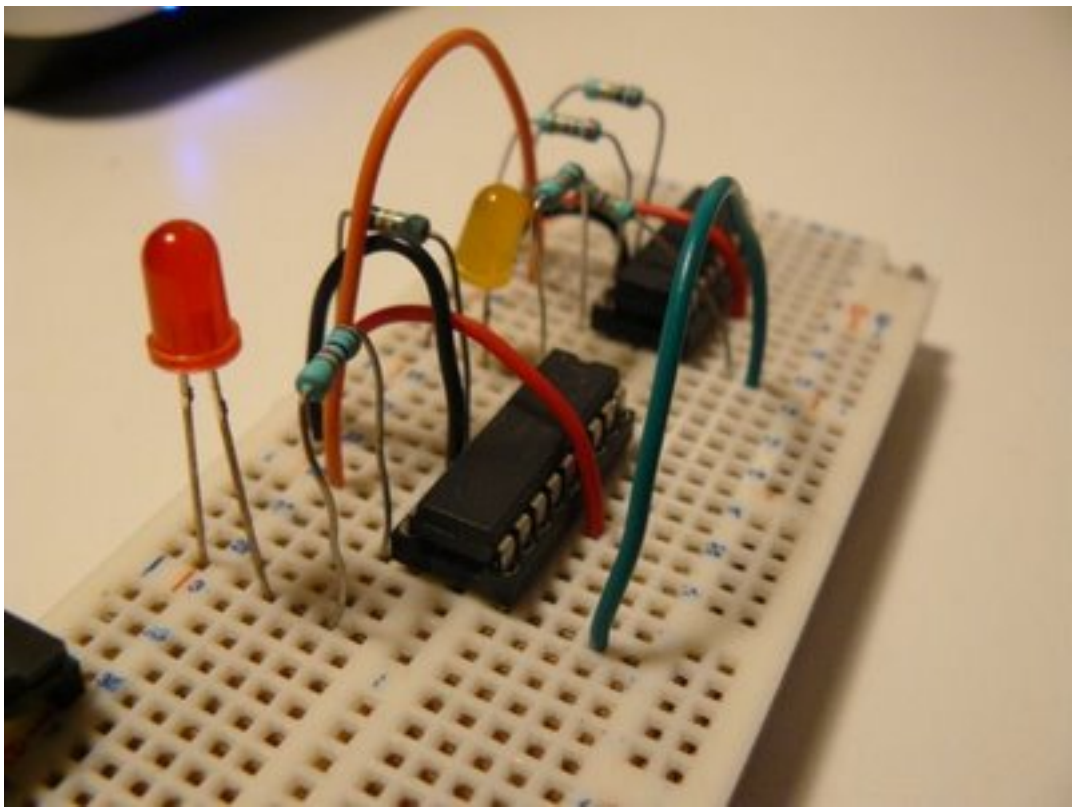
In this circuit, both PIC have a LED connected, which will help us understand what's going on. On a breadboard, this looks like that:



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 recommendations).

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 latest jallib pack, and check the following files (either in `lib` or `sample` directories):

- i2c_hw_slave.jal : main i2c library from the lib directory
- 16f88_i2c_sw_master_check_bus.jal: code for master from the sample directory
- 16f88_i2c_hw_slave_check_bus.jal : code for slave from the sample directory

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.

I²C (Part 3) - Implementing an I²C Slave

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:

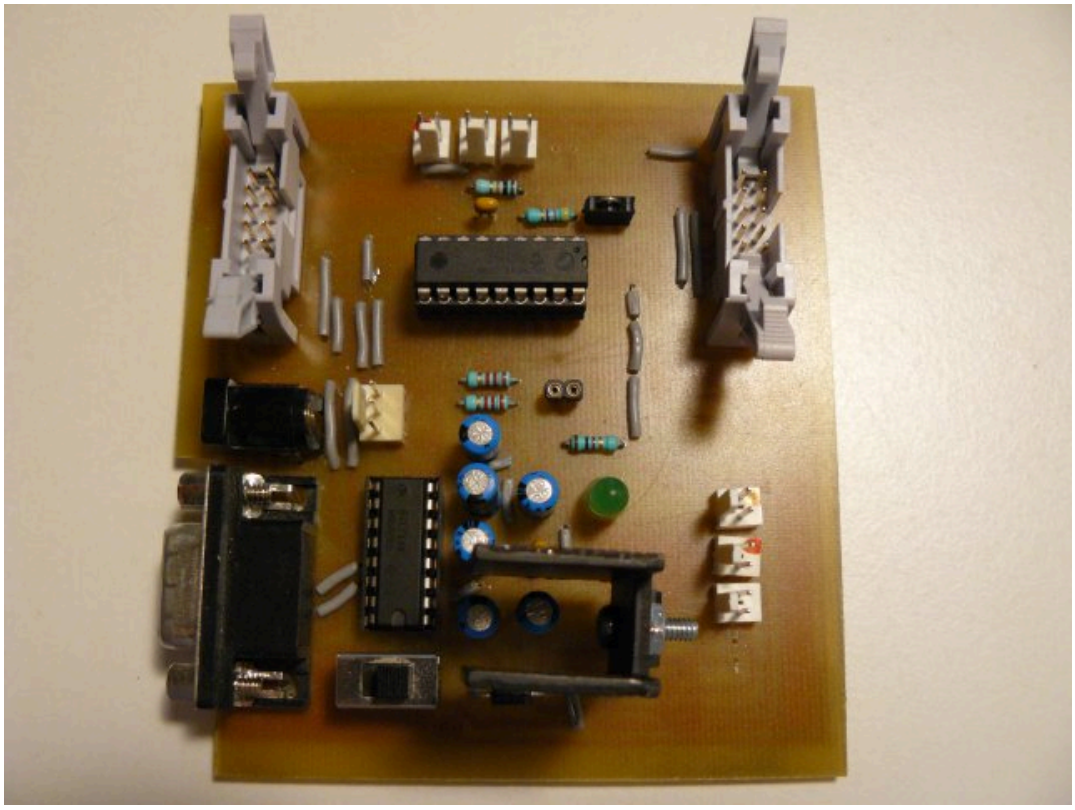
- if you send "a", it sends "b"
- if you send "b", it sends "c"
- if you send "z", it sends "{"[3]

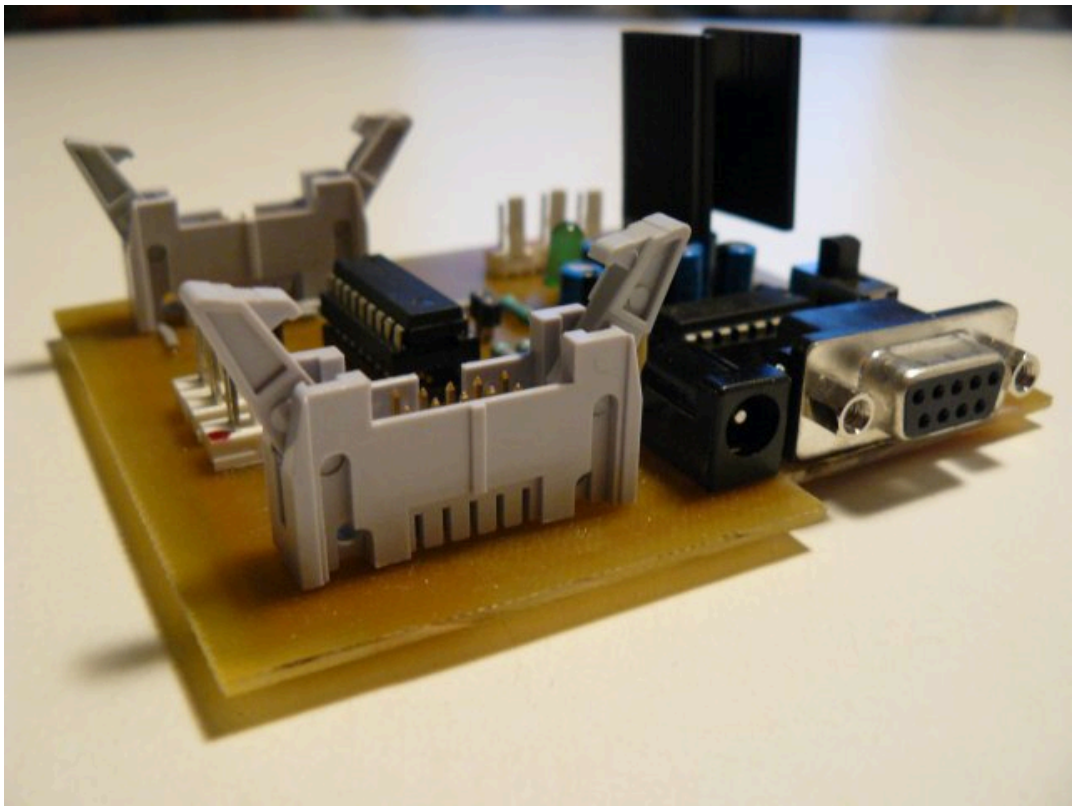
Building the i2c master

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.

My board looks like this:

[3] why "{" ? According to ASCII, "z" is the character for position 122. 123 is... "{")





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 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 under the name 16f88_i2c_sw_master_echo.jal in the sample directory of your Jallib installation.

Building the i2c slave

So this is the main part ! As exposed on first post, 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:

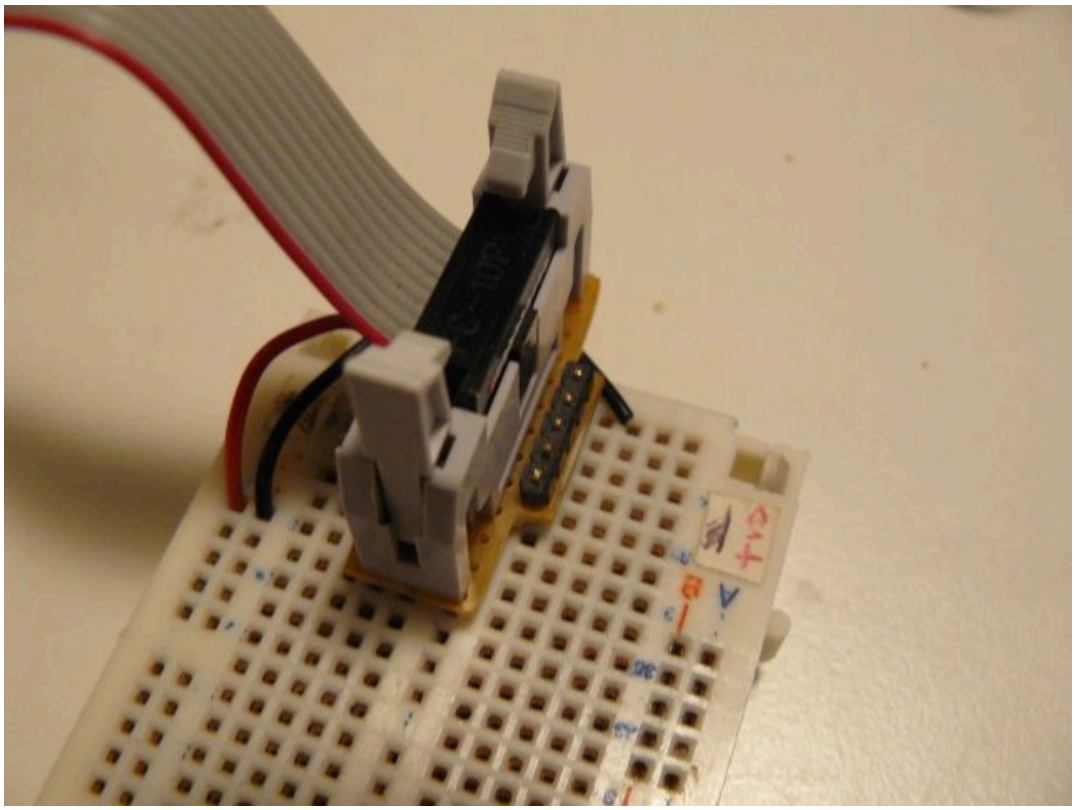
- i2c_hw_slave.jal
- i2c_hw_slave_isr.jal
- 16f88_i2c_sw_master_echo.jal
- 16f88_i2c_hw_slave_echo.jal

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

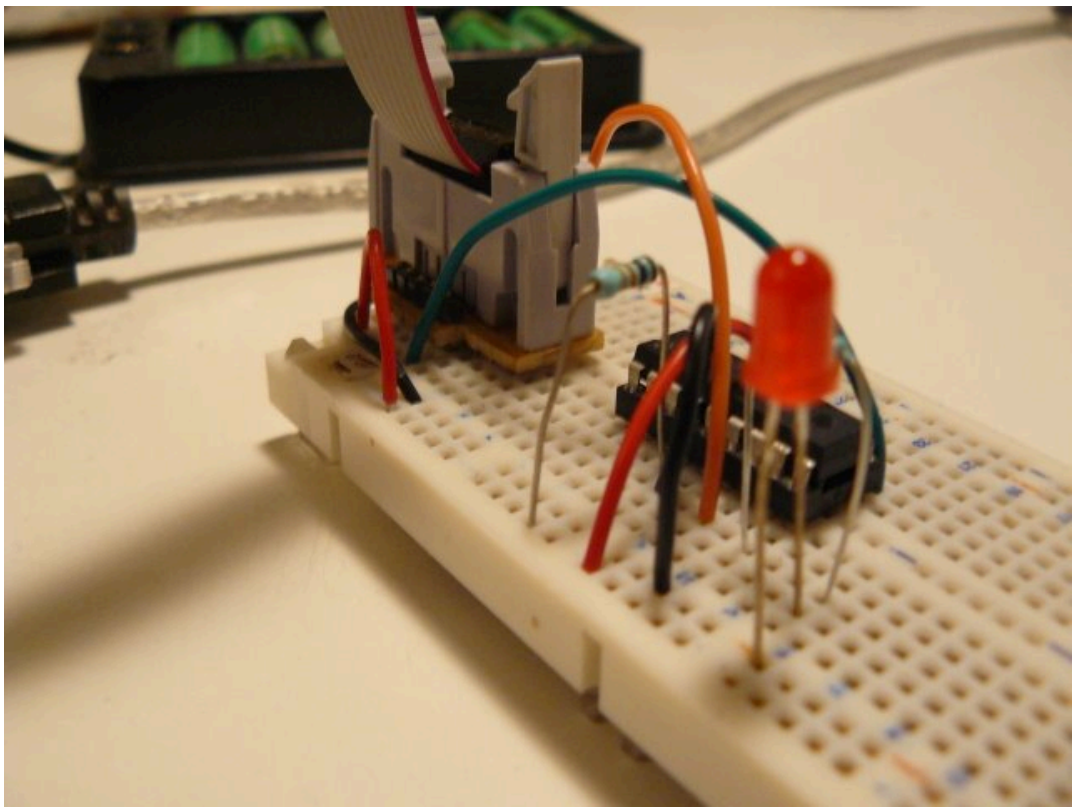
Connecting and testing the whole thing...

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 part 2 to know and see why...).

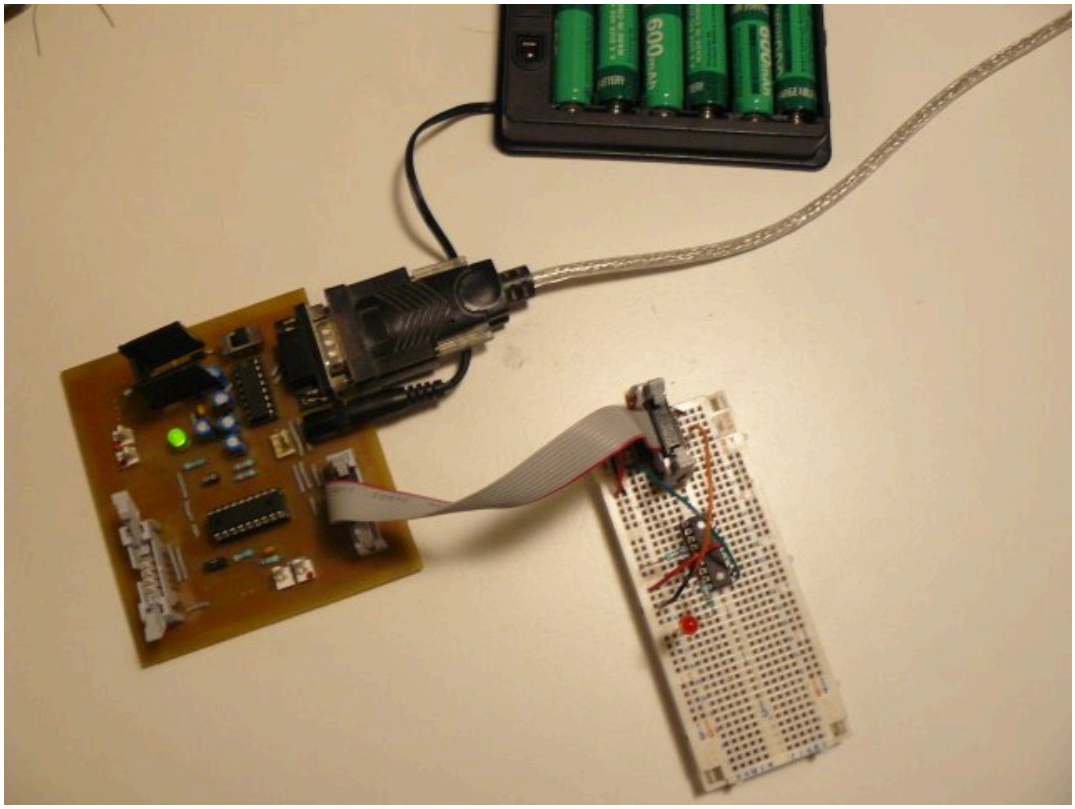
I use a connector adapted with a PCB to connect my main board with my breadboard. Connector's wires provide power supply, 5V-regulated, so no other powered wires it required.



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{01122334455667788899:
```

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.

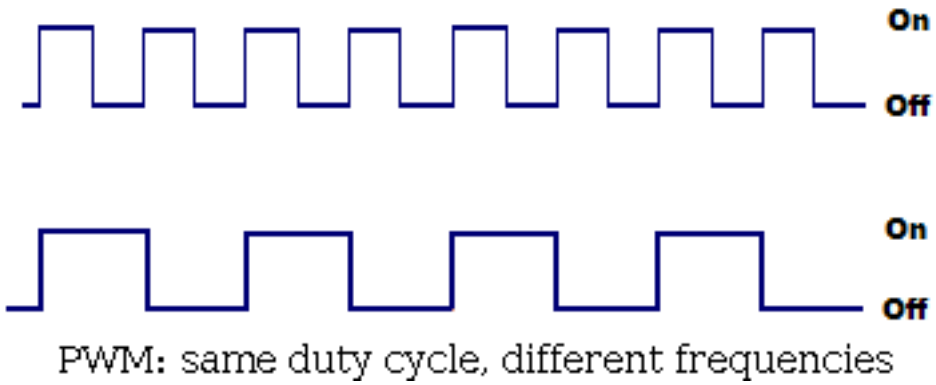
PWM Intro - Pulse Width Modulation

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 ?...

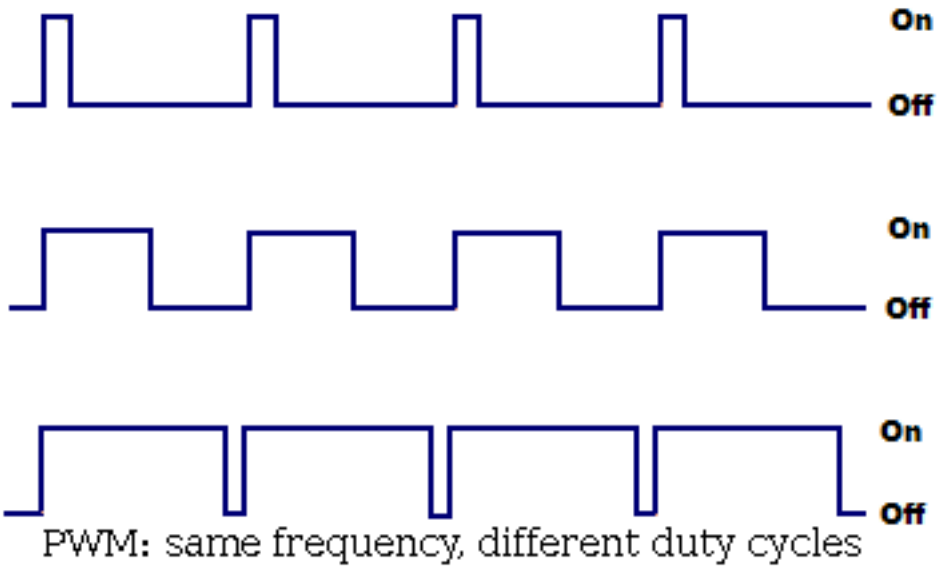
PWM is about switching one pin (or more) high and low, at different frequencies and duty cycles. This is a on/off process. You can either vary:

- 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 4: PWM: same duty cycle, different frequencies.



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

Figure 5: 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.

PWM (Part 1) - Dimming a led with PWM

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 circuit

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).

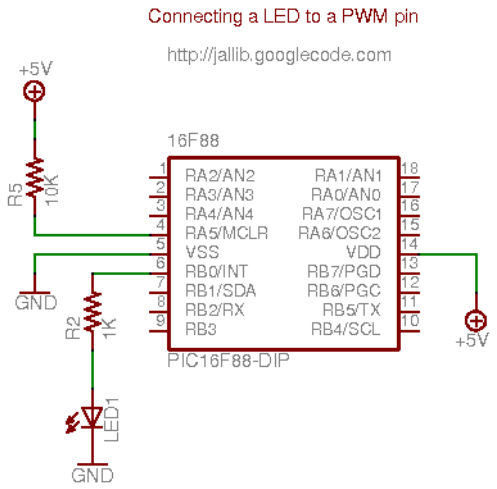
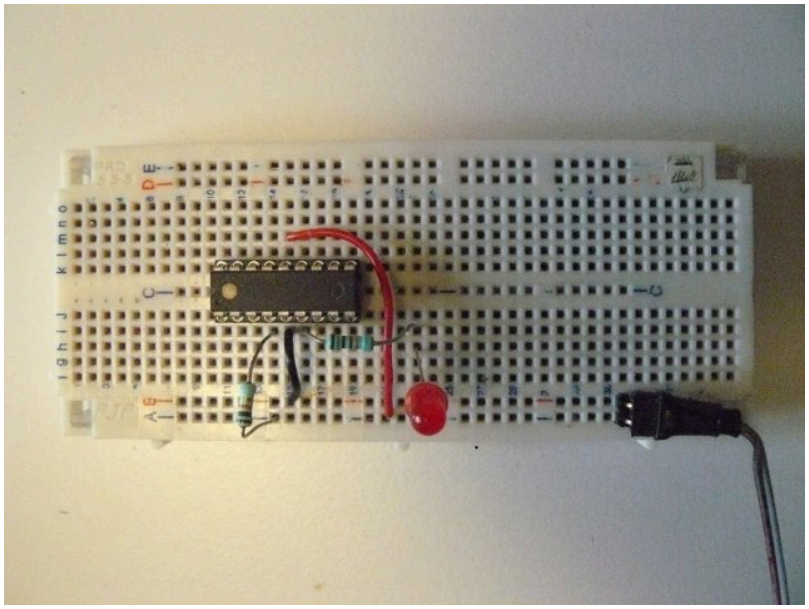
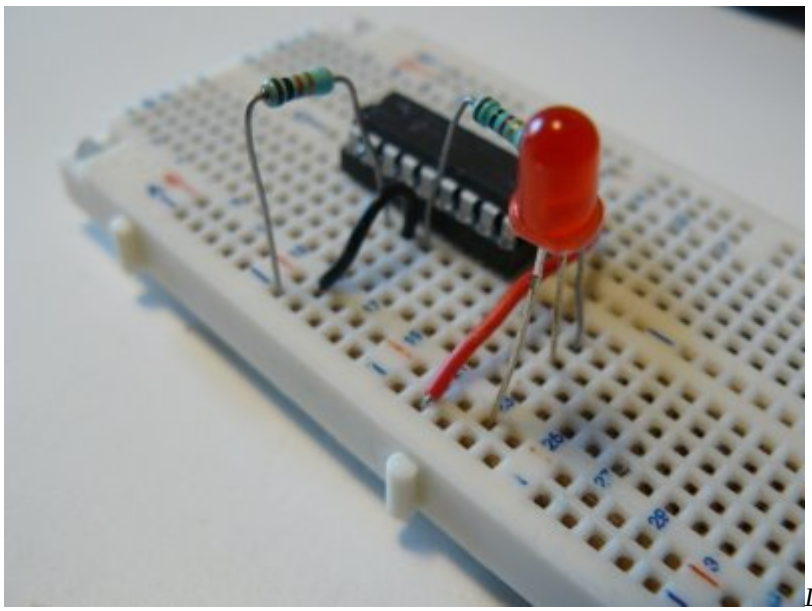


Figure 6: Connecting a LED to a PWM pin

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 latest jallib pack.

PWM (Part 2) - Sound and Frequency with Piezo Buzzer

In previous tutorial, 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 remember, 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.

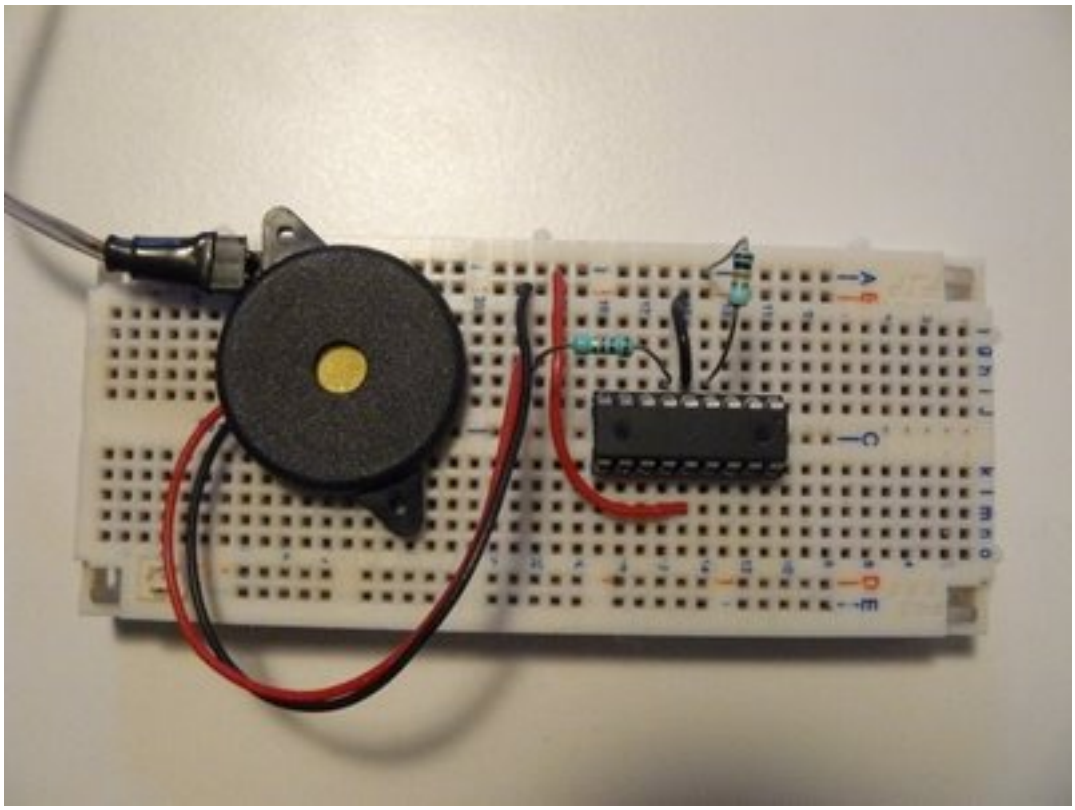
But why having a *constant duty cycle* ? What is the role of the duty cycle in this case ? Remember: when making a piezo oscillate, it's not the amount of volts which is important, it's how you turn the voltage on/off:[4]

- **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.

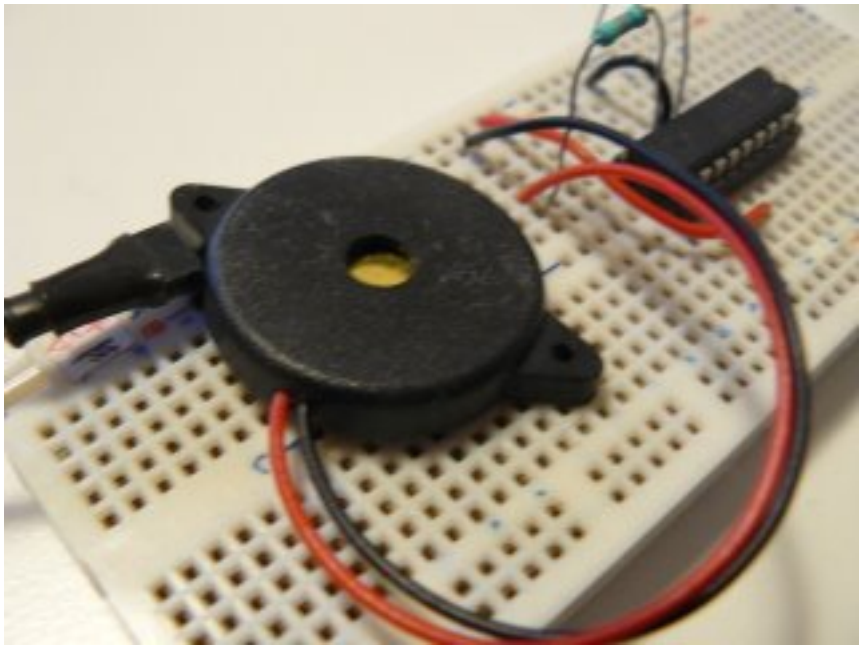
So, to summary, what is the purpose of the duty cycle in our case ? The **volume** ! You can vary the volume of the sound by modifying the duty cycle. 0% will produce no sounds, 50% will be the max volume. Between 50% and 100% is the same as between 0% and 50%. So, when I say when need a constant duty 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:



[4] 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 previous experiment. 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 latest jallib pack.

SPI Introduction

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?

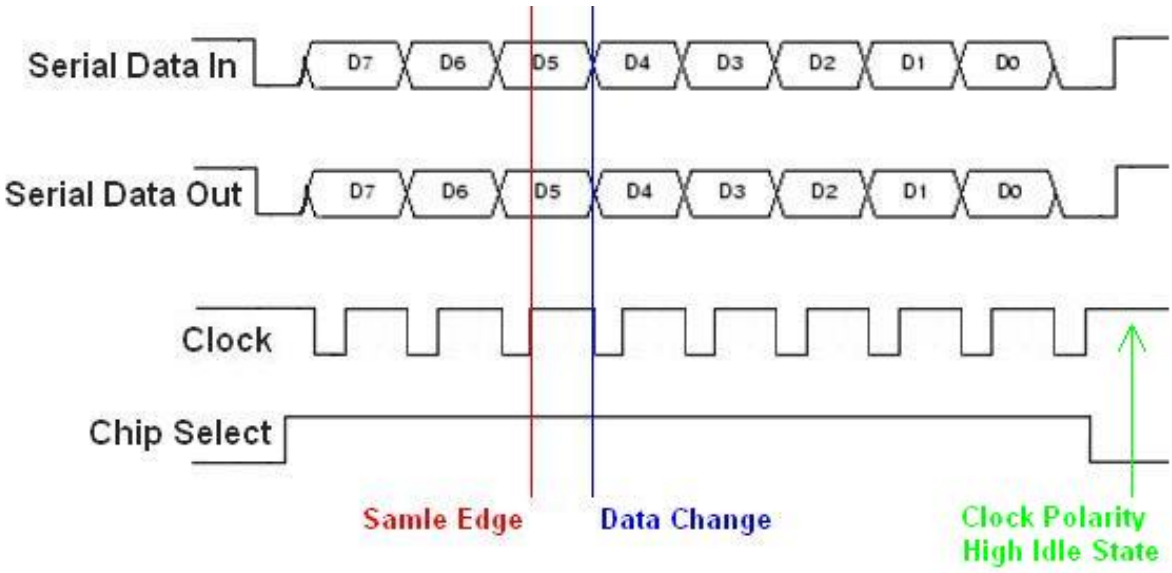
Firstly, SPI works in a master/slave setup. The master is the one that sends the clock pulses. At each pulse, data will be sent and received.

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 Support 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

USB (Part 1) - Introduction

What is USB?

USB stands for Universal Serial Bus and has replaced on may computer the standard (RS232) serial interface.

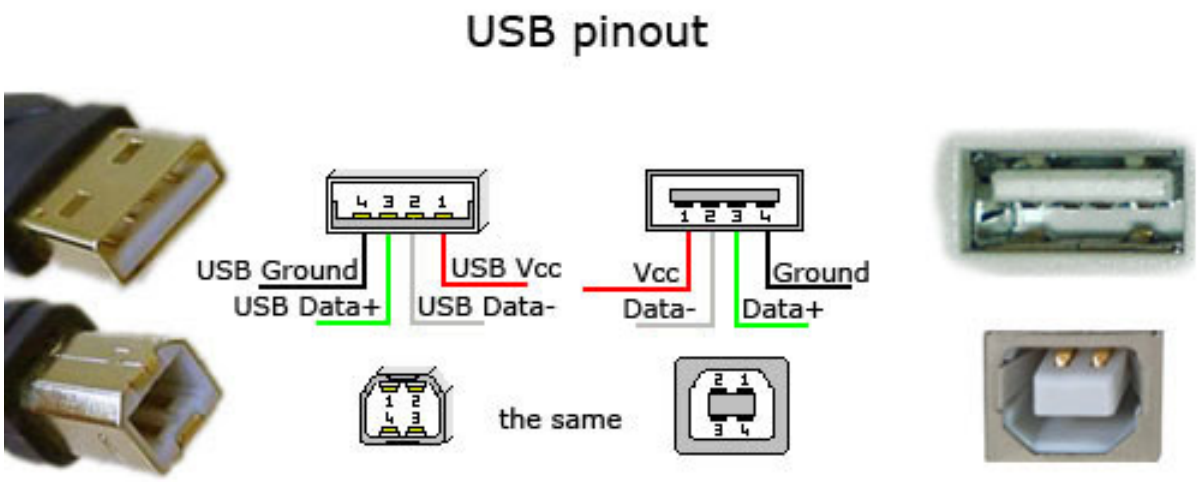
It is the most popular connection used to connect a computer to devices such as digital cameras, printers, scanners, and external hard drives. USB is a cross-platform technology that is supported by most of the major operating systems. On Windows, it can be used with Windows 98 and higher. USB is a hot-swappable technology, meaning that USB devices can be added and removed without having to restart the computer. USB is also “plug and play”. When you connect a USB device to your PC, your operating sytem should detect the device and even install the drivers needed to use it..

There are various versions of USB. The original version of USB is USB 1.0, supporting speeds of up to 11 Mbps and was used mostly to connect keyboards and mice. This is also the version that is supported by the JAL USB driver although the PIC USB hardware is able to support USB 2.0 which supports speeds up to 480 Mbps. If you want to know more about the higher USB version you can find all information on the Internet..

How does USB work?

That is a little bit complex to explain. If you want to know more about this, visit the official `USB website` where you can find all specifications of the Universal Serial Bus. You can also find websites which explain this very well like `USB in a nutshell`.

What you should know is that it uses 4 connections of which two are for power and the other two, called Data+ and Data-, are for communication. There are different types of connectors as shown in the following picture.



In the next section we will create a program that makes it possible to control the LEDs on a PIC from a (host) computer using the USB as serial port. For the serial interface from the computer to the PIC, a free terminal emulation program Termite is used, but any other terminal emulation program will do.

USB (Part 2) - The PIC as a serial port

Using the Jallb USB library

In the previous section some introduction as given about what the Universal Serial Bus is about. In this section we will make use of the Jallib USB driver to create an application using the USB.

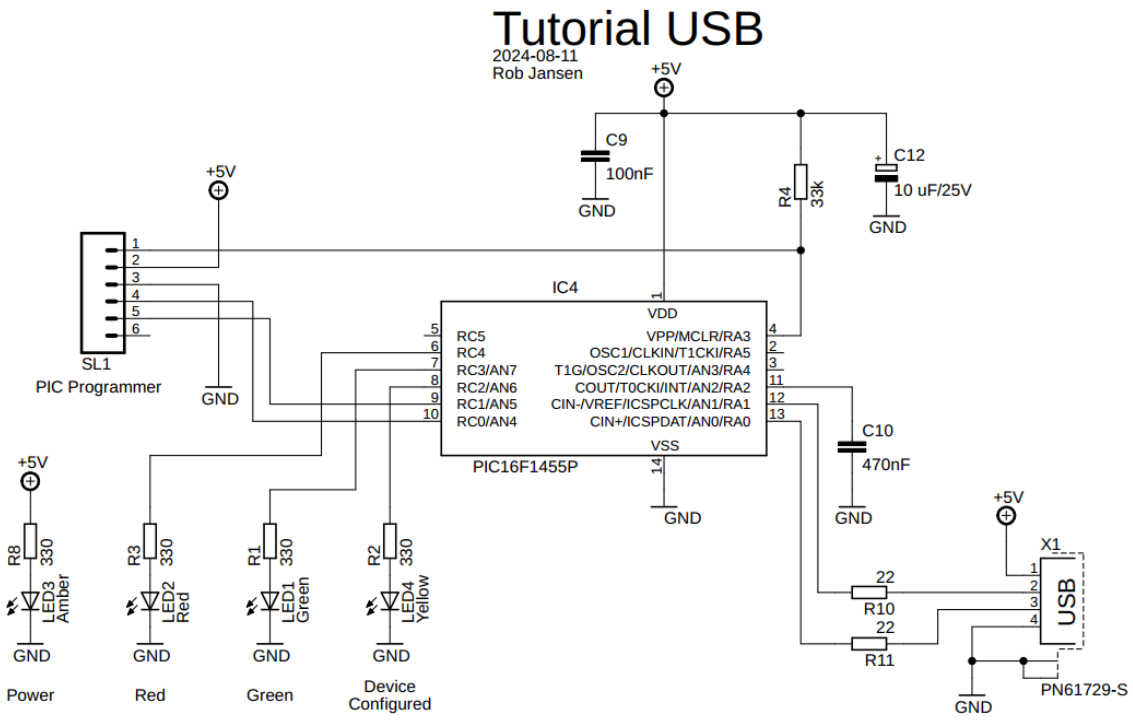
The Jallib USB library

Of course you can only use the Jallib USB library with PICs that have the USB hardware on board. This hardware is called the USB Serial Interface Engine (SIE). This SIE only supports the PIC to be a USB device, which means that it always needs a USB host like your computer. The host configures the PIC and initiates all USB transmissions.

There are various sample programs that demonstrate the capabilties of the Jallib USB library. Sample files are available to show the following features:

- Using the PIC as a serial (COM) port
- Using the PIC as a keyboard
- Using the PIC as a mouse
- Using the PIC as a Human Interface Device (HID)

To show you how this works, we need some hardware and software. The hardware used for this Tutorial consists of a PIC16F1455 with some LEDs that can be controlled via USB. The schematic diagram of this hardware is given below.



In the following sample program the PIC will act as a serial port. From the computer (the host) we will control two LEDs connected to the PIC. The PIC will also send the status of the LEDs back to the computer. On the computer side, the terminal emulation program is used for sending the control commands to the PIC and for showing the message returned by the PIC.

The sample program explained

We start with the configuration of the PIC. This is done via the setting the correct `pragma target` and we set the `target clock`. Note that for correct operation the USB hardware, the PIC has to run at 48 MHz,

```
include 16f1455             -- Target processor.

pragma target clock         48_000_000

-- Settings for internal clock and system clock 48 MHz.
pragma target OSC           INTOSC_NOCLKOUT -- Internal clock
pragma target CLKOUTEN      DISABLED -- CLKOUT function is disabled.
pragma target PLLMULT       N3X      -- PLL Multipler Selection Bit, 3x

-- Other fuses.
pragma target CPUDIV        P1       -- NO CPU system divide
pragma target USBLSCLK      F48MHZ   -- System clock expects 48 MHz
pragma target PLLEN         ENABLED  -- 3x or 4x PLL Enabled
pragma target FCMEN         DISABLED -- Fail-Safe Clock Monitor is disabled
pragma target WRT           DISABLED -- Write protection off
pragma target STVR          ENABLED  -- Stack Overflow or Underflow will cause a Reset
pragma target LPBOR         DISABLED -- Low-Power BOR is disabled
pragma target IESO          DISABLED -- Internal/External Switchover Mode is disabled
pragma target PWRTE         DISABLED -- power up timer
pragma target BROWNOUT      DISABLED -- no brownout detection
pragma target WDT           DISABLED -- Watchdog disabled
pragma target MCLR          EXTERNAL -- External reset
pragma target LVP           ENABLED  -- allow low-voltage programming
pragma target VOLTAGE       MAXIMUM  -- brown out voltage
pragma target CP            DISABLED -- Program memory code protection is disabled

OSCCON        = 0b1111_1100          -- Select PLL,3x, 16MHz internal oscillator
```

When a PIC is reset, all pins are set to input and are floating. A good practice is to use the weak pull-up feature of the PIC to pull the inputs high. When making a pin `output` the weak pull-up for that pin is disabled. For this PIC only port A has the weak pull-up feature and because of that we make port C output so that the pins are not floating.

