# Open-USB-IO Reference

The Open-USB-IO board allows you to control digital and analogue hardware from the USB port of a Windows or Linux PC. You can use a GUI, command line, script file, or your own C/C++ code (and most other languages).

You can also directly program the ATMEGA32 microprocessor, or add your code to the USB routines so your code can use the USB link to the PC. Your code can be downloaded using just the USB link (no STK-200 cable required).



V1.083

for downloads see
www.pjradcliffe.wordpress.com

for purchases see
www.interestingbytes.wordpress.com

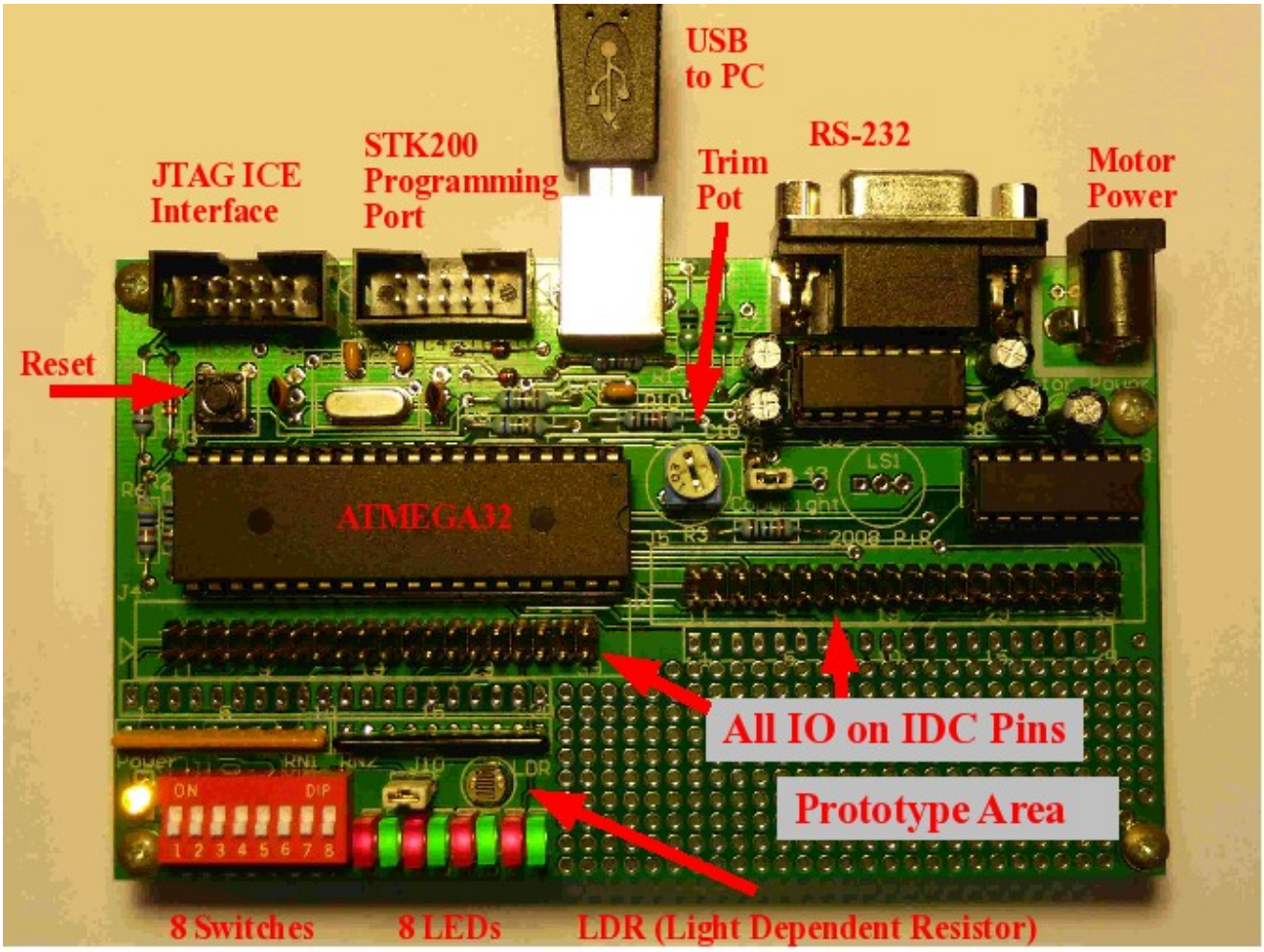


*Illustration 1: Open-USB-IO Printed Circuit Board*

# Table of Contents

# 1 Introduction

The Open-USB-IO board will plug into the USB port on a Linux or Windows (XP, Vista, or Windows 7) computer and provide a variety of digital and analogue inputs and outputs. It also has some remarkable software features.

## Software Features-

- Control the Open-USB-IO hardware from command line or GUI* from a Windows or Linux PC.
- Write you own code on the PC to control the board. You can use C/C++, bash or batch files, and most other languages. Speed is up to 250 commands per second using the new -multi or -file options.
- Download your own ATMEGA32 microprocessor code via the USB bootloader (no STK-200 cable required). This means you can develop ATMEGA32 code on a laptop without a parallel port. (An STK-200 programming interface is also provided but it requires the PC to have a parallel port.)
- Its easy to add your ATMEGA32 code to the USB interface code! In this way you can debug your own code using a powerful symbolic debugger, and use the USB link to control your ATMEGA32 code from the PC.
  This is a very powerful feature not offered by other products.
- There is a Linux live-DVD which has all the tool chains, all the code, this manual, and more examples. This can be run without altering your hard disk, or installed to make a dual boot Linux/Windows system. See www.interestingbytes.wordpress.com

## Hardware Features

- 8 input switches (these ports can alternatively be used as general purpose digital IO).
- 8 LEDs (can also be used as general purpose digital IO).
- 7 open collectors drivers rated at 500mA and 50v to drive coils, DC or stepper motors. There is a separate power input to supply these devices (small low power motors may use the USB +5v).
- RS-232 interface.
- Many uncommitted digital IO or analogue input lines. Some have special functions such as Pulse Width Modulators and SPI or I2C interface.
- One LDR for sensing light.
- One pot for generating a variable analogue input.
- JTAG ICE interface.
- Programming interface matching STK-200.

- ATMEGA32 processor with a 12 MHz crystal which can be programmed with the users own code, standard USB interface code, or a combination of the two.
- Power from USB (+5 Vdc, ~ 400mA on desktops and less on some laptops).
- Most IO pins are available on two 2x40 pin IDC plugs. The old flat IDC cables from computer IDE/ATA drives can be use to connect to other devices or circuits.
- Prototype area to place additional circuitry.

**Free!** The project is open source for both the hardware design and the software, see the web site www.pjradcliffe.wordpress.com for more details.
See www.interestingbytes.wordpress.com to purchase an Open-USB-IO board or the Linux live-DVD which has a huge number of development tools and enables you to write your own code for the ATMEGA32 microprocessor.

**Acknowledgements**: being a typical Open Source project contributions have come from many people. Most notable are my two project students Daniel Salby & Bowen Rees, and the open USB stack from Objective Development at-
http://www.obdev.at/products/vusb/index.html

**Errors or problems?** This document is a work in progress. If there are any errors or hard to understand bits, please contact the author.

Note that Mac versions will be following soon.*

* : feature currently under development.

# 1.1 Quick Start

This section will give you a quick start and enable you to see something working immediately on your Open-USB-IO board.

- Obtain a standard USB-AB cable.
- If you have the Linux live-DVD simply place it in your PC, restart the PC, and Linux will load up. Note this will not effect your hard disk. If the first boot option does not work then steadily try the other boot options down the boot list.
  If you do not have the live-DVD go to www.pjradcliffe.wordpress.com and download the appropriate ousb binary.
  - For Windows select the ousb.exe file and place it in a convenient directory, ideally in the path.
  - For Linux obtain the ousb binary and place it in /usr/local/bin ensuring the execute permissions are set ( you will need root access to achieve this). If you cannot be root, then use any convenient directory.
- Open a terminal and run the ousb program with no parameters to see help information.
  Under Windows select Start->Run, type `cmd` in the text box then hit enter. If the ousb.exe program is not in the current path then use the `cd` command to move to the directory that contains ousb.exe.
  Under Linux start a terminal ( under KDE open a file explorer window and hit F4).
  If the binary was placed in the path type `ousb`, if not placed in the path use `cd` to move to the right directory and type `./ousb` then enter.
  If ousb under Linux gives an error message about missing libraries then go back to the web site and obtain ousb_static (much larger but includes the libraries) and try again.
  If your Linux protects USB access try using a root terminal or logging on as root.
- Plug in the Open-USB-IO board and note the power LED should be lit.
  If your ATMEGA32 is not programmed then see the section on Compiling Code which gives details on how to program the ATMEGA32.

Assuming the USB firmware is loaded it's time to try a few commands from the command line-

- `ousb io PORTB 0xFF` # All LEDs should be lit. If not check the link directly # above the LEDs is connected.
- `ousb -b io PINC` # Switch state should be reported in binary (off = high).
- `ousb adc 5` # The trim-pot position should be reported, try changing it.
- `ousb -h adc 6` # The LDR reading should be reported in hex, # try changing the light level and reading again.
- `ousb_test` # If using the development Live-DVD, this test program # will report the state of several inputs. Ctrl-C to quit.

If these functions all work then your Open-USB-IO board is working as intended.

# 1.2 Changes and Issues

This section lists changes to the hardware and documentation, and lists known issues.

## Changes

- 17/6/2010: Windows 7 may not accept Open-USB-IO from 2/4/2010, whereas Win XP and Linux always will. Windows 7 appears to have USB timing issues which have been satisfied with the new firmware (V4). Co-USB on Win 7 is also affected, see the Co-USB_win7 folder for details.
- 2/4/2010: USB boot loader now a standard part of Open-USB-IO along with the USB interface to control the hardware. You can now develop your own ATMEGA32 code and download it into the Open-USB-IO board with just the USB cable, no STK-200 cable is required.
  "Cutting Free" section added. Other sections reorganized.
- 5/3/2010 : Co-USB shows how to integrate your own code with the USB code so you can run your code, read and write variables by name, and use breakpoints. See 7.3 Co-USB : Adding Your Code to the USB Code. The examples show this to be a very easy process.
- 20/1/2010 : command line tool updated to V1.2, PWM 2 and 3 now works reliably.
- 20/11/2009 : version 1.06 of documentation. More circuits added to section 9 and 10.

## Known Issues

- 7/3/2010 : Ubuntu and Kbuntu do not allow users access to the USB so ousb has to be run from root. There is a fix, see 8.2 Compiling ousb on Linux to see how to fix this.
- 7/3/2010 : on some machines the first USB command will cause a configuration error message but in fact everything works well.
- 7/3/2010 : some JTAG devices need one pin of the JTAG connector to be connected to +5v. See appendix on hardware schematics for details.
- 20/12/2009 : the stand-offs on some Open-USB-IO boards are thicker than intended. See the stand-off closest to the DC plug, it just overlaps a track. The PCB lacquer should stop any shorting but you may care to add an insulating washer to each leg.
- 18/11/2009 : plugging the ISP programming cable into the JTAG socket (or visa-versa) will short circuit +5v and 0v. This may cause the inductors to smoke and be damaged. Not much can be done about this as the pin-outs cannot be changed. Consider placing a dummy plug into an unused socket to stop accidental damage.

## Key Developments Planned

- GUI version of the command line tool.

# 2 Pin Function and Power Up State

On Power Up or reset the Open-USB-IO board has the following state. Most pins are multi-purpose (see the ATMEGA32 data sheet) and may be changed, but they are set up to a sensible initial state on power up/reset.

| Port | Purpose | Comments |
|---|---|---|
| PORTA | Analogue IO | PORTA with pins PA0 to PA7 can read an analogue value between 0 and +5v.<br>PA0-PA4 are available for any use.<br>PA5 is connected to the on board trim-pot.<br>PA6 is connect to the Light Dependent Resistor (LDR).<br>PA7 is either the Piezo buzzer or a user adjustable link. |
| PORTB | LED drive | PORTB with pins PB0 to PB7 is dedicated to driving the 8 LEDs.<br>PB3 is a basic Pulse Width Modulator and its state can be seen on the PB3 LED.<br>The LEDs can be detached from PORTB by removing the link just above the LED bar.<br>PB0-PB4 also drive Open Collector drivers which can be used to drive motors at up to 50v and 500mA. |
| PORTC | Switch input | PORTC with pins PC0 to PC7 is dedicated to sensing the 8 switches.<br>If a JTAG ICE interface is used then PC2-PC5 are used and the switches are ignored. |
| PORTD | Special Purpose | PORTD has several dedicated pins not available for normal use.<br>PD 0,1,4 and 6 are used for the RS-232 interface.<br>PD 2 and 7 are used for the USB interface.<br>PD4 and 5 are a PWM output which drive OC outputs.<br>PD3 and 6 are available for general use. |

**Protected inputs?** The ATMEGA32 IO pins are directly available on the board and IDC headers. Given that pins can be digital inputs or outputs, or analogue inputs, this gives you the user maximum flexibility. The microprocessor is surprisingly tough but severe abuse may damage pins. The microprocessor chip, in fact all chips on the board are mounted on sockets and a replacement will only cost a few dollars.

# 3 Command Line Interface

The Open-USB-IO board can be plugged into an XP, Vista, Windows 7, Linux (and other UNIX style operating systems) and does not require any special drivers. The command line program `ousb` can be used to control and read many of the ATMEGA32 microprocessor features including analogue IO, digital IO, and Pulse Width Modulators (PWM).

**Place binary**: On Linux place the binary executable in the path /usr/local/bin (do this as root). The alternative is to place it in the current directory and refer to the program as `./ousb` rather than just `ousb`. Click on a terminal icon to start a command line window.
On Windows copy the binary to any directory though it will be easier later on if that directory is in the path. To find the path directories use the command line, select Start, Run, type `cmd` then enter. In the command line window type `PATH`. If your selected directory is not in the path then to use ousb.exe you will have to open a command window, and then use the cd command to move to that directory before you can start the program by typing `ousb`.

| **Command** | **Comments** |
|---|---|
| ousb | Prints help information |
| ousb io reg | Prints the value of the named microprocessor register reg. If reg is omitted then ther values of PINA,PINB,PINC and PIND are printed out.<br>Example: read the switch values- `ousb io PINC` |
| ousb io reg value | Set the value of the microprocessor reg.<br>Any microprocessor hardware item such as timers and IO ports can be set up by one or more commands that write to registers.<br>Example: write to the LEDs- `ousb io PORTB 0x85` |
| ousb pwm-freq pin value | Set a Pulse Width Modular (PWM) to a particular frequency.<br>Example: set PWM 2 to 4 kHz- `ousb pwm-freq 2 4000` |
| ousb pwm pin value | Set the PWM to a percent on period.<br>Example: set pwm to 40%- `ousb pwm 1 40` |
| ousb adc pin | Read the analogue voltage on a pin.<br>Example: read trim pot- `ousb adc 5` |
| ousb io address (value) | Read (write) to the microprocessor RAM.<br>Example: write 85 to 0xFFF- `ousb 0xFFF 85` |
| ousb ee address | Read EEPROM address. `ousb ee 16` |
| ousb ee address data | Write byte to EEPROM address. `ousb ee 0x10 85` |
| ousb ver | Print version of firmware and ousb program. |

**Base options**: Any ousb command that results in an output can have a base option put immediately after the ousb command-

- No option: decimal
- -b : print output in binary.
- -h or -x : print output in hexadecimal
- -r : just print the decimal value without other text, useful for script files.

Example: `ousb -h io PINB` results in `PINB = 0x3a`

`ousb io PINB` results in `PINB = 58`

`ousb -b io PINB` results in `PINB = 0b00111010`

`ousb -r io PINB` results in `58`

Inputs on the command line may be anything the shell understands, notably decimal and hexadecimal (e.g. 0xFF).

**-multi option** : ousb can be started and allowed to stay in memory.

```
ousb  -multi          # start in multi-line mode.
io portb 0xFF         # turn all LEDs on.
io portb 0x55         # turn on alternate LEDs.
io portb 0            # turn LEDS off.
quit                  # quit multi-line mode.
```

Note that because ousb is running in memory it is not necessary to start each line with ousb. This option becomes much more useful when used in conjunction with script files as discussed in section 4 Script Control of Open-USB-IO or when controlling the board from your own code on the PC as shown in 5 C and C++ Control of Open-USB-IO.
Thank you to John F for contributing the base of this code.

**-file option**: ousb can take multiple commands from a file rather than the command line. It behaves much like the -multi option but the commands come from a file not the command line. The same output achieved in the -multi option above can be achieved as follows-

- File x.txt should contain the following text-

```
io portb 0xFF
io portb 0x55
io portb 0
quit
```

- The following command will cause the file to be executed-

```
ousb -file x.txt
```

# 3.1 Command Examples

This section will allow readers with no knowledge of the ATMEGA32 to drive the IO system, make LEDs flash and sense switches. To be more creative the reader will need to understand the register structure of the ATMEGA32. The best place to start is the data sheet for the chip which in on the live-DVD can be found in-

/home/user/projects/avr_info/data_sheets_and_compilers

Next look for examples on the web as to how each feature can be initialized and used.

**IO registers** : In the commands listed previously all registers are known by the identifiers used in the ATMEGA32 data sheets. To control the digital IO ports directly you will need the following identifiers.

- DDR*x* : data direction register for IO port *x* (A,B,C,D). Set 0 for all pins to be input, 255 for all pins to be output.
- PIN*x*: read the inputs from IO port *x* (A,B,C,D).
- PORT*x*: write for IO port *x* (A,B,C,D).

Examples:

- `ousb io PINC` # read the state of the switches.
- `ousb io PORTB 0xFF` # Write 255 to the LEDs, all LEDs on.
- `ousb io PORTB` # Read PORTB output latches (not the input pins).
- `ousb io DDRC 255` # Turn PORTC into an output ( on the J4 connector).
- `ousb io` # Print the inputs on PINA, PINb, PINC, PIND

**Analogue to Digital Converter (ADC)** : all 8 pins on PORTA can be analogue inputs. Conversion translates voltages from 0v to +5v to a integer between 0 and 1027.

- `ousb adc 5` # read the trim pot value.
- `ousb adc 6` # read the LDR value.

**EEPROM**: The ATMEGA32 has 1 kB of EEPROM which can keep values when power is removed ( the 2 kB of RAM loses its values when power is removed). This is most useful when user code is added to the USB code as shown in a later section.

- `ousb ee 100` # read the EEPROM values starting at address 100.
- `ousb ee 100 0x12` # write 18 to EEPROM address 100.

 

**PWM**: the ATMEGA32 has 3 Pulse Width Modulators though OC2 cannot be used if the USB link is to be active. The PWM command set timers and the PWM hardware to achieve the desired frequency and duty cycle.

| PWM Number | PWM, Port, Pin | Comments |
|---|---|---|
| 1 | OC0, PB3, pin 4 | Only 4 fixed frequencies, 8 bit duty cycle 0-255. |
| 2 | OC1A, PD5, pin 19 | Variable frequency, 16 bit duty cycle. |
| 3 | OC1B, PD4, pin 18 | Has the same frequency as PWM 2 but has its own 16 bit duty cycle. |
| 4 | OC2, PD7, pin 21 | Not used as interferes with the USB function. |

Examples-

- `ousb pwm-freq 1 2000` # Sets PWM 1 to closest available frequency, 732 Hz.
- `ousb pwm 1 30` # Sets PWM 1 to 30% duty cycle.
  # LED 3 glows at 30% brightness.
- `ousb pwm-freq 2 2000` # PWM 2 and 3 set to 2000 Hz.
- `ousb pwm 2 30` # PWM 2 has duty cycle 30% on then 70% off
  `ousb pwm 3 70` # PWM 3 has duty cycle 70% on then 30% off.