```
-- Enable weak pull-up for port a and and set port c to output just to
-- have no floating input pins.
OPTION_REG_WPUEN = FALSE             -- Enable weak pull-up for port a.
WPUA          = 0b0011_1111          -- Weak-pull up for all inputs.
TRISC         = 0b0000_0000          -- Port c output.
```

Now include the Jallib USB library. Since we will send information back to the computer we want to format it nicely so we also include the Jallib print library. We will also define the pins including some aliases to make the program more readable.

```
-- Include serial library and print library for formatting print output.
include usb_serial
include print

-- Aliases for LEDs, active HIGH.
alias led_red is pin_C4
pin_C4_direction = output            -- Pin 6 of 14 pin DIP.
alias led_green is pin_C3
pin_C3_direction = output            -- Pin 7 of 14 pin DIP.
alias led_yellow is pin_C2
pin_C2_direction = output            -- Pin 8 of 14 pin DIP.
```

This program uses a few variables, one for the character that is received from the computer and two bit variable to hold the status of the Red and Green LED.

```
-- Variables.
var byte character
var bit red_value, green_value
```

Last but not least is the main part of the program which does the following:

- Initialize the usb library
- Initialize (clear) all LEDs
- Report if the PIC USB device is configured (Yellow LED)
- Keep the USB going by frequently calling the function `usb_serial_flush()`

  Note: The USB driver can also be used on an interrupt basis as described in the USB library and the USB sample programs. In that case the `usb_serial_flush()` is not required
- Handle the commands sent by the computer and control the LEDs accordingly. A single character command will toggle the status of the LED from on to off or the other way around
- Return the status of the controlled LED to the computer after a command was received
- Return an error message if an incorrect command was received

The main program then becomes as follows.

```
-- ------------- The main program starts here  ------------------------

-- Setup the USB serial library.
usb_serial_init()

-- LEDs off.
led_yellow = FALSE
red_value = FALSE
green_value = FALSE

forever loop

   -- Update Red and Green LEDs.
   led_red = red_value
   led_green = green_value

   -- Poll the usb ISR function on a regular base,
   -- in order to serve the USB requests
   usb_serial_flush()
```

```
   -- Check if USB device has been configured by the host.
   if ( usb_cdc_line_status() !=  0x00 )  then
      led_yellow = TRUE
   else
      led_yellow = FALSE    -- disconnected
      red_value = FALSE     -- red led off
      green_value = FALSE   -- green led off
   end if

   -- Check for input character. If OK, toggle led value and report to host.
   if usb_serial_read(character) then
      if (character == "R") | (character == "r") then
         red_value = !red_value
         print_string(usb_serial_data, "Red LED is ")
         if red_value then
            print_string(usb_serial_data, "on")
         else
            print_string(usb_serial_data, "off")
         end if
      elsif (character == "G") | (character == "g") then
         green_value = !green_value
         print_string(usb_serial_data, "Green LED is ")
         if green_value then
            print_string(usb_serial_data, "on")
         else
            print_string(usb_serial_data, "off")
         end if
      else
         print_string(usb_serial_data, "Expecting 'r' or 'g'")
      end if
      print_crlf(usb_serial_data)
   end if

end loop
```

The complete program can be found in the sample directory of the latest Jallib release under the name `16f1455_tutorial_usb_serial.jal`.

In this video you can see the program in action. Commands are entered in the terminal emulation program, LEDs are switched on and off and the response of the PIC is returned to the computer. Some additional info that is worth mentioning:

- In order to initialize the PIC as USB device you have to enable RTS/CTS
- The USB serial baudrate and number of bits is not relevant
- Since we are using single character commands make sure to disable sending carriage return and/or linefeed after the character was entered, otherwise the PIC will return that as an unsupported command.

Chapter 3

Experimenting with external parts

You now have learned enough to be able to begin interfacing your PIC with external parts. Without being exhaustive, this chapter explains how to use a PIC with several commonly used parts, such as an LCD screen.

Hard Disks - IDE/PATA

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

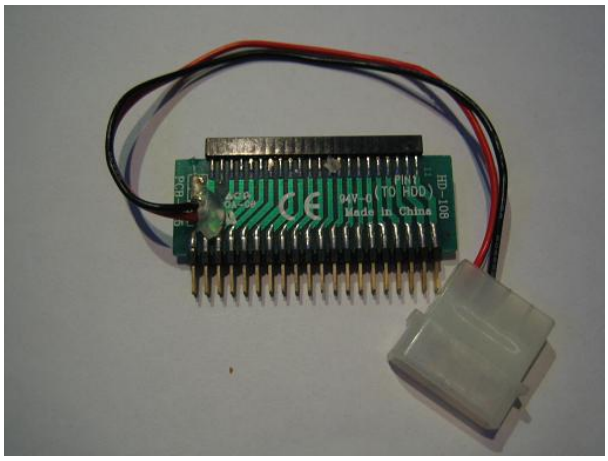
The current jallib library will accept drives up to 128GB. The 128GB limit is due to and addressing limitation, this is the 28 bit addressing limitation.The max address you will be able to reach is hex 0xFFFFFFF. If you multiply this address by 512 bytes (1 sector) you get a max size of 137,438,952,960 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 = 128PB (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:



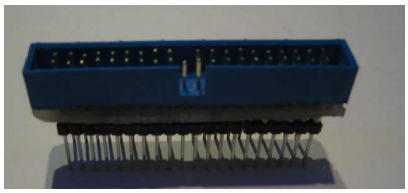
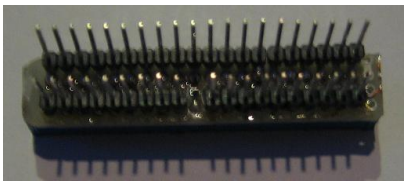
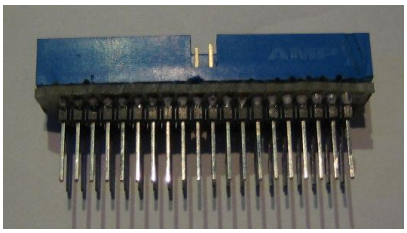


Build a breadboard connector

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:

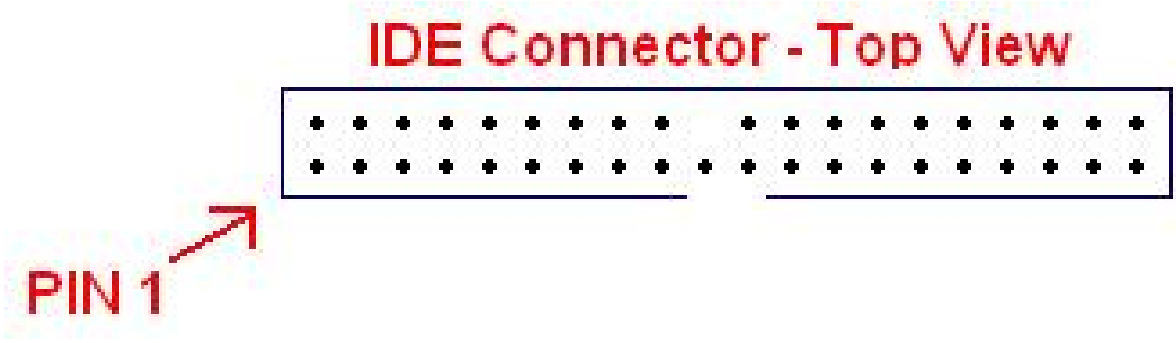




Circuit Power

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:

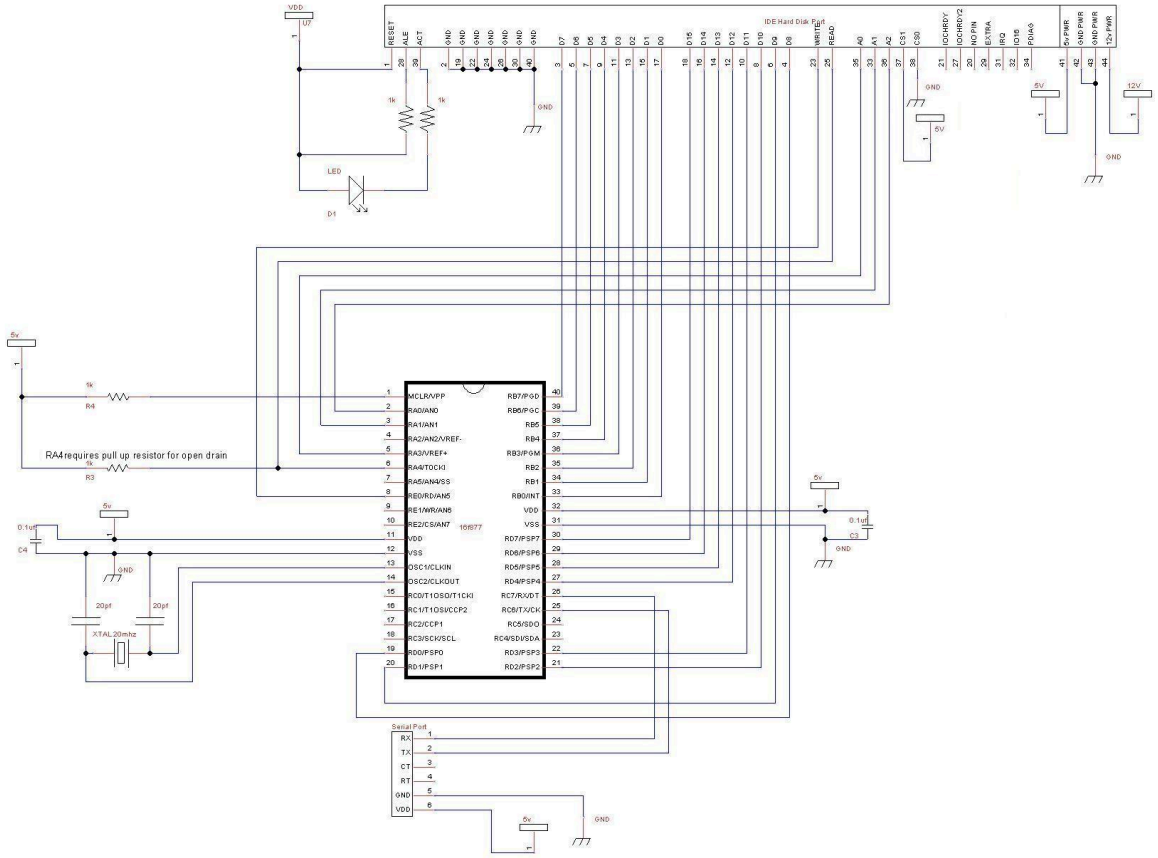


Build the circuit

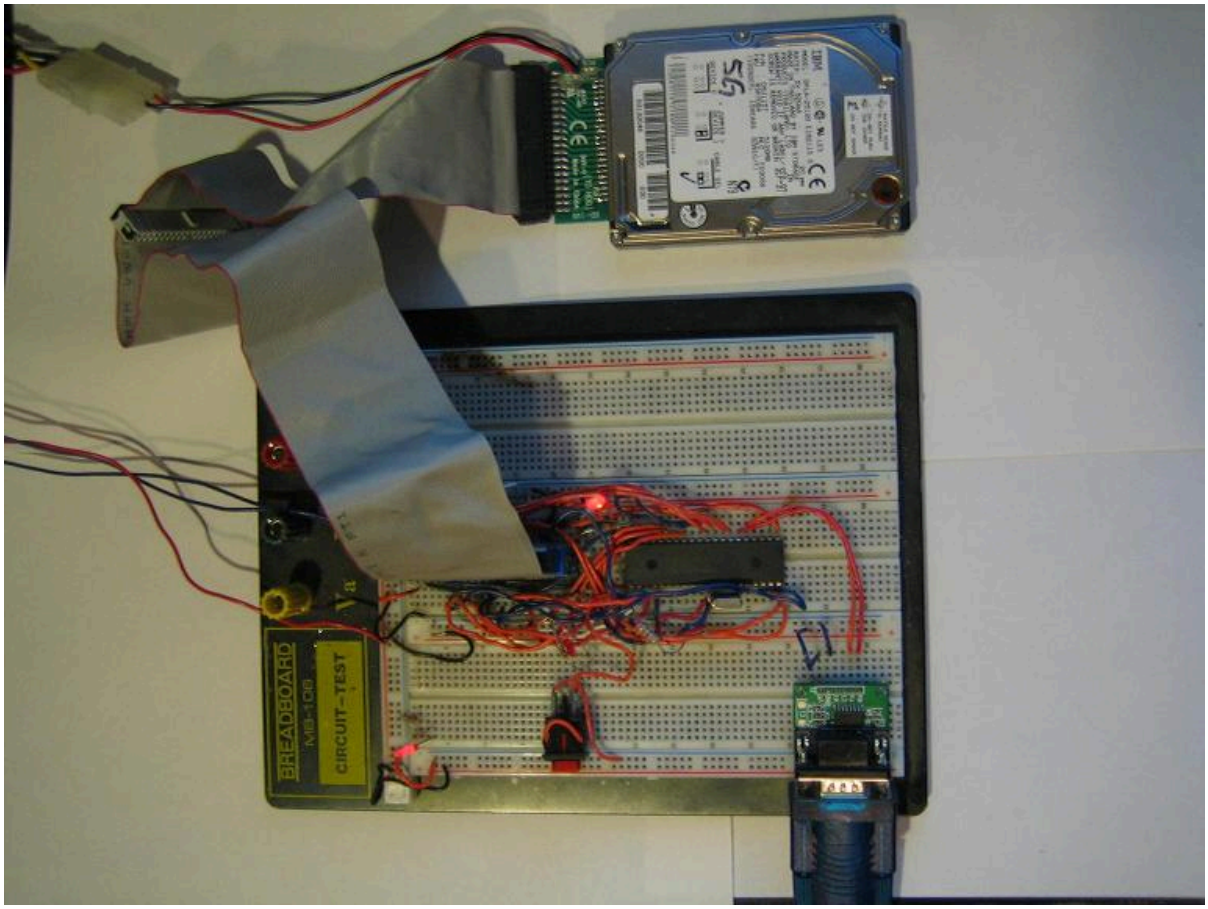
PIN	FUNCTION	PIN	FUNCTION
1	/RESET	2	GND
3	D7	4	D8
5	D6	6	D9
7	D5	8	D10
9	D4	10	D11
11	D3	12	D12
13	D2	14	D13
15	D1	16	D14
17	D0	18	D15
19	GND	20	NO PIN
21		22	GND
23	/IOWR - READ Pin	24	GND

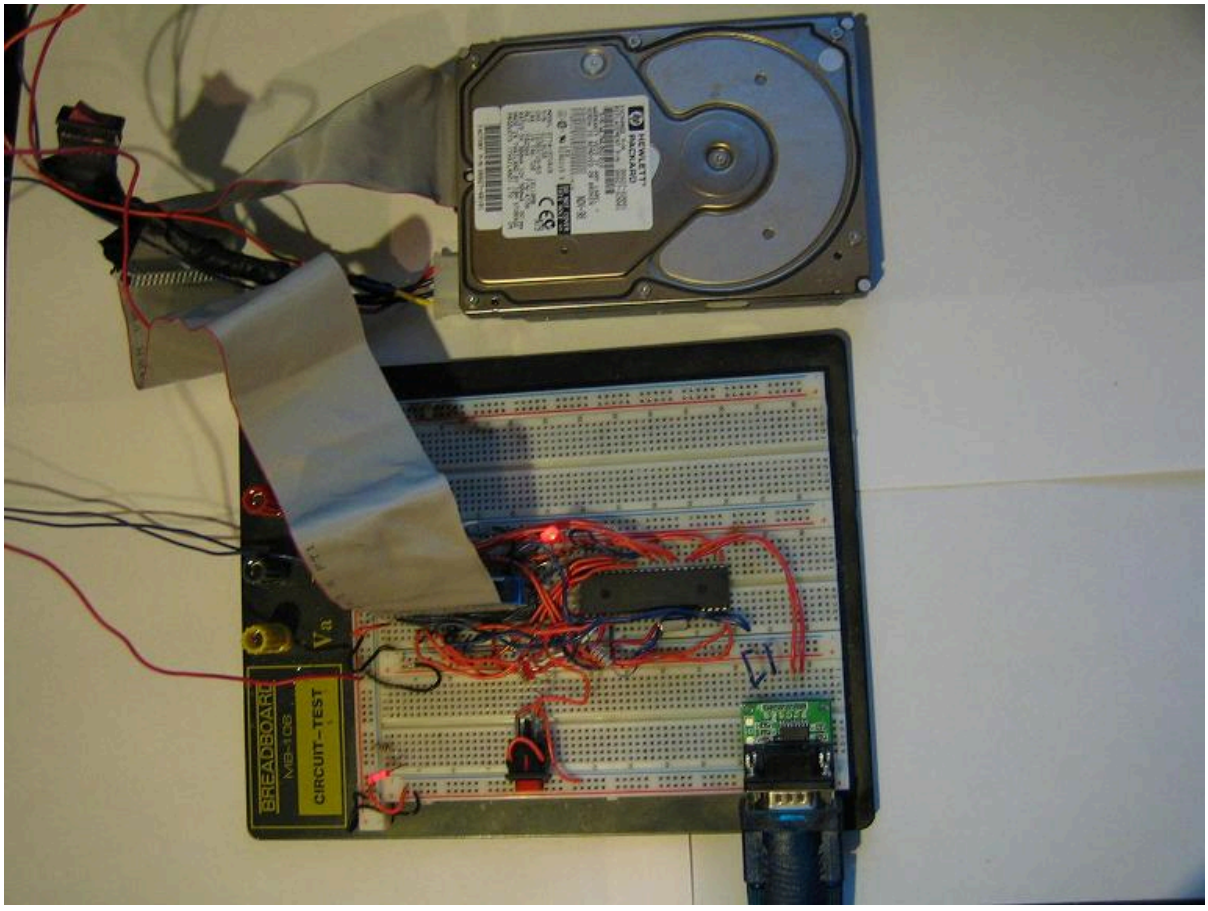
PIN	FUNCTION	PIN	FUNCTION
25	/IORD - Write Pin	26	GND
27		28	ALE - 1K resistor to 5v
29		30	GND
31		32	
33	A1	34	
35	A0	36	A2
37	/CS0 (to 5v)	38	/CS1 (to GND)
39	ACT - BUSY LED	40	GND

Build the circuit 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


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 in the jallib relase.

Sample file 16f877a_pata_hard_disk.jal in the lib directory.

Library file pata_hard_disk.jal in the sample directory.

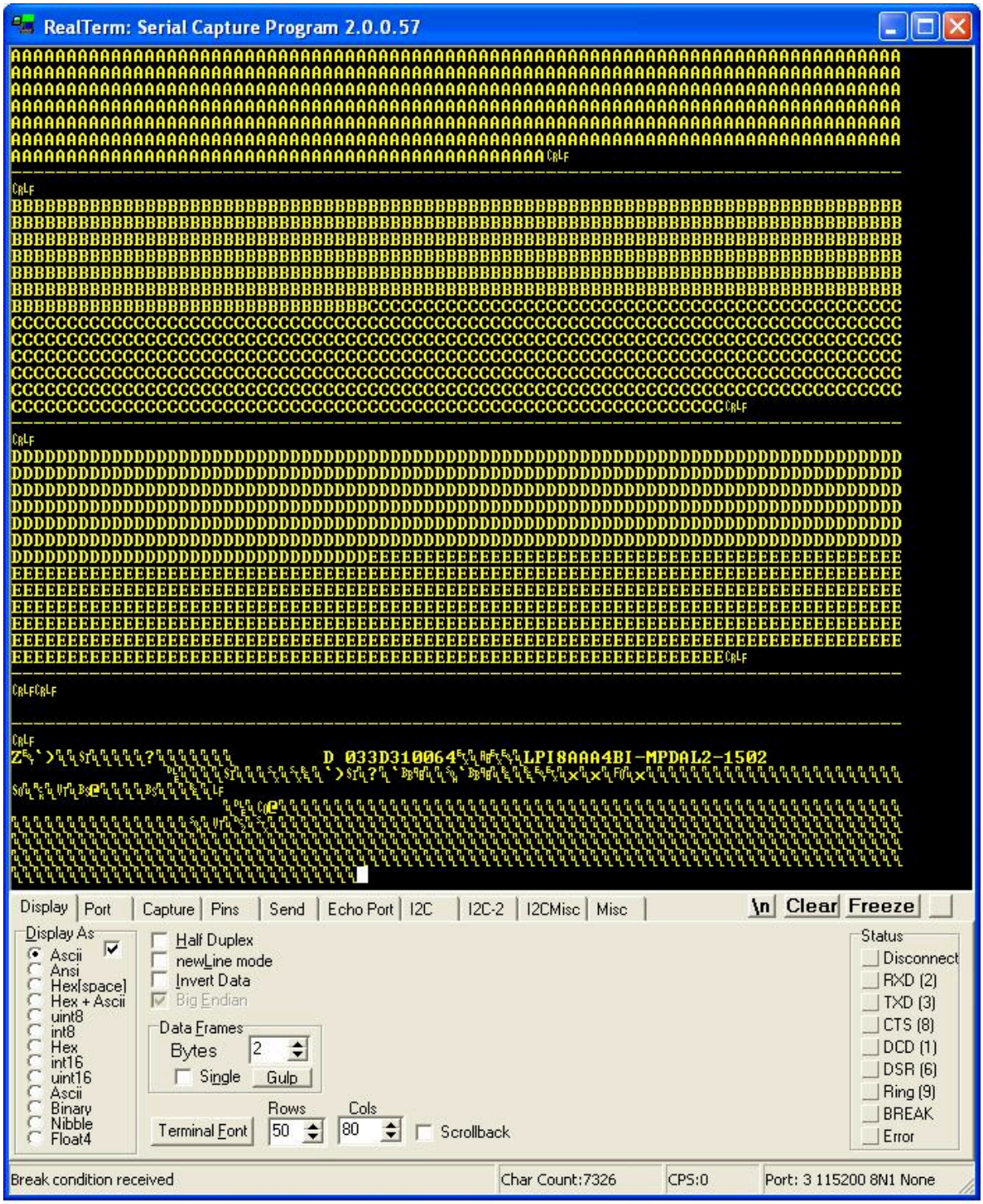
Now lets test it and make sure it works. Compile and program your pic with 16f877_sd_card.jal from your jallib samples directory. If you are using another pic, change the "include 16f877" line in 16f877_sd_card.jal to specify your PIC before compiling.

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 115200 baud rate. Now turn it on. You should get data similar to the image below onto your serial port. You will also hear the hard drive turn on and off because of one of the examples in the sample file.

Serial Port Output



Some of the data is not shown in the above image. If your disk is formatted with fat32 you may be able to see some readable data from the boot sector. On my drive formatted with fat32 I can read "Invalid partition table Error loading operating system" (not shown in the image). The AA BB CC DD EE are read/write examples that will be shown below. The last set of data is from the Identify Drive command.

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 16F877a                    -- target PICmicro
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
```

Disable all analog pins and wait for power to stabilize

```
enable_digital_io() -- disable all analog pins if any
_usec_delay (100_000) -- wait for power to stabilize
```

Setup serial port and choose baud rate 115200

```
-- setup uart for communication
const serial_hw_baudrate  = 115200   -- set the baud rate
include serial_hardware
serial_hw_init()
```

Include the print library

```
include print       -- include the print library
```

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

-- uses additional code space to add a speed boost to sector_read procedures
const bit PATA_HD_READ_EXTRA_SPEED = FALSE

-- 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 comment out cs1/cs0 and save pins because of the constant we set.

```
-- pin assignments
alias     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_a3
```

```
alias     pata_hd_a0_direction          is pin_a3_direction
alias     pata_hd_a1                    is pin_a1
alias     pata_hd_a1_direction          is pin_a1_direction
alias     pata_hd_a2                    is pin_a0
alias     pata_hd_a2_direction          is pin_a0_direction

alias     pata_hd_iowr                  is pin_e0
alias     pata_hd_iowr_direction        is pin_e0_direction
alias     pata_hd_iord                  is pin_a4
alias     pata_hd_iord_direction        is pin_a4_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 a separator procedure, This will be used to display "------" onto the serial port between examples.

```
-- procedure for sending "-----------------" via serial port
procedure separator() is
   serial_hw_data = 13
   serial_hw_data = 10
   const byte str3[] = "----------------------------------------"
   print_string(serial_hw_data, str3)
   print_crlf(serial_hw_data)
end procedure
```

It is always a good idea to send something to the serial port so we know the circuit is alive. Let's send "Hard Disk Sample Started"

```
-- Send something to the serial port
separator()              -- send "----" via serial port
var byte start_string[] = "HARD DISK SAMPLE STARTED"
print_string(serial_hw_data,start_string)
```

Declare some user variables

```
-- variables for the sample
var word step1
var byte data
```

EXAMPLES

OK, now that everything is setup, we are ready for some examples.

You will find that these examples are identical to the ones in the SD Card tutorial. This makes it easy for you to switch between using a hard drive and a SD Card.

Before we get stated, you may want to get to know your hard drive and it's size. This way you will know what the maximum addressable sector is.

On newer drives, you will see on the front sticker the number of LBA's. This is the number of sectors on the drive. We must subtract one from the number of LBA's to get the highest addressable sector since the 1st sector is at address 0. My drive says "60058656" LBA's, therefore the last sector is at 60058656 - 1.

Each sector is 512 bytes, so the actual size of this drive is 60058656 * 512 = 30GB

On the sticker of some older drives, you will see CYL, HEADS,SEC/T. You can calculate the number of sectors with: (cylinders * heads * sectors per track). Then you may multiply that by 512 if you wish to get the size of the drive in bytes.

I have left a few ways to read and write to hard disks. The usage you choose may will on the PIC data space you have, and what your application is. On a smaller PIC, you will only be able to run examples #1, #2, #5 and #6. I'll explain as I go.

Example #1 - Read data at sector 0

This is a low memory usage way of reading from the hard disk, however it is slower then some of the other examples later on. This method requires the use of pata_hd_start_read(), pata_hd_data_byte, and pata_hd_stop_read(). You'll see that the usage is quite simple.

Note: The variable pata_hd_data_byte is not actually a variable, it is a procedure that looks & acts like a regular variable. This is called a pseudo variable. You may use this variable to read data or write data, as shown in these examples.

The steps are:

1. Start reading at a sector address. In this case, sector 0 (the boot sector)
2. Loop many times while you read data. One sector is 512 bytes, we will read two sectors.
3. Store each byte of data into the variable "data". You can retrieve the data by reading the pseudo variable pata_hd_data_byte
4. Do something with the data. Let's send it to the serial port.
5. End the loop
6. Tell the hard disk we are done reading. The hard disk light will go out at this step.

```
pata_hd_start_read(0)      -- get sd card ready for read at sector 0
for 512 * 2 loop           -- read 2 sectors (512 * 2 bytes)
  data = pata_hd_data_byte -- read 1 bytes of data
  serial_hw_write(data)    -- send byte via serial port
end loop
pata_hd_stop_read()        -- tell sd card you are done reading
```

OK, we're done our example, so lets separate it from the next one with the separator() procedure to send some "-----" characters and a small delay.

```
separator()                -- separate the examples with "----"
_usec_delay(500_000)       -- a small delay
```

Example #2 - Writing data

This example is similar to example #1, but we will be writing data to the hard disk. It requires low memory usage. As with the first example, we will be required to use 3 procedures. pata_hd_start_write(), pata_hd_data_byte and pata_hd_stop_write()

Here are the steps:

1. Start writing at a sector address. I choose sector 20 since it seems that it will not mess up a fat32 formatted drive, I could be wrong!
2. Loop many times while you write your data. In this example, I am writing to 1 sector + 1/2 sector. The 2nd half of sector 2 will contain all 0's. The end of sector 2 will contain 0's because hard disks will only write data in blocks of 512, and therefore any data you have there will be overwritten.
3. Write some data. This time we are setting the value of the pseudo variable pata_hd_data_byte. Writing to this variable will actually send data to the hard disk. We are sending "A", so you will expect to read back the same data later on.
4. End your loop
5. Tell the hard disk we are done writing. The hard disk light will go out at this step.

```
pata_hd_start_write(20)    -- get sd card ready for write at sector 20
for 512 + 256 loop         -- loop 1 sector + 1 half sector (512 + 256 bytes)
  pata_hd_data_byte = "A"  -- write 1 bytes of data
```

```
end loop
pata_hd_stop_write()       -- tell sd card you are done reading
```

Now of course you will want to read your data back, which will be the same as in example #1, but at sector 20.

```
pata_hd_start_read(20)          -- get sd card ready for read at sector 20
for 512 + 256 loop         -- loop 1 sector + 1 half sector (512 + 256 bytes)
  data = pata_hd_data_byte       -- read 1 bytes of data
  serial_hw_write(data)    -- send byte via serial port
end loop
pata_hd_stop_read()             -- tell sd card you are done reading
```

Example #3 - Read and write data using a sector buffer (a 512 byte array)

In this example, we will use a 512 byte array for reading and writing. This 512 byte array is called a sector buffer. This method is very fast, however it will require a PIC that can fit the 512 bytes of data in it's ram space. I find it is also easier to use. I suggest PIC18f4620 with the same schematic.

For writing, You will need only need to write data to the sector buffer array, then use the pata_hd_write_sector_address() procedure.

Lets go through the steps, first for writing data:

1. Loop 512 times (the size of the sector buffer)
2. Set each data byte in the array
3. End your loop
4. Write the data to the hard disk at a sector address.
5. Repeat the above to write more sectors.

```
-- fill the sector buffer with data
for 512 using step1 loop                    -- loop till the end of the sector buffer
   pata_hd_sector_buffer[step1] = "B"       -- set each byte of data
end loop
-- write the sector buffer to sector 20
pata_hd_write_sector_address(20)
```

Here we will write another sector (to sector 21, the next sector)

```
for 512 using step1 loop                    -- loop till the end of the sector buffer
   pata_hd_sector_buffer[step1] = "C"       -- set each byte of data
end loop
-- write the sector buffer to sector 21
pata_hd_write_sector_address(21)
```

OK, it's time to read back the data, which is exactly the opposite of writing. For reading, we will use the pata_read_sector_addres() procedure first, then we can read data from the sector buffer array.

1. Request data from the hard disk at a sector address.
2. Loop 512 times (the size of the sector buffer)
3. Send each byte to the serial port.
4. End your loop.
5. Repeat the above to read more sectors.

```
-- read back the same sectors
-- read sector 20 into the sector buffer
pata_hd_read_sector_address(20)
-- now send it to the serial port
for 512 using step1 loop                    -- loop till the end of the sector buffer
   serial_hw_write(pata_hd_sector_buffer[step1]) -- send each byte via serial port
end loop
```

Here we will repeat the above to read the next sector (sector 21)

```
-- read sector 21 into the sector buffer
pata_hd_read_sector_address(21)
-- now send it to the serial port
```

```
for 512 using step1 loop                    -- loop till the end of the sector buffer
   serial_hw_write(pata_hd_sector_buffer[step1]) -- send each byte via serial port
end loop
```

EXAMPLE #4 - Another method for reading and writing sectors

Example #4 is pretty straight forward. I am not going to go into too much detail on this one. It is a combination of examples 2 and 3. It is about the same speed as example #3.

```
-- get sd card ready for write at sector 20
pata_hd_start_write(20)
-- fill the sector buffer with data
for 512 using step1 loop                  -- loop till the end of the sector buffer
  pata_hd_sector_buffer[step1] = "D"      -- set each byte of data
end loop
-- write the sector buffer to the sd card
pata_hd_write_sector()
-- fill the sector buffer with new data
for 512 using step1 loop                  -- loop till the end of the sector buffer
  pata_hd_sector_buffer[step1] = "E"      -- set each byte of data
end loop
-- write the sector buffer to the sd card
pata_hd_write_sector()                    -- write the buffer to the sd card
-- tell sd card you are done writing
pata_hd_stop_write()
--
-- read back both of the same sectors
-- get sd card ready for read at sector 20
pata_hd_start_read(20)
-- read the sector into the sector buffer
pata_hd_read_sector()
-- now send it to the serial port
for 512 using step1 loop                  -- loop till the end of the sector buffer
  serial_hw_write(pata_hd_sector_buffer[step1]) -- send each byte via serial port
end loop
-- read the next sector into the sector buffer
pata_hd_read_sector()
-- now send it to the serial port
for 512 using step1 loop                  -- loop till the end of the sector buffer
  serial_hw_write(pata_hd_sector_buffer[step1]) -- send each byte via serial port
end loop
pata_hd_stop_read()                         -- tell sd card you are done reading
```

EXAMPLE #5 - Sending a command to the hard disk (Spin Up/ Spin Down)

Hard drives have other features that may be useful. In this short example, I will show how to turn on and off the hard disk motor.

To turn on/off the hard disk motor, you will be writing to the "command register", and you will be sending the "spin down" command. If you browse through the hard disk library file pata_hard_disk.jal, you will see some constants that you may use for other commands. For more information, you can read "connecting ide drives by tilmann reh" at http://www.gaby.de/gide/IDE-TCJ.txt

With this "spin down" command, you will actually hear the hard drive motor turn off

```
pata_hd_register_write (PATA_HD_COMMAND_REG,PATA_HD_SPIN_DOWN)    -- turn off motor
```

Now give some delay.

```
_usec_delay(5_000_000) -- 5 sec delay
```

Then of course turn the drive motor back on

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

EXAMPLE #6 - 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 in "connecting ide drives by tilmann reh" at http://www.gaby.de/gide/IDE-TCJ.txt

You will have to follow these steps to receive this drive information:

1. Send the "Identify Drive" command to the command register
2. Wait till the hard drive is ready for you to read it
3. read the data, and do something with it (send it to the serial port)

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

-- 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)

-- Read 512 bytes
for 512 loop                            -- 256 words, 512 bytes per sector
   data = pata_hd_data_byte
   serial_hw_data = data
end loop                                -- drive info high/low bytes are in reverse order
```

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!

IR Ranger with Sharp GP2D02

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.