After the command is typed, and the enter key is pushed, then the value of the duty cycle or frequency actually used is printed out. This may vary from what was typed in and reflects the limits of the real hardware in the ATMEGA32.

**MORE COMMANDS**: see the section 7.3 Co-USB : Adding Your Code to the USB Code which shows how your code can be added to the existing USB code. This allows you to read and write RAM and EEPROM by name or address. There is also a breakpoint capability.

**The other ports?** This is where you need to start reading the Open-USB-IO circuit diagram! The ATMEGA32 ports are clearly labelled and you can see what pins they go to. Here are some things you should find-

- ADC: PA0-5 are general purpose pins that can be set as digital IO or ADC. See where they appear on J5 (directly above the prototype area).
- Open collector drives are PB0-PB4 (also driving LEDs) and PD4-5. These also appear on J5. Note that devices such as relays and motors can be driven from here (500mA maximum). An external DC plug pack is required and may be up to 50 volts peak value.

## 3.2 Special Commands

The live-DVD has several special commands that can be called from any terminal. These include-

- **ousb_test** : run a test program that exercises the ousb firmware and hardware. It toggles unused outputs on ports A,B and D. It also looks for links between TX and RX, and CTS and RTS on the RS-232 port. Try changing the switches and see the new values on screen. Try covering the LDR and changing the trim pot to see changes there. Use ctrl-C to quit. Hit reset to change ports back to their default settings.
- **ousb_all** : use the STK-200 cable to program in the bootloader and ousb firmware. The link can stay on J9.
- **ousb_reset** : if the STK-200 cable is present reset the Open-USB-IO board and then bring out of reset. Useful when first plugging in the STK-200 cable and the default parallel port setting is to hold the ATMEGA32 in reset.
- **ousb_prog** : program just the ousb firmware into the Open-USB-IO board using the STK-200 cable.
- **ousb_usbprog** : program just the ousb firmware into the Open-USB-IO board using the USB bootloader.
- **stepper_motor_demo** : with no parameters it bgives help on connecting a stepper motor to the Open-USB-IO board.
- **win_start windows_program_name** : start a Microsoft Windows program under wine.
- **lp_tty_start** : allow a Linux program to directly access the old legacy parallel and serial ports. See the readme file on the live-DVD desktop for more details.

# 4 Script Control of Open-USB-IO

The previous section dealt with command line control of Open-USB-IO. Command lines can be placed in a file to automate operation, and eliminate the need to remember and enter long sequences of commands. Typically under Linux the BASH shell would be used, and a BAT file under Windows. The Linux BASH shell is much more powerful than BAT files and is a programming language in its own right.

Under Windows the cygwin package can be added which provides a BASH shell interface. Note if you install the WIN-AVR development package which compiles Atmel AVR microprocessor code, the MinGW compiler, or Dev-C++ then this has a BASH shell included. Do not install cygwin and WIN-AVR on the same host as WIN-AVR will stop working. To stop the problems rename the cygwin directory to say xcygwin.

**cygwin** : if you are a Windows user and do not intend to compile the Open-USB-IO firmware or other ATMEGA32 programs then go to www.cygwin.com, download setup.exe and run it. Start a Windows terminal and type "sh" to start a BASH shell.

**BASH**: is not the easiest script language to learn but very powerful.

For a beginners tutorial go to http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html

For a complete guide go to http://tldp.org/LDP/abs/html/

Both documents are available as pdf or HTML downloads, as well as being web pages.

**Calling issues**: in the examples below the program ousb is assumed to run from the Linux path. For Linux only, if the version of ousb to use must come from the current working directory then use `./ousb` rather than `ousb` which is used in the examples below.

For Windows use `ousb` which can be either in the current path or the active directory.

Under Linux the `ousb` program must have its permissions set to executable, there is no such requirement under Windows.

**Examples** in this section include-

- LED flash which shows basic output and a loop.
- Reading the Light Dependent Resistor (LDR) and showing its value on the LEDs.
- PWM (Pulse Width Modulator) control.
- RS232 serial communications control.
- High speed operation for Linux using the -multi option.
- High speed operation for Windows and Linux using the -file opion.

**Example 1: LED Chaser.** Implement an LED chaser and read the input switches.

```
#!/bin/bash

#----- stop auto declaration of variables.
set -u

#----- forever led chaser loop and input, ctrl-C to stop.
PATTERN=1
READ=0
until  [ 0 != 0 ]
do
 ousb io PORTB $PATTERN
 READ=$( ousb io PORTC)
 echo "   LED Output = $PATTERN,   Input on PORTC = $READ."
 sleep 0.3
 let "PATTERN = PATTERN + PATTERN"
 if [ $PATTERN == 256 ]
  then PATTERN=1
 fi
done
```

Things to try and know:

- The writing to PORTB creates an unwanted print output, how can that be eliminated?
- Why is "set -u" used?
- The pattern is a moving on LED, try making the pattern all LEDs on and a moving off LED. Try making alternate patterns of a moving on, then a moving off.
- Try changing the switch settings and note the response.
- Try changing the cycle period.

**Example 2: LDR.** Read the LDR and write to the LEDs.

```
#!/bin/bash
#
#-----  Read the LDR light sensor and write the value to the LEDs.
set –u    # stop auto-declaration of variables.
LDR=
until  [ 0 != 0 ]     #  A forever loop,  control-C from the keyboard to stop.
do
 sleep 0.3            # pause for 300 ms.
 LDR=$(ousb adc 6)  # get the LDR reading from Open-USB-IO
 let "LDR = LDR/4"    # scale the 10 bit ADC back to 8 bits.
 ousb io PORTB $LDR #  write the value to the LEDs
done
```

Things to try and know:

- Initially the LED's show a wildly varying value. What could cause this? How can it be solved? (Hint look at the physical arrangement).
- Why does high light levels give a low reading?
- Change the code to make the trim-pot drive the LEDs.
- If the trim-pot conversion is done repeatedly the values differ by a few counts. Why is this so? What methods can be used to eliminate this problem?
- Speed here is limited by the sleep command. If this is removed the loop can work at about 25 ousb commands per second.

**Example 3: PWM.** Drive the PWM on PORTB3 with a command line parameter.

```
#!/bin/bash
set -u

#------ Take command line parameter and set PWM.
ousb io PORTB 0
ousb pwm-freq 1 4000
ousb pwm 1 $1
```

Things to try and know:

- Change the code to input both the frequency and duty cycle from the command line.
- Try different values and watch the #3 LED.

- Try hooking up a small DC motor to see the speed variation as the duty cycle changes. With reference to the circuit, and using links or clips, connect J5 pin 39 to pin 37 (motor to use +5v), the motor should be connect to J5 pin 37 and pin 27.
  For larger motors omit the 37-39 link and plug an external supply into the DC power socket with the centre pin as positive. See the section Open-USB-IO Hardware for details.

**Example 4: RS-232.** This script sets up the RS232 port, receive a character, increment it by one, and transmit it back. To test this setup Windows Hyper Terminal or Linux gTerm to be 9600 baud terminal with no flow control. Connect a serial cable from the PC to the Open-USB-IO board. When the script is run any chracter typed on the terminal should be echoed as the next character, for example pressing A would result in displaying B.

```
#!/bin/bash
set -u

#------ Initialize ports and the uart.
DDRD=$(ousb -r io DDRD)
let "DDRD = (DDRD & 0xFE) | 2"   # TX to output, RX to input.
ousb io DDRD $DDRD

ousb io UBRRL 75                # baud rate to 9600 baud.

UCSRB=$(ousb -r io UCSRB)
let "UCSRB = UCSRB | 0x18"       # Enable TX and RX.
ousb io UCSRB $UCSRB

#------ Loop to receive, add one, and transmit back.
while  [ 0 == 0 ]
do
  #--- wait for RX of byte.
  UCSRA=0
  while [ $UCSRA == 0 ]
   do
     UCSRA=$(ousb -r io UCSRA)
     let "UCSRA = UCSRA & 0x80"
   done
  #--- get received byte, incr, and send.
   UDR=$(ousb -r io UDR)
   let "UDR = UDR + 1"
   ousb io UDR $UDR
done
```

Things to try and know:

- Try sending a string to the terminal.
- Try receiving a string from the terminal, use Enter as the terminator. When the whole string is received send it back.

**Example 5: fast ousb commands with -multi for Linux.** Previous scripts can only execute about 25 ousb commands a second due to the delays involved in loading the ousb program, starting it, and stopping it. This can be dramatically sped up to about 200-250 commands per second by allowing the ousb program to remain in memory.

The -multi option tells ousb to stay in memory until the command "quit" is given.

Consider the text file ousb_commands that contains-

```
-r io portb 0
-r io portb 1
-r io portb 2
…
-r io portb 255
quit
```

This can be sent to the Open-USB-IO board as follows-

```
ousb -multi < ousb_commands
```

The results will be printed to the screen.

Additionally the output can be sent to a file call ousb_result-

```
ousb -multi < ousb_commands > ousb_result
```

This could be used to make a low speed logic analyser. Consider a text file sample with 1000 lines of-

```
-r io portc
```

With a `quit` at the end. Now start the program as follows-

```
ousb -multi < sample > result
```

The file result will have a 1000 samples from port C sampled at about 4 milliseconds each.

**Example 6: fast ousb commands with -file for Linux & Windows.** The -file option works much like the -multi option except that the ousb program obtains its commands from a file rather than stdin. Such an approach works for both Linux and Windows.

Consider the text file ousb_commands that contains-

```
-r io portb 0
-r io portb 1
-r io portb 2
…
-r io portb 255
quit
```

This can be sent to the Open-USB-IO board as follows-

```
ousb -file ousb_commands
```

The results will be printed to the screen.

Additionally the output can be sent to a file call ousb_result-

```
ousb -file ousb_commands > ousb_result
```

# 5 C and C++ Control of Open-USB-IO

The Open-USB-IO board can be directly controlled by a program running on the PC. This section will show how to use C/C++ but the same principles can be applied to almost any programming language.

Using C/C++ the system and popen library calls allow the execution of a command line as if it came from a terminal command. For the GCC compilers on Windows and Linux-

**System()** : this allows execution of a shell from C or C++ but not the interrogation of the returned result. See the manual pages for details.

```
#include <stdio.h>
#include <stdlib.h>
main()
 {  system("ousb io PORTB 0x55");  // Light alternate LEDs
    printf("Command done!");
 }
```

Note that in Linux if the ousb executable is in the same directory as this program then "./ousb" should be used not "ousb".

**popen()** : a little more complex but it allows reading of any returned results. See the manual pages for details ( from a command line: `man 2 popen` ).

```
#include <stdio.h>
#include <stdlib.h>
main()
 { FILE *fpipe;
   char *command="ousb io PINC";  // Read the switch state.
   char line[256];

   if ( !(fpipe = (FILE*)popen(command,"r")) )
    {  // error if fpipe returns NULL
       perror("Problems with pipe");
       exit(1);
    }

   while ( fgets( line, sizeof line, fpipe))
    { printf("%s", line);
    }
   pclose(fpipe);
 }
```

**-r option** : ousb has an option -r which results in a single decimal number being printed not a whole message. For example here are two reads and a write-

| | | |
|---|---|---|
| `ousb io pinc` | results in | `PINC = 28` |
| `ousb -r io pinc` | results in | `28` |
| `ousb -r io portb 22` | results in | `22` |

This can be used to simplify reading something from ousb-

```
#include <stdio.h>
#include <stdlib.h>

int port_read ;

main()
 { FILE *fpipe;                       // Pipe to command line.
   char *command="ousb -r io PINC";  // Read the switch state.
   char line[100];                    // string to p;lace return text.

   if ( !(fpipe = (FILE*)popen(command,"r")) )
    {//--- error if fpipe returns NULL
       perror("Problems with pipe");
       exit(1);
    }

   port_read = atoi(fgets( line, sizeof line, fpipe)) ;
   printf("\n   Port C read returned %i\n\n", port_read) ;

   pclose(fpipe);
 }
```

Providing the "-r" option is always used then a single function can provide both read and write operations-

```
int do_ousb_command(char* command) //-----------------------------------
{ char line[100] ;
  FILE *fpipe ;
  if ( !(fpipe = (FILE*)popen(command,"r")))
   { cout << "pipe error" << endl ;
     exit(0) ;
   }
  fgets( line, sizeof line, fpipe) ;
  pclose(fpipe) ;
  return( atoi(line)) ;
}

//--- how to call, a write then a read-
do_ousb_command("ousb -r io portb 0x55") ;
printf("  Value of PORTB is now %i\n", do_ousb_command("ousb -r io portb") ;
```

The code above gives about 25 ousb commands per second. The USB access time is the limiting factor not the C code.

 

**Faster and faster** : as discussed previously the repeated starting and stopping of the ousb program is slow and limits the speed of this system to about 25 commands per second. Using the -multi or -file option can speed things up dramatically, to about 200-250 commands per second.

- For Linux using named pipes: the basic operation is to start ousb such that it takes its input from one named pipe and sends its output to another named pipe. Named pipes are essentially FIFOs that look like a file. The user C/C++ code can then talk to the running ousb program using these pipes.
  In the directory that holds the source code for the command line program ( on the live-DVD this is /home/user/projects/open-usb-io_command_line/ ) there is a subdirectory called ousb_multi. Read the read_me file there and try executing the script file run_fast (`./run_fast`) to see a demonstration of writing to the ATMEGA32 PORTB at about 200 commands per second.
  To create your own program copy run_fast and fast_c_ousb.c to your own folder. Find the for loop in the main() function, and replace it with your code.
- For Linux and Windows using files: the following code shows how the ousb program can use a file of commands, and send any output to another file. This works on both Windows and Linux with the same speed improvement noted above. See the directory mentioned in the dot point above for full details and code examples.

```
#include <stdlib.h>
#include <stdio.h>

#ifdef _WIN32
 #define CMD "..\\ousb -file ousb_commands.txt > ousb_result.txt"
#else
 #define CMD "../ousb -file ousb_commands.txt > ousb_result.txt"
#endif

//============================ main ================================

int main(int argc, char **argv)
{  system( CMD) ;
}
```

# 6 Open-USB-IO Hardware

The Open-USB-IO hardware schematics are copyright Dr. Pj Radcliffe but are available for general use under the terms of the GPL 2 licence. These may be found at www.pjradcliffe.wordpress.com or on the live-DVD under /home/user/projects/avr_info/ousb_related. Please note the following files-

- open-USB-IO_schematic.pdf : the full circuit schematic. To do advanced programming you will need to read this circuit to understand what the various ATMEGA32 port pins are connected to.
- stk-200_programming_cable.pdf : schematic of a simple cable that allows for the programming of many Atmel microprocessors from a PC parallel port.
  Note this is not required if you use the USB bootloader that is now standard with the Open-USB-IO board.
  A simpler and clearer wiring diagram can be found in this document under 7.5 The STK-200 Programming Cable.

The hardware may be purchased from www.interestingbytes.wordpress.com at a very reasonable price.

To understand the following discussion you will need to refer to the circuit schematic diagram.

**Interfacing to Hardware**: the Open-USB-IO board has several connectors that make it easy to interface to external circuitry. With reference to the board and the circuit diagram look for-

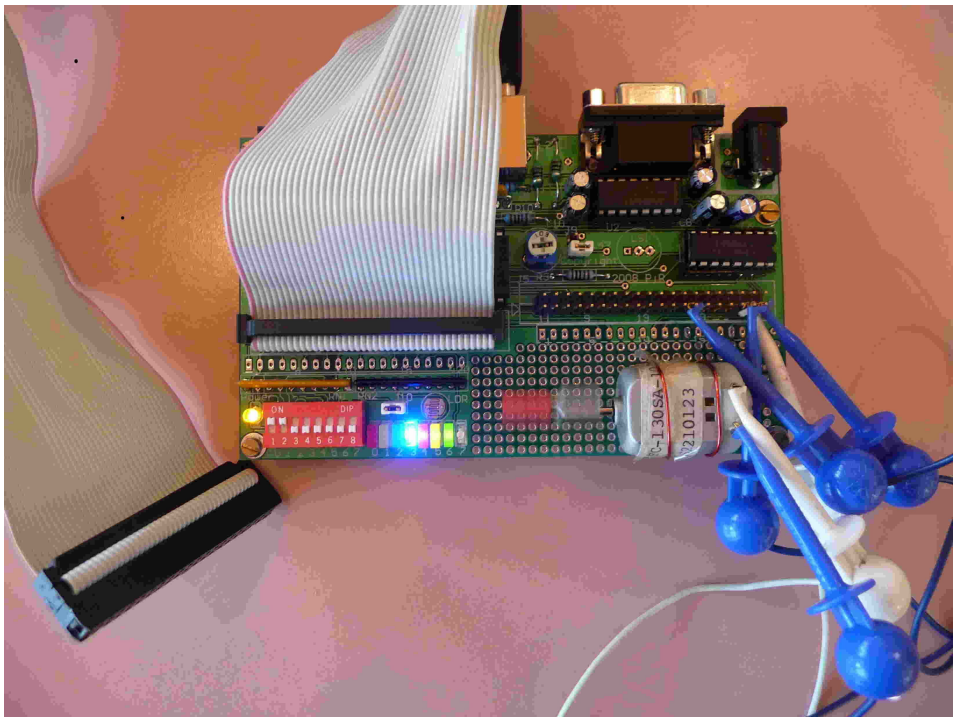

*Illustration 2: Ezy Hooks (on J5) and IDC Cable (on J4) Connections*

- J5 is directly above the prototype area that connects to the PORTA digital or analogue IO pins and the Open Collector drive pins. This is a 2x20 IDC connector which can be accessed using easy hooks or an old flat IDE cable available from most old PCs.
  Note the row of pins furthest away from the prototype area are all connected to zero volts. All signals are on the row of pins closest to the prototype area.

- J3 is a row of holes directly below J5 and has all the signals that appear on J5. Each J3 hole is connected to the J5 pin directly above it. These holes are intended to be useful when building circuits in the prototype area.
- J4 is another 2x20 pin IDC connector directly next to the 40 pin microprocessor chip. It has all the digital input (switches) and digital output (LED) signals.
  The LEDs can all be disconnected by removing the link above the LEDs thus making the output drives all available for external circuits. The switches have either a 100kohm pull-up or a 4.7kohm pull-down. Most external circuits can easily overdrive these inputs, or the entire port can be reconfigured as an output.
- J2 is like J3, a string of holes below J4 which is connected to the J4 signal lines.

**Detail on the circuit diagram**: In order to follow this section you will need to have a copy of the circuit diagram for Open-USB-IO. The top left of the circuit shows the USB interface where the zener diodes D1 and D2 and associated resistors R1,R2 and R4 act as voltage limiters and present the correct load to the PC USB port. The USB lines carry both DC power and high frequency data signals. The inductor L1 and capacitors C1 and C6 filter out the high frequency data to provide DC power, Vcc, for the board to use. On a desktop computer the USB port can supply up to 500 mA but laptops can provide rather less. The DC power is clean enough for digital circuits but still has too much noise for analogue circuitry so the combination of L2 and C2 gives an extra level of filtering to provide the clean power source AVcc which is used for all analogue circuits. The USB data interface is handled by firmware on the ATMEGA32 (12MHz xtal) which uses interrupt PD2 and pin PD7 to receive or drive signals to the USB line.