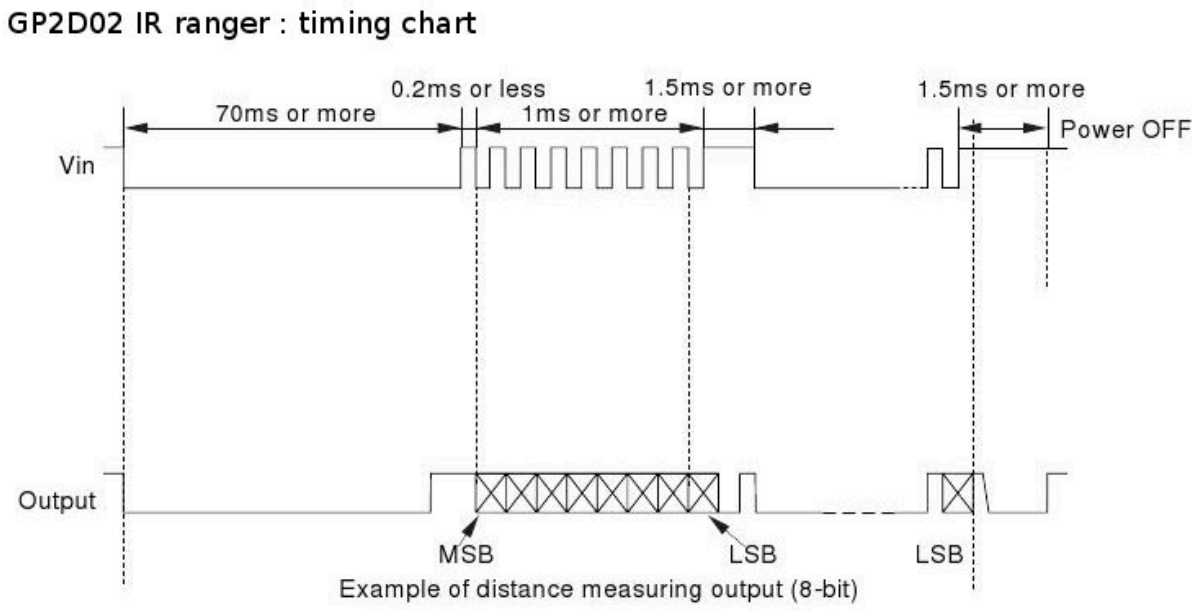
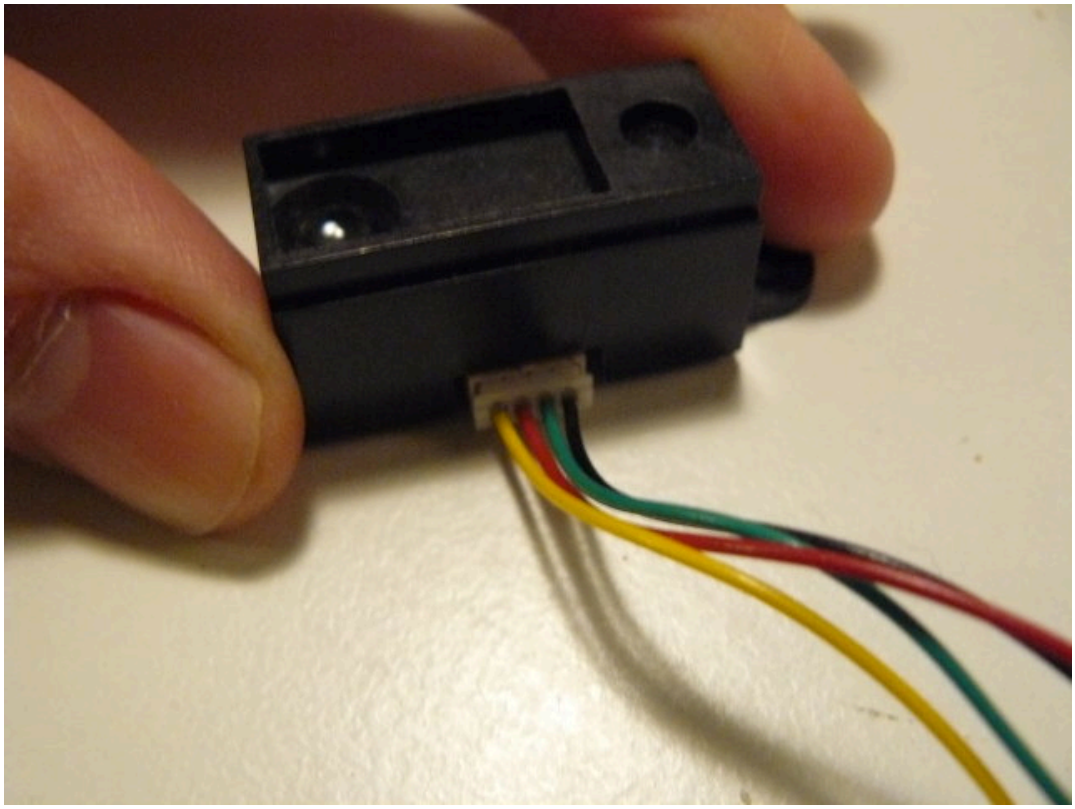


Figure 7: GD2D02 IR ranger : timing chart

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.

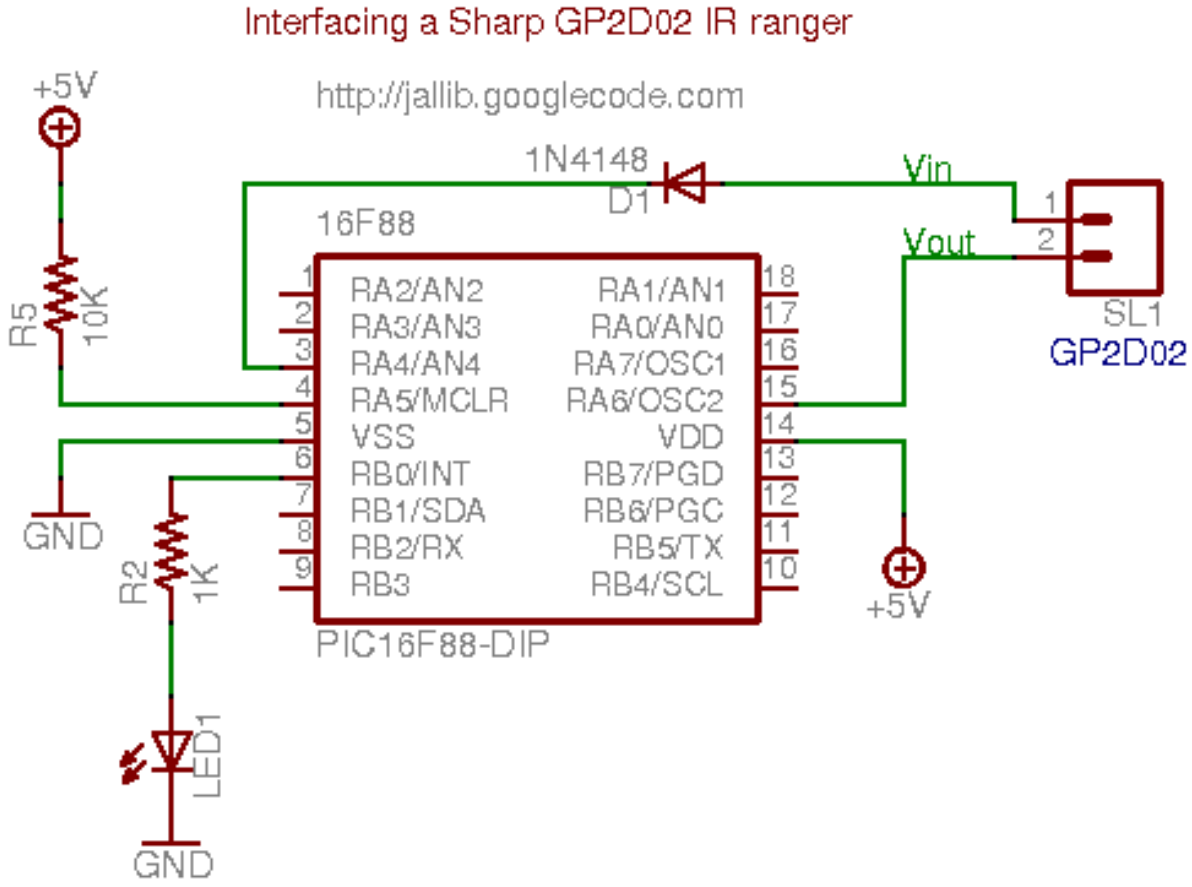
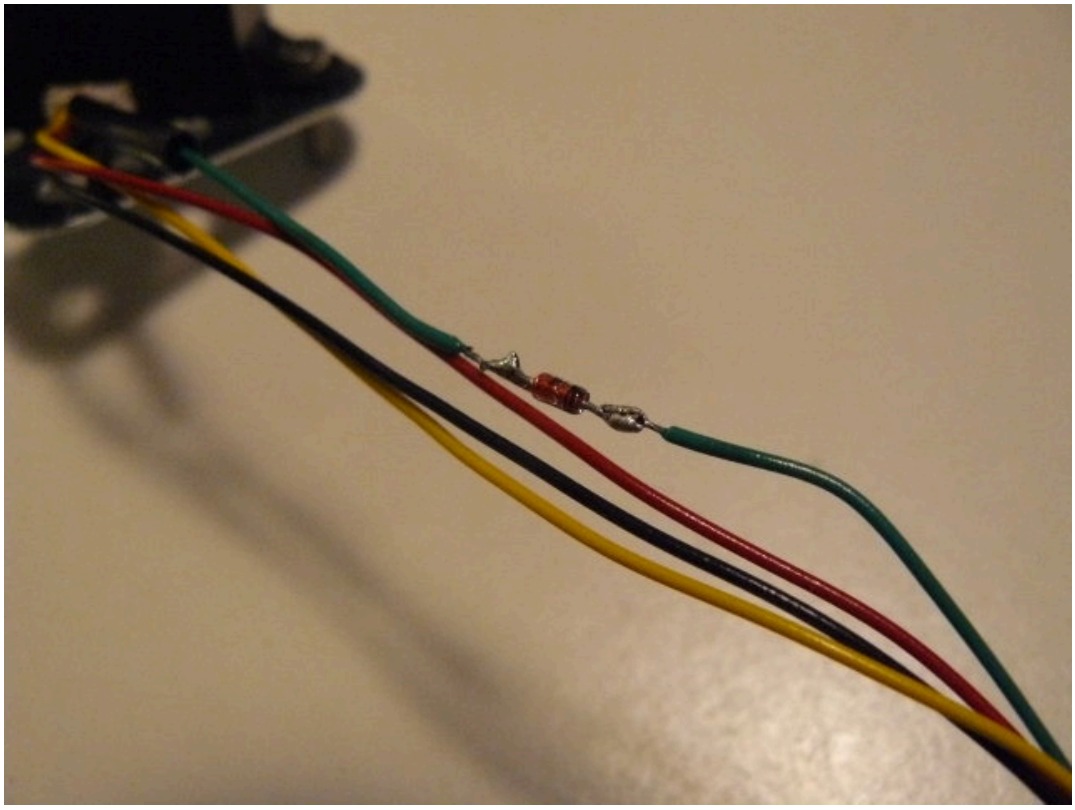
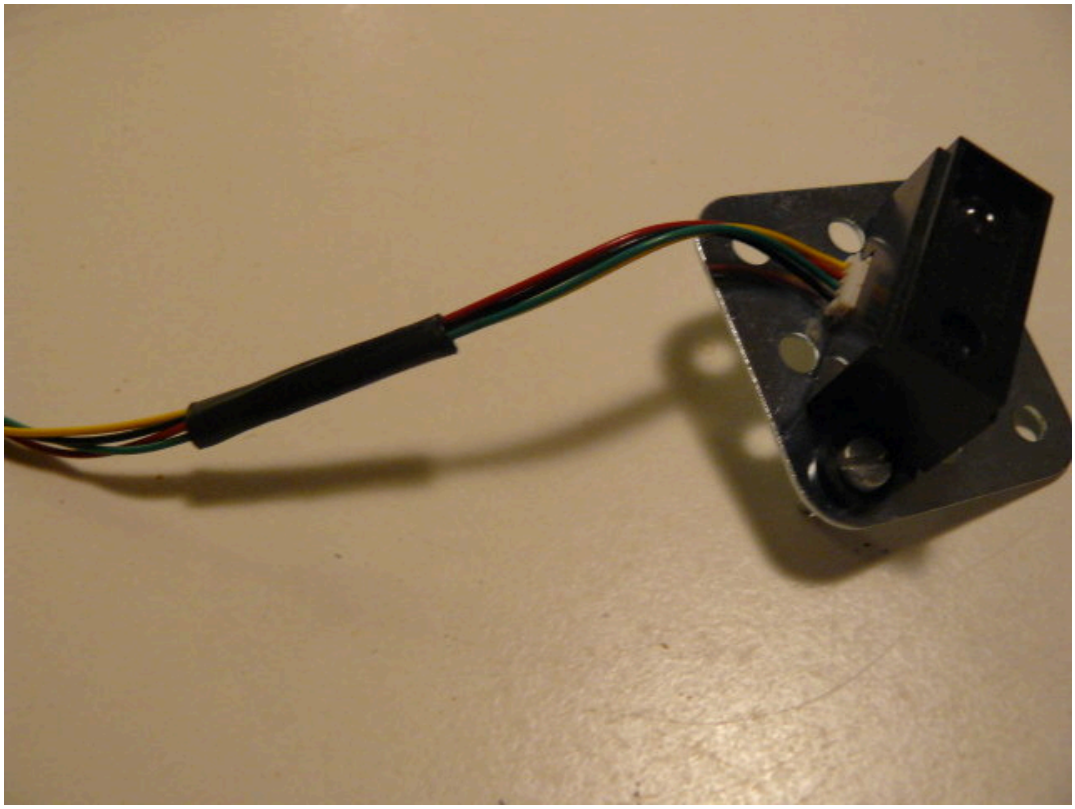


Figure 8: Interfacing Sharp GP2D02 IR range : schematic

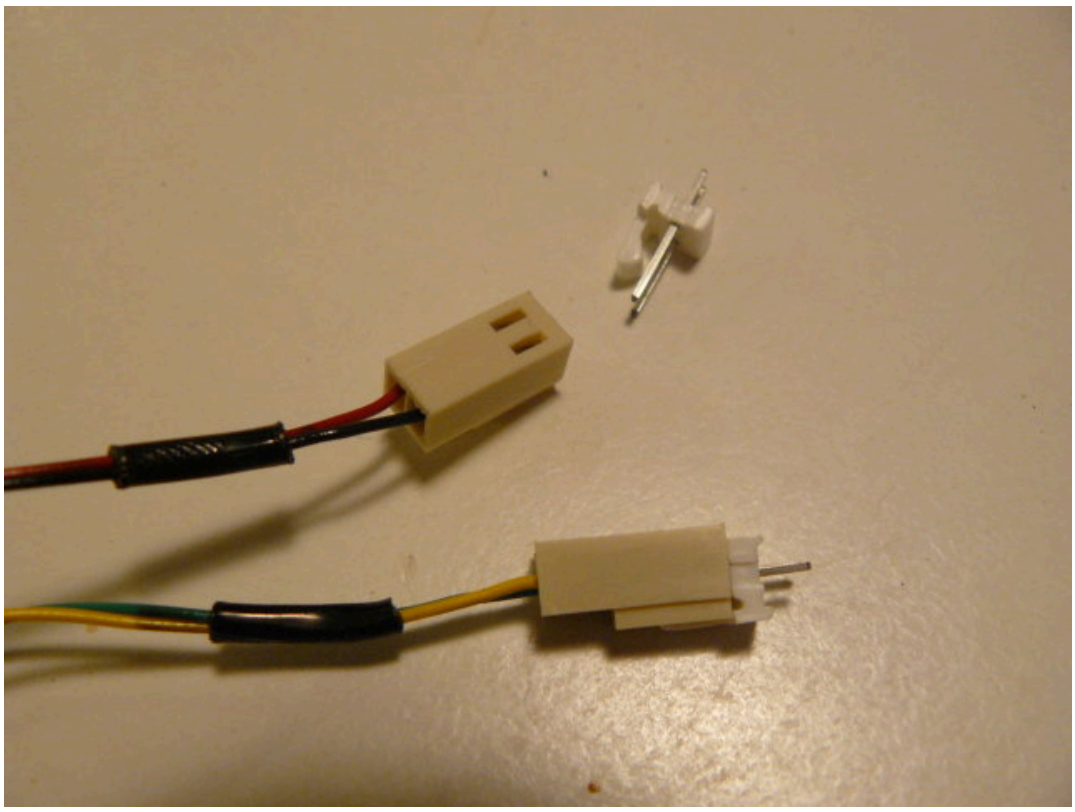
Here's the ranger with the diode soldered on the green wire (which is Vin pin, using your previously created mental note...):



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 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

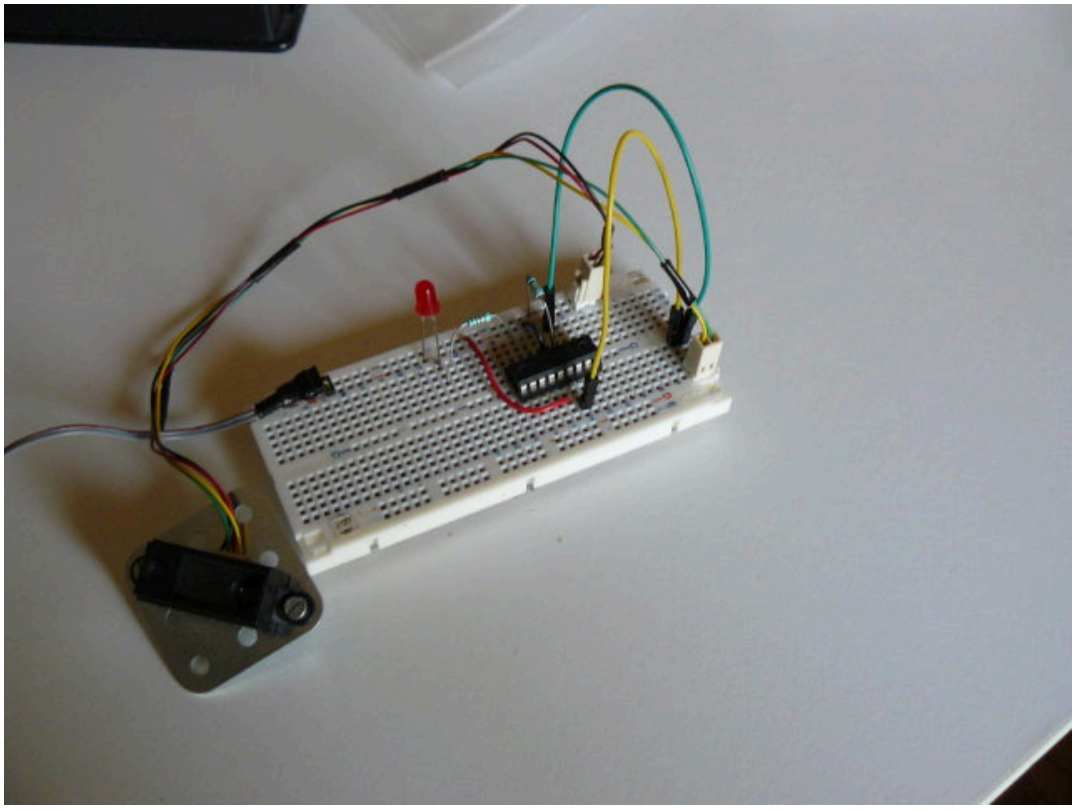
var byte measure
forever loop
   -- read distance from ranger num. 0
   measure = gp2d02_read_pins(gp2d02_vin,gp2d02_vout)
   -- results via serial
   serial_hw_write(measure)
   -- now blink more or less
   pwm1_set_dutycycle(measure)
end loop
```

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

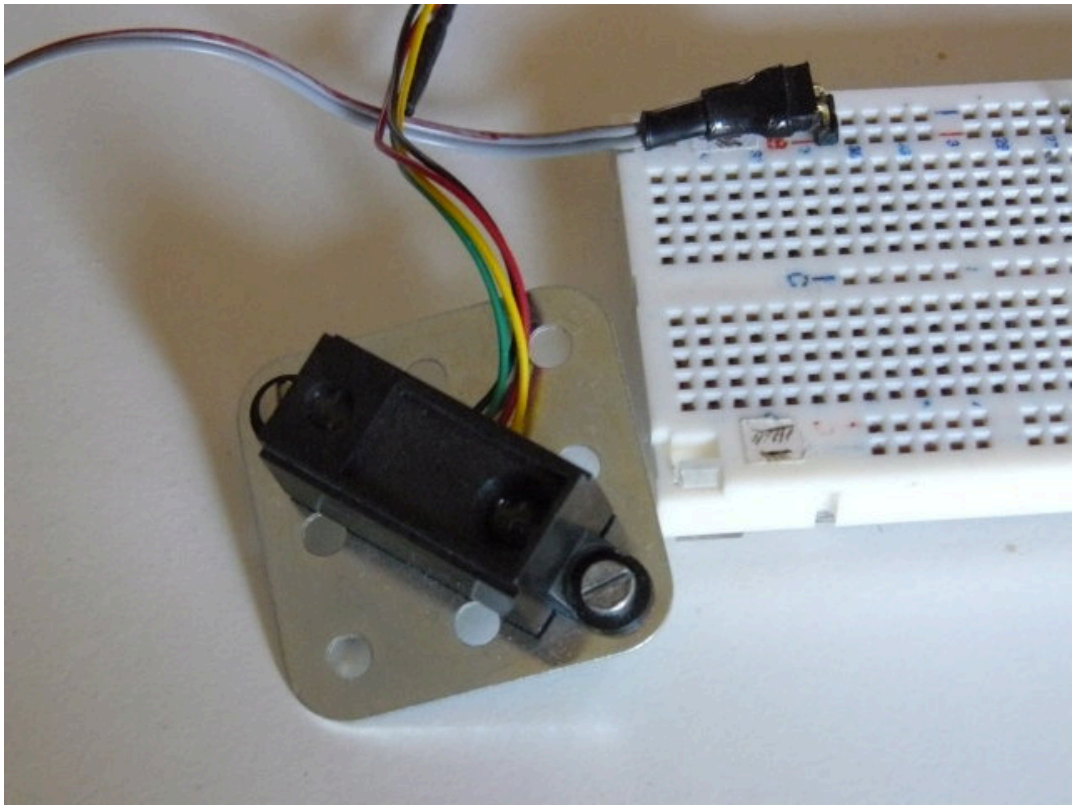
A sample file 16f88_ir_ranger_gp2d02.jal, is available in the sample directory of the jallib released packages.

Building the circuit on a breadboard

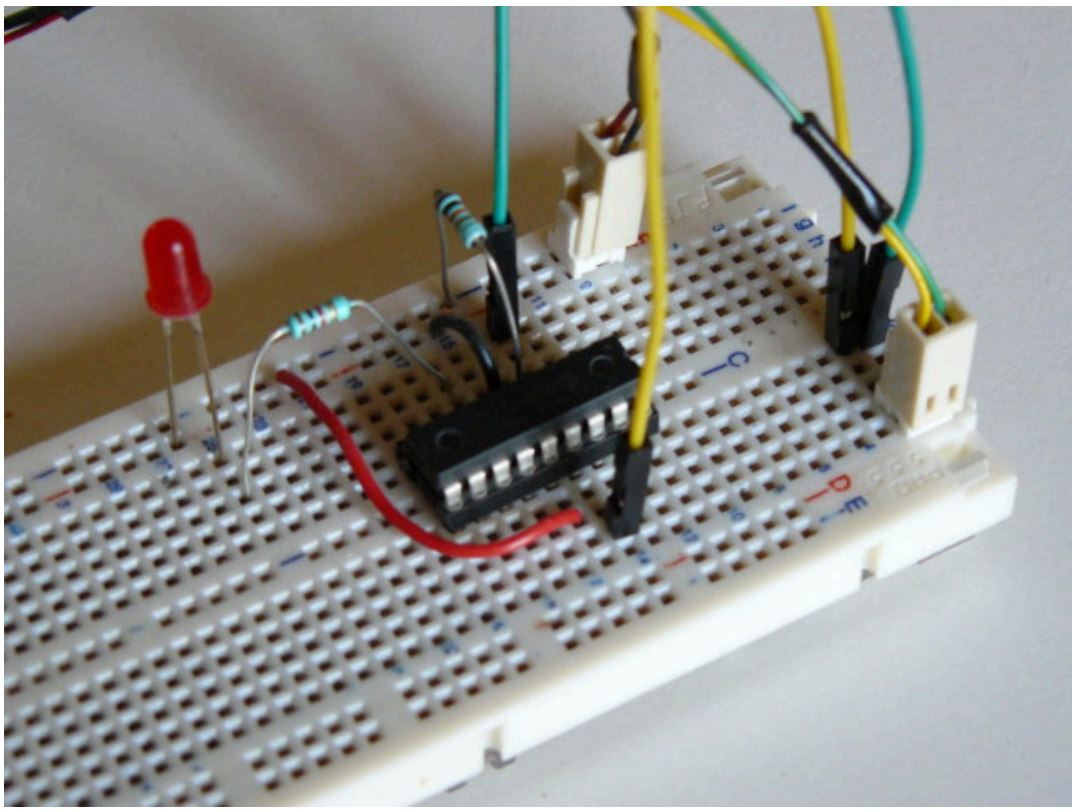
Building the circuit using a breadboard



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

LCD Display - HD44780-compatible

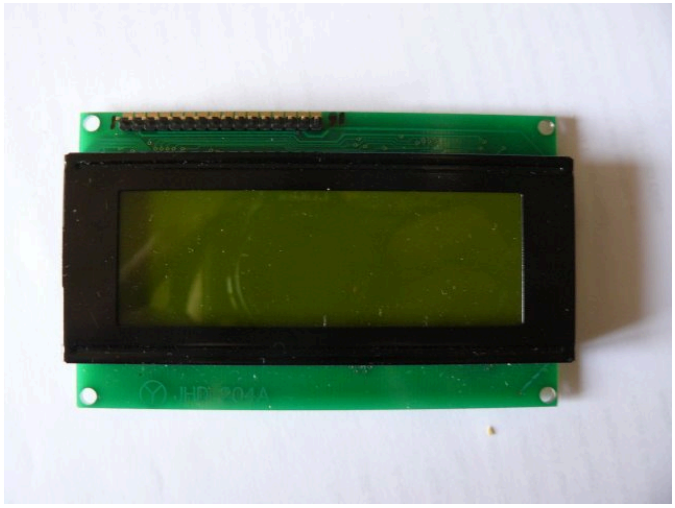
In this "Step by Step" tutorial, we're going to (hopefully) have some fun with a LCD display. Particularly one compatible with HD44780 specifications, which seems to be most widely used.

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...

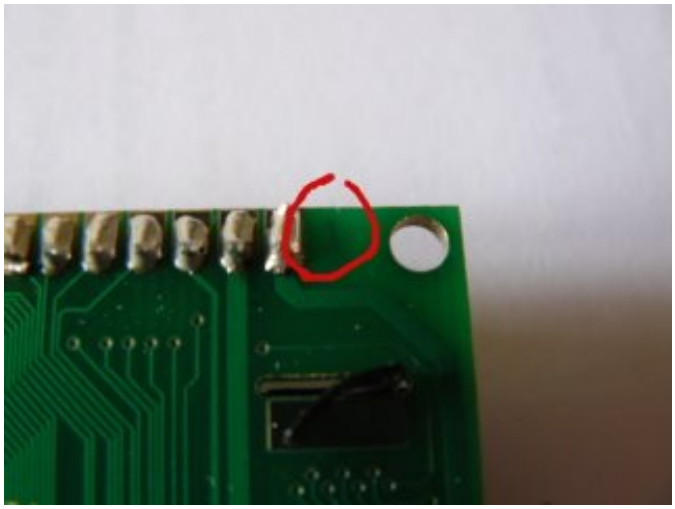
So, I've never used LCD, to be honest. Most guys consider it as an absolute minimum thing to have, since it can help a lot when debugging, by printing messages. I tend to use serial for this... Anyway, I've been given a LCD, so I can't resist anymore :)

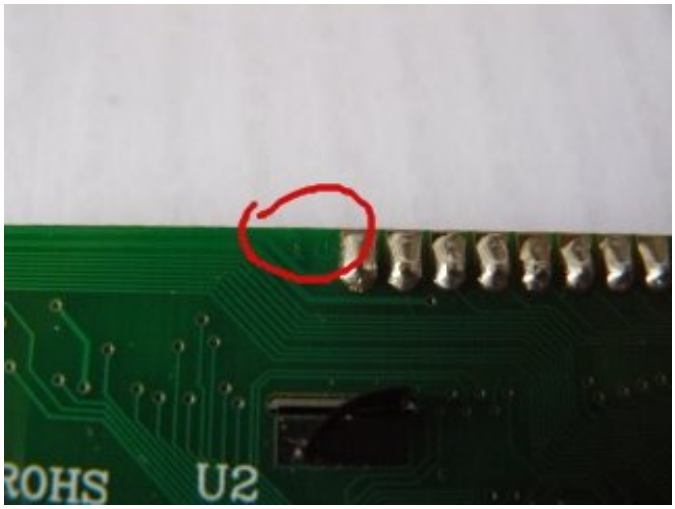
The LCD I have seems to be quite a nice beast ! It has 4 lines, is 20 characters long.



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**

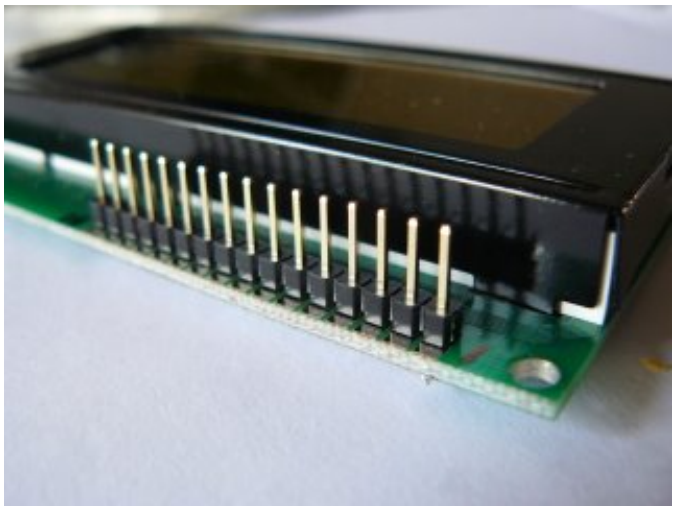
Pin NO.	Symbol	Function		Remark
1	GND	Power supply	0V	
2	Vdd		+5V	
3	V5		For LCD	Variable
4	RS	Register Select(H=Data,L=Instruction)		
5	R/W	Read/Write L=MPU to LCM,H=LCM to MPU		
6	E	Enable		
7	DB0	Data bus bit 0		
8	DB1	Data bus bit 1		
9	DB2	Data bus bit 2		
10	DB3	Data bus bit 3		
11	DB4	Data bus 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:

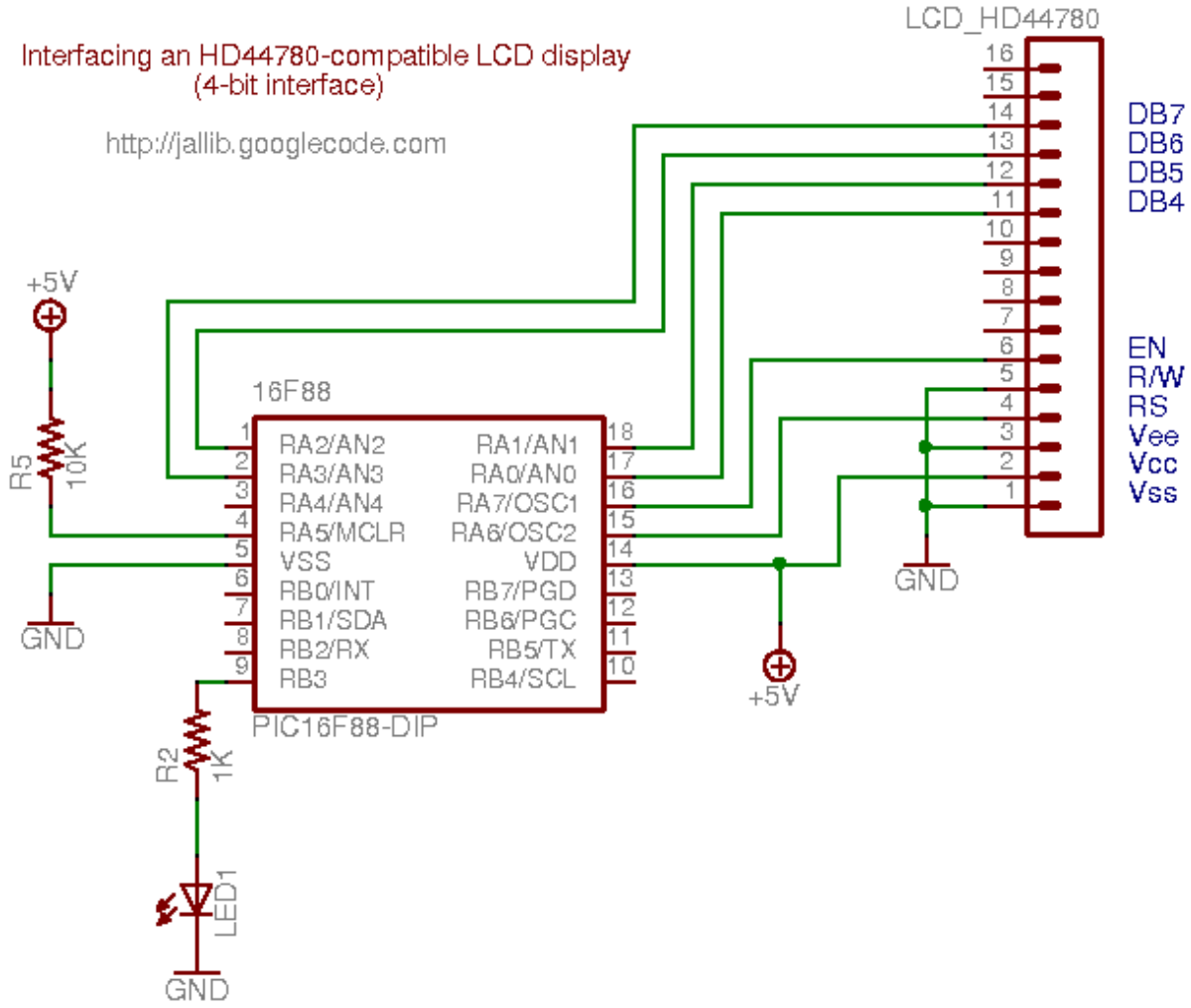
- as previously said, we'll use 4-bit interface. This means we'll use DB4, DB5, DB6 and DB7 pins (respectively pin 11, 12, 13 and 14).
- we won't read from LCD, so R/W pin must be grounded (pin 5)
- we won't use contrast as well, V5 pin (or Vee) must be grounded (pin 3)

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).

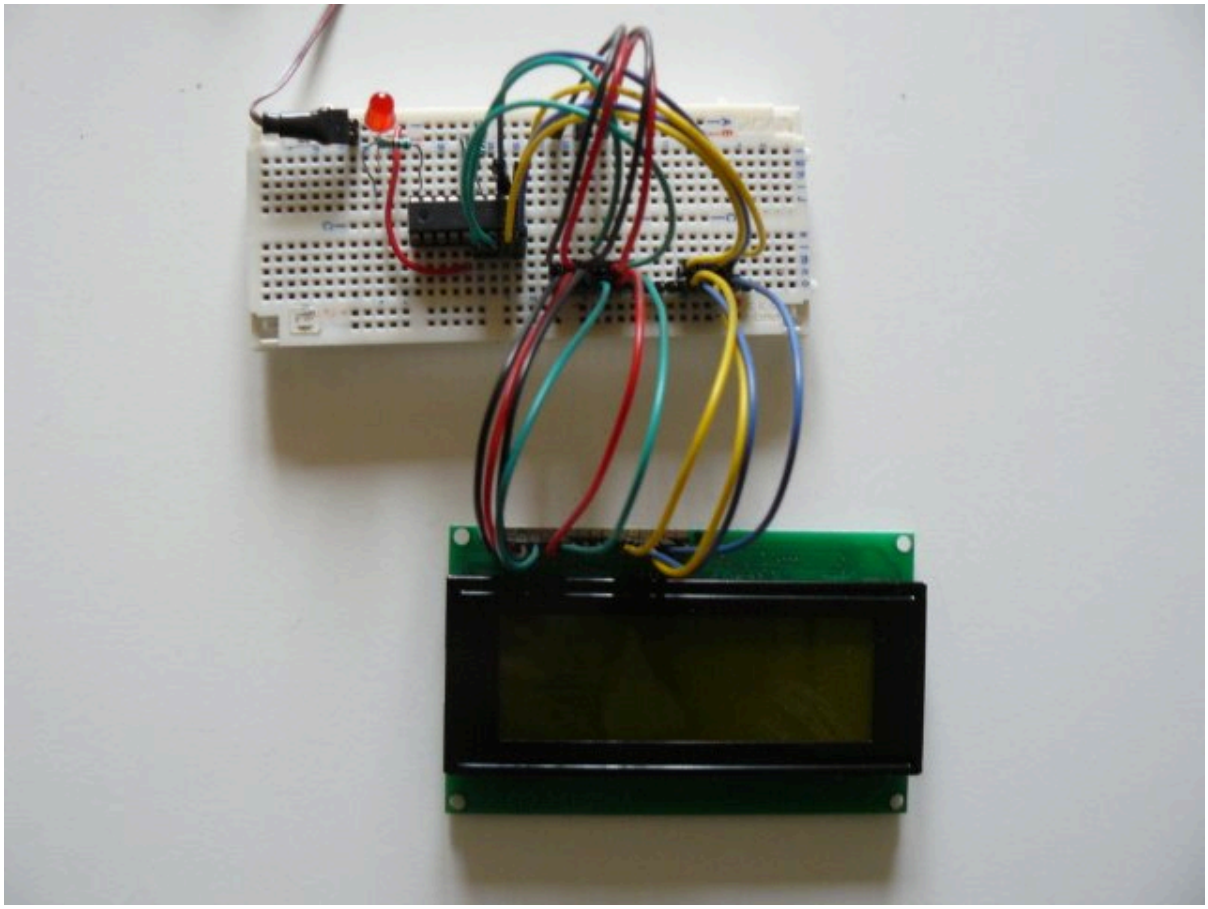
For convenience, I soldered a male connector on the LCD. This will help when building the whole on a breadboard.

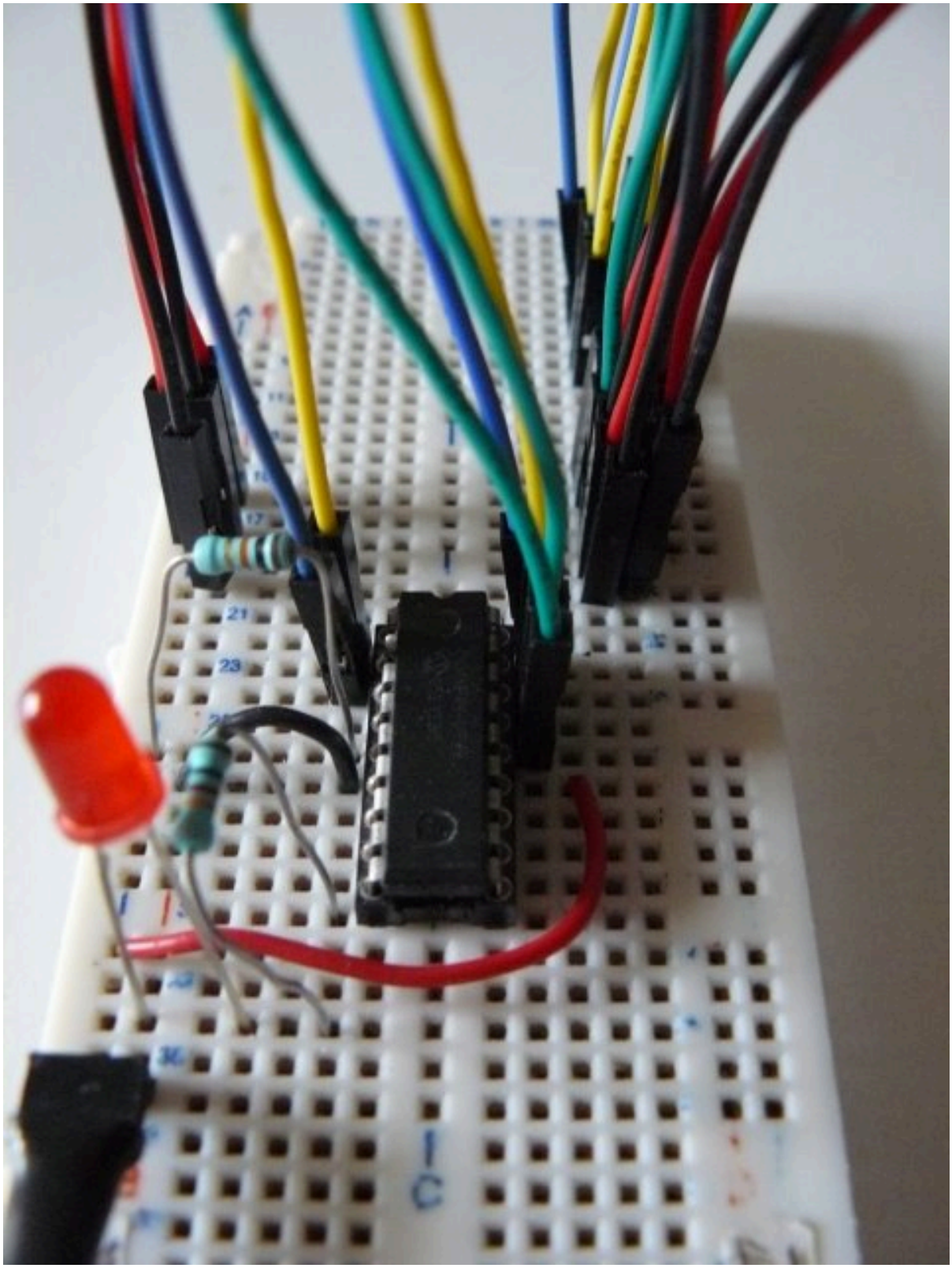


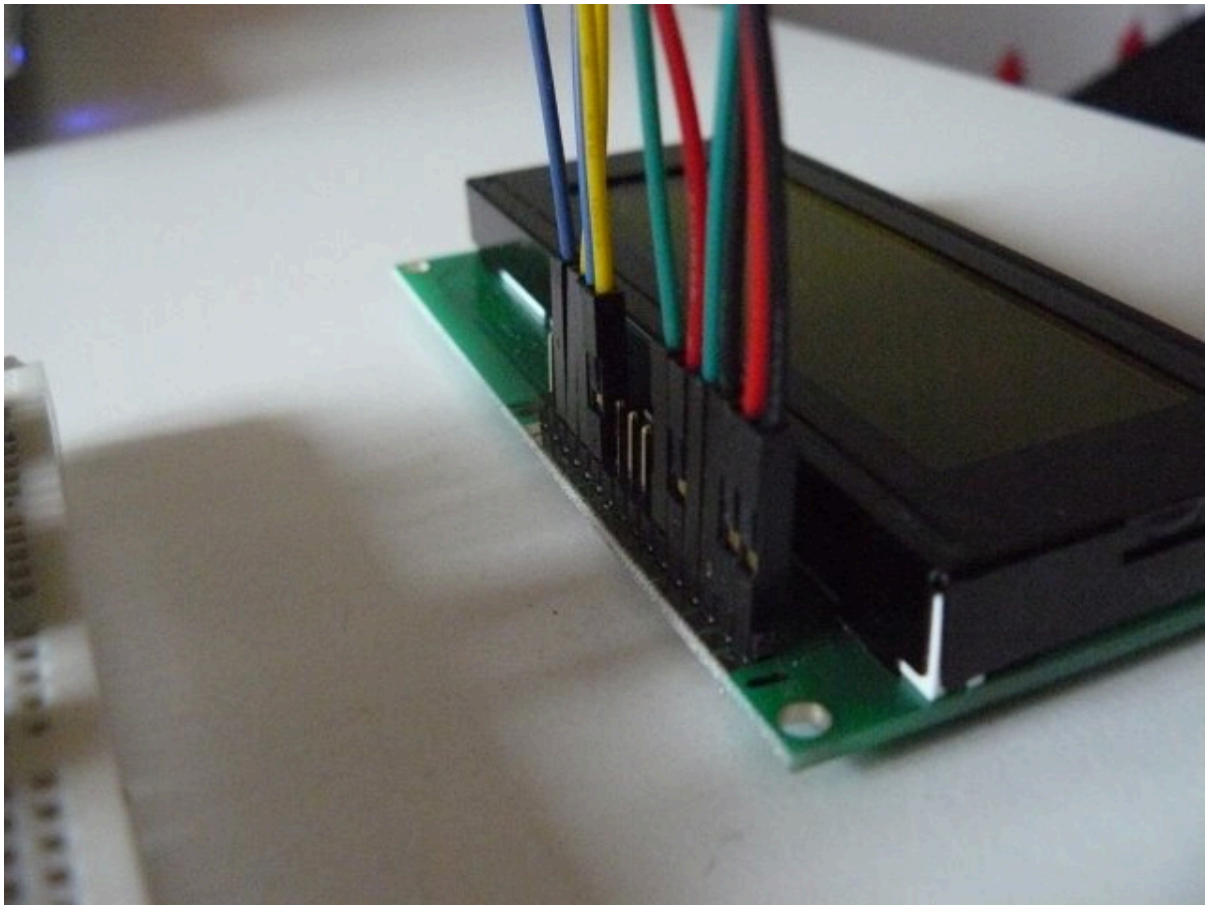
So we now have everything to build the circuit. Here's the schematic. It also includes a LED, it will help us checking everything is ok while powering up the board.



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 in the jallb release.

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

You'll need latest 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

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:

https://www.microchip.com/en-us/product/23K256

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

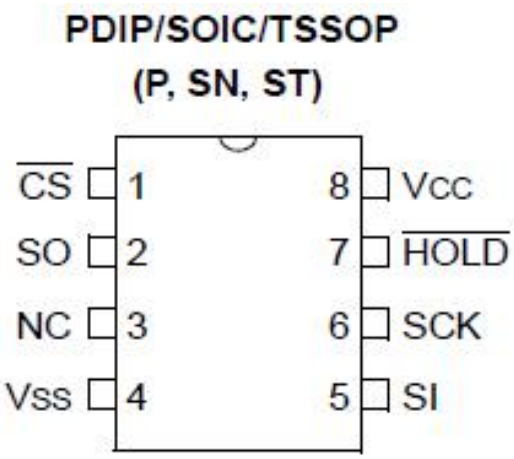
I suggest you start by compiling and writing the sample file to your PIC. We must make sure your circuit is working before we continue. As always, you will find the 23k256 sample file in the sample directory of your jallib installation "16f877_23k256.jal"

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.

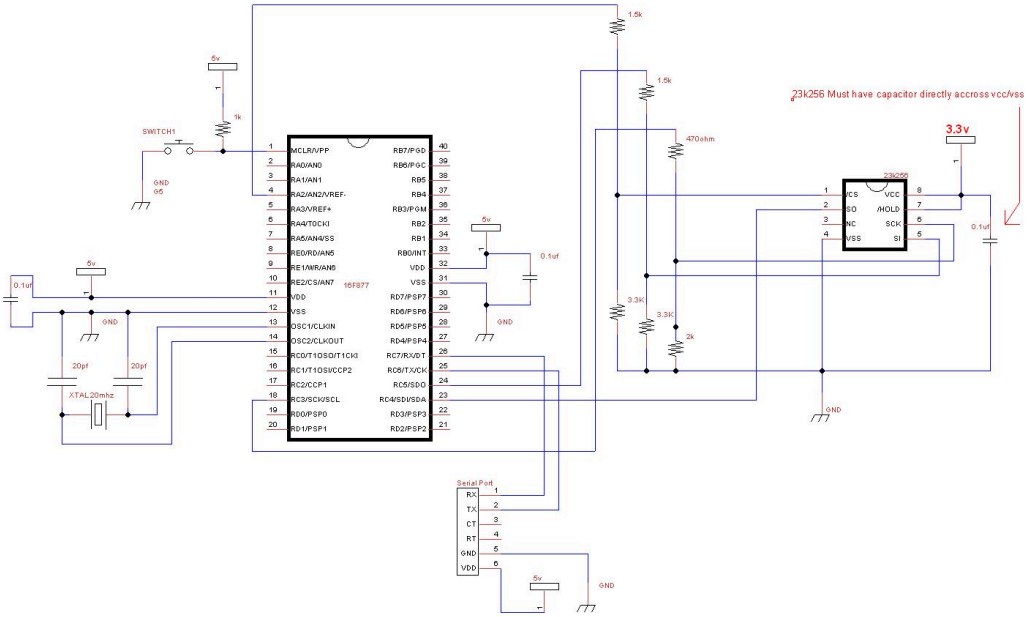
```
const bit SRAM_23K256_ALWAYS_SET_SPI_MODE = TRUE
```

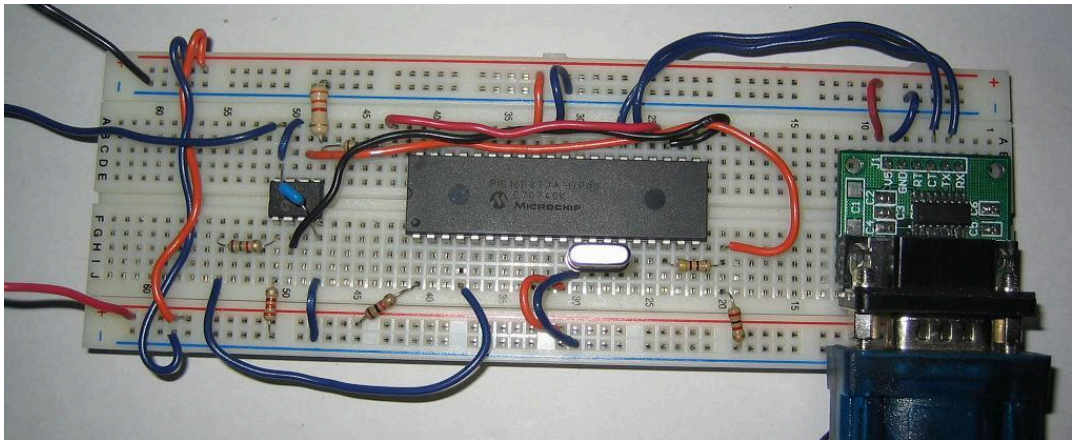
Here's the 23K256 pin-out diagram:



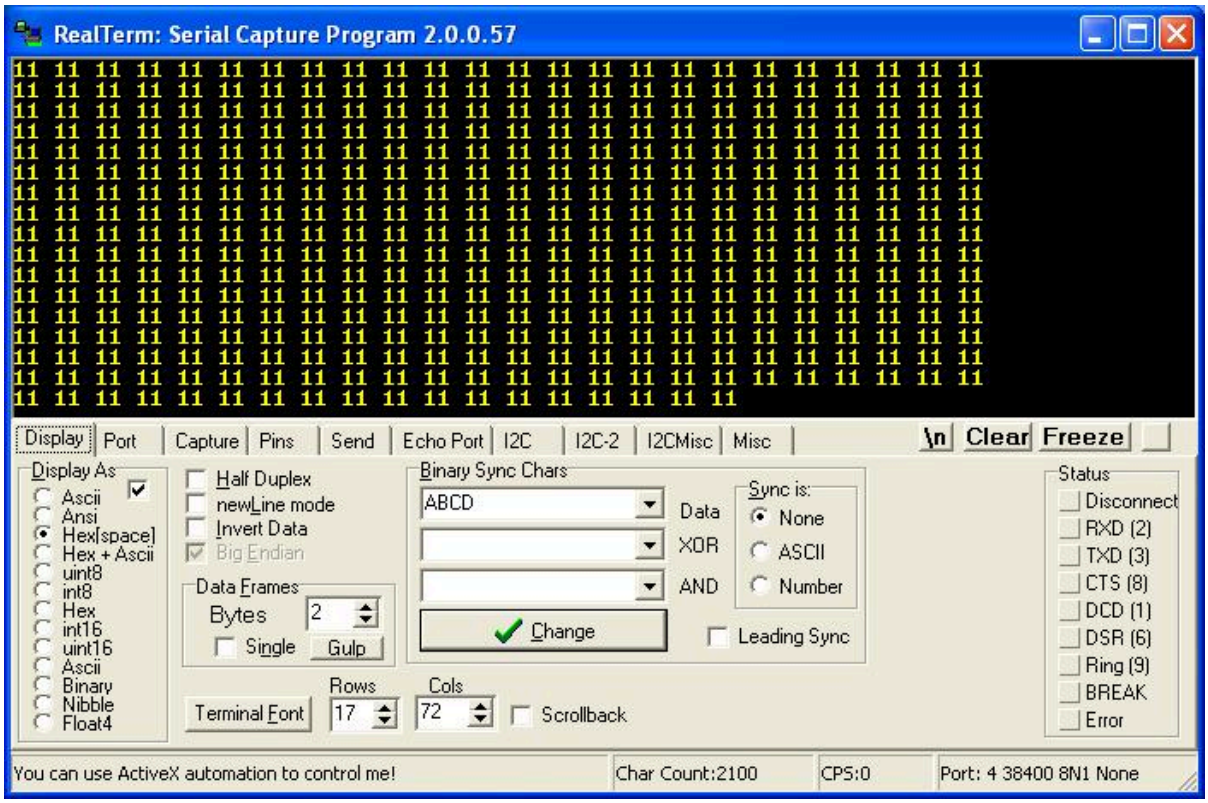
Now build this schematics. Notice the resistors for 5v to 3.3v conversion. Also notice that we are using the PIC's hardware SPI port pins. These pins are named (SDO, SDI, SCK) + One chip select pin of your choice.

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 on page 93 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 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
-- 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
-- init 23k256 in sequential mode
sram_23k256_init(SRAM_23K256_SEQUENTIAL_MODE, SRAM_23K256_HOLD_DISABLE)
```