The top right of the circuit has S1, a bank of 8 switches which can be read by the microprocessor. The microprocessor provides a 100 kohm pull-up on each port C pin which sets the pin to logic high, the switch can add a 4.7k pull-down resistor to force the input to logic low. These inputs are available on the J4 connector ( and the J2 holes below the connector). Any external output capable of driving the 4.7 kohm resistor could be connected here and be read by the microprocessor. If all the switches were set to off the external input would only have to drive the 100 kohm pull-up. Port B of the microprocessor drives 8 LEDs (DS2-DS9) through resistors, then a link to zero volts. If the link is removed then the LEDs will not light. This can be useful if port B pins on connector J5 are intended to drive external devices. Alternatively the LEDs may be left connected when driving external circuitry as the ATMEGA32 outputs are capable of driving 20mA and the LEDs only take around 12mA thus leaving spare drive for external devices. The ATMEGA32 should not drive more than 200mA for the entire chip as an absolute maximum but given the chip only requires some 12 mA for its internal uses this leaves a lot of drive for external devices.

The RS232 interface at the bottom left of the circuit uses a standard MAX232 chip to interface to the RS232 lines and to provide the plus and minus 3 volt power supplies needed to drive the RS232 outputs. The device not only handles transmit and receive but also one status line in and one status line out. If the RS232 port is not needed for serial data then the two output lines can be used as general purpose outputs that drive around +3v and -3 volts.

The bottom right of the circuit shows the open collector drive chip ULN-2003A which has seven open collector drivers, the circuit of one is shown opposite in illustration 6. An input of

 

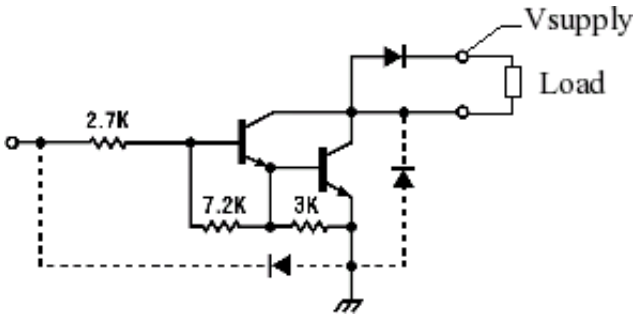


*Illustration 3: ULN-2003 Driver*

3 volts or more applied to the 2.7k resistor will turn on the Darlington transistor and current can flow from Vsupply through the load to ground. If the input goes to zero volts the Darlington turns off and the load current drops to zero. If the load is inductive the built in diode connected to Vsupply will short circuit the inductive current and ensure there are no large voltage spikes that could destroy the chip. Vsupply is not tied in any way to the board +5v and can range from zero volts to 50 volts. The Darlington can handle 500mA and so each of the seven drivers can control a small DC motor or a coil in a stepper motor. Our students at RMIT have used such a configuration to drive one 6 wire stepper motor (using 4 outputs) and 3 DC motors or servo units. The power for these motors is usually connected to the 2.5 mm DC socket (centre pin positive) which corresponds to Vsupply above. For small 5 volt motors it is possible to link pins 37 and 39 on J5 and so use the 5 volts provided by the USB link to drive a motor. Desktop computers can usually supply 400-500mA on the USB power line but laptops provide less and may be incapable of driving a motor. If your commands to Open-USB-IO start to generate errors then probably the motor is drawing too much current from the USB port.

The two IDC connectors of 2x20 pins, J4 and J5, provide access to most of the microprocessor pins and all the open collector drives. The back row of these pins is all connected to zero volts. When a cable is connected this means each signal wire has a zero volt wire on each side. This helps to stop interference both to and from the signal wire. Without such an arrangement a signal on one wire will usually create glitches on the wire next to it in the cable.

**Motor Warning**: only small motors should be powered by the USB +5v for several reasons.

- If the motor draws too much current the voltage to the microprocessor may dip too low and cause it to crash. This can make the board misbehave and `ousb` commands may fail.
- DC motors put lots of electrical noise and voltage spikes on the power supply. Again this can cause the microprocessor to fail. In theory it is possible for a large DC motor to generate large enough spikes to destroy the ATMEGA32.

The moral is to only use a small DC motor with USB +5v, and preferably use an external power supply. Motors can be made to generate less noise by soldering a 0.1 uF capacitor across the terminals. This will help to soak up spikes and noise but the dangers listed above are still an issue.

**Motor Connections**: the image opposite shows how a small motor is connected to use the USB +5v.

- If an external power source is being used then the small red link between pin 37 and 39 of J5 is not required.
- One side of the motor goes to J5 pin 37.
- The other side of the motor goes to J5 pin 27 which is connected to an open collector driver, driven by PB3 which is a PWM (Pulse Width Modulator).

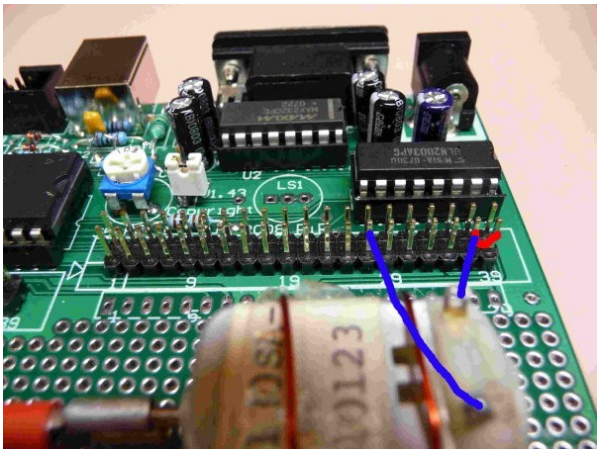

*Illustration 4: Small Motor Connection*

To control the motor speed try the following commands-

- `ousb pwm-freq 1 2000` # Sets PWM 1 to closest available frequency, 732 Hz.
- `ousb pwm 1 30` # Sets PWM 1 to 30% duty cycle, motor may just go.
  # LED 3 glows at 30% brightness.
- `ousb pwm 1 70` # Sets PWM 1 to 70% duty cycle, motor may just go.
  # LED 3 glows at 70% brightness.
- `ousb pwm 1 100` # Sets PWM 1 to 100% duty cycle, full speed.
  # LED 3 glows at full brightness.

# 6.1 Interesting Projects

The Open-USB-IO board can be used for a vast variety of hardware interface and control applications, either controlled from a PC via the USB or from on-board ATMEGA32 code. This section lists a few ideas for useful projects. Many of these projects are on the web.

- Stepper motors : use 4 of the seven Open Collector drive pins on J5 to control a stepper motor. Try control from the PC and ATMEGA32 code.
- DC motors: Use the PWM on PB3 or PD4 and PD5 to drive DC motors.
- Speed control: try sensing the speed of the DC motor using an LED/Sensor device and implement speed control for the DC motor.
- PID controller: extend the motor control to a full PID motor control.
- Temperature sensing: try using a thermistor to sense temperature. Use the proper compensation equations to get a linear reading.
- Burglar alarm, temperature control: program the board to be a complete burglar alarm and temperature control system for a house.
- Hardened interface: design interface circuitry to harden the inputs and outputs against voltage spikes.
- Gray water pump: check the local regulations for grey water storage and pumping. Create a grey water pumping system.
- Solar controller: control the charging of batteries from solar cells.
- Keyboard interface: read the output from a PS2 keyboard. Try also a 4x4 matrix style keyboard.
- Play voice : encode voice as PCM or delta-modulation form and replay via a PWM to a piezo transducer. Make your AVR talk!
- Record voice: interface a microphone to an analogue input and record voice. Consider some compression formats.
- CRO, logic analyser: turn your AVR into a chart recorder, or low speed CRO or logic analyser.
- Data logger: similar to above turn your AVR into a data logger.
- SD card: interface the AVR to an SD memory card.
- Dual slope DVM: add circuitry to make a dual slope ADC with high resolution.
- Robots: control model cars and other robots.
- Guitar and piano tuner with a good user interface.
- Ultra-sonic distance ranging.
- Resonant destructor to shake mechanical assemblies to bits.

# 7 Compile Your Own ATMEGA32 Code

The simplest way to develop ATMEGA32 code is to use a Live-DVD based on PCLinuxOS 2009 (and soon Kbuntu) which has been created especially for code development. See www.interestingbytes.wordpress.com for details. It contains many tools but notably the entire avr-gcc compiler tool chain already installed, along with the entire ousb project and sample projects.

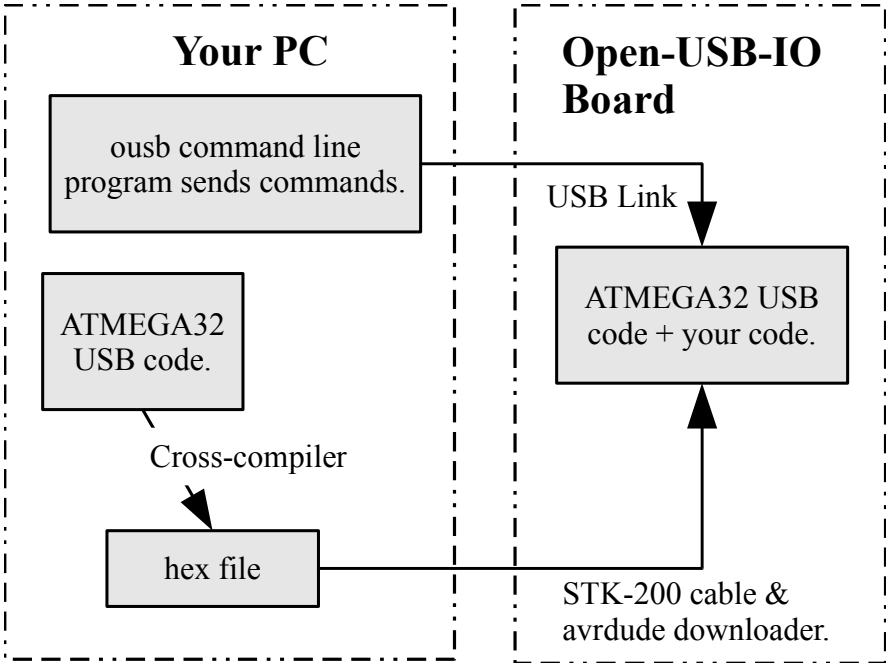


*Illustration 5: Firmware Downloading*

The DVD also has a huge range of other development tools and other free software including the Eclipse IDE for C, C++, Java, python, Perl, and C for the ATMEGA32. Other tools include the Apache web server, the MySQL data base, PHP, web editors such as Kompozer, Qt Designer for GUI development, and much more. There is also a large range of network tools, drawing tools, Open Office, audio-visual programs, and a few games.

This section describes how to write your own programs for the ATMEGA32 using the live-DVD.

**Programming the ATMEGA32**: prior to April 2010 it was necessary to use an STK-200 cable from a parallel port to program the ATMEGA32. Now all Open-USB-IO boards come with a built in USB bootloader that can load your code via the USB cable. Boards previous to this date can have the USB bootloader added, alas by using an STK-200 cable.

**Live-DVD development issues**: the live-DVD is an excellent way to run Linux as it does not change your hard drive, though\ it is slow to start up. A major issue is that when you close the live-DVD all changes are lost! The best solution is to copy the folder you wish to change to a USB memory stick and do all your development from the memory stick.

Live-DVD development should use a USB memory stick as persistent storage.

The alternative is to install the live-DVD to your hard drive. See the read me file on the live-DVD desktop for more details.

# 7.1 Getting Started: Flash the LED and Beyond

To get started let's use the live-DVD to download a simple program to the Open-USB-IO board that will flash an LED.

- Put the Live-DVD into any PC and boot to the DVD.
  Double click on the read_me.html file and read through it.
- Double click on the home icon, then projects. The folder avr_info is well worth reading but for now go into AAA_simple_ousb_projects then the folder led_flash.
- Hit F4 to raise a terminal window, type "make help" to see what the make file can do.
- Type "make clean" then “make” to recompile the code.
- Connect a USB cable from the PC to the Open-USB-IO board.
- Find the link on J10 directly above the LEDs and move it to J9 in the middle of the Open-USB-IO board next to the trim pot.
- Press the reset button on the Open-USB-IO board ( near the microprocessor).
- Type "make usbprog" to download the program, wait for the download to finish.
  Remove the link on J9 and move it back to J10. (If you don't complete this move the your application wont run or the LEDs wont light.)
  The user code should start executing and LED 0 should now be flashing.

Note that this code has overwritten the Open-USB-IO USB interface code which links to the PC command line tool ousb. The USB bootloader will not have been effected. The ousb interface code can easily be restored-

- Move the link on J10 to J9.
- Hit the reset button on the Open-USB-IO board.
- Use any terminal window and type "ousb_usbprog".
- Move the link on J9 back to J10.
- Try a test such as "ousb io portb 255" which turns on all LEDs.

**Your code plus USB interface?** You add your own ATMEGA32 code to the USB interface code and so have available a powerful debugging environment, and have your ATMEGA32 code communicating with code on the PC. See 7.3 Co-USB : Adding Your Code to the USB Code for more details.

**Your development on a memory stick** : the live-DVD does not write anything to the hard disk and so cannot save files permanently, when the PC powers down the changed files are lost. To avoid this problem do your development on a memory stick (or install PCLinuxOS to your hard drive) or at least copy your files to memory stick before powering down.

- Insert a memory stick into the computer and wait until a window to it appears.
- Drag the led_flash folder onto the memory stick and rename as appropriate (right click ...).
- Open the new folder and hit F4 to get a terminal window.
- If you intend to change any file names or add files the edit the Makefile (right click, open with, Kwrite), rename the variables TARGET (the name of the final hex file) and the source files OBJECTS.
  Save and quit.
  Rename the source file from led_flash.c to the name you you set in the OBJECTS variable. For example if the OBJECT variable is my_code.o then the source file must be my_code.c.
- Now recompile and program, first "make clean" then "make", then "make usbprog". Check the LED still flashes.
- Now start developing your own code.

Note that most Linux systems block script files from executing on external drives but this should not effect your ATMEGA32 development.

**Live-DVD RAM problems.** The host machine should have at least 512 MB of RAM, and ideally much more.
A live-DVD does not use a hard drive in any way. Every time a program requires to read/write a file it is placed in RAM, and so RAM steadily gets used up. To see this open a terminal (double click on the terminal icon), and type the command "df". Run any program such as OpenOffice, and try "df"' again to see how available RAM is used up.

When using any live-DVD do a regular check with the "df" command to ensure that RAM is not about to be exhausted. Save to memory stick on a regularly basis, and always if RAM is getting low.

**Useful folders** on the live-DVD are listed below. Start by double clicking on the read_me.html file on the desktop, then start browsing these folders.

- /home/user/projects has a huge range of information and example projects. Its worth spending time to find what is here.
- /home/extras has a large amount of documentation about Linux, useful lecture notes, and example code for Linux.

## 7.2 Development Hints

This section offers some brief hints that will help you when developing your ATMEGA32 code.

**Extreme Programming approach.** This lifecycle dictates a sensible development practice that decreases the number of faults, and makes their detection easier.

1. Plan a small change to code, around 20 lines or less.
2. Devise tests to check the code works.
3. Write the code.
4. Run the tests to check it works.
5. Go back to step 1.

**What debugger to use?** Debuggers provide a way for you to see the value of variables, change variables, set breakpoints, and perform other useful functions. Without a debugger its very hard to tell why your code is misbehaving. There are a variety of options-

- Don't use debugger. Rely on LED's, LCD displays, or a serial communications port to output basic information. This may be useful on very very little projects but soon becomes frustrating and unhelpful.
- Use a built-in debugger. The Open-USB-IO board can use the Co-USB option which enables you to add your code to the USB interface code. The ousb program on the PC can then provide powerful debugging functions, and allow your code to work with the USB link.
- Use a hardware debugger, an In Circuit Emulators ( ICE), to provide full debugging similar to the debuggers that run under Windows or Linux. Full speed debuggers for microprocessors are very expensive and the cheaper JTAG debuggers, while very useful, cannot work at full speed.

**When to get a hardware debugger?** Debuggers cost money, between US$50 and US$500, and they take some time to master. Debuggers make sense in a commercial environment where time is money, and larger applications mean debugging will take some time.
In most cases the Co-USB debugging will be adequate and a hardware debugger is not needed.

**PC then Board**: as described in an earlier section it is possible to write a C or C++ program on the PC that directly controls the Open-USB-IO ports. This can be debugged with all the usual PC debugger tools. When the PC program is working then compile the code for the ATMEGA32 but change all "system" or "popen" commands to real IO port accesses. This might best be done with C/C++ Macros (#ifdef and #define) to make it painless to move between PC and ATEMGA32 versions. This approach will flush out nearly all the problems on the PC where debugging is much easier.

 

# 7.3 Co-USB : Adding Your Code to the USB Code

The ousb code on the PC has been re-arranged so that you can easily add you own code to run your own application. Co-USB, uniting your code and USB code, has some major advantages-

- Your application can be running at the same time as the USB link to a PC.
- You can use the ousb commands to read and write RAM, registers, and EEPROM, while your code is running.
- Your code can have breakpoints which can be enabled and disabled.
- You can refer to RAM and EEPROM variables by name and type, for example you can read an array of real numbers, and write an array of real numbers. Other data types include 8, 16 and 32 bit data both signed and unsigned, characters and strings.
- By adding only a little code you can enable the "ousb user" command to send data to your own functions. This can be very useful for controlling your own code from the PC.
- The USB code provides a 500 microsecond periodic interrupt call that can be used to perform user timing activities.

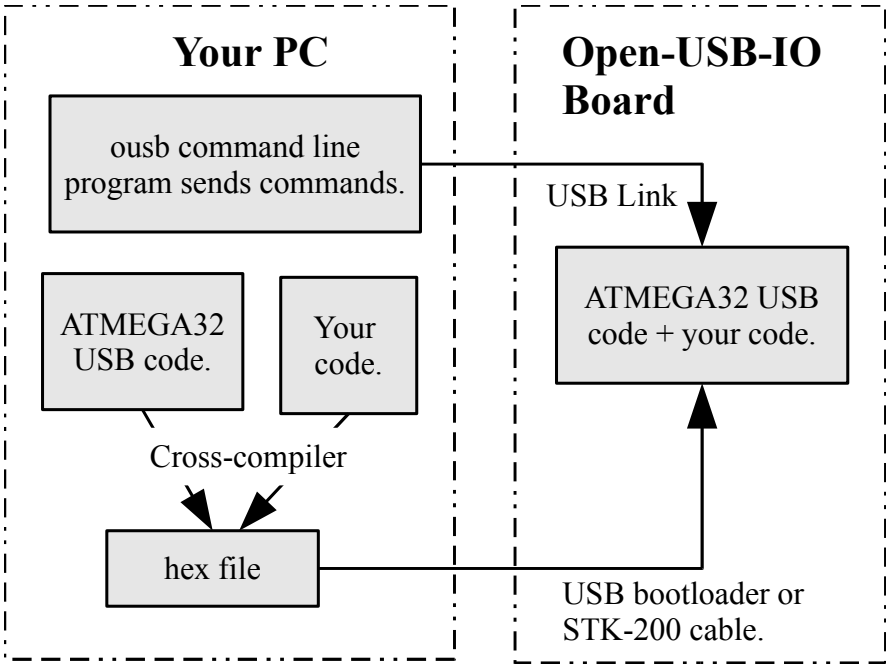


*Illustration 6: Mixing Your Code and USB Code*

With all of this do you need a hardware emulator any more? Co-USB can provide all the data read/write features and breakpoints you need, but emulators have a few advantages-

- An emulator does not need to have breakpoints written into code. An emulator allows breakpoints to be added without the code being altered and re-downloaded.
- An emulator can single step through code.
- Co-USB interrupts can stop your code for 100 microseconds at a time in a random manner. In many cases this does not matter.