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!

RC Servo Control & RC Motor Speed Control

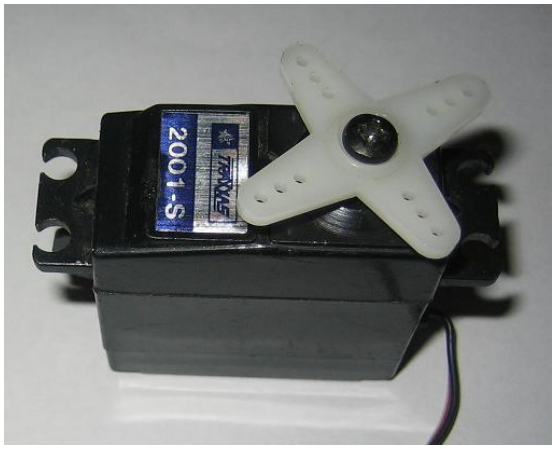
PIC RC servos and RC speed controllers used in the Radio Control hobby.

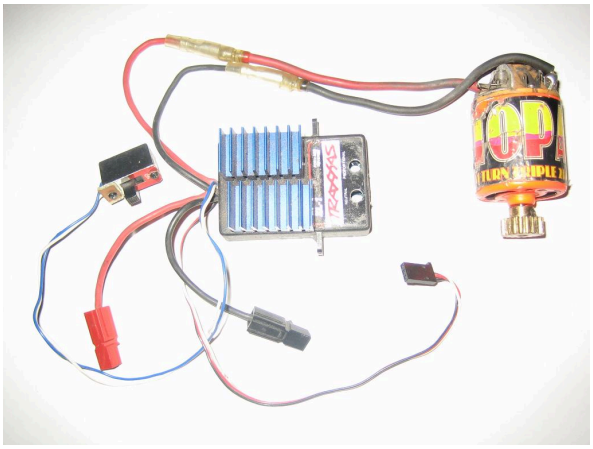
Servo Control Intro

RC or R/C (Radio Control) servos and RC motor speed controllers are used in the radio control hobby to control things like RC Cars, RC airplanes, boats, robots, etc.

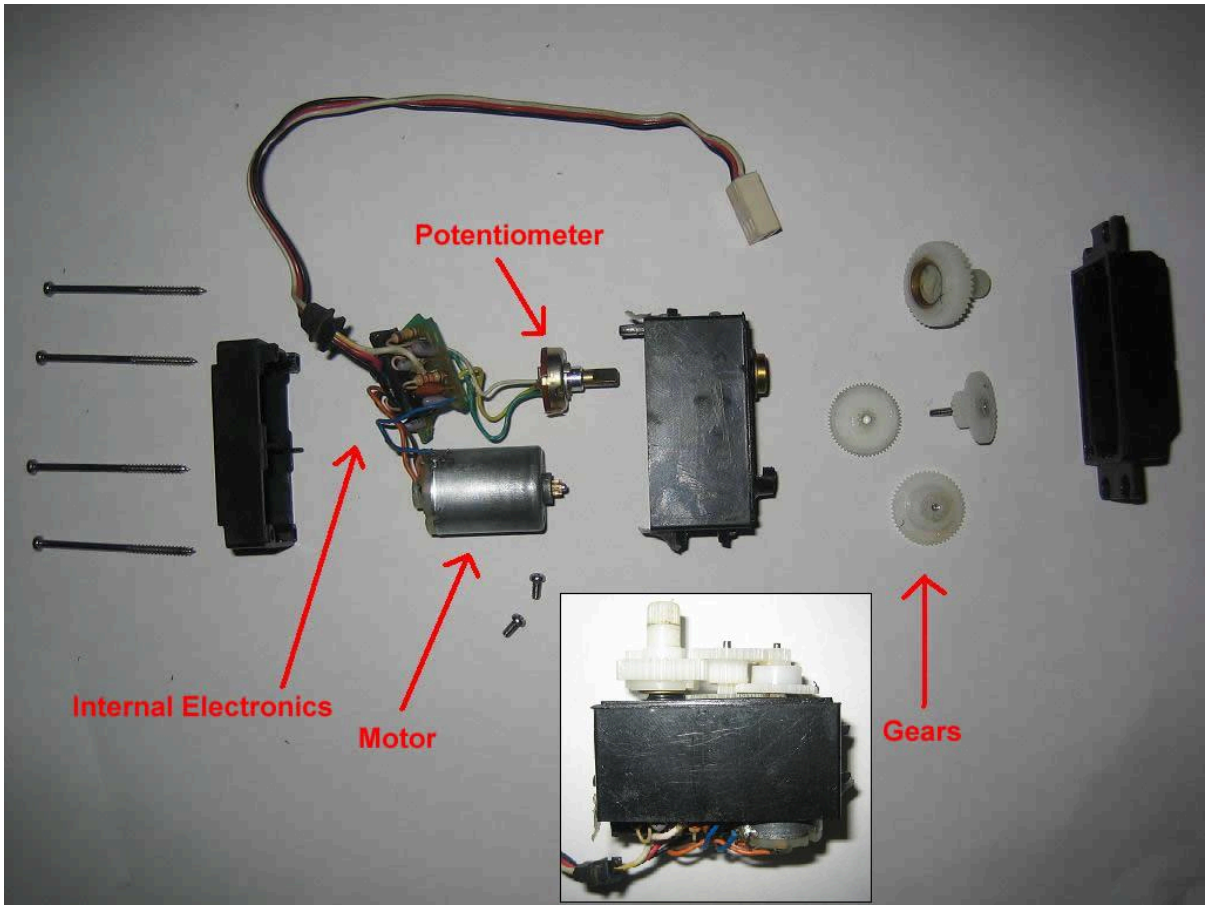
Servos are used for there positioning capability and strength. Small, regular sized servos can be bought at a local hobby store for $10 or less. Of course there are more expansive ones depending on the quality and size. These servos normally plug into your radio control receiver, but today we will connect it to your PIC.

I will mostly be talking about RC servos, but you will also be able to connect a RC speed controller since they use the same technology. These speed controllers are made up of power MOS FETs to allow 12v at 50A+ to control the speed of a motor via PWM.



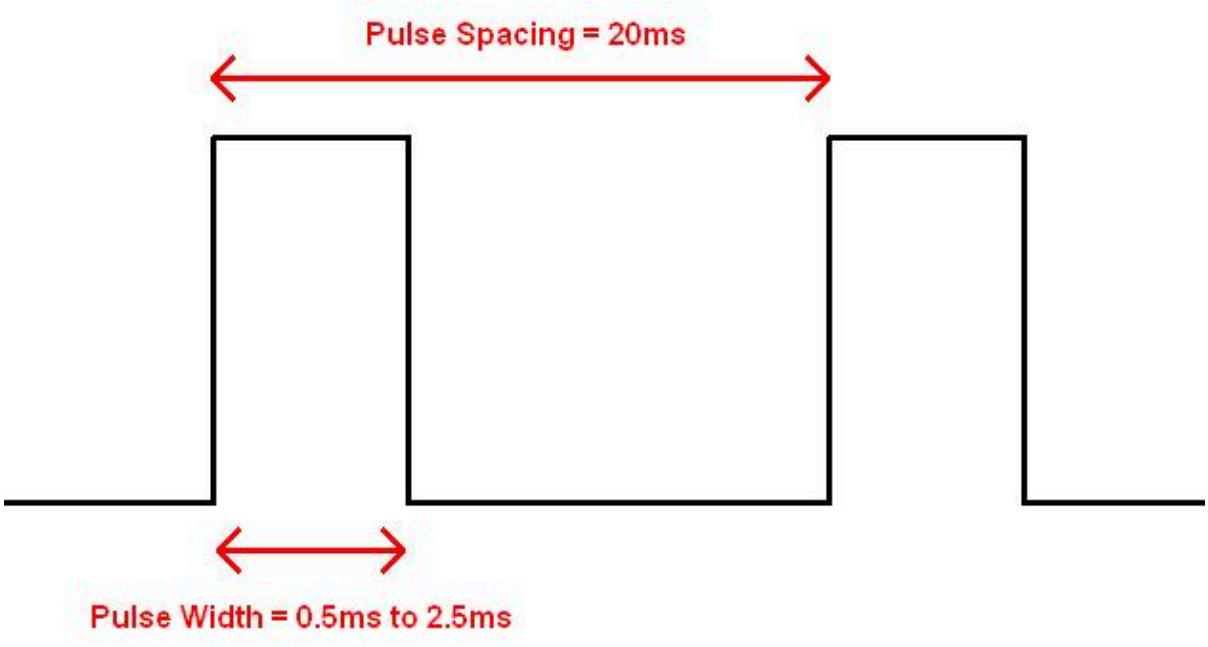


The only way to really know how something really works is to take it apart! I found some gears, a potentiometer some electronics and a motor. The servos gears are to give it the strength it needs to move whatever it is you want to move in your project. The servo knows it's position by reading a voltage off the potentiometer that gets turned by the gears which of course gets turned by the motor. After a signal is given, the servo will move to the correct location.



To control a servo, we need to send it a PWM (pulse width modulation) signal. Thankfully it will all be taken care of by the Jallib library I have created. A pulse will be sent every 20ms, and each pulse will be a width between 0.5ms and 2.5ms. The pulse width will vary depending on the position you have chosen. Servo pulse width required can

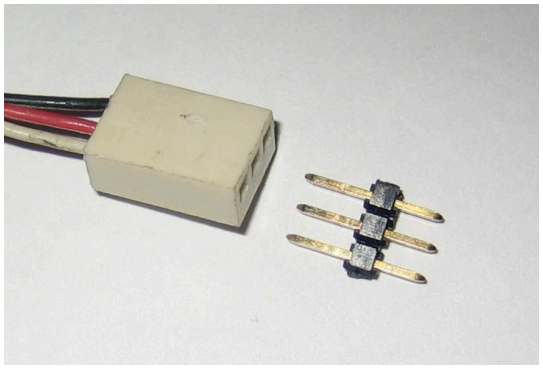
very depending on the servo manufacturer, therefore the library has been created with some default values that you may change to get a full movement from left to right.



Here's a YouTube video of one servo moving and it's signal on my oscilloscope: http://www.youtube.com/watch?v=zA3anG0YZD4

Servos come with a verity of connector types but always have 3 wires. One is ground, one is power and the other is signal.

Here's an image of my RC servo connector (left), I will use some pins (right), to plug my servo into my breadboard.



There are two ways of implementing servos into your project. You may either have your servos connected to your main PIC, or to an external PIC. For smaller projects you will choose to control your servo from your main PIC, this is the method I will show you.

If your main PIC is needed to run some heavy code, or if you need more then 24 servos, you may wish to use external PIC(s) via I2C interface. There is a library and two samples for using an external PIC via I2c bus. I will not discuss this method here.

Servo control via your main PIC

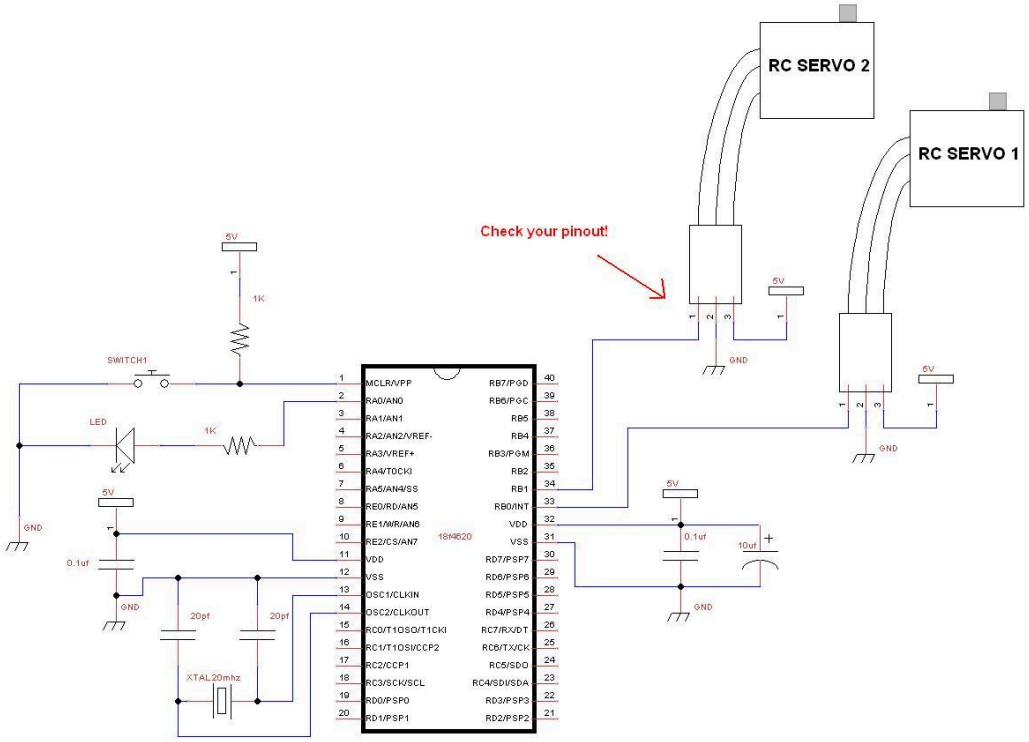
This method will allow you to plug up to 24 servos into your main PIC. Any digital capable I/O pin on your PIC should be able to run your servo signal, always lookup your pin in the datasheet. Use a pull up resistor on open drain pins. You will need to choose a PIC with a hardware timer, 8 servos can run on each timer module.

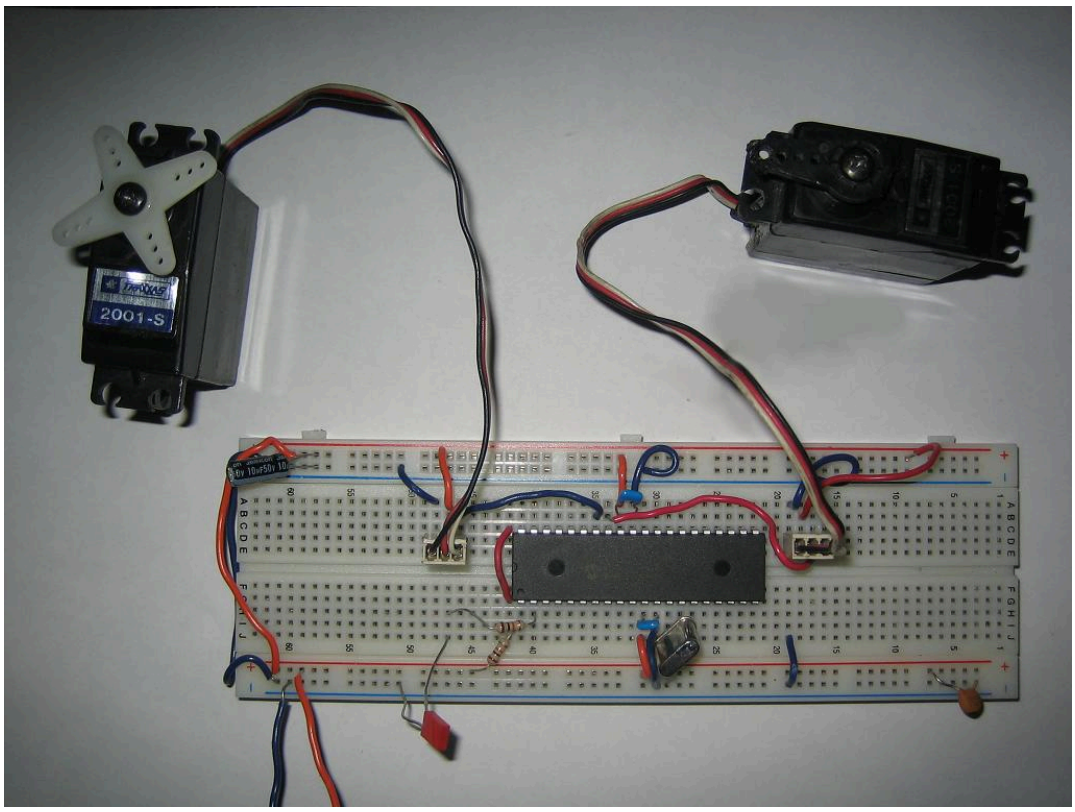
The library supports timer0, timer1 and timer3. You can do a quick search for "timer0 module" in your datasheet to see if you have a timer, most PICs do have at least one timer. The library will give you an error if you if you do not have a timer when you try to compile your code.

I have chosen 18F4620 (3 timers), but I have also tested it on 16f877a (2 timers), and 18f452(3 timers) with the same schematic.

Build your circuit

The schematic is very very simple, just take your blink circuit and plug in your servo.





The Code

Since your main PIC will be controlling the servos, the PIC is acting as a master device and therefore we will be using the servo_rc_master library.

The library can be found under Jallib SVN at trunk\include\external\motor\servo\servo_rc_master.jal

The sample can be found in the Jallib SVN at trunk\sample\18f4620_servo_rc_master.jal

You can access the Jallib dowload site at http://justanotherlanguage.org/downloads

Let's start by including your PIC, and disable all analog pins

```
-- include chip
include 18f4620                    -- target PICmicro
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
pragma target MCLR external        -- reset externally

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

This sample file will require 1 led, so define it now

```
-- led definition
alias led                is pin_a0
alias led_direction      is pin_a0_direction
--
led_direction = output
```

Now you may define the pins that will be used for each of your servos, I have defined 8 although I have not connected them all in my circuit.

```
-- setup servo pins
alias servo_1             is pin_b0
alias servo_1_direction   is pin_b0_direction
```

```
servo_1_direction = output
--
alias servo_2             is pin_b1
alias servo_2_direction   is pin_b1_direction
servo_2_direction = output
--
alias servo_3             is pin_b2
alias servo_3_direction   is pin_b2_direction
servo_3_direction = output
--
alias servo_4             is pin_b3
alias servo_4_direction   is pin_b3_direction
servo_4_direction = output
--
alias servo_5             is pin_b4
alias servo_5_direction   is pin_b4_direction
servo_5_direction = output
--
alias servo_6             is pin_b5
alias servo_6_direction   is pin_b5_direction
servo_6_direction = output
--
alias servo_7             is pin_b6
alias servo_7_direction   is pin_b6_direction
servo_7_direction = output
--
alias servo_8             is pin_b7
alias servo_8_direction   is pin_b7_direction
servo_8_direction = output
--
-- commenting out 9th servo
;alias servo_9             is pin_a0
;alias servo_9_direction   is pin_a0_direction
;servo_9_direction = output
```

Here we will define the min & mas movement. These values can be changed to limit the amount your servos can move. We will talk about this in detail later on, this is an important step. Changing these values will change the pulse width for all servos.

```
-- choose min & max servo movment / pulse size
const byte SERVO_MIN   = 50  -- default is 50  (0.5ms min pulse)
const byte SERVO_MAX   = 255 -- default is 255 (2.5ms max pulse)
```

Choose the timers your PIC will be using to control your servos. Each timer will take care of 8 servos. I have defined 8 servos, so I need only 1 timer. I have commented out the other 2 timers that I may use later on.

```
-- choose pic internal timers
const byte SERVO_USE_TIMER = 0            -- timer for servo's 1 to 8
;const byte SERVO_9_TO_16_USE_TIMER = 1    -- timer for servo's 9 to 16
;const byte SERVO_17_TO_24_USE_TIMER = 3   -- timer for servo's 17 to 24
```

I may now include the servo_rc_master library, and initialize the servos. Within the init() procedure, all servos will be centered.

```
include servo_rc_master -- include the servo library
servo_init()
```

If you wish to turn off a servo at any point in your program. This will turn off the servos motor by keeping the signal line low. You may set or unset the on/off bit for any servo as follows:

```
-- use this to turn off a servo
;servo_1_on = FALSE
```

Sometimes the servo you have may move in the opposite direction that you would like it to, so here you have an option of switching a servos direction. I have also noticed that some types of servos move in the reverse of others.

```
-- use this to reverse a servo
;servo_1_reverse = TRUE
```

The init procedure does center all servos, but you may want to start your servo at another location. Since I did not leave any delay yet, the servos did not actually have time to move to there center position.

I am going to center the servos again to show you an example of the correct way to move a servo. After I give the servos there move position, I will wait for 1sec so they have time to move to center.

You can use various delays or move in increments to slow the servo movement speed. 127 is center.

```
-- example center all servos
servo_move(127,1) -- center servo 1
servo_move(127,2) -- center servo 2
servo_move(127,3) -- center servo 3
servo_move(127,4) -- center servo 4
servo_move(127,5) -- center servo 5
servo_move(127,6) -- center servo 6
servo_move(127,7) -- center servo 7
servo_move(127,8) -- center servo 8
;servo_move(127,9)

_usec_delay (1_000_000) -- wait for servos to physically move
```

Now I will create my main loop and have 2 of the servos move to various positions.

```
-- example moving servos one and two and blink led
forever loop

   servo_move(255,1)
   servo_move(0,2)
   _usec_delay (1_000_000)
   led = !led

   servo_move(127,1) ;servo 1 centered
   servo_move(127,2) ;servo 2 centered
   _usec_delay (1_000_000)
   led = !led

   servo_move(0,1)
   servo_move(255,2)
   _usec_delay (1_000_000)
   led = !led

   servo_move(127,1) ;servo 1 centered
   servo_move(127,2) ;servo 2 centered
   _usec_delay (1_000_000)
   led = !led
end loop
```

So, that's it for the code. Simple right? I wish writing the library was that easy!

You can go ahead and turn on your circuit, you should see the led blink and the servos should be moving. Change and test your servo pinouts if needed. The two servos will move in opposite directions since my servo_move() procedure call values are different for each servo.

At this point, you should turn off the power when your servos are at the center position. We are turning the power off so you may remove the moving part on the top of your servo, and place it back on so it looks centered.

Here's a YouTube video of my two moving servos http://www.youtube.com/watch?v=QS8M07uuagY

Setting Your Servo Max & Min Movements

For my projects, I feel that it is important to set the servo min/max values. You may choose to either use the default values that I have set, or set your own. There are two reasons for setting these values correctly:

1. You can get more movement out of your servo (far right to far left)
2. You do not want your servo to try to move out of it's range. If your servo moves out of it's range for a long period of time, you may burn out the servos motor.

All manufacturers create there servos differently, there is no official specification for RC servos (that I know of).

So here are the steps:

1. Set SERVO_MIN = 0 and SERVO_MAX to 255
2. Set your servo_min values to restrict movement to one side. Directly after you call servo_init(), you should place this code:

```
servo_move(0,1)
forever loop
end loop
```

   This will move your servo all the way to one side. You will hear your servo motor being ON all the time (not good for the motor).
3. Now run your circuit and gradually increase servo_min value so the servo is at the correct location on one side. Try to get the servo to be 1mm from it's min location. You should not hear the motor running.
4. Repeat step 2 by decreasing servo_max but use this:

```
servo_move(255,1)
forever loop
end loop
```

Well, looks like your all set. I hope your having fun with Jalv2/Jallib!

SD Memory Cards

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

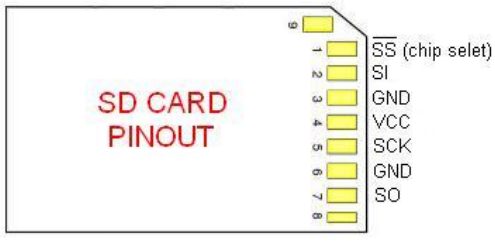
SD Card Introduction

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.

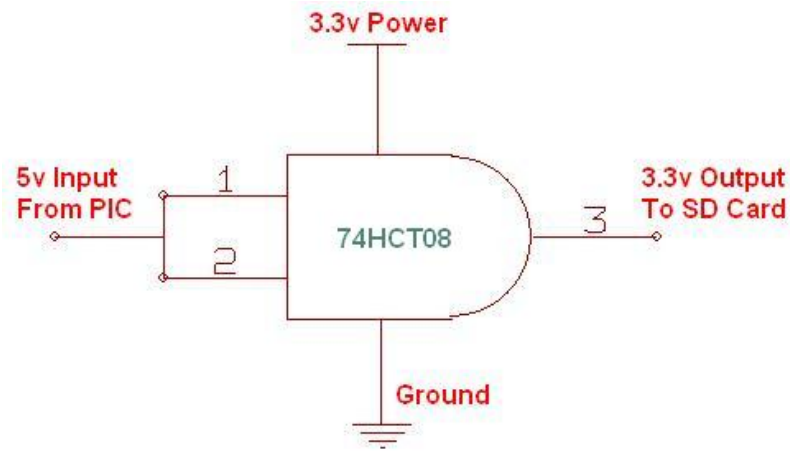
We are not responsible for your data or SD card. Make sure you have nothing important on your SD card before you continue.





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 https://ww1.microchip.com/downloads/en/DeviceDoc/chapter%208.pdf for more information.

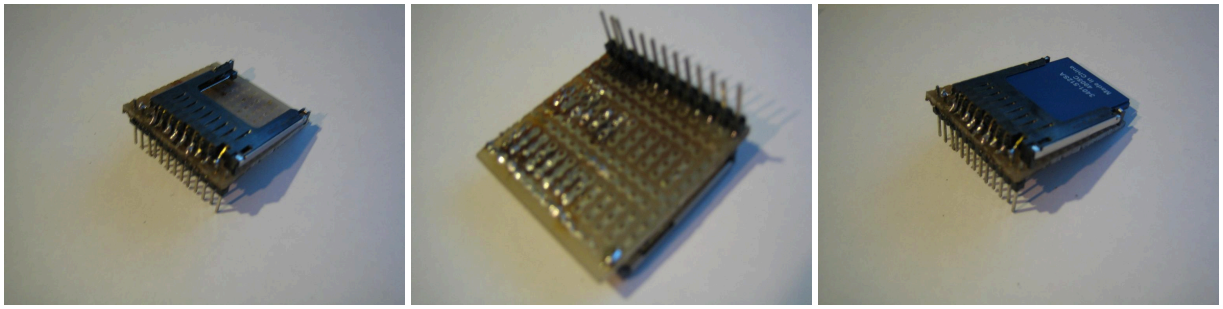
My favorite way of converting 5v to 3v is with 74HCT08. It must be HCT (not LS). 74LS08 will not work on a 3.3v power supply.



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.

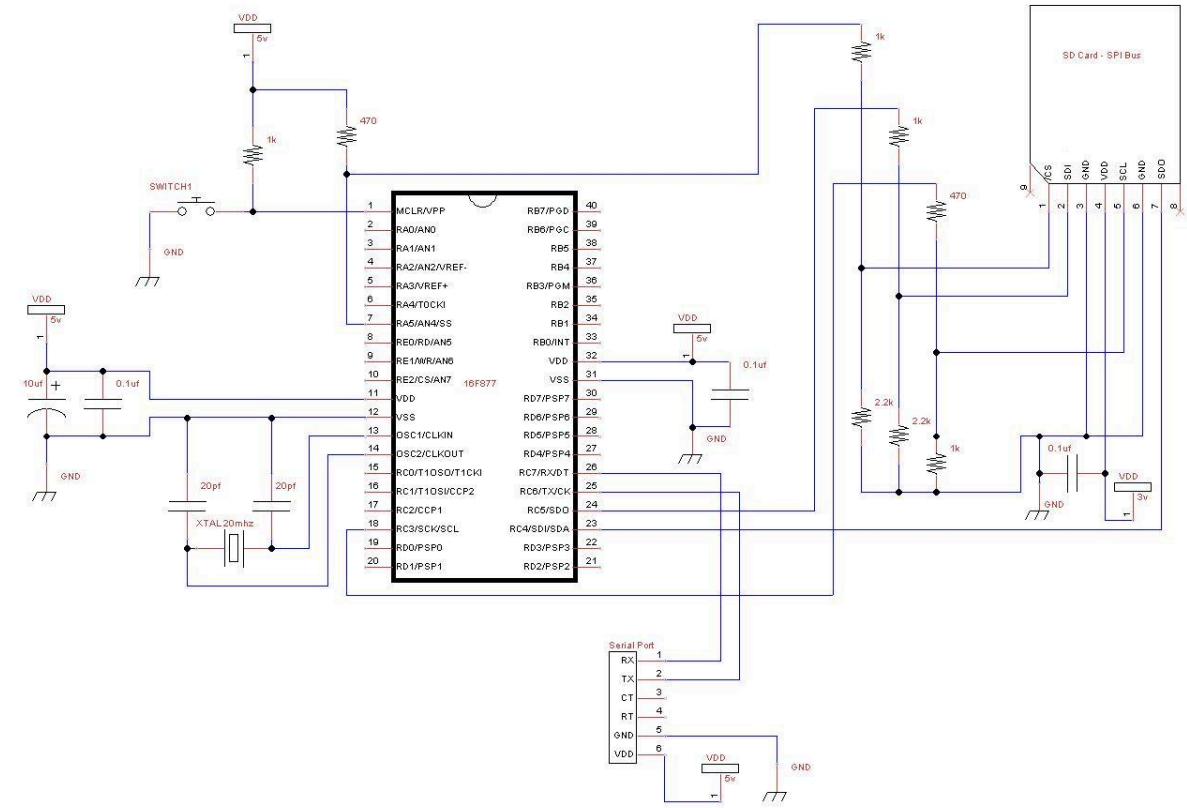
Build a SD Card Slot

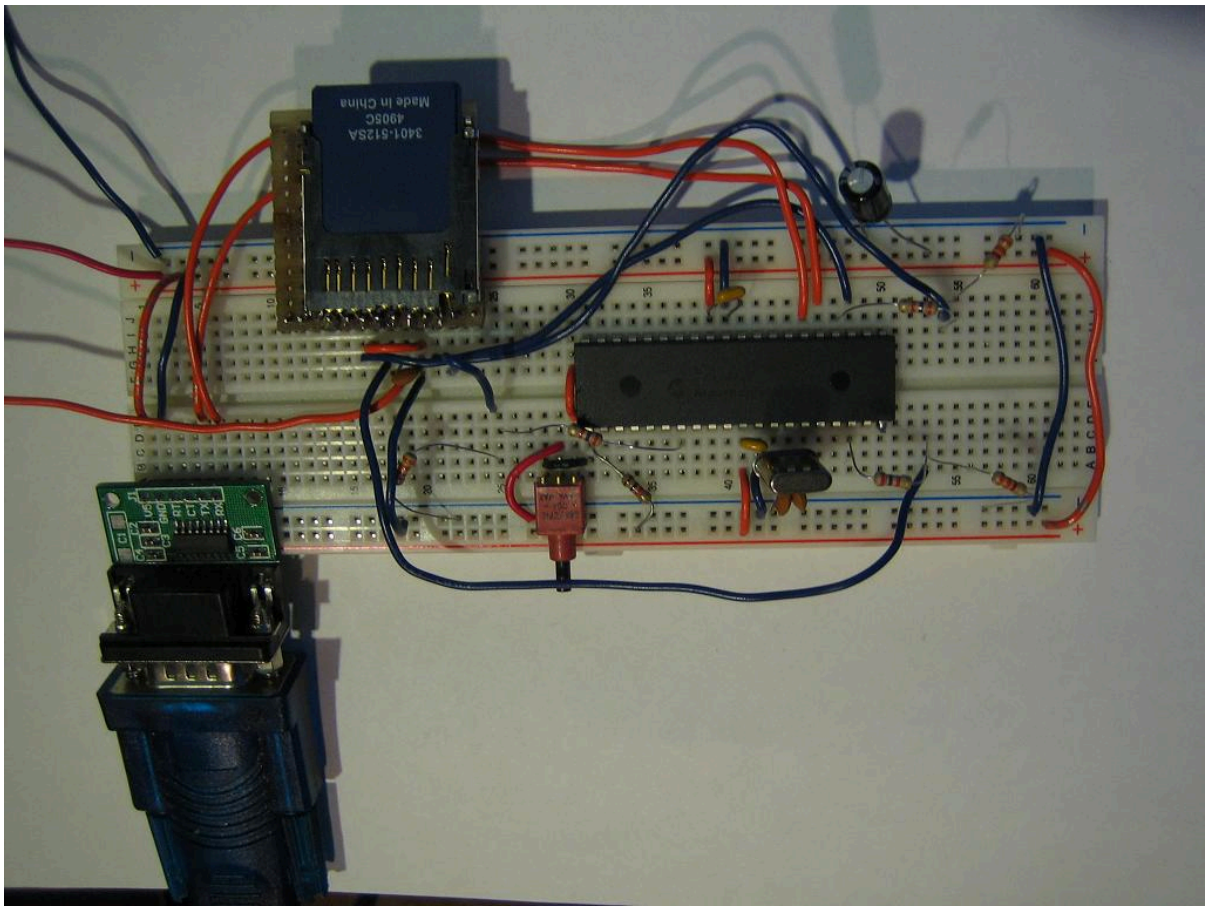
Before we can build our circuit, we will need to find ourselves an sd card slot that can plug into our breadboard. You can find pre-made sd card slots on ebay and other places around the net. It is quite easy to make your own anyways. I took one out of a broken digital camera and placed it on some blank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


VDD
5v
1k
470
SWITCH1
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
0.1uf
20pf
XTAL 20mhz
2.2k
3v
SD Card - SPI Bus
/CS
SDI
GND
VDD
SCL
GND
SDO
Serial Port
RX
TX
CT
RT
GND
VDD


Made in China
4905C
340-512SA

Compile and write the software to your PIC

With the use of the sd card lib (sd_card.jal) and a sample file 16f877a_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 16f877a_sd_card.jal from the sample directory.

Library file sd_card.jal from the lib directory.

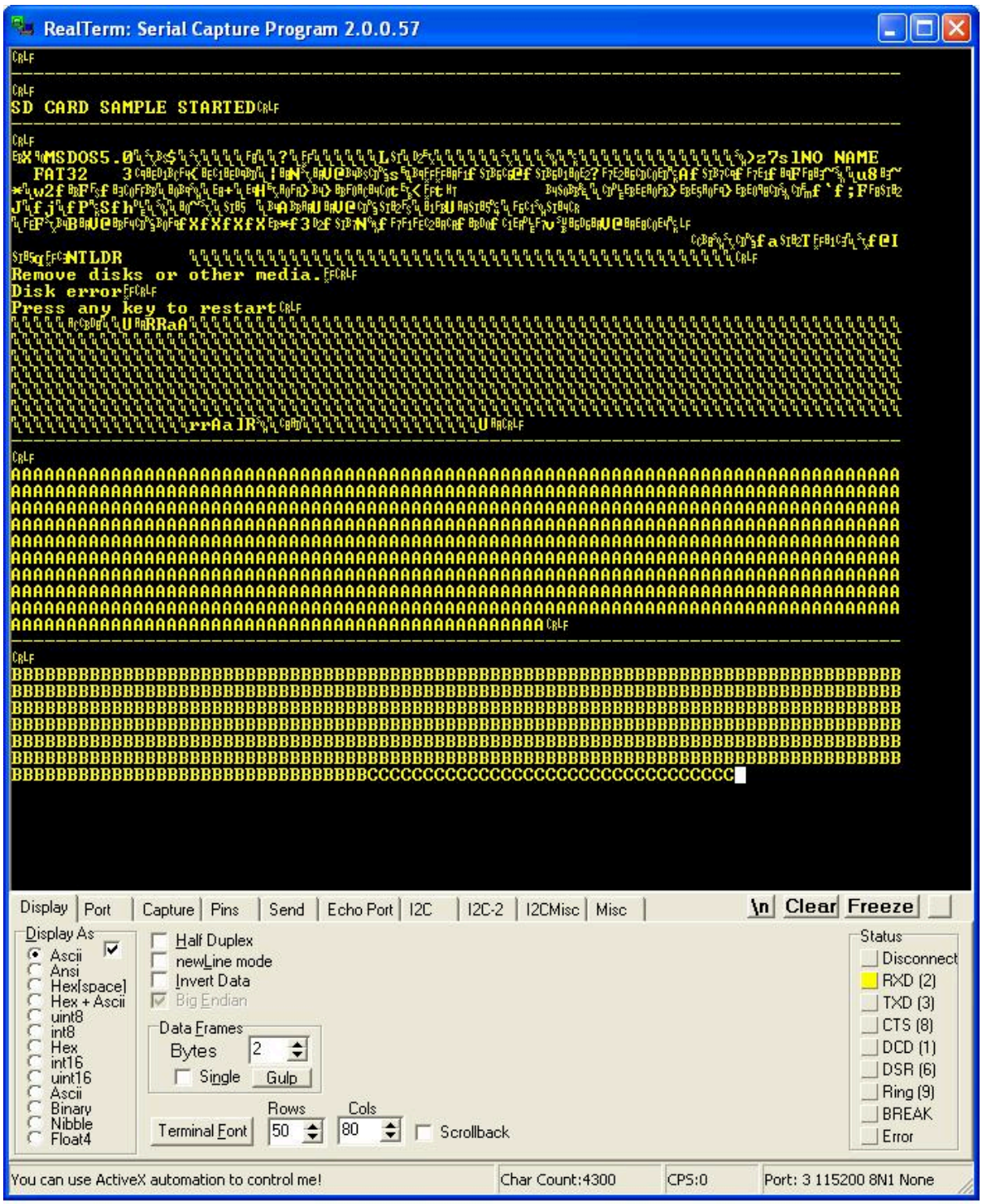
Now that our circuit is built, lets test it and make sure it works before we continue with more details. Compile and program your pic with 16f877a_sd_card.jal from your jallib samples directory. If you are using another pic, change the "include 16f877a" line in 16f877a_sd_card.jal to specify your PIC 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:

Serial Output



As you can see from the image, we got some actual readable data off the sd card as well as a bunch of junk. My sd card is formated with fat32, this is why I can actually read some of the data.

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 16f877a
--
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 and wait for power to stabilize

```
enable_digital_io() -- disable all analog pins if any
_usec_delay (100_000) -- wait for power to stabilize
```

Setup serial port and choose baud rate 115200

```
-- setup uart for communication
const serial_hw_baudrate  = 115200   -- set the baud rate
include serial_hardware
serial_hw_init()
```

Include the print library

```
include print       -- include the print library
```

Setup SPI Settings - The data transfer bus.

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" because 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). For the fastest speeds, it is a good idea to keep your SD Card as close to the PIC as possible.

```
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_init(SPI_MODE_11,SPI_RATE_FOSC_4) -- init spi, choose mode and speed

alias sd_chip_select is pin_a5
alias sd_chip_select_direction is pin_a5_direction
sd_chip_select_direction = output
```

Setup the SD Card library and settings

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

Some sd cards may require a 10ms (or more) delay every time you stop writing to the sd card, you can choose weather 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 the sd card
const bit SD_ALWAYS_SET_SPI_MODE = TRUE
const bit SD_DELAY_AFTER_WRITE = TRUE
include sd_card              -- include the sd card ide hard disk library
sd_init()                    -- initialize startup settings
```

Add a separator procedure, This will be used to display "------" onto the serial port between examples.

```
-- procedure for sending 80 "-----------------" via serial port
procedure seperator() is
   serial_hw_data = 13
   serial_hw_data = 10
   const byte str3[] =
 "--------------------------------------------------------------------------------"
   print_string(serial_hw_data, str3)
   print_crlf(serial_hw_data)
end procedure
```

It is always a good idea to send something to the serial port so we know the circuit is alive. Let's send "Hard Disk Sample Started"

```
-- Send something to the serial port
seperator()             -- send "----" via serial port
var byte start_string[] = "SD CARD SAMPLE STARTED"
print_string(serial_hw_data,start_string)
```

Declare some user variables

```
-- variables for the sample
var word step1
var byte data
```

EXAMPLES

OK, now that everything is setup, we are ready for some examples. I have left a few ways to read and write to SD Cards. The usage you choose may will on the PIC data space you have, and what your application is. On a smaller PIC, you will only be able to run examples #1, #2, #5 and #6. I'll explain as I go.

You will find that these examples are identical to the ones in the hard disk tutorial. This makes it easy for you to switch between using a SD Card and a hard disk.

Example #1 - Read data at sector 0

This is a low memory usage way of reading from the sd card, however it is slower then some of the other examples later on. This method requires the use of sd_start_read(), sd_data_byte, and sd_stop_read(). You'll see that the usage is quite simple.

Note: The variable sd_data_byte is not actually a variable, it is a procedure that looks & acts like a regular variable. This is called a pseudo variable. You may use this variable to read data or write data, as shown in these examples.

The steps are:

1. Start reading at a sector address. In this case, sector 0 (the boot sector)
2. Loop many times while you read data. One sector is 512 bytes, we will read two sectors.
3. Store each byte of data into the variable "data". You can retrieve the data by reading the pseudo variable sd_data_byte
4. Do something with the data. Let's send it to the serial port.
5. End the loop
6. Tell the sd card we are done reading.

```
sd_start_read(0)           -- get sd card ready for read at sector 0
for 512 * 2 loop           -- read 2 sectors (512 * 2 bytes)
```

```
  data = sd_data_byte      -- read 1 bytes of data
  serial_hw_write(data)    -- send byte via serial port
end loop
sd_stop_read()             -- tell sd card you are done reading
```

OK, we're done our example, so lets separate it from the next one with the separator() procedure to send some "-----" characters and a small delay.

```
separator()                -- separate the examples with "----"
_usec_delay(500_000)       -- a small delay
```

Example #2 - Writing data

This example is similar to example #1, but we will be writing data to the sd card. It requires low memory usage. As with the first example, we will be required to use 3 procedures. sd_start_write(), sd_data_byte and sd_stop_write()

Here are the steps:

1. Start writing at a sector address. I choose sector 20 since it seems that it will not mess up a fat32 formatted sd card, I could be wrong!
2. Loop many times while you write your data. In this example, I am writing to 1 sector + 1/2 sector. The 2nd half of sector 2 will contain all 0's. The end of sector 2 will contain 0's because SD Cards will only write data in blocks of 512, and therefore any data you have there will be overwritten.
3. Write some data. This time we are setting the value of the pseudo variable pata_hd_data_byte. Writing to this variable will actually send data to the hard disk. We are sending "A", so you will expect to read back the same data later on.
4. End your loop
5. Tell the SD Card we are done writing.

```
sd_start_write(20)         -- get sd card ready for write at sector 20
for 512 + 256 loop         -- loop 1 sector + 1 half sector (512 + 256 bytes)
  sd_data_byte = "A"       -- write 1 bytes of data
end loop
sd_stop_write()            -- tell sd card you are done reading
```

Now of course you will want to read your data back, which will be the same as in example #1, but at sector 20.

```
sd_start_read(20)          -- get sd card ready for read at sector 20
for 512 + 256 loop         -- loop 1 sector + 1 half sector (512 + 256 bytes)
 data = sd_data_byte       -- read 1 bytes of data
  serial_hw_write(data)    -- send byte via serial port
end loop
sd_stop_read()             -- tell sd card you are done reading
```

Example #3 - Read and write data using a sector buffer (a 512 byte array)

In this example, we will use a 512 byte array for reading and writing. This 512 byte array is called a sector buffer. This method is very fast, however it will require a PIC that can fit the 512 bytes of data in it's ram space. I find it is also easier to use. I suggest PIC18f4620 with the same schematic.

For writing, You will need only need to write data to the sector buffer array, then use the sd_write_sector_address() procedure.

Lets go through the steps, first for writing data:

1. Loop 512 times (the size of the sector buffer)
2. Set each data byte in the array
3. End your loop
4. Write the data to the hard disk at a sector address.
5. Repeat the above to write more sectors.

```
-- fill the sector buffer with data
for 512 using step1 loop                    -- loop till the end of the sector buffer
   sd_sector_buffer[step1] = "B"            -- set each byte of data
end loop
```

```
-- write the sector buffer to sector 20
sd_write_sector_address(20)
```

Here we will write another sector (to sector 21, the next sector)

```
for 512 using step1 loop                    -- loop till the end of the sector buffer
   sd_sector_buffer[step1] = "C"            -- set each byte of data
end loop
-- write the sector buffer to sector 21
sd_write_sector_address(21)
```

OK, it's time to read back the data, which is exactly the opposite of writing. For reading, we will use the pata_read_sector_addres() procedure first, then we can read data from the sector buffer array.

1. Request data from the SD Card at a sector address.
2. Loop 512 times (the size of the sector buffer).
3. Send each byte to the serial port.
4. End your loop.
5. Repeat the above to read more sectors.

```
-- read back the same sectors
-- read sector 20 into the sector buffer
sd_read_sector_address(20)
-- now send it to the serial port
for 512 using step1 loop                    -- loop till the end of the sector buffer
   serial_hw_write (sd_sector_buffer[step1]) -- send each byte via serial port
end loop
```

Here we will repeat the above to read the next sector (sector 21)

```
-- read sector 21 into the sector buffer
sd_read_sector_address(21)
-- now send it to the serial port
for 512 using step1 loop                    -- loop till the end of the sector buffer
   serial_hw_write (sd_sector_buffer[step1]) -- send each byte via serial port
end loop
```

EXAMPLE #4 - Another method for reading and writing sectors

Example #4 is pretty straight forward. I am not going to go into too much detail on this one. It is a combination of examples 2 and 3. It is about the same speed as example #3.

```
-- get sd card ready for write at sector 20
sd_start_write(20)
-- fill the sector buffer with data
for 512 using step1 loop                   -- loop till the end of the sector buffer
  sd_sector_buffer[step1] = "D"            -- set each byte of data
end loop
-- write the sector buffer to the sd card
sd_write_sector()
-- fill the sector buffer with new data
for 512 using step1 loop                   -- loop till the end of the sector buffer
  sd_sector_buffer[step1] = "E"            -- set each byte of data
end loop
-- write the sector buffer to the sd card
sd_write_sector()                          -- write the buffer to the sd card
-- tell sd card you are done writing
sd_stop_write()
--
-- read back both of the same sectors
-- get sd card ready for read at sector 20
sd_start_read(20)
-- read the sector into the sector buffer
sd_read_sector()
-- now send it to the serial port
for 512 using step1 loop                   -- loop till the end of the sector buffer
  serial_hw_write(sd_sector_buffer[step1]) -- send each byte via serial port
end loop
-- read the next sector into the sector buffer
sd_read_sector()
-- now send it to the serial port
```

```
for 512 using step1 loop                    -- loop till the end of the sector buffer
  serial_hw_write(sd_sector_buffer[step1]) -- send each 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 is a FAT32 library and there will be a tutorial soon 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

DFPlayer Mini

Introduction

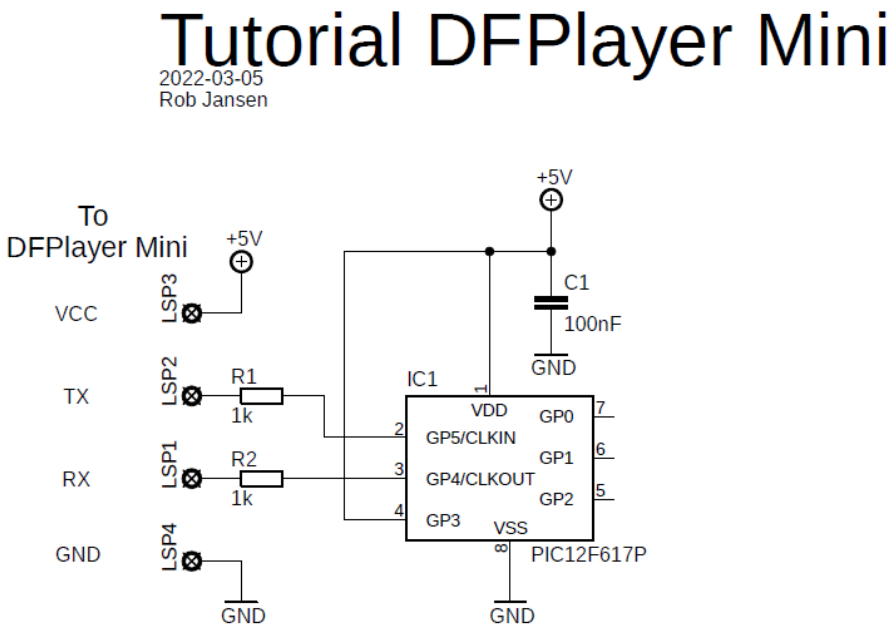
This JAL Library supports all features of the DFPlayer Mini, an audio playback device.

The library is set up in a way that the main program determines which serial interface is being used for controlling the DFPlayer Mini. Three sample files are available, a simple version called `16f1823_dfplayer.jal` that plays the first track, waits for 10 seconds and then plays the next track. The other sample program called `16f19176_dfplayer.jal` uses a menu structure with which all features of the library can be controlled. The third sample called `12f617_dfplayer.jal` shows the same functionality as the 16f1823 version but this version is meant to show that it can work on a smaller PIC that has no on-board USART so it uses the JAL software serial interface library to control the DFPlayer Mini.

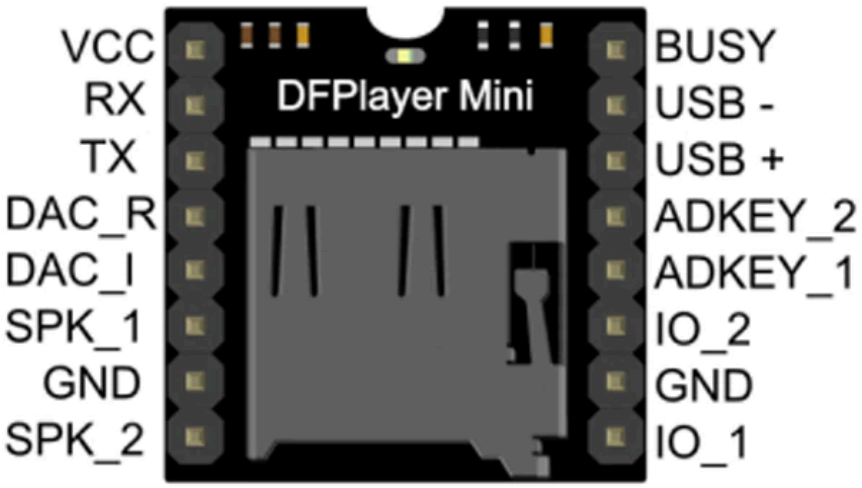
The Hardware

The DFPlayer Mini is controlled via the serial interface using a fixed baudrate of 9600 baud. It can operate on 5 Volt but it is recommended to use 1k resistors in the serial communication lines between the PIC and the DFPlayer Mini. The DFPlayer Mini plays audio files in mp3 or wav format, stored on e.g. a micro sd card that can be inserted into the device. A loudspeaker can be connected directly to the DFPlayer Mini of which the volume can be controlled via the API provided by the library.

The schematic diagram below shows how the DFPlayer mini can be connected to the pic12f617.



The connections of the DFPlayer mini are as follows:



The loudspeaker is connected directly to the pins SPK_1 and SPK2.

Sample program for the DFPlayer Mini using a 12f617

The sample file always start with a header to explain what it is about.

```
-- -------------------------------------------------------------------------
-- Title: Sample program for the DFPlayer Mini.
-- Author: Rob Jansen, Copyright (c) 2020..2020 all rights reserved.
-- Adapted-by:
-- Compiler: 2.5r4
--
-- This file is part of jallib (https://github.com/jallib/jallib)
-- Released under the BSD license (http://www.opensource.org/licenses/bsd-license.php)
--
-- Description: Demonstrates the playback feature of the DFPlayer Mini library.
--              For a completed demo of all features see 16f19176_dfplayer.jal.
--
-- Sources:
--
-- Notes: The DFPlayer uses a serial interface. This program uses the software
--        serial library since this PIC has no on-board USART.
--
```

Include the device file for the pic12f617.

```
include 12f617                          -- Target PICmicro
```

Set the pragma's. We use the 4 MHz internal oscillator.

```
-- Use internal clock and internal reset.
pragma target OSC      INTOSC_NOCLKOUT -- Internal Clock
pragma target PWRTE    ENABLED         -- Power up timer
pragma target MCLR     INTERNAL        -- Reset internal
pragma target WDT      DISABLED        -- No watchdog
pragma target BROWNOUT DISABLED        -- Disable to save power
pragma target IOSCFS   F4MHZ           -- Set internal oscillator to 4 MHz
pragma target clock    4_000_000       -- Oscillator frequency 4 MHz
```

At reset all pins of the PIC are set to analog input. Switch them to digital. Note that we also wait some time here. It can happen that a PIC needs some more time to stabilize after power up so we add a delay here before we continue with the rest op the program..

```
enable_digital_io()
-- Give the hardware some time to stabilize.
_usec_delay(100_000)
```

This PIC does not have a serial interface on board, the so called USART. Instead we use a software library that emulates a serial port. We have to define which pins we are going to use for the serial interface. After that we can include the serial software library and initialize it.

```
-- Setup the serial software interface for communication with the DFPlayer.
alias serial_sw_tx_pin is pin_GP4   -- Pin 3 of 8 pin DIP.
pin_GP4_direction = output
alias serial_sw_rx_pin is pin_GP5   -- Pin 2 of 8 pin DIP.
pin_GP5_direction = input
const serial_sw_baudrate = 9600
include serial_software
serial_sw_init()
```

The next step is to include the library of the DFPlayer mini and start playing an audio track. In this example we first set the volume, start playing track 1 and play a next track after every 10 seconds. For the 10 seconds delay we use a `delay_1s()` function from the delay library so we need to include that library too .

```
include delay

-- Now we can include the DFPlayer.
include dfplayer
dfplayer_init()

-- Initial volume is quite loud, so lower it.
dfplayer_set_volume(15)

-- Play the first track that the DFPlayer found on the storage device.
```

```
dfplayer_play(1)

forever loop

    -- Play each track for 10 seconds the go to the next available track.
    dfplayer_play_next()
    delay_1s(10)

end loop
```

The following video shows this sample program in action. The micro as card has one folder called '01' which has five mp3 tracks labeled '001.mp3' to '005.mp3'.

Some more info about the DFPplayer Mini that you should know

Audio files are stored in a certain folder structure. This structure is documented in the dfplayer.jal library and is defined as follows:

- Folders used must be named 01 to 99, ADVERT or MP3
- Audio files are numbered 001 to 255 or 0001 to 3000 with extension .mp3 or .wav
- Folders 01 to 15 can contain audio files named 001 to 255 and 0001 to 3000, these are the so called special 3000 track folders
- Folders 16 to 99 can contain audio files with names 001 to 255
- Folder ADVERT can contain advertisement audio files named 0001 to 3000
- Folder MP3 can contain mp3 only audio files named 0001 to 3000

So audio files (tracks) are always numbers and they have to be exactly 3 or 4 digits with the extension .mp3 or .wav. Note that some API functions only work on files with 3 digits and some other on audio files with 4 digits. It is possible to combine these different files in folders 01 to 15, so you could have audio files in these folders like 001.wav, 1234.mp3.

To play track 011.wav from folder 02 you the procedure call is `dfplayer_play_folder(2,11)` but if you want to play track 1234.wav from the same folder with number 02 the procedure call is `dfplayer_play_3000_folder(2, 1234)`. When calling procedure dfplayer_play_advertisement(2000), the DFPlayer will interrupt the playback of the current track, plays the advertisement track 2000 (.wav or .mp3) from the ADVERT folder and will resume play after the advertisement track has completed.

The module has some other special features like a repeat track, repeat folder and an equalizer. It also has a sleep function that can be called via the API of the library but the only way to wakeup the DFPlayer after sleep is by switching the power off and on again.

The tests of the library where done with the folders and files stored on a micro sd card but the module and the library also supports other storage devices.

This library was immediately used by one of the JAL users. The result of this project can be seen in the video of Bobby .

Chapter 4

PIC software

Learn about software libraries that will help you on your journey. These software libraries may run on hardware you already connected from other tutorials.

Print & Format

Formatting output data with Print & Format

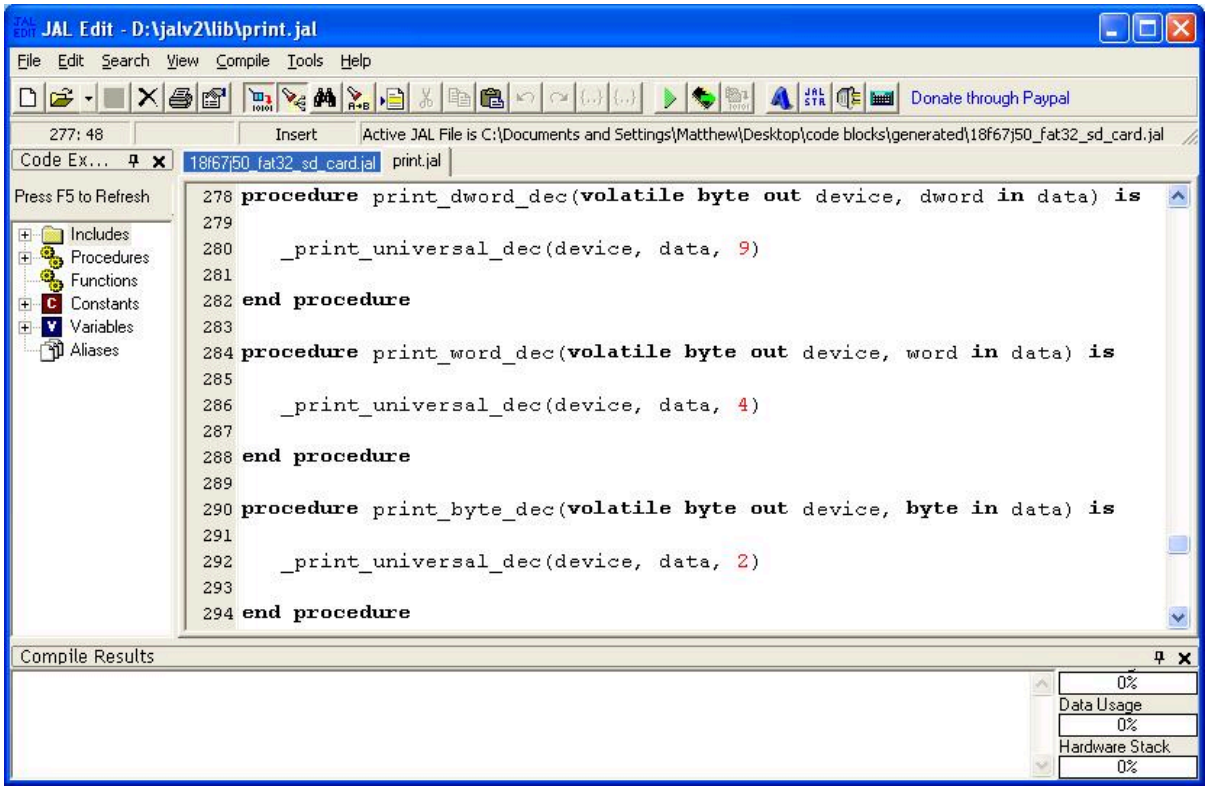
Print & Format

In a previous tutorial (Serial Port and RS-232 for communication on page 21) You probably noticed it is annoying to have to change RealTerm from ASCII to HEX in order to see output of numbers. It is also not easy to view raw HEX data. You will probably want to be able to output decimal numbers to make it easy on us humans. This is where the print and format libraries come in!

The print and format libraries are quite easy to use, start off by including them into your main file (after your serial port include block).

```
include print
include format
```

Of course, now you need to know what procedures are available. This is when it is a good idea to open up a library file. Scroll through the file and note procedure names as well as their input parameters.



Print and format numbers

In the previous image, you can see 3 procedure names:

1. **print_byte_dec()** - Prints a byte in decimal format.
2. **print_word_dec()** - Prints a word in decimal format.
3. **print_dword_dec()** - Prints a dword in decimal format.

Each requires the following parameters:

1. **device** - The device the data will be outputted to. Usually you will put serial_hw_data or serial_sw_data. However, you can put any pseudo variable (fake variable, that is actually a procedure) as an input. This pseudo variable usually allows writing to an output device such as and LCD or protocol SPI, I2C, etc.
2. **data** - The data to be sent to the device.

The following example will set the value of the word x to 543 and send it to the serial serial port in ASCII format:

```
var word x = 543
print_word_dec(serial_hw_data, x)
```

Now let's have a look at the format library. you will see 3 alike procedures with names:

1. **format_byte_dec** - Formats a byte in decimal.
2. **format_word_dec** - Formats a word in decimal.
3. **format_dword_dec** - Formats a dword in decimal.

These format procedures are able to format a byte, word or dword. Here are the input parameters:

1. **device** - Same as print library.
2. **data** - The data to be sent to the device.
3. **n_tot** - The total length of the outputted number (Including sign' +/-', and decimal point)
4. **n2** - The number of decimal places.

The following example will write 61.234 to the serial port in ASCII format:

```
var dword_dec x = 61234
format_dword_dec(serial_hw_data,x,6,3)
```

Printing Strings

The print library also has a procedure for printing strings. called print_string. It requires 2 inputs:

1. **device** - same as above.
2. **str[]** - The string to print to the serial port (an array of characters).

Here's an example that will output "Hello World" to the serial port in ASCII format:

```
const byte hello_string[] = "Hello World"
print_string(serial_hw_data, hello_string)
```

Note: the constant array must be a **byte** array.

Last of course, you may need to go to the next line with carriage return + line feed (CRLF).

```
print_crlf(serial_hw_data)
```

CRLF can also be put directly into your string with "\r\n". This will put Hello and World on 2 separate lines.