Co-USB can replace an emulator for most purposes and is usually entirely adequate for developing code. Once the code is debugged then it can easily be turned into user code without the USB code.

## SETTING UP USER CODE

- The Co-USB firmware folder is on the live-DVD or can be download from www.pjradcliffe.wordpress ). Copy this folder to your work area.
- The folder contains a file called user_code.c which has several alternative versions-

blank_user_code.c : this is used for the standard ATMEGA32 firmware and has no functional user code, thus the USB features are all that exist. This is the firmware shipped with assembled Open-USB-IO boards.

led_flash_user_code.c : adds some basic user code to show how to to flash an LED, read and write RAM, and do breakpoints. The user_code.c file in the Co-USB folder is a copy of this file.

all_features_user_code.cpp : all features of Co-USB shown and explained.

- To see what these files do first rename user_code.c to old_user_code.c and copy the selected file to user_code.c. Next compile and download into the Open-USB-IO board. Using the live-DVD-
  - ➢ Plug the Open-USB-IO board into the USB socket of your computer.
  - ➢Find and open the Co-USB folder, open a terminal window by hitting F4.
  - ➢Compile the code by typing `make clean` then `make`.
  - ➢If using the USB bootloader then move the link on J10 to J9, hit the reset button, then type
    ```
    make usbprog
    ```
    The next command is needed for all_features_user_code.c to load the EEPROM file.
    ```
    make usbeep
    ```
    Finally move the link on J9 back to J10.
  - ➢If using the STK-200 programming cable, plug it in right next to the USB socket, and type
    ```
    make prog
    ```
    The next command is needed for all_features_user_code.c to load the EEPROM file.
    ```
    make progeep
    ```
    If using Windows remove the STK-200 cable (to remove the reset signal).
  - ➢Read through the source code and try the ousb commands suggested.
  - ➢Start adding your own code bit by bit. Regularly do a `make`, `make prog`, and test your code.

**NEW COMMANDS**

There are several new ousb commands that support user code-

| Command | Function & Capabilities |
|---|---|
| `ousb bp` | There are 8 possible breakpoints numbered 1 to 8.<br>List the status of all breakpoints.<br>Breakpoints may be set or cleared.<br>Continue from a breakpoint.<br>Note breakpoints must be written in the user code. |
| `ousb symr` | Read the value of RAM or EEPROM variables.<br>The variables may be selected by name or address.<br>The display mode may be selected and includes signed and unsigned 8, 16 and 32 bit numbers, characters, strings, and float.<br>Arrays of variables may be printed out.<br>The command line can take multiple variables. |
| `ousb symw` | Write data to RAM or EEPROM variables.<br>Features similar to symr. |
| `ousb user` | Calls user code with data from the command line.<br>Data returned from user code may be displayed. |

## Breakpoint commands

- A breakpoint must be written in source code. For example-
  ```
  BREAKPOINT(N) ;
  ```
  Where N is from 1 to 8. If a breakpoint is added or altered then the firmware must be compiled (`make`) then downloaded (`make prog`).
- `ousb bp` : print the breakpoints which are active, and whether the code is stopped at a breakpoint.
- `ousb bp clear` : clear all breakpoints.
- `ousb bp N` : set this breakpoint to be active (N is numbers 1 to 8).
- `ousb bp -N` : stop this breakpoint being active.
- `ousb bp cont` : continue from any breakpoint once it has been reached.

**Example** : the following code comes from the example led_flash_user_code.c-

```
//=============== user constants and variables ==========================

volatile uint8_t counter ; // view this with the command
                           //     "ousb symr -u08 counter"
                           // volatile is needed to ensure the variable is
                           //   visible to any debugger.

volatile float fa = 123.456 ; // read with "ousb symr -f fa",
                              //    try "ousb symw -f fb 12.3 45.9"

volatile float fb [] = { 56789.1, 0.000345, 1000001} ;
                             // read with "ousb symr -f fb 3",
                             //    try "ousb symw -f fb 12.3 45.9"
                             //    try "ousb symr -e fb 2"


//=============== SYSTEM hooks, leave and ignore ========================
void user_system_500us_interrupt()
{}
void user_command( uint8_t* get_ctrl, uint16_t* get_addr, uint16_t* val)
{}


//============== Executive Loop ==========================================

void user_forever_loop()
{//--- User initializations.

 //--- Forever executive loop.
   for ( ; ; )
    {//--- put your tasks here.

     //--- example tasks, remove these when you add your own code.
       _delay_ms(200) ;
       PORTB ^= 0x01 ;  // flash the PB0 LED.
       ++counter ;      // read this with "ousb symr counter"
       BREAKPOINT(3) ;  // set breakpoint with
                        //     "ousb bp 3", continue "ousb bp cont"
                        // the PB0 led will change state on each continue.
                        // Stop breakpoint with "ousb bp -3" or
                        //    "ousb bp clear"
                        //     then "ousb bp cont" to continue.
    }
}
```

- If this code is running then the LED PB0 will flash.
- To make the breakpoint active use the command `ousb bp 3`.
  The LED will stop flashing as execution is halted at the now active breakpoint.
- To see the status of the breakpoints type `ousb bp`. Note that the output
  `At breakpoint #3. Active breakpoints (1 to 8) = 3`

- Watch the LEDs and try the command `ousb cont`.
  The LED PB0 will change state as execution continues but stops when it next reaches the breakpoint.
- To disable the breakpoint use the command `ousb bp -3`.
  To continue the user code type `ousb cont`.
  The LED should now start flashing.

## Symbol Read with symr

This is a powerful command with many possible parameters and terms which may be entered in any order. Each parameter must be of form "-text".

| Parameter/Term | Meaning |
|---|---|
| -ee | Read from EEPROM memory not RAM. |
| -q or -r | Quiet mode, do not print descriptions just the data. |
| -h or -x | Data printed in hexadecimal if unsigned type. |
| -d | Data printed in decimal if unsigned type. |
| -b | Data printed in binary unsigned type. |
| -c | Print as ASCII character. |
| -s | Print as null terminated string. |
| -uc or -u08 | Print as 8 bit unsigned number, if no type is given this is the default. |
| -s08 or -sc | Print as signed 8 bit number. |
| -u16 or -ui | Print as unsigned 16 bit number. |
| -i or -si | Print as signed 16 bit number. |
| -uli or -u32 | Print as unsigned 32 bit number. |
| -li or -s32 | Print as signed 32 bit number. |
| -df or -f | Print as double or float real number, ( uses IEEE 754 format) |
| -e | As -f or -df but print out in exponential format. |
| variable_name | A variable name as stated in the source code. |
| number | The number of variables to be printed starting at the variable address. |
| =address | Define variable by address. |

**Examples** : using the example code given above-

- `ousb symr -u08 counter` : print the value of the counter. If the code is running and the LED is flashing then this value should be incrementing.
- `ousb symr -f fa` : print the floating point variable fa.
- `ousb symr -f fb 3` : print fa and the next 2x4 bytes as a floating point number.
- `ousb symr -u08 counter -f fb 3` : several reads on one line.
- `ousb symr =0x68 -f 2` : print two float numbers starting address 0x68.
- `ousb symr -ee =100 5` : read 5 bytes in EEPROM starting at address 100.

*Note*: in order to use program symbol names the working directory for the terminal must contain the *.sym file created by the cross compiler.

 

## Symbol Write with symw

This is a powerful command with many possible parameters and terms which may be entered in any order. It is identical in form to symr except any numbers, text or strings in the command line are written to the RAM or EEPROM destination.

| Parameter/Term | Meaning |
|---|---|
| ... | See all the symr terms and parameters. |
| number | A number to be written to RAM or EPROM. |
| '"text string' | Write the text "text string" to RAM or EEPROM |
| /x | Write the character "x" to RAM or EEPROM. |

**Examples** : using the example code given above-

- `ousb symw -u08 counter 100` : write 100 to the counter.
- `ousb symw -f fb 1.1 222.2  0.00033` : write three real numbers starting at fb.
- `ousb symw -ee =200 '"Put this in eeprom!'` : write a string to EEPROM.
  `ousb symr -ee -s =200` : read what was written above.
- `ousb symw =400 /a /b /c /d 0`: write 4 characters to RAM location 400 then a zero.
  `ousb symr =400 -c 5` : read 5 characters just written (0 does not appear).
  `ousb symr -s =400` : read data as null terminated string.

## User Command and Interrupt Issues

The user command calls the following function in user_code.c-

```
void user_command(   uint8_t*  get_ctrl,
                     uint16_t* get_addr,
                     uint16_t* val)
```

The three variables hold the numbers sent from the ousb command line-

```
ousb user 1 2 3
```

This would set *get_ctrl to 1, *get_addr to 2, and *val to 3.

User code in the function user_command can write to these variables which are then returned to the PC for display. If the user code did not modify these variables the response on the screen would show the variables unchanged-

```
user code return:  1   2   3
```

Display mode can be binary, hex, decimal, or raw as with symr.

**Interrupt problems** : all ousb commands including the user command run at interrupt level driven by the USB interrupt. The user code runs at background level and is stopped at some random point when the USB interrupt occurs. The USB code then acts ( for example a data read or write, or the user command), then control goes back the the background user code.

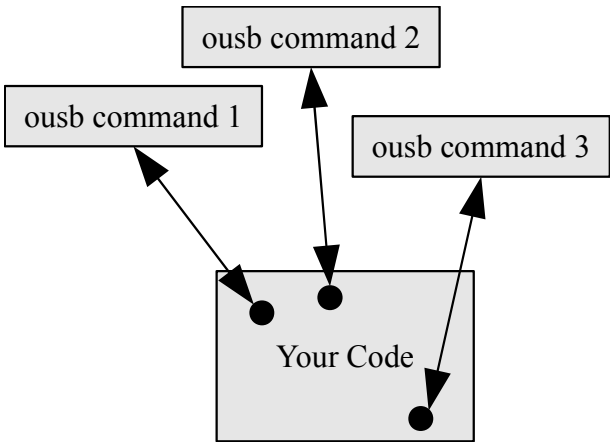


*Illustration 7: Interrupt Acting At Random Points*

The interrupt actions may interfere with the background code in unexpected ways that will be random in nature. For example consider background user code

```
var_char *= 2 ;
```

This consists ofseveral atomic actions (actions that cannot be interrupted), a read, a multiply, then a write. An interrupt level ousb command may set var to 31 after the read but before the multiply. The multiply then occurs and var is now 62 not 31. This will not always happen, just occasionally.