```
const byte hello_string[] = "Hello\r\nWorld"
print_string(serial_hw_data, hello_string)
```

Put it all together

```
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
-- 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)

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

include print
include format

const byte start[] = "Start of main program...\r\n"
print_string(serial_hw_data,start)

var dword x = 61234

const byte string1[] = "Let's print a dword: "
print_string(serial_hw_data,string1)
print_dword_dec(serial_hw_data,x)
print_crlf(serial_hw_data)

const byte string2[] = "Let's print a dword with 3 decimal places: "
print_string(serial_hw_data,string2)
format_dword_dec(serial_hw_data,x,6,3)
print_crlf(serial_hw_data)

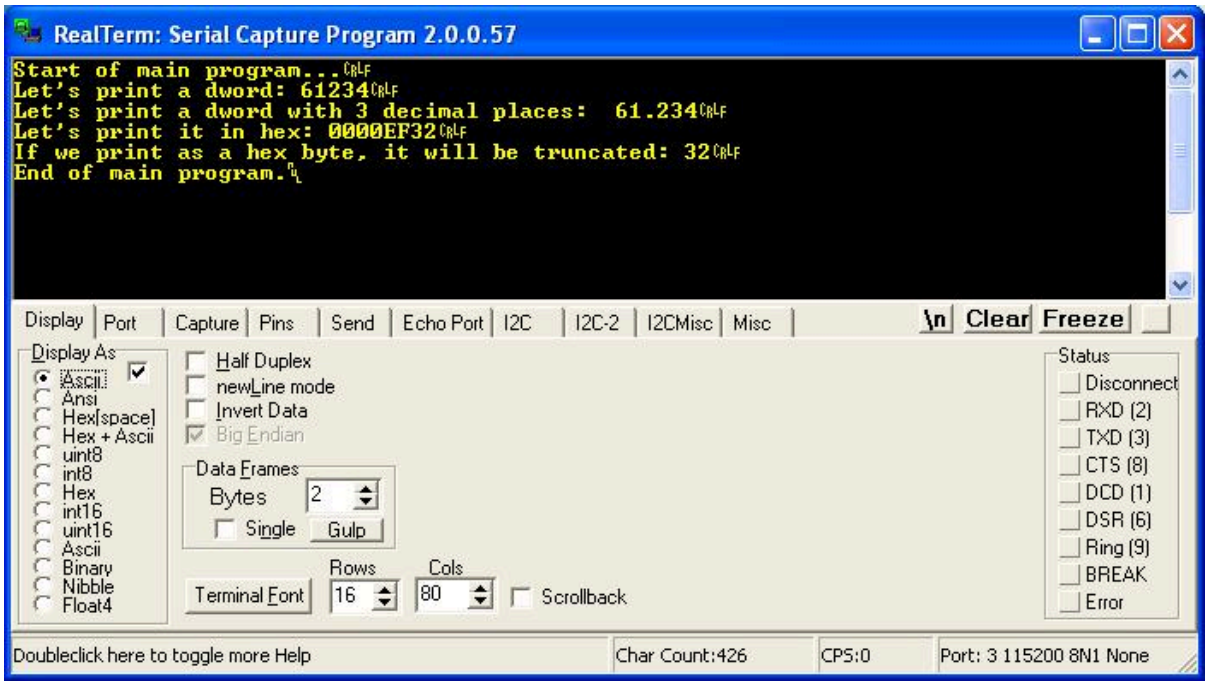
const byte string3[] = "Let's print it in hex: "
print_string(serial_hw_data,string3)
print_dword_hex(serial_hw_data,x)
print_crlf(serial_hw_data)

const byte string4[] = "If we print as a hex byte, it will be truncated: "
```

```
print_string(serial_hw_data,string4)
print_byte_hex(serial_hw_data, byte(x) )

const byte end[] = "\r\nEnd of main program..."
print_string(serial_hw_data,end)
```

Here's the output. Take special note of how and why our number 61234 in hex (0x0000EF32) got reduced into a byte (0x32) in the 5th line shown below.



There you go... that's print and format! Doesn't this make life so easy!

FAT32 File System

FAT32 Intro

If you made it here, you either have or are thinking about putting some storage device into your project. Of course you may want to use FAT32 or some type of file system if you are using a device such as a SD card or hard disk with a huge amount of storage space. You could read or write directly to your storage device if you wanted wish, so why use a file system such as FAT32? It is an easy way to manage data and to access data via a PC, it is user friendly for building your product and also good for users of your product.

Fat32 can be read by the most popular operating systems. Of course, FAT32 is a Microsoft product used in Windows, but Linux and MAC OS can also read FAT32. Jallib also has Minix V2 file system available.

Before you start & Requirements

Before you get started, there are a few things you need to know about the current Jallib FAT32 libraries. You will need to choose a library based on your project requirements. Be sure to choose a PIC that will suit your needs. I suggest PIC 18f4620 since it has loads of RAM & program memory. There are 2 FAT32 libraries to choose from. Here's a list of features the libraries currently have:

Features	**FAT32**	**FAT32 SMALL**
RAM (Minimum)	1500 bytes	256 bytes
Program Memory (Minimum)	20k	5k

Features	FAT32	FAT32 SMALL
Max Files:	Dependant on RAM available (Can use external memory)	1
List files	YES	YES
Create files	YES	NO
Read & Write to files	YES	YES
Read long file names	YES	YES
Write long file names	NO	NO
Max Partitions	4 primary, 0 extended	1 primary
Read/Write fragmented files & directories	YES	NO
Max file fragments	Dependant on RAM available (Can use external memory)	1
Max directory fragments	Dependant on RAM available (Can use external memory)	1

This tutorial will concentrate on the normal FAT32 library. There will be a separate tutorial for FAT32 SMALL whenever I find time!

Note: Always get the newest library and sample. I suggest you download the newest Jallib Bee package from http://justanotherlanguage.org/downloads

Choose a storage device

SD cards - SD cards are popular thanks to their small size. They are quite easy to connect to your circuit. The hookup will cost you 4 PIC pins via SPI port. They are slower then hard disks due to serial data transfer via SPI port. Most SD Cards also have an endurance of 100,000 write cycles. They run on 3.3v.

Hard Disk - Hard drives are fast but large. The main sizes of hard disks are 3.5", 2.5" and 1.8". Connecting them to your circuit is simple, but will require 21 pins. I did actually fit sd card + hard disk in one circuit on 18f4620. Hard drives have an unlimited number of write cycles. They run at 5v TTL levels, but will accept 3.3v on it's inputs. You can run your PIC at 3.3v or 5v.

Suggested Tutorials

1. Getting Started
2. Blink a led
3. Serial Communication
4. SD Card OR PATA Hard DIsk
5. 23k256

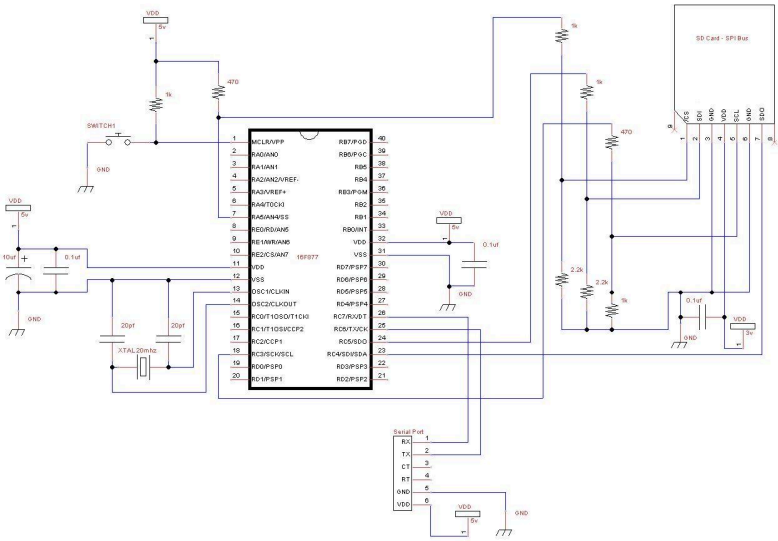
Benefits of ICSP

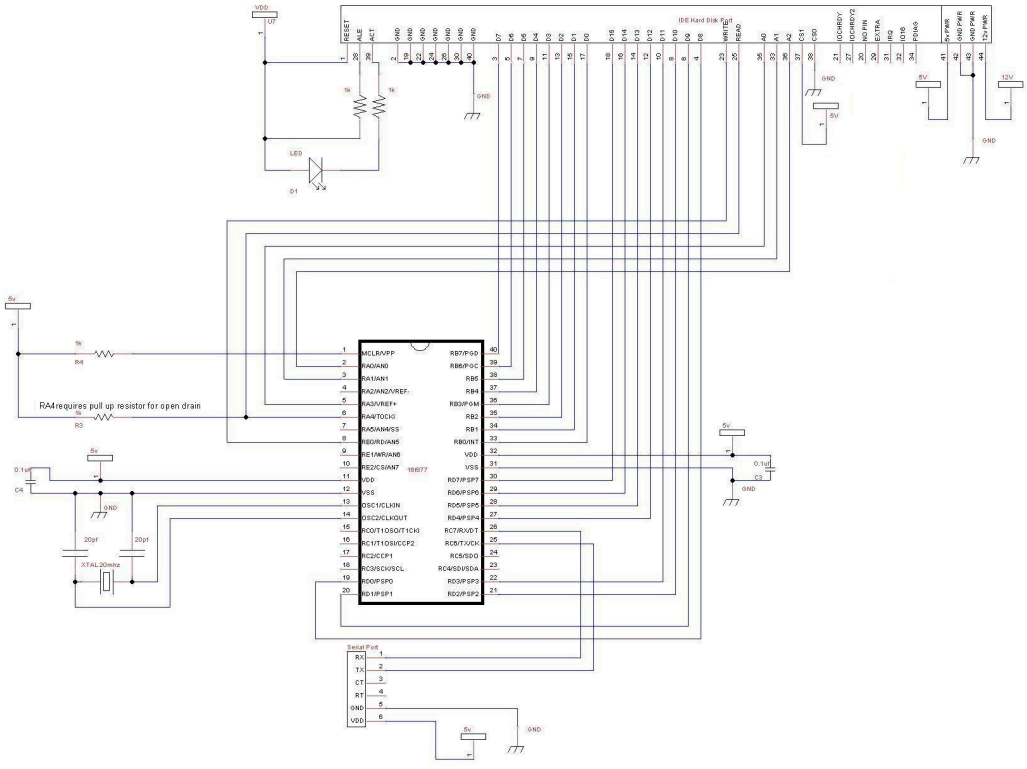
1. You may program your PIC while it is in your breadboard circuit
2. You may program your PIC while it is on a soldered circuit board
3. You will save time programming so you can write more code faster
4. You can reset your circuit from your PC
5. You can program surface mount PIC's that are on soldered 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 living room on my laptop while I watch tv with my wife! (I keep my mess in my office)

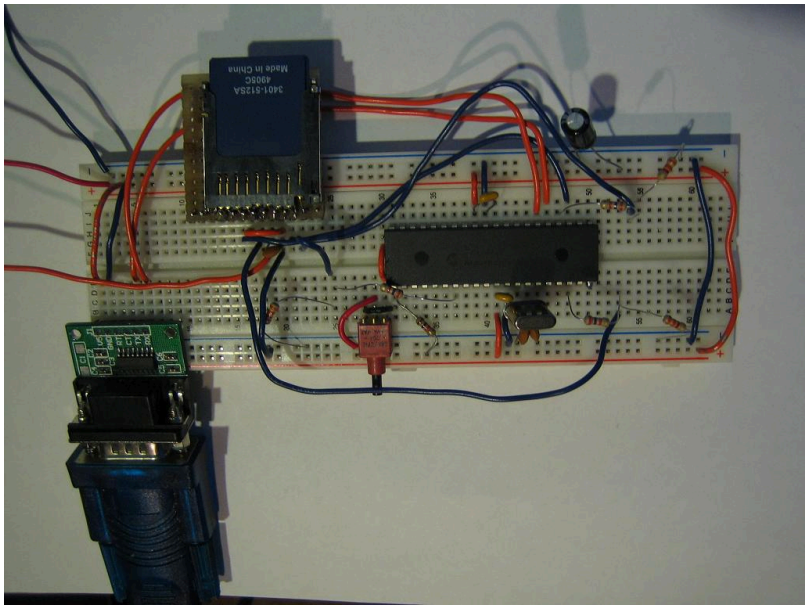
The Schematic

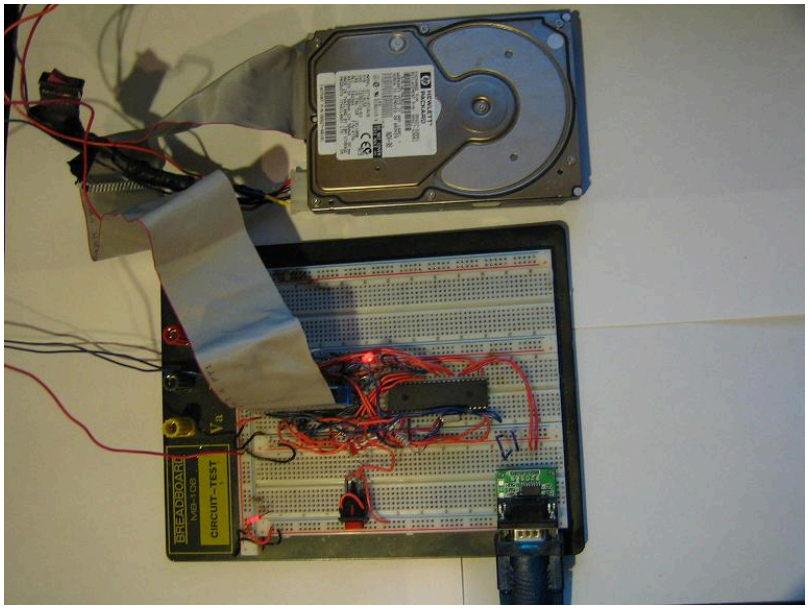
Your schematic will be the same as either the hard disk or sd card tutorial. If you wish to use external memory, you can add a 23k256. I'll explain external memory later on. Although the schematics show 16f877, you can replace it with 18f4620. 16F877 is not large enough.





Some Images





Run a sample

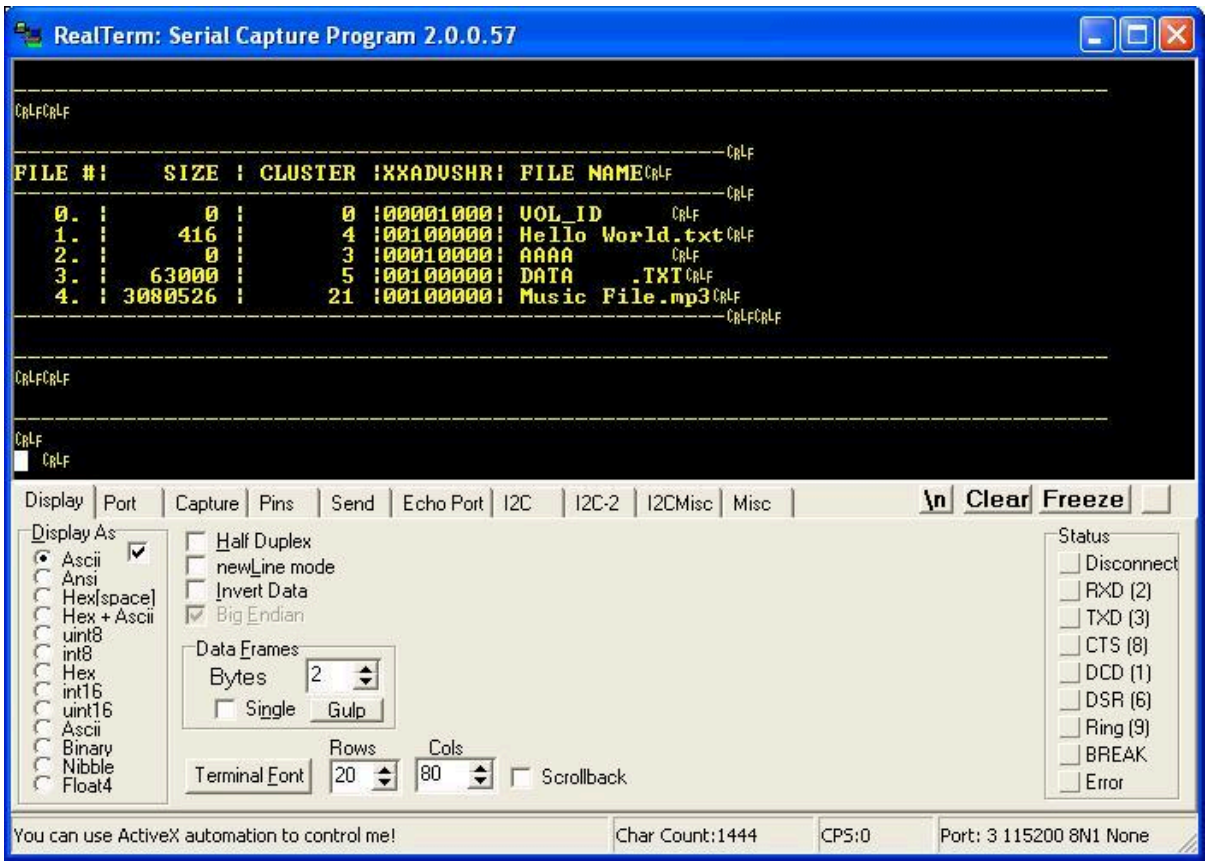
If you've already done a previous sd card or hard disk tutorial, you know you have a working circuit. I suggest you try one of the following samples:

18f4620_fat32_pata_hard_disk.jal

18f4620_fat32_sd_card.jal

Start by formatting your storage device in windows. Then put a few files and a directory on it. In this example I put 4 files and one directory.

Set your serial port to 115200 baud. If all is good, you should get a directory listing on your serial port software:

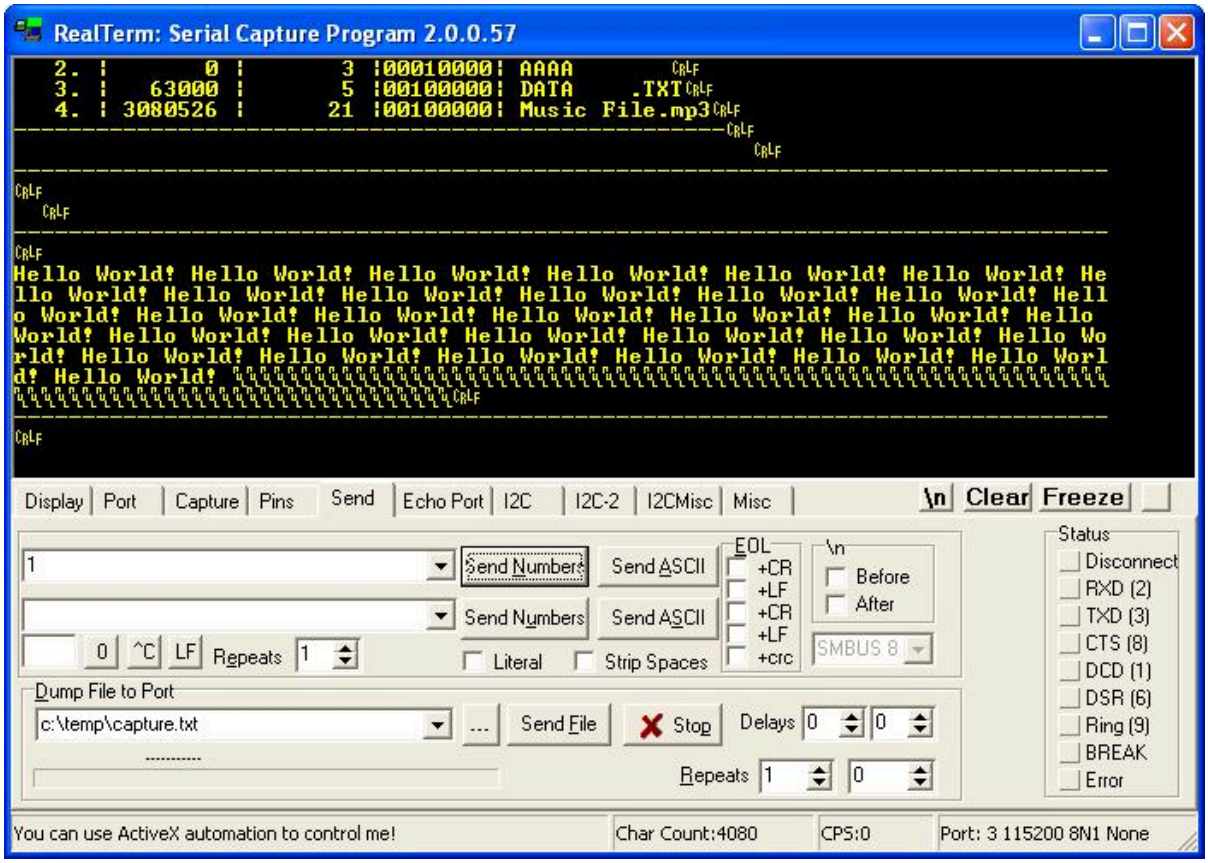


While we have this sample running, let's try it out!

Each file is identified by a number. The first one "0" is the volume id. The partition loaded is named "VOL_ID". You will also see 3 files and one directory. The directory is named "AAAA" (number2). You can identify directories by their attributes "00010000".

Now just send a number via Realterm's "Send" tab. If you send the number 0, you will list the same root directory again. If you send the number of a file, It's contents will be displayed. If you send the number of a directory, you will go into that directory and it's contents will be listed. Here's what I get when I send the number "1".

You will notice some junk at the end of the file. Usually you'll get a bunch of 0's. They are there because fat32 reads & writes in 512 byte chunks, and the file is only 415 bytes long. You can fix your software to stop at the correct byte at the end of the file.



The Code

Alright then, we're ready to get our hands dirty. Let's take a look

Required Includes

The first part of the sample is just a bunch of includes, I'm not going to cover the first includes to much since they are covered in other tutorials. When we get to the fat32 include, I'll give more detail.

```
-- Title: FAT32 library for reading fat32 filesystem
-- Author: Matthew Schinkel, copyright (c) 2009, all rights reserved.
-- Adapted-by:
-- Compiler: >=2.4k
--
-- This file is part of jallib (http://jallib.googlecode.com)
-- Released under the BSD license (http://www.opensource.org/licenses/bsd-license.php)
--
-- Description: this example reads files & folders from a fat32 formatted sd card
--              using the fat32 library.
--
-- Sources:
-- http://www.microsoft.com/whdc/system/platform/firmware/fatgen.mspx
-- http://www.pjrc.com/tech/8051/ide/fat32.html
-- http://en.wikipedia.org/wiki/File_Allocation_Table
--

-- include chip
include 18f4620                    -- target picmicro
-- this program assumes a 20 mhz resonator or crystal
-- is connected to pins osc1 and osc2.
pragma target osc INTOSC_NOCLKOUT              -- hs crystal or resonator
;pragma target osc hs             -- hs crystal or resonator
;pragma target clock 20_000_000    -- oscillator frequency
pragma target clock 32_000_000    -- oscillator frequency
--
pragma target wdt  disabled
pragma target lvp  disabled
pragma target MCLR external        -- reset externally
```

```
--;
;OSCCON_IRCF = 0b110   -- set int osc to 4mhz
OSCCON_IRCF = 0b111   -- set internal osc to 8mhz
OSCTUNE_PLLEN = true  -- multiply internal osc by 4
;--
enable_digital_io()               -- make all pins digital I/O
--
_usec_delay(100_000) -- wait for power to settle


include delay

-- setup uart for communication
const serial_hw_baudrate  = 115200   -- set the baudrate
include serial_hardware
serial_hw_init()
-- some aliases so it is easy to change from serial hw to serial sw.
alias serial_write is serial_hw_write
alias serial_read is serial_hw_read
alias serial_data is serial_hw_data
alias serial_data_available is serial_hw_data_available


include print

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_init(SPI_MODE_11,SPI_RATE_FOSC_4) -- init spi, choose mode and speed
alias spi_master is spi_master_hw


-- setup 23k256 for external memory
-- setup chip select pin
ALIAS sram_23k256_chip_select             is pin_a1
ALIAS sram_23k256_chip_select_direction   is pin_a1_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)
-- initalize 23k256 in byte mode
alias sram_23k256_force_spi_mode is spi_master_hw_set_mode_00 -- always set spi mode to 0,0
include sram_23k256 -- setup Microchip 23k256 sram
sram_23k256_init(SRAM_23K256_SEQUENTIAL_MODE, SRAM_23K256_HOLD_DISABLE)  -- init 23k256 in
 sequential mode


-- setup the sd card pins
alias sd_chip_select is pin_SS
alias sd_chip_select_direction is pin_SS_direction
sd_chip_select = high
sd_chip_select_direction = output
--
-- setup the sd card library
alias sd_force_spi_mode is spi_master_hw_set_mode_11 -- always set spi mode to 1,1
;const bit SD_EXTRA_SPEED = TRUE
const bit SD_ALWAYS_SET_SPI_MODE = TRUE
const bit SD_DELAY_AFTER_WRITE = TRUE
include sd_card              -- include the sd card ide hard disk library
sd_init()                    -- initialize startup settings
```

Include the FAT32 library

Now for the fat32 include. You will need to modify some of the constants to fit your need. Here's a long description of each constant so you have some idea of what to use in your project.

FAT32_WRITE_LONG_FILE_NAMES - Enables writing of long file names. This is not currently implemented. Keep it FALSE for now.

FAT32_FILES_MAX - The maximum number of files per directory. Each file will take up 2 bytes of ram. You can use external memory for this. I'll explain more about external memory soon.

FAT32_FILE_NAME_SIZE - The maximum size of a file name. larger sized file names take more RAM space. If a file name is larger then this constant, some of it's characters will get cut off during listing of a directory.

FAT32_DIR_FRAGMENTS_TO_ALLOW - The number of fragments a directory may have. Requires 6 bytes of RAM space per fragment allowed. This will use PIC's internal memory at the moment. I'll probably change this so you can use an external memory source.

FAT32_FILE_FRAGMENTS_TO_ALLOW - The number of fragments a file may have. Requires 8 bytes of RAM space per fragment allowed. This will use PIC's internal memory at the moment. I'll probably change this so you can use an external memory source.

FAT32_ENTRIES_MAX - highest file entry address can be 65535. Changing this is only for advanced users. I'll have to give a better description of this sometime. Basically each file entry (not file name) is 32 bytes long. FAT32 lib reads all entries and stores important entry locations into RAM. The important entries are entries that are the beginning of a file name. When a file number is called, the library will go to the entry address to read the file name, size, cluster address etc. This way the entire file name list does not need to be stored in RAM. Only the location of the file name gets stored.

FAT32_USE_INTERNAL_MEMORY - Choose where to store FAT32's file location table (internal memory or external memory). This is related to FAT32_FILES_MAX. If you have 50 files max, each file will take 2 bytes of ram, so 100 bytes ram. Choose weather you want this 100 bytes of ram to be used internally or on an external memory device. For external memory, I very much suggest external RAM or something fast and with a high endurance (write cycles) like 23k256. This will of course affect the "if" statement that follows this constant.

Within the FAT32_USE_INTERNAL_MEMORY "if" statement, you will need to define an array named "fat32_entry_location". This array can be a real array, a large array (through the large array lib) or a pseudo array. If you look in the 23k256 lib, you will see that sram_23k256_word[] is a pseudo (fake) array. For external memory we simply alias a pseudo array "alias fat32_entry_location is sram_23k256_word"

And of course, here is a sample block for including fat32. You can keep the defaults for now and mess around with them later. I kept the values low to save RAM space.

```
-- setup fat32 --
-- include the required files
;include pic_data_eeprom
-- change these vaues to save memory
const bit FAT32_WRITE_LONG_FILE_NAMES = FALSE -- support writing of long file names
const word FAT32_FILES_MAX = 20            -- the max number of files allowed in a directory
const byte FAT32_FILE_NAME_SIZE = 80        -- choose max file_name size. if a file_name is
 longer the this, beginning chars will be cut. short file_names are 12 bytes.
const FAT32_DIR_FRAGMENTS_TO_ALLOW = 5     -- uses 6 bytes memory per fragment allowed (0 not
 allowed)
--                                          -- windows defrag does not defragment directories.
const FAT32_FILE_FRAGMENTS_TO_ALLOW = 5    -- uses 8 bytes memory per fragment allowed (0 not
 allowed)
--
-- experts may change the following values
;const byte FAT32_ENTRIES_MAX = 1           -- highest file entry address can be 256
const byte FAT32_ENTRIES_MAX = 2           -- highest file entry address can be 65535
--
-- choose a memory source for the file list
const bit FAT32_USE_INTERNAL_MEMORY = TRUE -- Use internal memory for file location list
IF FAT32_USE_INTERNAL_MEMORY == TRUE THEN
   -- Setup a large array for storing sector data, This is where file_name locations are stored
   const dword LARGE_ARRAY_2_SIZE = FAT32_FILES_MAX              -- choose number of array
 variables
   const dword LARGE_ARRAY_2_VARIABLE_SIZE = FAT32_ENTRIES_MAX    -- choose bytes size of
 variables
   include large_array_2                   -- include the array library
   ALIAS fat32_entry_location is large_array_2
elsif FAT32_USE_INTERNAL_MEMORY == FALSE THEN
   -- put your own code here if you wish to allow massive amounts of files per directory

   -- example usage of 23k256 for external memory

   -- alias the 23k256 device word array
   ;alias entry_location is sram_23k256_byte  -- highest file entry address can be 256
   alias fat32_entry_location is sram_23k256_word   -- highest file entry address can be 65535
END IF
--
include fat32   -- include fat32 library -- include fat32
```

You may want to filter out some files such as hidden or system files which you may not need in your project.

```
-- CHOOSE FILE ATTRIBUTES TO FILTER OUT
fat32_filter_is_read_only  = FALSE
fat32_filter_is_hidden     = FALSE
fat32_filter_is_system     = FALSE
fat32_filter_is_volume_id  = FALSE
fat32_filter_is_directory  = FALSE
fat32_filter_is_archive    = FALSE
```

The main program

Well then, your ready for the main program. First we have a few inital settings.

I have created a simple seperator procedure to send a line "----------" via serial port.

```
-- procedure for sending 80 "-----------------" via serial port
procedure separator() is
   serial_data = 13
   serial_data = 10
   const byte str3[] =
 "--------------------------------------------------------------------------------"
   print_string(serial_data, str3)
   print_crlf(serial_data)
end procedure

-- start of main program
separator()-- send "----"
```

We can now initialize the library. The input to fat32_init is the partition number, which will usually be the first (partition 1).

```
fat32_init(1) -- initialize fat32, go to 1st primary partition's root dir "\"
```

Now some good coding stuff, let's print a directory to the serial port. The first dir listing is the root dir. There are some options we can choose for the print dir procedure.

FAT32_PRINT_LONG_FILE_NAME - print the file name

FAT32_PRINT_NUMBER - print the file number

FAT32_PRINT_SIZE - print the file size

FAT32_PRINT_CLUSTER - print the file's cluster address

FAT32_PRINT_TABLE - print a table around everything

FAT32_PRINT_DATE - print the file date

FAT32_PRINT_ATTRIBUTES - print the files attributes (such as read only, hidden, etc).

The file attributes are "ADVSHR" (you will see this on your serial port software), and each can be true or false.

A - Is Archive

D - Is a directory

V - Is a volume ID

S - Is a system file

H - Is a hidden file

R - Is read only

```
fat32_print_directory(serial_data, FAT32_PRINT_ALL) -- sends dir listing via serial port
```

Of course, you could just print one file name at a time instead. This could be useful on an LCD where you can only view a certain number of files at a time, and scrolling is needed.

```
-- read 3rd file's name, location, size, attributes into memory
```

```
fat32_read_file_info(3)

-- now send the filename via the serial port (file number and file name)
fat32_print_file_info(serial_data,FAT32_PRINT_NUMBER + FAT32_PRINT_LONG_FILE_NAME)
```

Declare some variables we're going to use, then start our main loop

```
var byte data
var dword step1
var word step2
var byte file_number = 0

forever loop
   separator() -- send "-----" then loop and wait for user input
```

This example is a user program, so we will wait for the user. The user can send data to the device via serial port to select a file number. So, let's wait for data.

Note that the serial software library does not contain a serial_data_available variable.

When we get data from the user, it will be placed into our variable file_number.

```
file_number = serial_data
```

Now we can either check the file's attributes to see if it is a file or directory (see fat32_file_attrib in fat32.jal), or we can simply try to go into the directory. If we fail to go into the directory, it must be a file.

If we do go into the directory, we will print the new directory to the screen.

```
   -- choose a file for reading or dir for opening
   if fat32_cd(file_number) then             -- if change directory is successful
      fat32_print_directory(serial_data, FAT32_PRINT_ALL) -- print dir listing
```

Otherwise (if it is not a directory), we will open the file.

```
   elsif fat32_file_open(file_number) then   -- if go into file is successful
```

Calculate the number of sectors in a file

The fastest way to read files is by reading them sector by sector. We'll need to calculate how many sectors are in the file so we can use a for loop later on.

```
      -- calculate number of sectors in file
      var dword sectors_available
      if (fat32_file_size) == (fat32_file_size / 512) * 512 then
         sectors_available = (fat32_file_size / 512)
      else
         sectors_available = (fat32_file_size / 512) + 1
      end if
```

Reading & writing

While writing the FAT32 library, I wanted to give some different ways for the code writer to read & write to a file.

1. Read any byte from a file at any address.
   a. fat32_read_file_byte_address(address) - function that returns a byte from any byte address in the file.
2. A faster way to read byte by byte. Still slower then sector reads. It is not so user friendly since you must call a few procedures. Only reads starting at the beginning of a file. If you are using an sd card and have other SPI devices connected, do not use them until you do stop_file_read.
   a. fat32_start_file_read() - start reading a file from the beginning
   b. fat32_read_file_byte() - read one byte
   c. fat32_stop_file_read - stop reading
3. A very fast & user friendly way to read and write to a file at any sector address of the file.
   a. fat32_sector_buffer[] - a 512 byte array that stores data to be read or written

   b. fat32_read_file_sector_number(address) - Call this to read data into the buffer from any address.
   c. fat32_write_file_sector_number(address) - Call this to write sector buffer to media at any address.

4. The fast way to read and write to a file. A little bit faster then #3. Not as user friendly. You must start reading or writing at the beginning of the file. Read and write sector by sector by filling fat32_sector_buffer[]

   a. fat32_sector_buffer[] - a 512 byte array that stores data to be read or written
   b. fat32_start_file_write() - start writing to a file, starting at the beginning of the file.
   c. fat32_stop_file_write() - stop writing to a file
   d. fat32_start_file_read() - start reading from a file, starting at the beginning of the file.
   e. fat32_stop_file_read() - stop reading from a file

Well, as you can see you have a lot of options. Our sample uses #3 (fast and user friendly). It really depends on your application. If you are recording sound, you may want #4 since you'll be starting at the beginning of the file. If you are jumping around in the file, you may want #3. Of course #3 would be perfectly sufficient for recording sound. You'll just have to try them out!

I'll give a bit more explanation of #3. You will obviously either read from the storage device or write to the storage device.

For writing, you will first fill the sector buffer with data, then when you call fat32_write_file_sector_number(address), the library will transfer the data from the buffer onto the storage device.

```
      -- EXAMPLE 3 WRITE (fast and user friendly)
      -- Read from any sector number in the file, in any order you wish.
      for sectors_available using step1 loop

         -- set the data to be written
         for 512 using step2 loop
            fat32_sector_buffer[step2] = "E"
         end loop

         -- write one sector to the disk
         fat32_write_file_sector_number(step1)
      end loop
```

For reading, you will first choose the sector to read from, then call fat32_read_file_sector_number(address), the library will transfer data from the storage device into the sector buffer. After the data is in the sector buffer array, you can do what you like with it.

```
      -- EXAMPLE 3 READ (fast and user friendly)
      -- Read from any sector number in the file, in any order you wish.
      for sectors_available using step1 loop
         -- read one sector from the disk
         fat32_read_file_sector_number(step1)
         -- send the sector via serial port
         for 512 using step2 loop
            serial_data = fat32_sector_buffer[step2]
         end loop
      end loop
```

Well, that was awsome, let's wrap this up! You should close the file when your done with it, just to ensure your storage device goes ready. On a hard disk, you will see the disk LED go off.

```
      fat32_file_close()
   end if
end loop
```

What's that now? You want to see an example of this working? I guess it's time for a Youtube Video! You can see me easily moving around in files & directories, and even transferring an MP3 to my PC via Realterm!

http://www.youtube.com/watch?v=ar7DkTPriNk

Have fun!

Large Array

Introduction

JAL support to type of arrays:

- Arrays with constant data. Data is stored in program memory of the PIC
- Arrays with variable data. Data is stored in data memory of the PIC

The maximum size of the constant data arrays depends on the size of the program memory of the PIC and can at most be 64 kbytes. For an array with variable data there are limitation caused by how a PIC is organized. Data memory in the PIC is stored in banks which are limited in size:

- For PIC12F and PIC16F, banks are 80 bytes
- For PIC18F banks are 256 bytes

The total data memory of a PIC is a total of all banks that are present in the PIC.

Working with large data arrays

In applications that need arrays with the size larger than the bank size of the PIC a library is available. This large array library supports the following:

- For controllers with PIC14 core (labeled PIC16(l)f... or PIC12(l)f...)
  - Supports byte array with up to 4800 entries
  - Supports word array with up to 2400 entries
  - Supports dword array with up to 1200 entries
- For controllers with PIC16 core (labeled PIC18(l)f...)
  - Supports byte array with up to 14848 entries
  - Supports word array with up to 7424 entries
  - Supports dword array with up to 3712 entries

The total size of a large array cannot be larger than the available - remaining - data memory. The JAL compiler will generate an error when the program runs out of data memory.

Defining the large array library

There are four large array libraries present in Jallib so at most four large arrays can be used in one application. The type of array (byte, word, dword) and size are defined using constants. If, for example, library 'large_array_1.jal' is used, the following needs to be defined before including the library:

- `const LARGE_ARRAY_1_SIZE` defining the number of entries in the array
- `const LARGE_ARRAY_1_VARIABLE_SIZE` defining the type of array, where:
  - A value of 1 gives a byte array
  - A value of 2 gives a word array
  - A value of 3 gives a dword array

Unlike the standard arrays in JAL, there is no range checking available for large arrays. This means that when an entry is written which lies outside of the maximum size of the array, the user is not warned. So the user has to be extra careful to make sure this does not happen since it may result in unwanted side effects.

Using a large array in your application

The following - part of a - sample program shows how to include the array in your application. Usage of the array is the same as the use of a standard array of the JAL programming language. In this example we skip the include of the

device file the pragmas and other libraries. Instead we focus only on the definition and use of the large array. Here we use large array 4.

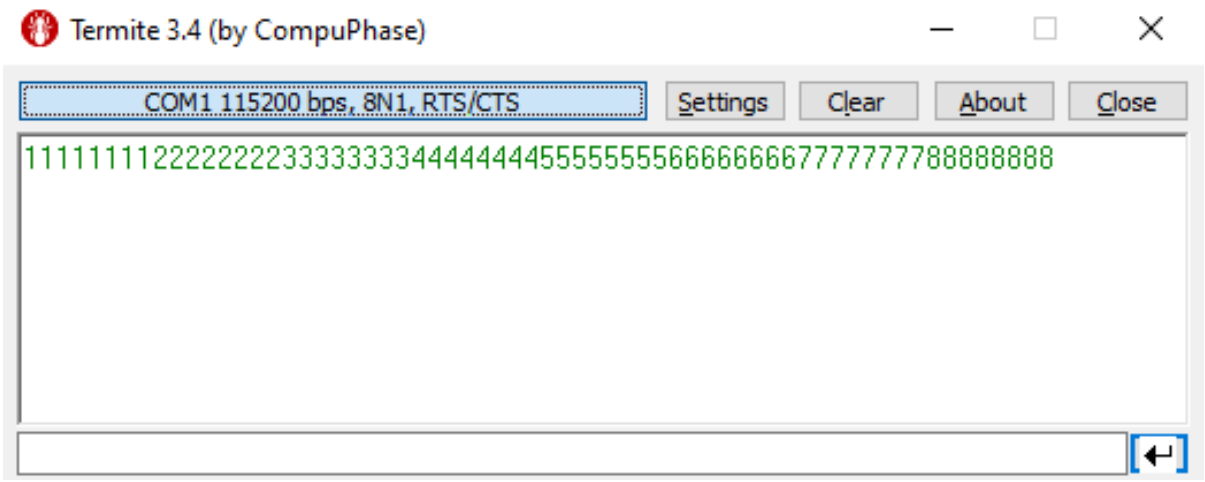
```
-- Setup the large array
const LARGE_ARRAY_4_SIZE = 400         -- choose number of array variables
const LARGE_ARRAY_4_VARIABLE_SIZE = 4  -- choose size of variables (byte*4)
include large_array_4                  -- include the array library
alias test is large_array_4            -- rename/alias the array to test
```

Now we store some data and print it the computer.

```
-- store some values
test[50] = 0x1111_1111
test[200] = 0x2222_2222
test[26] = 0x3333_3333
test[27] = 0x4444_4444
test[00] = 0x5555_5555                      -- This is the first byte of the array.
test[57] = 0x6666_6666
test[300] = 0x7777_7777
test[LARGE_ARRAY_4_SIZE - 1] = 0x8888_8888  -- This is the last byte of the array.

-- read some values and print them
print_dword_hex(serial, test[50])
print_dword_hex(serial, test[200])
print_dword_hex(serial, test[26])
print_dword_hex(serial, test[27])
print_dword_hex(serial, test[00])
print_dword_hex(serial, test[57])
print_dword_hex(serial, test[300])
print_dword_hex(serial, test[LARGE_ARRAY_4_SIZE - 1])
```

There output printed to the serial port should look something like this:



There are many large array sample files in the Jallib sample directory.

Appendix
A

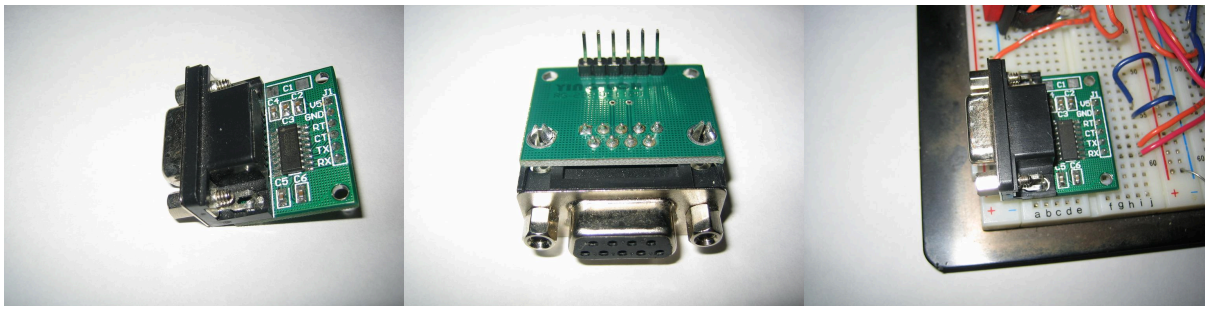
Appendix

Materials, tools and other additional how-tos

Building a serial port board with the max232 device

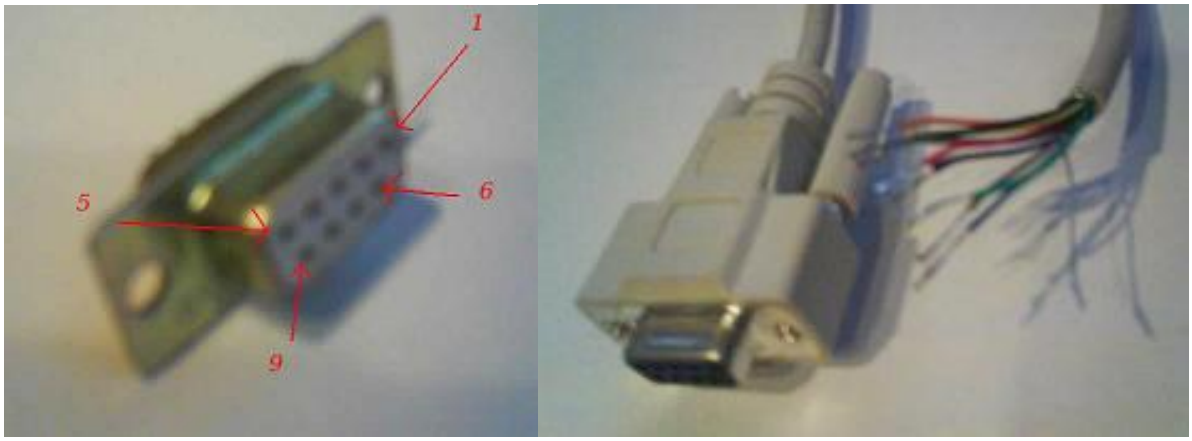
In this tutorial, we're going to build a serial port that can connect your PIC's TX and RX pins to your pc or other hardware using a max232 chip.

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.



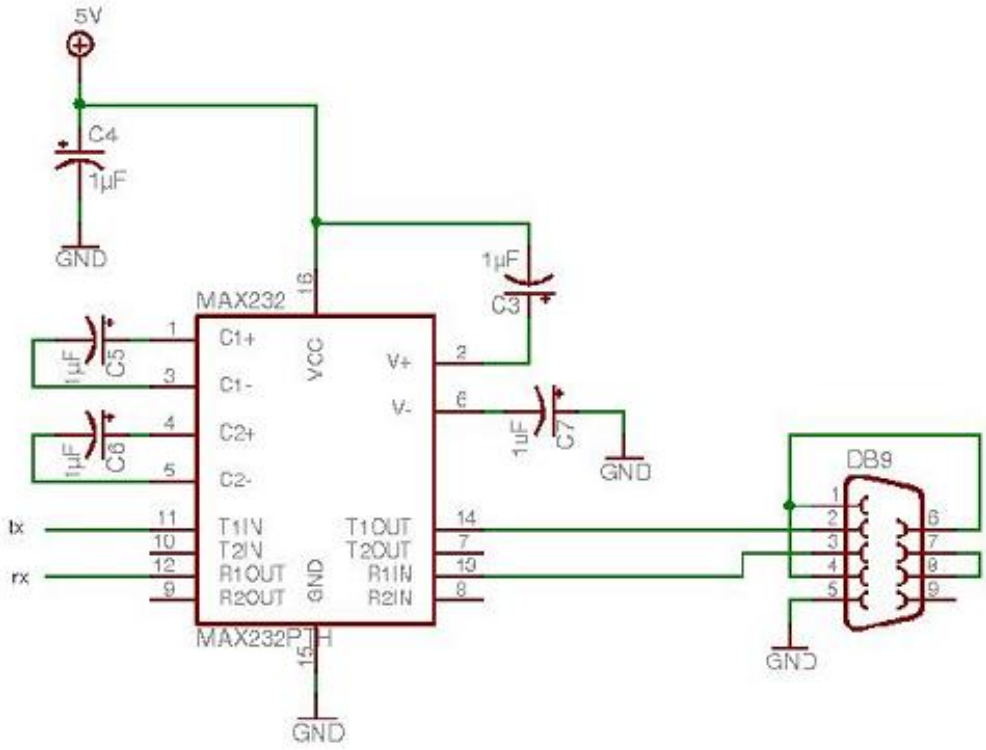
Now, lets build our own!

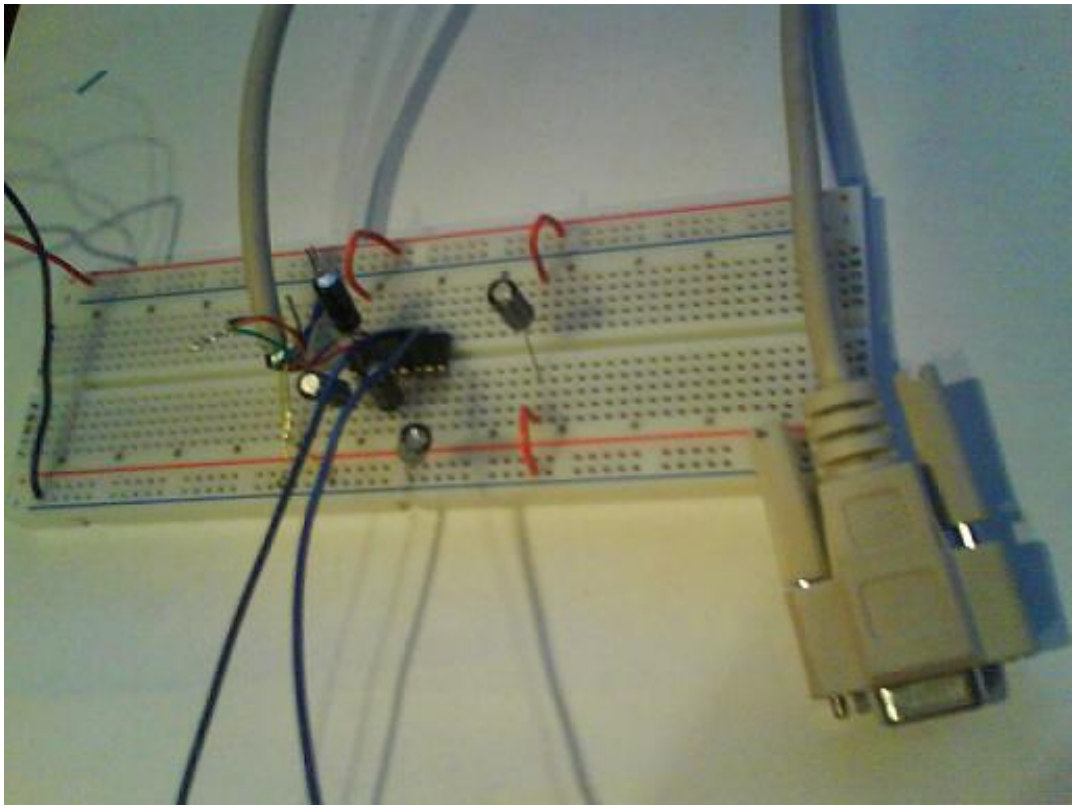
First get yourself a RS232 port, you can cut up one of your serial port cords, or buy a port from the store for a dollar or two.



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.

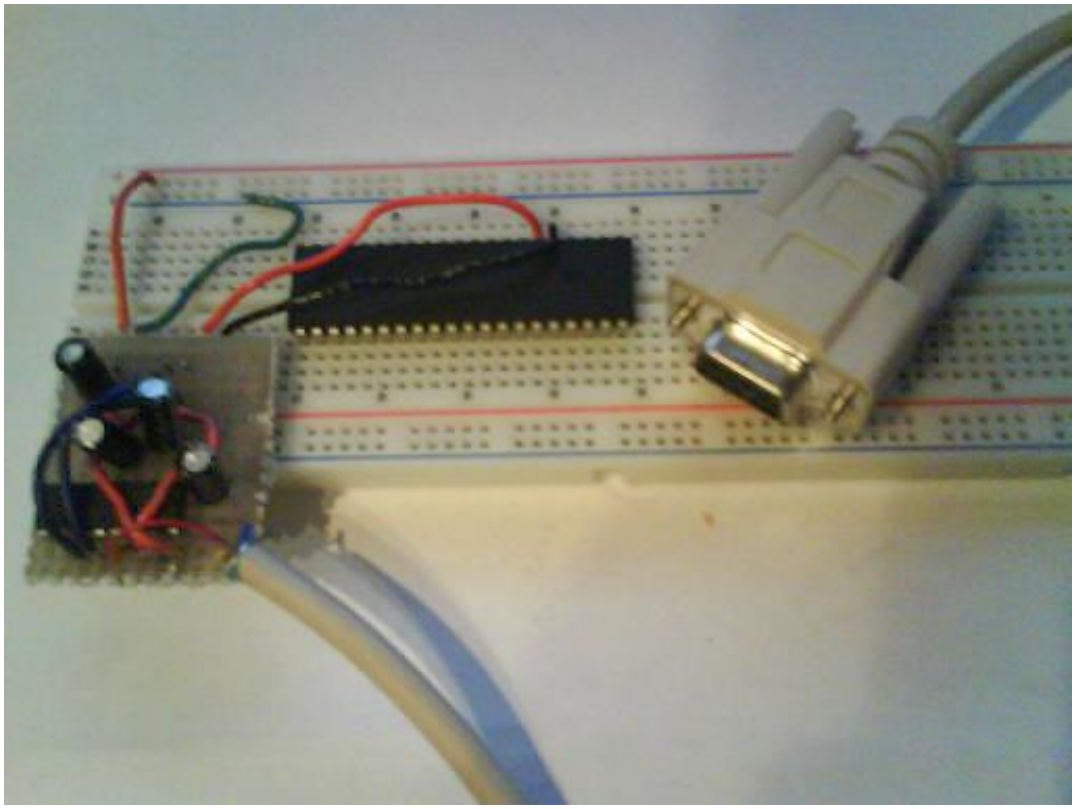




Great job, now connect the RX and TX pins to your circuit, and plug the rs232 port directly your pc, or to a usb-to-serial adapter, or even to a bluetooth-to-serial adapter for short range wireless.

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:



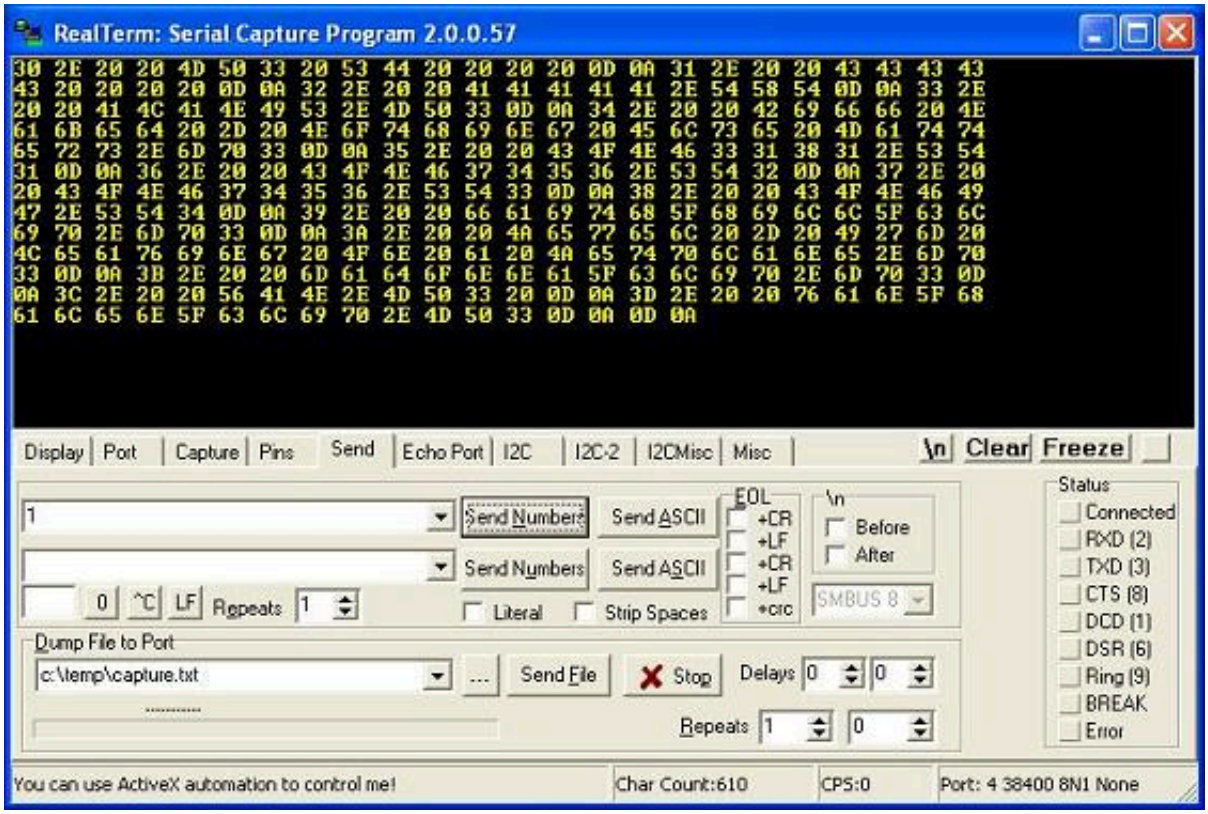
You can use serial_hardware lib or serial_software lib to transmit data to your pc, check for it in the other jallib projects. I suggest the software realterm for sending/receiving data to your PIC

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

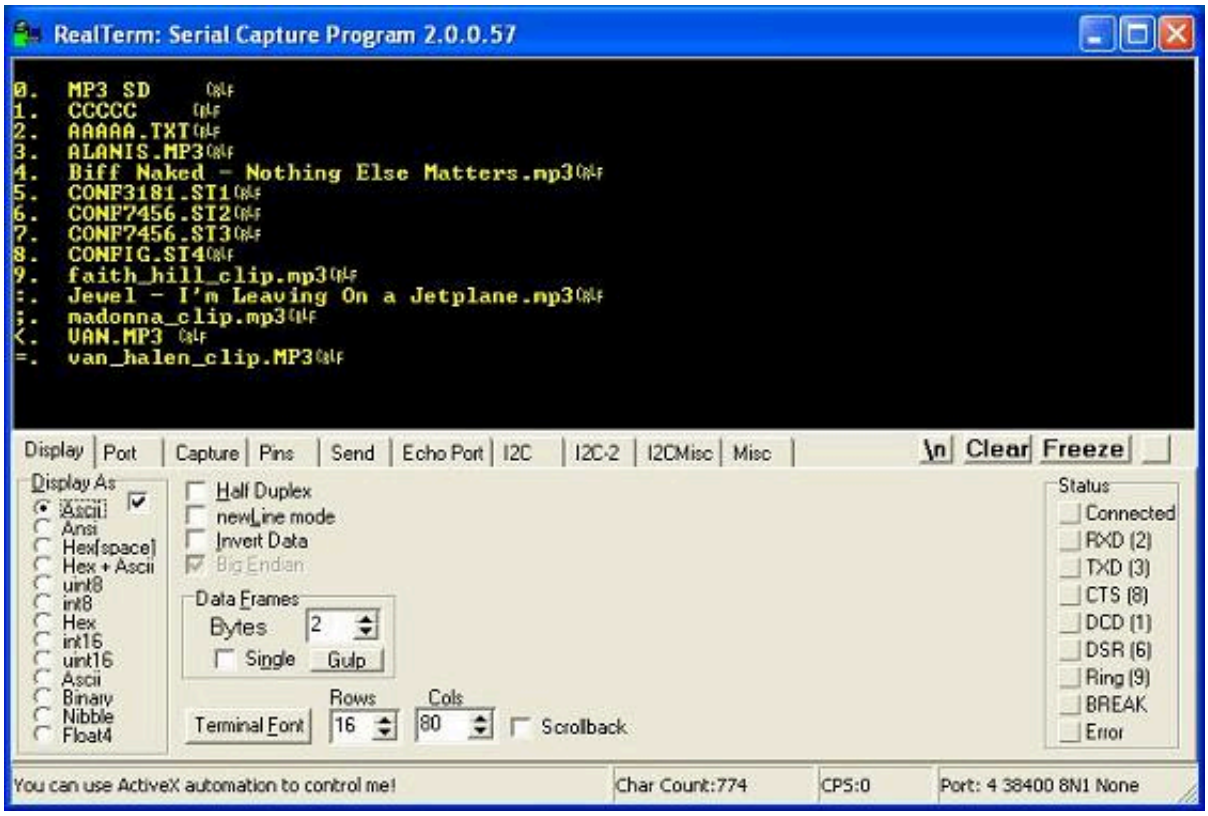
It can be downloaded for free from http://sourceforge.net/projects/realterm/files/

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

Hex output



ASCII output



In Circuit Programming

This tutorial introduces in-circuit serial programming (ICSP) for PIC microcontrollers and explains how to integrate it into your hardware.

What is ICSP?

In-Circuit Serial Programming (ICSP) lets you load firmware onto a PIC while it remains in the circuit. The programmer communicates through a small header, so there is no need to remove the device from the board each time you make a change.

For Microchip's formal definition of the protocol, see Microchip's ICSP Programming Specification.

Benefits of ICSP

ICSP streamlines development by letting you:

1. Program your PIC while it remains on a breadboard circuit
2. Program your PIC after it is soldered into a finished board
3. Load new firmware quickly so you can iterate on code faster
4. Reset the circuit directly from your development PC
5. Work with surface-mount PIC devices without removing them
6. Avoid bending or breaking pins from repeated insertion
7. Prevent damage caused by misaligning the PIC in a socket or breadboard
8. Program a remote project using tools such as VNC
9. Stay comfortable while developing - my favorite setup is programming from the couch while the hardware stays in the lab

Intro to ICSP & in-circuit programming

When I first started working with microcontrollers and JAL, I chose a programmer at random from an online auction site. It supported many PIC devices, which seemed good enough at the time.

For years I removed a PIC16F877 from my breadboard, programmed it, and then placed it back into the circuit. That routine bent pins and cost a lot of development time. I eventually noticed the programmer had an ICSP output - the front panel even said "ICSP" - but I still did not understand how to use it.

After hearing more about ICSP, I examined the six-pin header on the programmer. The connections were poorly labeled, so I used a multimeter, a logic probe, and occasionally an oscilloscope to watch each pin while programming. Pin 3 was clearly ground, and pin 6 appeared unused. If you already know your programmer's ICSP pinout, feel free to skip ahead to the circuit diagrams.

Because ICSP pin assignments vary by programmer, I measured the signals while a chip was idle and again during programming. The readings looked like this:

PIN #	While Idle	While Programming
1	0v	12v
2	0v	5v
3	0v	0v
4	5v	Pulsing 0v to 5v (random)
5	0v	Pulsing 0v to 5v (square wave)
6	not connected	0v - can see no connection on PCB

Get the pin names

The typical ICSP signals are VPP (program enable), VDD (target power), GND, PGD (data), PGC (clock), and an optional AUX or PGM pin. Matching these names to the voltages above reveals which signal each physical pin carries:

Pin 6 stays at 0 V, so it must be the unused AUX/PGM pin.

Pin 3 is the ground reference.

Pin 1 rises to about 12 V during programming, indicating VPP.

Pin 2 provides 5 V to the target, so it is VDD.

Pins 4 and 5 pulse while programming; they correspond to PGD (data) and PGC (clock). If all you have is a multimeter, you may need to guess which is which and confirm with a data sheet.

Most six-pin ICSP connectors follow this mapping:

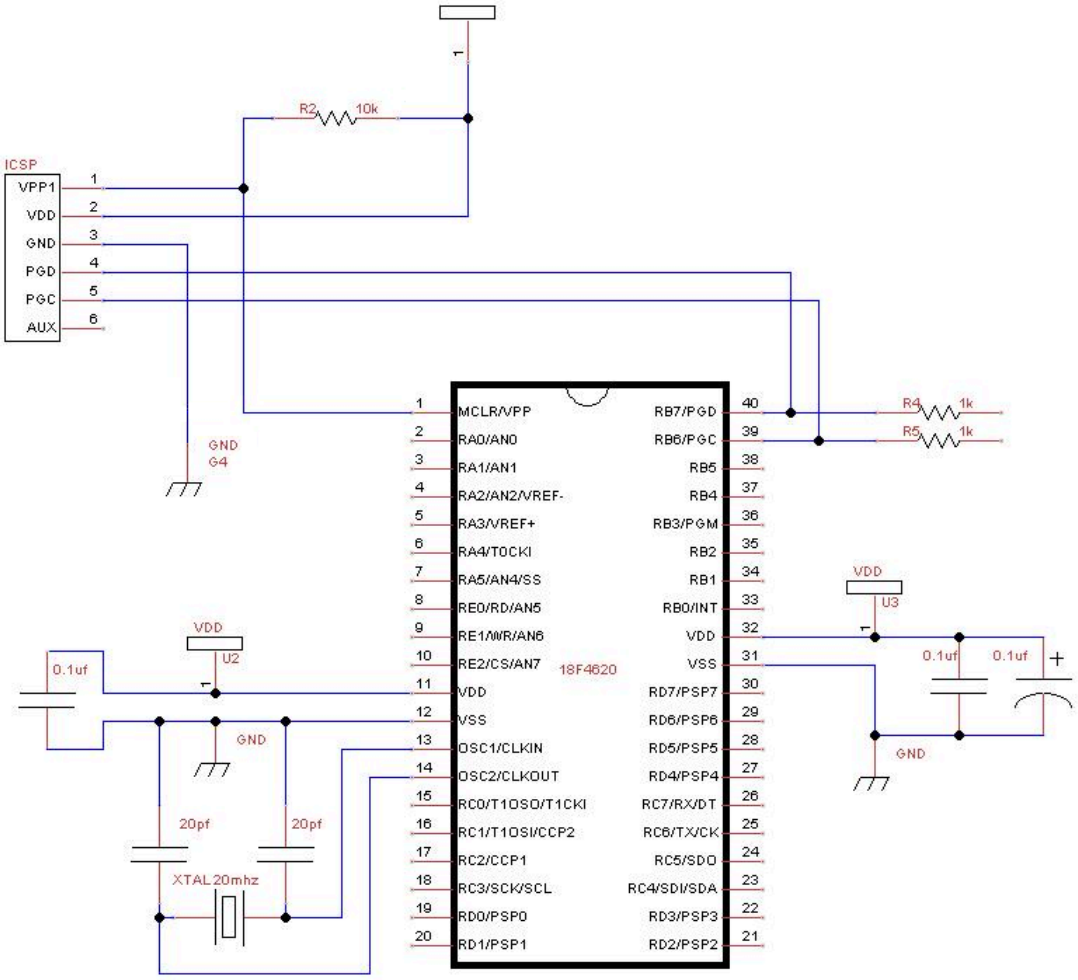
PIN #	PIN NAME	While idle	While Programming
1	VPP	0v	12v
2	VDD	0v	5v
3	GND	0v	0v
4	PGD	5v	Pulsing 0v to 5v (random)
5	PGC	0v	Pulsing 0v to 5v (square wave)
6	AUX/PGM	not connected	not connected

Build a circuit with ICSP

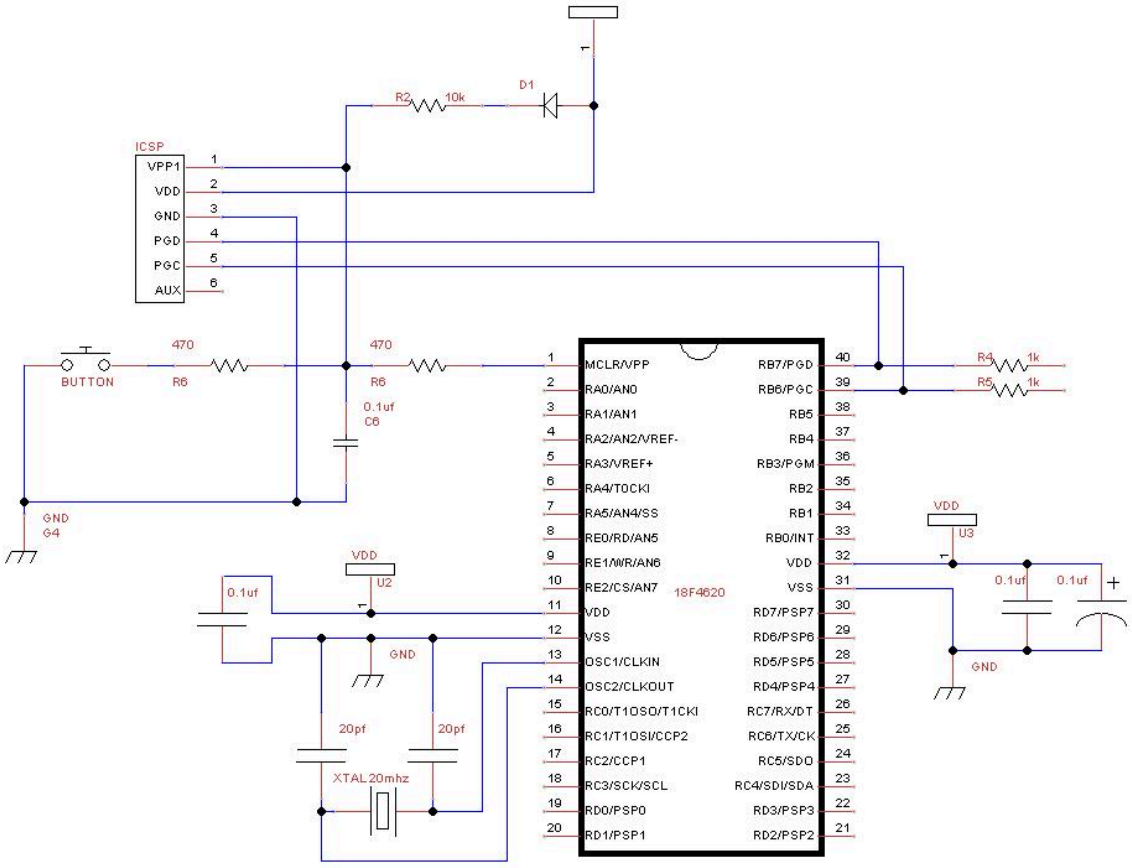
You can tie the ICSP VDD pin to your circuit's 5 V rail, as shown in the simplified schematic below. Some programmers supply power on this pin while others expect the target to provide its own supply. If your programmer does not require VDD, disconnect the wire once everything is working.

I rarely rely on power-off programming because my projects draw too much current when the supply is removed. Instead, I keep the circuit powered and let the programmer share the same ground.

GND must always connect to the project ground. Follow one of the schematics below to wire the header:



Note: This simplified schematic omits the reset button; the detailed version below includes it.



You're ready to go - apply power to the circuit and try programming your PIC.

Using Visual Studio Code with JAL

Introduction

As explained in the `Getting Started` section, any text editor can be used to develop JAL programs. To make life easier JAL Edit can be used, which supports syntax highlighting and much more. This editor, however, only works for Windows.

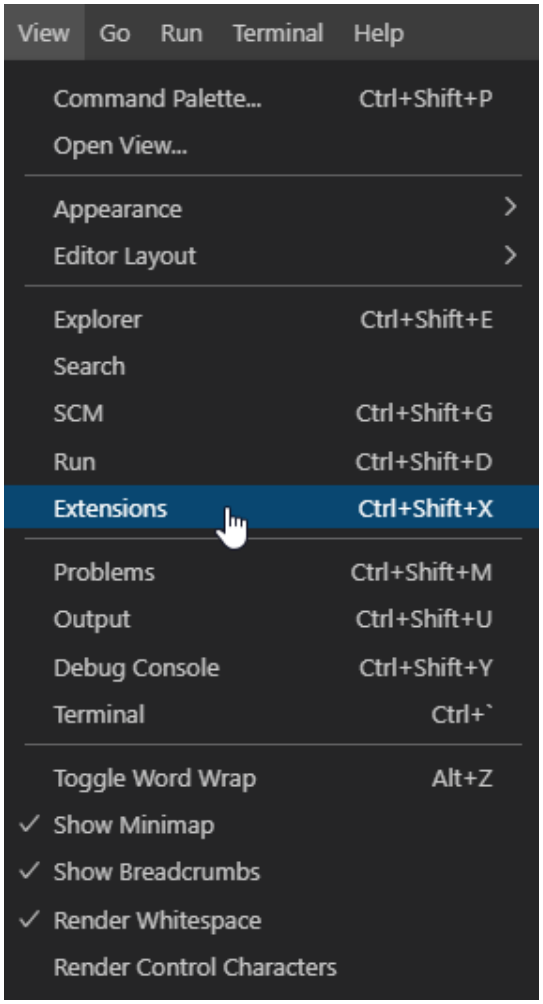
A good editor helps writing good programs and syntax highlighting makes life easier. In order to support more than only the Windows platform, Microsoft Visual Studio Code can be used, which runs on Windows, Linux and MacOS. Visual Studio Code can be `downloaded` for free. A JAL Visual Studio Code extension is available to support programming in JAL.

This section describes how to install the Visual Studio Code JAL extension and how to use Visual Studio Code for your JAL development.

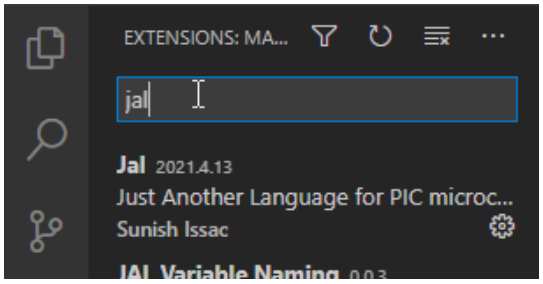
Downloading the JAL Visual Studio Code extension

After having installed Visual Studio Code you can download the VS Code JAL extension from the VS Code extension market place. The steps are as follows.

From View --> Extensions open the extensions bar. which will appear bar on the left side of the screen.



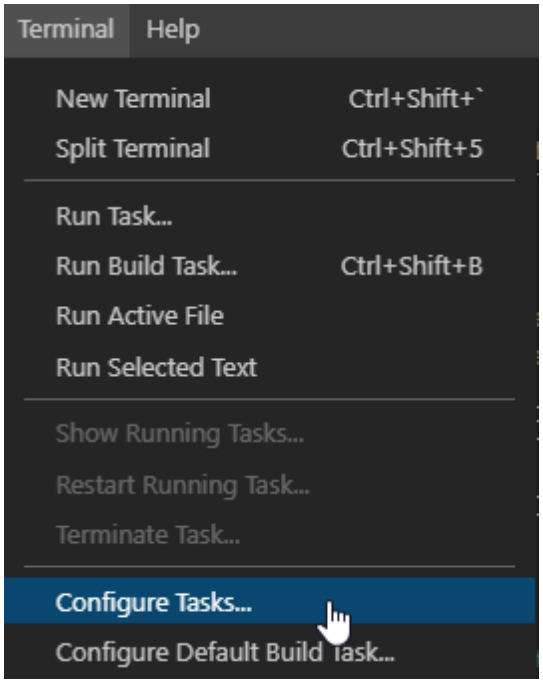
Type ‘jal’ in the search bar and you will find the JAL extension.



Now install the extension.

Installing a task to run the compiler

Select Terminal --> Configure Tasks ...



You can create a default `tasks.json` from a template. You can download the `tasks.json` from the GitHub or you replace the example by the following:

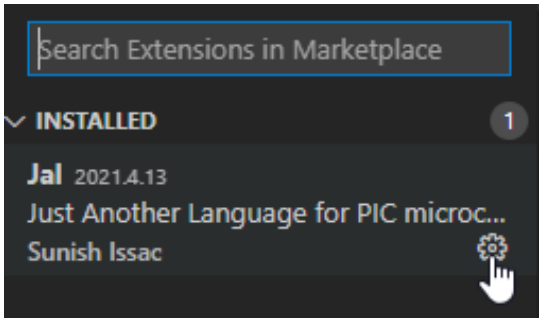
```
{
    // See https://go.microsoft.com/fwlink/?LinkId=733558
    // for the documentation about the tasks.json format
    // prefilled tasks.json for compiling a JAL file
    "version": "2.0.0",
    "isBuildCommand": true,
    "tasks": [
        {
            "label": "Compile JAL File",
            "type": "process",
            "command": "${config:jal.paths.exePath}",
            "args": [
                "${file}",
                "-s",
                "${config:jal.paths.LibPath}"
            ],
            "presentation": {
                "reveal": "always",
                "panel": "new"
            },
            "problemMatcher": [],
            "group": {
                "kind": "build",
                "isDefault": true
            }
        }
    ]
}
```

You can download this piece of code from `GitHub` or copy-paste the code below in the `tasks.json`.

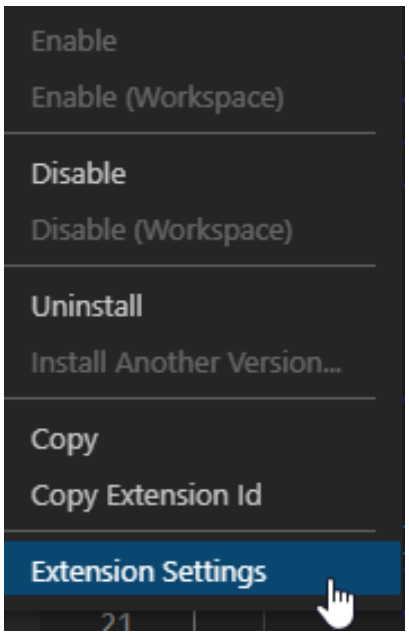
```
{
    // See https://go.microsoft.com/fwlink/?LinkId=733558
    // for the documentation about the tasks.json format
    // prefilled tasks.json for compiling a JAL file
    "version": "2.0.0",
    "isBuildCommand": true,
    "tasks": [
        {
            "label": "Compile JAL File",
            "type": "process",
            "command": "${config:jal.paths.exePath}",
            "args": [
                "${file}",
                "-s",
                "${config:jal.paths.LibPath}"
            ],
            "presentation": {
                "reveal": "always",
                "panel": "new"
            },
            "problemMatcher": [],
            "group": {
                "kind": "build",
                "isDefault": true
            }
        }
    ]
}
```

Configuring the JAL extension

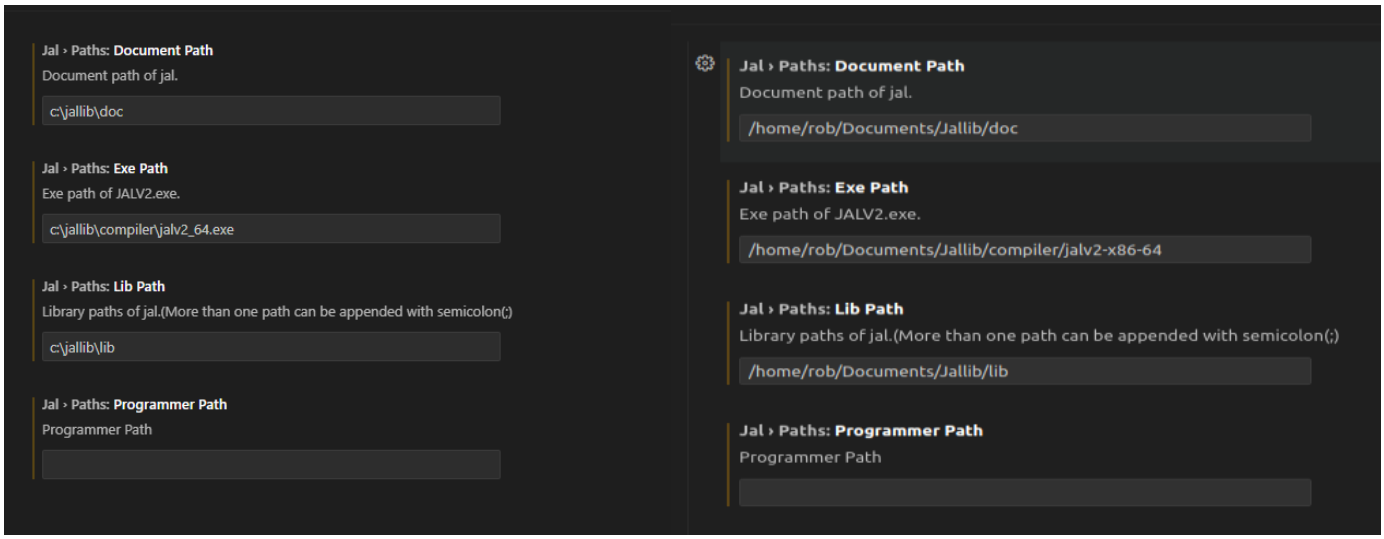
On the installed plug-in click on the ‘manage’ (or the configuration) wheel.



Select Extension Settings.



Set the paths to the JAL documentation, the JAL compiler, the JAL libraries and programmer (if present). The example below on the left is for Windows, the example below on the right is for Linux:



The final result

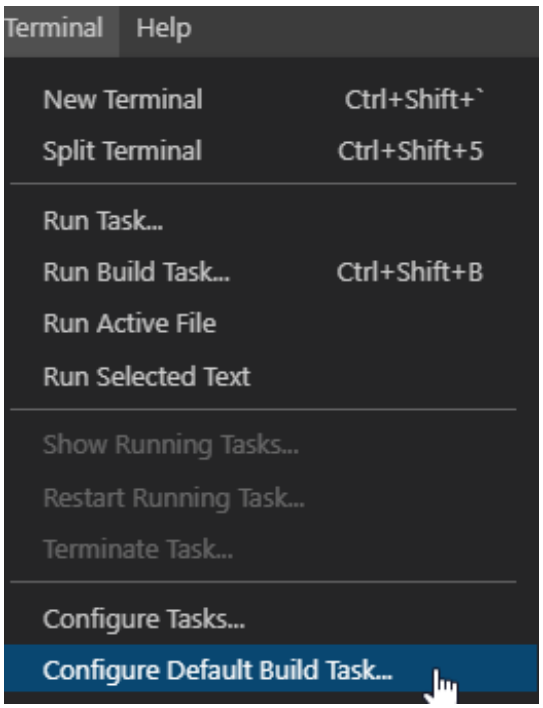
When done, your JAL blink-a-led program could look like this:

```
--
include 12f617                    -- target PICmicro
--
-- This program assumes that a 20 MHz resonator or crystal
-- is connected to pins OSC1 and OSC2.
pragma target clock 20_000_000      -- oscillator frequency
--
pragma target OSC      HS                       -- crystal or resonator
pragma target WDT      DISABLED                 -- watchdog
pragma target BROWNOUT DISABLED                 -- no brownout reset
pragma target MCLR     EXTERNAL                 -- external reset
--
-- The configuration bit settings above are only a selection, sufficient
-- for this program. Other programs may need more or different settings.
--
--
enable_digital_io()                -- make all pins digital I/O
--
-- A low current (2 mA) led with 2.2K series resistor is recommended
-- since the chosen pin may not be able to drive an ordinary 20mA led.
--
alias  led       is pin_A0          -- alias for pin with LED
--
pin_A0_direction = OUTPUT
--
forever loop
   led = ON
   _usec_delay(100_000)
   led = OFF
   _usec_delay(400_000)
end loop
--
```

You can select your own theme to suit your needs.

Activating the compiler

You can set the default build task to run the compiler under Terminal --> Configure Default Build Task ...



The compiler can be activated by:

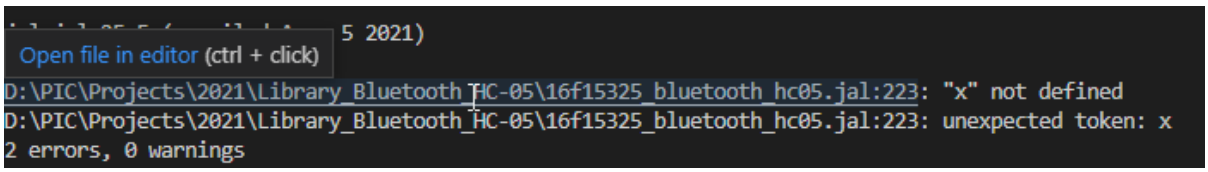
- Terminal --> Run Task … and selecting the task Compile JAL File
- Terminal --> Run Buld Task ...
- Using the shortcut `ctrl-shift-b`

Note: If you have more files open in your editor, always activate the compiler form the main program.

The compiler output appears at the bottom of the screen in a terminal window and may look like this:

```
jal jalv25r5 (compiled Apr  5 2021)
generating p-code
1597 tokens, 375210 chars; 8263 lines; 13 files
generating PIC code pass 1
generating PIC code pass 2
405 branches checked, 0 errors
1073 data accesses checked, 0 errors
276 skips checked, 0 errors
writing result
Code area: 4102 of 8192 used (words)
Data area: 334 of 1024 used
Software stack available: 80 bytes
Hardware stack depth 5 of 16
0 errors, 0 warnings
```

If a compiler error occurs you can jump to that error in the editor by pressing `ctrl-click`.



Changing highlighted keywords

The Visual Studio Code extension for JAL comes with a default highlighting keywords. If you want to change this you can edit the file `jal.tmLanguage.json` which you can find in a hidden folder under `.vscode`

`\extensions\sunish.vscode-jal-2021.4.13\syntaxes`. An updated version of this syntax file can also be found on GitHub.

JAL VS Code extension features

Features that work well:

- Syntax Highlighting
- Fast opening and saving of files irrespective of the size
- Linux and Windows support Code folding
- Auto completion (More work needs to be done to have functions/procedures included from include files)
- Code Snippets (Only very few are added, but not very difficult to add)
- Compiling to Hex file
- Ctrl-Click to go to error line (It doesn't go automatically)
- Searching/Replacing any word within files and across folder
- Direct Github Push/Pull/Diff other commands
- Side by side View and file comparison
- Theme selection/switching
- Folder/explorer view

Some of the main features not in the extension:

- Code Explorer for include files,procedures,functions,variables,constants,aliases
- Opening include files with Ctrl-Enter
- Auto Backup with time stamp/compilation
- Backup project as zip file
- Go to error line after compilation
- Running programmer on successful build
- Serial Terminal
- Compile/Program buttons/keys
- Detecting PIC name from code and passing it as variable to Programmer executable
- Set file as Active JAL file and compile that irrespective of file you are editing

Making a Tutorial

Introduction

This tutorial gives a brief instruction on how to create a new tutorial for the Tutorial Book. It will not explain the elements used within a tutorial since that is already explained in detail in the file `template.xml` that can be used as a starting point.

Before starting with a new tutorial you can have a look at the existing tutorials to see how they are built up. When you start, you can make a copy of the previously mentioned template file or make a copy of a tutorial that has more or less the format and layout that you want to use. Rename the xml that you copied file to the subject of your tutorial. The format of such a tutorial xml file is `tutorial_my_subject.xml` where the filename is only using small caps. It is recommended to use a logical name for the subject that matches the subject of your tutorial.

Required tooling

It is not very easy to write xml documents so we will be using a tool for writing our tutorial. A nice XML editing tool that is available for Windows, MacOS and Linux is XMLmind which can be downloaded and used for free for personal use or for open source development. Before writing your tutorial open the `template.xml` file to see which features can be used like creating sections, paragraphs, unsorted lists, etc.

Writing your tutorial

When writing a tutorial make use of references (external and internal), pictures and videos where possible. A picture says more than a thousand words. Keep the 'instructions' in mind as given in the template. For example external references should have the `scope external` and pictures should have the `otherprops clickable`. Of course the clickable feature is only available in the HTML version of the Tutorial Book on the JAL site.

Adding your tutorial to the Tutorial Book

Once you completed your tutorial it has to become part of the Tutorial Book. Depending on the topic it can be in the section about PIC internals, external parts or another section. In order to add your tutorial to the Tutorial Book you have to perform the following steps:

- Open the file `tutorials.ditamap`
- Insert a new `topicref`
- Fill in the name of your tutorial xml file in section `href`
- Save this file

After having done all the work you should upload both your new tutorial xml file and the `tutorials.ditamap` file to GitHub. If you where using pictures, make sure that you also upload them to the `images` directory. When a new Tutorial Book is recreated, your part will be in it.

Updating the change log

At the end of the Tutorial Book, the change log can be found which is stored in the file `tutorial_changelog.xml`. You should add an extra entry in the most recent table and write what you did, e.g. adding a new tutorial or updating an existing tutorial.

Note: While the Tutorial Book is updated, it remains a draft until it is released. When released, the release date is completed and the Tutorial Book gets a new version number.

Note: The version number and the release date are also mentioned in the `bookmeta` on the first page of the Tutorial Book in the file `tutorials.ditamap`. Before releasing, this data should be updated too.

Making sure your tutorial is published on the JAL Website

In the dita folder on GitHub you find a file called `TOPUBLISH`. This is a plain text file and you should add your tutorial to this file in the format that it is described in the file. It indicates which tutorials should be published as HTML to the JAL Website.

Jallib Style Guide

Why a style guide?

The **Jallib Style Guide** (JSG) defines the standards used to write Jalv2 code. It is recipe to write a standard jalv2 library.

There are many ways to write code, whatever the programming language is. Each language has its preferences. For instance, java prefers *CamelCase* whereas python prefers *underscore_lowercase*.

While this seems a real constraint, not necessarily needed, it actually helps a lot while sharing code with everyone: it improves **readability**, and readability is important because code is read much more often than it is written.

Finally, more than a how to write code, this guide is here to help you not forget things like *author(s)*, *licence*, and remind you of some basic principles.

Headers in a library

Every jal file published on this repository must have the following headers (comments), as the very beginning of the file:

```
-- Title: [very small description if needed]
-- Author: [author's name], Copyright (c) YEAR..YEAR, all rights reserved.
-- Adapted-by: [adapter author's name, comma separated]
-- Compiler: [which version of compiler is needed. Ex: >=2.4g, 2.5]
--
-- This file is part of jallib (https://github.com/jallib/jallib)
-- Released under the ZLIB license (http://www.opensource.org/licenses/zlib-license.html)
--
-- Description: [describe what is the functional purpose of this lib]
--
-- Sources: [if relevant, specify which sources you used: website, specifications, etc...]
--
-- Notes: [put here information not related to functional description]
--

[code starts here...]
```

The **author** is the original author's name. The library may have been modified and adapted by **adapters**. The **compiler** helps readers to know which compiler version is needed to use this file (no space between operator and version: >=2.5r6). **Description**, **sources** and **notes** fields must be followed by an empty line (just `--`) to declare the end of the field content. As a consequence, those fields cannot have empty lines within them.

JSG Header example:

```
-- -----------------------------------------------------------------------------
-- Title: Library for the DS3231 Real Time Clock IC.
-- Author: Rob Jansen, Copyright (c) 2021..2022, all rights reserved.
-- Adapted-by:
-- Compiler: 2.5r6
--
-- This file is part of jallib (https://github.com/jallib/jallib)
-- Released under the ZLIB license (http://www.opensource.org/licenses/zlib-license.html)
--
-- Description: Library for controlling the SD3231 Real Time Clock IC.
--              The chip uses an IIC interface. The library provides all
--              functions and procedures to support the rtc_common.jal library and
--              includes extra functions and procedures specific for the DS3231.
--              For all common rtc procedures and functions see rtc_common.jal.
--
-- Sources: Maxim datasheet rtc. 19-5170; Rev 10; 3/15
--
-- Notes: This library supports the control of the DS3231 via IIC.
--        The default is hardware IIC control but this can be overruled using
--        software IIC control by defining the following constant:
--        -) const RTC_SOFTWARE_IIC = TRUE
--
--        The following pins must be defined by the main program before
--        including this library. Common pins for using IIC:
--        -) alias rtc_sck             -- IIC to sck of rtc
--        -) alias rtc_sck_direction
--        -) alias rtc_sdo             -- IIC to sda of rtc
--        -) alias rtc_sdo_direction
--
```

Note: if you need to create a new paragraph within a multiline field, use the "--" special chars. See example in Notes field: "The following pins ..." is part of the Notes field, but visually separated from the beginning of the field content.

In the `/tools` directory on GitHub you'll find `jallib.py` and `jallib3.py` for Python version 3 and higher. Among many things, you can run the "validate" action, and check lots of JSG requirements. You can (must) use it to make sure that all your jal files are JSG compliant. This script will help you to identify problems:

Example:

```
bash$ python jallib.py validate my_file.jal

File: my_file.jal
4 errors found
        ERROR: Cannot find field Title (searched for '^-- Title:\s**(.**)')
```

```
        ERROR: Cannot find field Author (searched for '^-- Author:\s**(.**)')
        ERROR: Cannot find field Compiler (searched for '^-- Compiler:\s**(.**)')
        ERROR: Cannot find field Description (searched for '^-- Description:\s**(.**)')

0 warnings found
```

Filenames naming convention

A library must be named as:

- `<function>_<implementation|other>.jal` for PIC-specific libraries (peripherals). `function` gives clues about what the library is about. Then `implementation` or `other` is here to differentiate libraries, and is more about implementations (`serial_hardware.jal`, `serial_software.jal`), things specific to the function (`pwm_ccp1.jal`, `pwm_ccp2.jal`, ...).
- `<device-family>_<device>.jal` for external libraries. `device-family` describes the device family (...), and is often the directory name where the lib is. `device` precisely sets the part (`lcd_hd44780_4.jal`, `rtc_ds1302.jal`, `co2_t6603.jal`.).

Variables, constants and procedures naming convention

All **external names** (of global variables, constants, procedures and functions available to application programs) **must** start with a prefix unique to the library. Names of other global entities (not supposed being used by application programs) should use this prefix and use an additional underscore at the beginning.

Variables, constants, procedures and functions must be named as:

- `<device>*<whatever>` if you want to avoid name space collision
- `<device-family>*<whatever>` if you want to have a common API

For example, `co2_t6603.jal` library have all its procedures starting with `t6603_` (and `*t6603*` for internal names). This makes all these procedures very specific to this library. If you have another CO2 sensor, you'll be able to use both at the same time, because they'll be no name space collision. This is the purpose of the `<device>*<whatever>` naming convention.

Another example: the names of the procedures in the LCD libaries start with `lcd*` (and `*lcd*` for internal names). There are many different LCD types, but all implements the same API, because procedures, variables, etc... are named according to the device-family, not the device itself. This is the purpose of the `<device-family>_<whatever>` naming convention.

Now, how do you know which to follow? Ask and we will discuss.

Note: Following the same principle, naming const/var/procedure/function in a PIC-specific libraries (peripherals) can include the *function* and/or the *implementation*. This depends whether you want to have more than one function within a same PIC.

Example: There are two implementations of i2c and serial: hardware and software. Having both i2c implementation within a same PIC is not useful, since i2c is addressable. Thus, all const/var/... are prefixed by `i2c_<whatever>.jal`. On the contrary, it can be useful to have two serial implementation within a same PIC (eg. one talking a PIC, another talking to a external device). Thus, serial libs' const/var/... are prefixed by `serial_hw_<whatever>.jal` or `serial_sw_<whatever>.jal`.

Headers of procedures and functions

It is recommended to use the following header layout for procedures and functions. Return values are of course only applicable for functions. Notes are optional. When no parameters are passed, mention 'None'.

```
--------------------------------------------------------------------------------
-- Description: <purpose of this procedure or function>
-- - <this may be a long description, continue on this line>
-- Parameters:
-- - <parameter> - <description>
-- - <parameter> - <description>
-- Returns:
-- - <returnvalue> - <description>
-- Notes: <some optional notes can go here for this procedure>
-- - <this may be a long description, continue on this line>
--------------------------------------------------------------------
```

So for example:

```
--------------------------------------------------------------------------------
-- Description: Calculate CRC7 for byte array (one-shot calculation)
-- Parameters:
-- - data   - array of bytes to process
-- - length - number of bytes
-- Returns:
-- - calculated CRC7 value
-- Notes:
--------------------------------------------------------------------------------
```

Pin names naming convention

The pins are named as:

- `<device>_<external_pin_name>` if you want to avoid names pace collision
- `<device-family>_<external_pin_name>` if you want to have a common API

This is almost the same as for variables, contants, ... except the `<whatever>` part now corresponds the pin name of the external device (usually found in datasheets). Using the `<device-family>_<external_pin_name>` convention to build a common API may cause problems, if pin names aren't named the same in all supported devices. In that case, the pin name should be as explicit as possible.

Important: See also the very important rules about pin names within a library: "Don't use port and pin names".

Samples and test naming convention

Tests are named as `test_<whatever>.jal`. That is, they should starts with the prefix `test_`. That is, samples must **not** start with `test_`.

Board files are named as `board_<pic>*<whatever>.jal`.

Samples are named as `<pic>*<whatever>.jal`. `<whatever>` can be whatever, but should give users hints about what the sample is (e.g. `16f88_serial_hardware.jal`).

Why such a pain?

The main purpose of this is to control the naming conflicts between libraries and application code. Bear in mind that this is about source-level libraries which are combined by the compiler to form a single application program.

Having naming convention is also a great optimize process, saving time, by scripting and generating code. This is good.

Don't use port and pin names

Don't use port and pin names like `portA` or `pin_a5` in your great library, because someone may (will) want to use your library on another port or pin. It also helps to make your great library PIC independent.

Name your pins according to the context, to what your library is doing. Client code, i.e. users, will have to define those variables before actually include your great library.

Let the user set the pin directions, except if the library is supposed to modify direction during execution.

Example: how to use your library (doing amazing things with the GP2D02 IR ranger):

```
-- declare in/out pins for the ranger
alias ranger_pin_in          is pin_A0
alias ranger_pin_out         is pin_A1
```

and make sure the pins work as required:

```
-- specify the direction of the pins
-- Since directions won't change during execution, this is
-- done here, during the setup, before including the library
pin_A0_direction = input
pin_A1_direction = output
```

and now include the library:

```
-- now include the library which uses ranger_pin_in and ranger_pin_out
include gp2d02            -- ranger library
```

Exception: If your library uses a special PIC feature, it may use the name defined in the device files / datasheet. Not so much an exception, as you'll use the pin name given the context (feature, peripheral).

Note: Syntax `"var ... is ..."` is now deprecated in favor of `"alias ... is ..."` and must not be used anymore. The `"alias"` keyword is more powerful as it allows to create synonyms for any type of names (variables, constants, procedures, functions, pseudo-variables).

Example: An i2c hardware library (using built-in PIC i2c) may refer to `SCK` and `SDA`. Those pin names are set into the device include file (prefixed with the portname!).

Let the user initialize the library

Most of the time, a library needs to be configured (you define variables/constants before including the file), then initialized (you call `<libname>_init()`). While having the init step automatically called when the library is called can be convenient, this results in a lack of flexibility. Indeed, you may want to initialize one library or the other, or initialization step can take quite a long time, so you want to have control about when you can "waste" such time.

So, **a library must never call its own init procedures, the user will**. And the init procedure must be named either as `<device>_init` or `<function>_init`, whether you want to avoid names pace collision, or on the contrary, if you want to have different implementation for the same API (see rules about naming convention above).

Avoid weird default values in library

Don't put default values in your library, someone may (will) have a different opinion about what's a *default value* . Even if it's tempting because it can save time writing the same value again and again. Remember, your library is to be shared, nasty default value can be a real obstacle using it. If for some reason default values have to be used, make sure that you provide a means for the user to overrule oror change them.

Write examples

Write examples to show the world how to use your great library. Without it, people may (will) not use your library, because it's too complicated and too time-consuming reading code to actually discover what it does. Also remember writing examples can help *you* to design a usable, simple and clear API.

Assembler

Avoid the use of inline Assembler. If you cannot do without it use **standard asm opcodes** and avoid nasty Assembler statements. So:

Good

```
btfsc STATUS_Z
```

Bad

```
skpnz
```

Warnings are errors

Don't be tempted to ignore warnings. **Consider warnings as errors**, until you've completely understand why there should be a warning (or not). Warnings can mask more relevant warnings and errors, so track them and try to avoid them. A library **should** compile **without any warnings**... if possible but it **must** compile **without any errors**.

Code layout

Indent your code. It helps following the code structure (flows). Code must be indented using **3 spaces** (no tab). You can use `python jallib3.py reindent <file.jal>` for this.

Good

```
var byte char
forever loop
   if serial_hw_read(char) then
      echo(char)
   end if
end loop
```

Bad

```
var byte char
forever loop
if serial_hw_read(char) then
echo(char)
end if
end loop
```

Use lower_case_with_underscores ...

Good

```
var byte this_is_a_variable
var byte another_one
```

Bad

```
var byte ThisIsAVariable
var byte Another_One
```

... except for constants

Uppercase variables should be used for constants, internal PIC function registers or for external PIN names, if they are uppercase in the datasheet as well.

Good

```
const RESET_CHAR = "*"
const SSPCON1_CKP = 1
```

Bad

```
const reset_CHAR = "*"
const sspCON1_Ckp = 1
```

Be explicit when calling procedures and functions

When a procedure (or a function) does not take any parameters, be explicit and help your readers: put parenthesis so everyone knows it's a call. Same when defining the function/procedures. Also note no space is allowed between the procedure/function name and the opening parenthesis. Finally, pseudo-variable must be defined with parenthesis, but not when used (heh, these are functions/procedures behaving like variables!).

Good

```
-- Defining
procedure do_it_please() is
    -- I will do it
end procedure

-- Calling
do_it_please()

-- pseudo-var
function my_pseudo_var'get() return byte is
    -- I promise I'll do it
end function

var byte what = my_pseudo_var
```

Bad

```
procedure do_it_again is
    -- this is bad
end procedure

do_it_again

function my_pseudo_var'get () return byte is
    -- this is bad, too because there's a space !
end procedure
```

Filenames are lower cased, include statements too

All jal files must be lowercased. So:

Good

```
$ ls 16f88.jal
```

Bad

```
$ ls 16F88.jal
```

Being consistent, include statements are lower cased, too:

Good

```
include 16f88
```

Bad

```
include 16F88
```

Inform readers what should be considered private

Functions, procedures, variables, etc... starting with an underscore is a warning to users saying "you shouldn't use me, I'm for internal use only". Play carefully with this, remember users are quite curious and may want them anyway :).

Comment your code

It helps readers to understand what's going on. The comment should describe **why** your code does its thing, not what is does. That should be obvious from the code itself.

External data

When developing a library, you may need to collect and organize external / 3rd party data. For instance, the relation between a datasheet reference and the PICs described in this datasheet is what we call external data: it's not jal code, but *often* used to generate some, and *always* a source of information everyone can refer too.

External data must store in a **structured format** so everyone potentially is able to use it. Before we, developers, are also (kind of) humans, we want this format to be readable, and even writable, but also structured enough so a computer can also use and exploit it. That's why this format is `JSON` (and not XML), which is available in many languages. This is a way to share information, among the many scripts used to deal jal code base.

Making a Library

Introduction

In addition to the Jallib style guide this tutorial gives some additional information on how to create a JAL library. The purpose of a JAL library is to make it easy for the user to use JAL or to use a specific hardware device in an easy way. A library normally converts all kind of - sometimes complex - hardware register settings into easy to use procedures or functions. There are also pure software libraries that, for example a library that generates a random number.

Mandatory library requirements

The most important requirement of a JAL library is that it complies with the Jallib style guide and that it compiles without any warnings and errors. As mentioned in the style guide this can verified by running the validate function of jallib3.py.There may be no warnings or errors when running this validation.

As mentioned in the style guide, the user has to define the pins - and the pin direction - used by the library using aliases. If a library uses an interface, e.g. the SPI interface, the initialization of this interface has to be done by the user and not by the library. The library has to described the settings it needs for this interface.

Support as many PICs as possible

It is important that a library is easy to use, easy to maintain and can be used by a variety of PICs. The following sections describe some ways how you can obtain that.

Various PICs have different registers for the same on-board hardware like a Timer. In order to support as many PICs as possible check if certain registers are present and use aliases to cover the variants, for example:

```
-- Support for newer PICs (based on 16f15325).
if defined(T1CLK) then
   alias stopwatch_cs is T1CLK_CS -- timer clock source bit
   alias stopwatch_ie is PIE4_TMR1IE -- interrupt enable
   alias stopwatch_if is PIR4_TMR1IF -- interrupt overflow bit
else
   alias stopwatch_cs is T1CON_TMR1CS -- timer clock source bit
   alias stopwatch_ie is PIE1_TMR1IE -- interrupt enable
   alias stopwatch_if is PIR1_TMR1IF -- interrupt overflow bit
end if
```

Write clean code

This in not only valid for libraries but for all code that you write. Writing clean code makes it easier to read and maintain the library. My favorite book on how to write clean code is the Clean Code book written by Robert C. Martin. Some clean code practices are:

- Use logical names for constants, variables, procedures and functions that explain what the constant or variable means or what the procedure or function does.
- A procedure or function does only one thing. Do not pass for example a boolean variable to control the functionality of the procedure or function.
- Limit the size of a procedure or function. There is no strict rule on this but use common sense. If a procedure or function becomes too big, split-it up in several smaller procedures or functions.
- Only use comments when needed. Code should be simple, self explanatory and easy to understand without the need for additional comments.

Separate the definition of the API from the implementation

When using a library, it can be quite problematic to get a complete overview of all provided functionality. In order to improve this you can define all public procedures and functions at the start of the library. I call this section the Public API of the library and should contain only the functions and procedures that are relevant for the user of the library. This overview can be obtained by using `function prototypes` (which is the definition of a procedure or function without the `is`). Always include in the definition what the procedure or function does, for example:

```
-- -----------------------------------------------------------------------------
-- Intialize the RDA5807M. The device is initialized for Europe as follows:
-- -) Set the band to RDA5807M_BAND_US_EUROPE
-- -) Set the spacing to RDA5807M_SPACING_100_KHZ
-- -) Set the de-emphasis to RDA5807M_DEEMPHASIS_50_US
-- The device is reset and powered up.
-- -----------------------------------------------------------------------------
procedure rda5807_init()

-- -----------------------------------------------------------------------------
-- Enable the RDA5807M.
-- -----------------------------------------------------------------------------
procedure rda5807m_enable_power()

-- -----------------------------------------------------------------------------
-- Set the volme of the RDA5870M. Volume must be in range RDA5807M_VOLUME_MIN
-- to RDA5807M_VOLUME_MAX.
-- -----------------------------------------------------------------------------
procedure rda5807_set_volume(byte in volume)
```

When using global variables in your library of which the value is also needed by the main program, use a procedure to change that variable and where possible a function to return its value. So in fact you are hiding the implementation of your library from the user of the library. This can also help to keep the library backwards compatible in case you want to change the implementation. For example:

```
-- -----------------------------------------------------------------------------
-- Set the library to support the extended protocol.
-- -----------------------------------------------------------------------------
procedure nec_rc_set_protocol_extended() is
   _nec_rc_protocol_standard = FALSE
end procedure

-- -----------------------------------------------------------------------------
-- Returns TRUE when a valid Remote Control message was received.
-- -----------------------------------------------------------------------------
function nec_rc_message_received() return bit is
  return _nec_rc_available
end function
```

If, for example, the library changes, it could be that the implementation of the above function changes as follows:

```
-- -----------------------------------------------------------------------------
-- Returns TRUE when a valid Remote Control message was received.
-- -----------------------------------------------------------------------------
function nec_rc_message_received() return bit is
   return _nec_decoder_state == _NEC_MESSAGE_AVAILABLE
end function
```

Although the implementation of the library function changes, the interface of library the remains the same and so there is no change for the user of the library.

Note: As mentioned in the style guide, it is recommended to start library specific constants, variables, functions and procedures with an underscore '_'. These are meant for internal use and should not be used by the user of the library. Instead, always use the public functions and procedures provided by the library.

Create a sample program

Each library must at least have one sample program. In this sample program try to use as much as possible the functionality of the library you created. When you are done you can add your sample files and library to Jallib.

Changing the contents of a Jallib release

Introduction

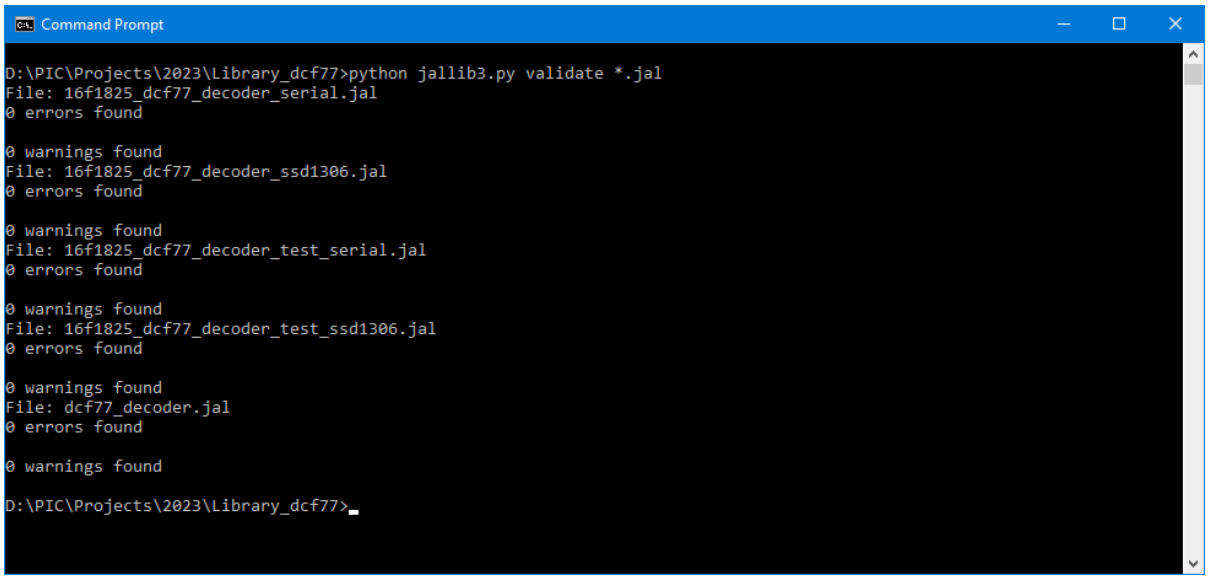
When changing the contents of a Jallib release, the following files must be updated in the root directory of Jallib on GitHub:

- The file `TORELEASE` describing which files are part of the Jallib release
- The file `CHANGELOG` describing the changes of the Jallib release

Prerequisites

When adding libraries and/or sample files to a Jallib release you must be sure that all files comply the Jallib style guide and that all files compile without any warnings and errors. Compliance with the style guide can be verified by running the `jallib3.py` Python script.

You can check one file using the command `python jallib3.py validate myfile.jal` or check all your files in the same directory using `*.jal` as parameter, as follows:



Before you can add any files to Jallib you need to install GitHub on your computer or you can use GitHub Desktop. All JAL sample files and libraries must be uploaded to Jallib on GitHub. There you find a directory structure. In most cases the following directories are used:

- `sample` for your sample files
- `include` for your library files. There are sub directories in this directory. See if your library fits in one of the existing sub directories. If not, create a new sub directory for your library files.

Important: Before making any changes to GitHub make sure that you have fetched the latest Jallib files from GitHub.

Updating the file TORELEASE

Not all files that are on GitHub are part of a Jallib release. In order to add your files to the upcoming Jallib release you have to add all uploaded files from the previous section to the `TORELEASE` file. This is a text file in which you have to add your new files using the correct file location used on GitHub. Make sure to add the files in alphabetic order.

You can, or course, also remove files from the Jallib release but that will not happen very often.

Updating the file CHANGELOG

This text file describes the changes of each Jallib release. Changes can, for example, be new libraries, new sample files, new device files or changes to existing files like bug fixes or additions.

Creating a new Jallib build

After having added all your sample files, library files and updated the files `TORELEASE` and `CHANGELOG` you can commit your changes to GitHub. After your commit, a build will automatically be started on the Jallib server where all sample files in the TORELEASE folder are validated and compiled. The result of the build is sent to the Jallib Google Group[5]. When an error occurs in making a new build you have to fix the error as soon as possible and upload your fixes.

If all goes well you did a great job and you made the next Jallib release richer!

Notice: Not every commit to GitHub results in a new release. Instead a bee-package of the latest build is created every week. This bee-package can be downloaded from the JAL website and can be used for testing and/or for using the latest JAL libraries. About once every year an official Jallib release is created

Miscellaneous

Changelog

Jallib Tutorial Book changes and updates.

Unreleased

Changes since 2024-08-11

Date	Comments	Change By
2026-01-30	Update tutorial_installation.xml	Rob Jansen
2026-01-30	Small cleanup (license, blink-a-led, getting started, installation)	Rob Jansen
2026-01-23	Jallib Tutorial Book page break support (pdf.xml, tutorial_style_guide.xml)	Matthew Schinkel
2026-01-23	Update of bmp180.jal (style guide)	Rob Jansen
2026-01-21	Update tutorial_style_guide.xml	Rob Jansen
2025-11-11	Updated ICSP tutorial to include a simplified schematic	Matthew Schinkel
2024-08-23	Change Jallib Tutorial Book to small caps (html.xml, pdf.xml)	Rob Jansen

Version 1.0

Release date: 2024-08-11

[5] Make sure you are a member of the Jallib Google Group and the Jallist Google Group.

Date	Comments	Change By
2023-12-10	Added making a library and added changing the contents of a Jallib release	Rob Jansen
2022-03-19	Added making a tutorial	Rob Jansen
2022-03-13	Added Visual Studio Code tutorial	Rob Jansen
2022-03-12	Added PICKit2 and PICKit3 software to the getting started tutorial	Rob Jansen

Version 0.5

Release date: 2022-03-06

Date	Comments	Change By
2022-03-05	Added MPLABX IPE to the getting started tutorial	Rob Jansen
2022-03-05	Added style guide tutorial	Rob Jansen
2022-02-26	Added USB tutorial and Large Array tutorial	Rob Jansen
2022-02-20	Updated tutorials of ADC, blink-a-led	Rob Jansen
2022-02-19	Restored/Updated broken links	Rob Jansen
2022-02-12	Added DFPlayer tutorial	Rob Jansen

Version 0.4

Release date: 2011-05-23

Date	Comments	Change By
2011-05-23	Added print & format library	Matthew Schinkel
2011-05-19	Added FAT32 tutorial	Matthew Schinkel
2010-04-22	Fixed English and integrate Youtube flash object in HTML output	Matthew Schinkel
2010-04-06	Alphabetical TOC	Matthew Schinkel
2010-04-05	Added RC servo & motor speed control tutorial	Matthew Schinkel
2010-04-01	Changed hard disk tutorial schematic	Matthew Schinkel
2010-03-25	Updated ICSP schematic and tutorial to reflect PIC Kit 2 pinouts. ICSP schematic matches Microchip Specification	Matthew Schinkel
2010-03-12	Updated SD Card schematic. Added pull-up resistor on chip-select line, changed resistor values for 5v-3v conversion	Matthew Schinkel

Version 0.3

Release date: 2010-03-12

Date	Comments	Change By
2010-01-27	Fixed I²C bus schematic and modified I²C titles	Sébastien Lelong
2010-01-21	Added ADC introduction, re-organized PWM tutorials and titles	Sébastien Lelong
2010-01-20	Better quality Images on Getting Started, serial board, blink a led tutorials.	Matthew Schinkel
2010-01-19	Added serial & rs-232 tutorial	Matthew Schinkel

Date	Comments	Change By
2010-01-15	New ICSP Schematic	Matthew Schinkel

Version 0.2

Release date: 2009-12-30

Date	Comments	Change By
2009-12-06	Added SD Card tutorial	Matthew Schinkel
2009-12-06	Added PATA Hard Disk tutorial	Matthew Schinkel
2009-12-06	Added ICSP tutorial	Matthew Schinkel

Version 0.1

Release date: 2009-11-22

Date	Comments	Change By
2009-11-22	Initial Release	Sébastien Lelong

License

We, the Jallib Group, want this book to be as *open* and *free* as possible, so we have decided to release it under the ZLIB License license.

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

Basically you can:

- **Use it commercially**
- **Modify**
- **Distribute**
- **Sublicense**

The Jallib group cannot be:

- **Hold Liable**

You must:

- **Include Copyright**
- **Rename**
- **Include License**