**** Interactions between interrupt level and background level are a major cause of intermittent and hard to trace problems. The design of such interfaces must be very carefully done. The most common solution is a control flag, lets call it `intrOwn`, that controls access to some data.

- If `intrOwn` is zero the associated data is owned by the background level and should not be accessed by the interrupt level.
- If `intrOwn` is one then the associated data is owned by the interrupt level and should not be read or written by the background level.

There are other solutions such as turning off interrupts and only working with atomic actions but these solutions require expert knowledge if they are to work reliably.

**Background access to user command** : The all_features_user_code.c file shows how to use the function-

```
void user_command_to_background()
```

which allows background level user level code to safely access the data written from the USB interrupt level. The variable `new_user_command` is in effect the ownership flag. In this case the user code cannot return data via the user command.

**User command example**: the file all_features_user_code.c has some basic user code inside the `user_command` function. Please have a careful look at this before writing your own user code.

**Aside: why no function call command?** The GCC compiler will perform a bewildering array of optimizations such as putting variables into registers, putting variables on to the stack, making functions inline, sharing code, and more. Variables can be forced into a viewable state by adding the key word "volatile" in front, this is explained in the examples. Code is more of a problem and it is not always reliable to call a user function. To add to these problems many functions have parameters which must be placed in the calling stack.

If you need to call one of your functions using the ousb program on the PC then see the "user" command function above. At the PC the command is of form-

```
ousb user int8 int16 int16
```

Add code to the user_command() routine in the ATMEGA32 Co-USB code to call your function, with parameters from the command line if required.

**Windows 7 Issues**: Windows 7 appears to have timing issues with USB. Things that work well under Win XP, Linux, and the Mac do not work under Windows 7. See the folder Co-USB_win7 for details of the small changes necessary for Co-USB to work under Windows 7.

Quite frankly – its much easier to do the development of Co-USB under Linux so you should probably stick to the live-DVD.

## 7.4 Cutting Free : Open-USB-IO Without the PC

The discussion so far has assumed an Open-USB-IO connected to a PC using the USB cable. If you require the Open-USB-IO board to work by itself then simply unplug the USB cable and add a power supply.

**ATMEGA32 code stand alone**: all the example ATMEGA32 code shown will run if the Open-USB-IO board is standing alone. Even the Co-USB code which has the USB interface will work just fine without the USB cable.

Even if the application does not require USB its useful to leave it in the code. If anything goes wrong then you can simply plug in a PC and debug what is happening.

**Low power**: the Open-USB-IO board is relatively low power , consuming typically 35 ma with all LEDs off and 125 ma with all LEDs on.

Power consumption can be further reduced in the following ways (see the Open-USB-IO circuit)-

- Remove the power LED.
- Set the bank of eight switches to all be off.
- Remove J10 so the LEDs do not light.
- Remove the trim pot and LDR.
- Remove the smaller ICs (MAX232 RS232 driver, ULN2003 motor drive).
- Remove the zener diodes (this will disable USB interface).

Removing the power LED, the two small ICs, and having LEDs off, reduces power consumption to 22ma. The other savings suggested above will not make much difference.

A significant reduction in power will occur if the crystal frequency is changed. The crystal can be as low as 32 kHz, even 1 MHz will make a significant saving, but the USB interface will not work.

**Using a regulated plug pack**: the easiest way to externally power Open-USB-IO is with a 5 volt regulated plug pack. These produce a smooth +5v supply with no peaks. Note that unregulated plug packs will not work, and most likely will destroy your board. If in doubt put a multimeter on the output of the plug-pack. An unregulated plug pack will show a voltage significantly above 5 volts, a regulated plug pack will be very close to 5 volts.

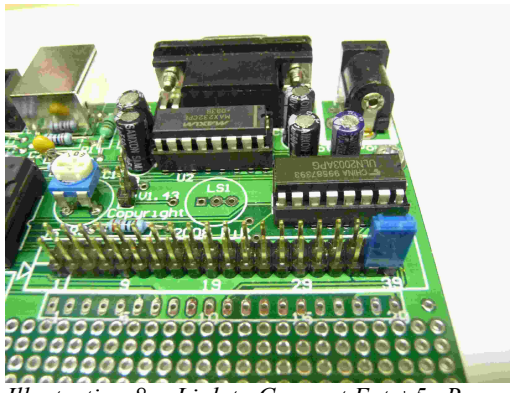

*Illustration 8: Link to Connect Ext +5v Reg*

Ensure the plug pack has a central positive and plug it into the DC motor socket (top right of illustration opposite).
Also put a shorting link between pin 39 and pin 37 of J5, as shown by the blue link.

**3 terminal regulator**: A simple 3 terminal power regulator will enable you to use 9 volt and 12 volt batteries, or unregulated DC plug packs of 9v DC to about 18v DC. There are a large number of suitable chips such as the venerable LM7805 and the low drop out LM2940CT-5. Any 5v regulator in a TO-220 package should work.

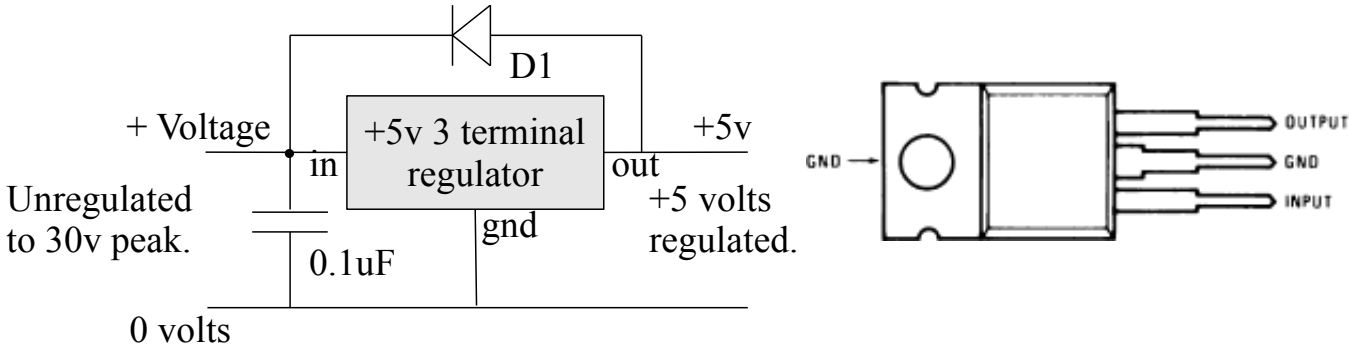


*Illustration 9: 3 Terminal Regulator Circuit*

D1 in the circuit above prevents issues if the unregulated power is suddenly removed. Ideally the diode is a Schottky diode but silicon will do providing it can take 200 ma or above . Normally there are filter capacitors on the outputs but the Open-USB-IO board already has these in its circuitry. Provided you can do basic soldering the circuit is quite simple as shown in the illustration opposite. Find the J3 pads, directly below J5, Use J3 pin 20 (end pad) for +Voltage input, and J3 pin 19 (next pad) for +5v output, for 0 volts use the pad between J3 pad 20 and the edge of the board.

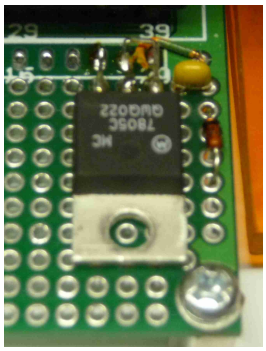

*Illustration 10: +5v Regulator*

**Basic power calculations**: power dissipation calculations are fundamental to electronic design but too often ignored, resulting in equipment that fails during operation at the customer site after a period of time, due to overheating.

For silicon devices the junction temperature must stay below 150 °C. In our example the following equation must hold-

150 >T ambient + Pd * Rt

T ambient=air temperature in °C,
Rt = Thermal resistance °C/Watt
Pd = power dissipation of regulator in Watts.

For a TO-220 package Rt= 50 °C/W.

Pd = V * I = (Plug pack voltage – 5) * ( maximum Open-USB-IO current which is 0.13 amps)

For a 12v DC plug pack then-

150 > T ambient + (12-5)*0.125*50 resulting in T ambient < 44 °C

Now 44°C is above room temperature, but inside a small box with no ventilation it could easily get much higher than this. One solution may be to ensure that the LEDs do not all light, or none because J10 is removed. Another solution is to add a heat sink to reduce Rt.

Try the same power calculations for the popular TO-92 package (small round with a flat side) which has 310 °C/Watt thermal dissipation. You will find it practically useless.

# 7.5 The STK-200 Programming Cable

Most ATMEGA32 code development on the Open-USB-IO board can use the built in USB bootloader to download the ATMEGA32 hex files. There are several occasions when the USB bootloader is not adequate-

- The bootloader has been damaged.
- You are working with blank ATMEGA32 chips.
- You want to use a different crystal (eg 32 kHz for low power).
- Your project needs the USB interface pins, PD2 (INT0) or PD7 (OC2) or PA7.

One of the cheapest solutions is the STK-200 programming cable between a parallel port on the PC and the Open-USB-IO board.

**The STK-200 programming cable** can be built in a few minutes if you have a 25 pin male D connector, a 10 pin IDC socket, and a soldering iron. These may be available from your junk box or a nearby hobby shop. There are also four resistors that can be fitted inside the plastic shell on the back of the 25 pin connector.

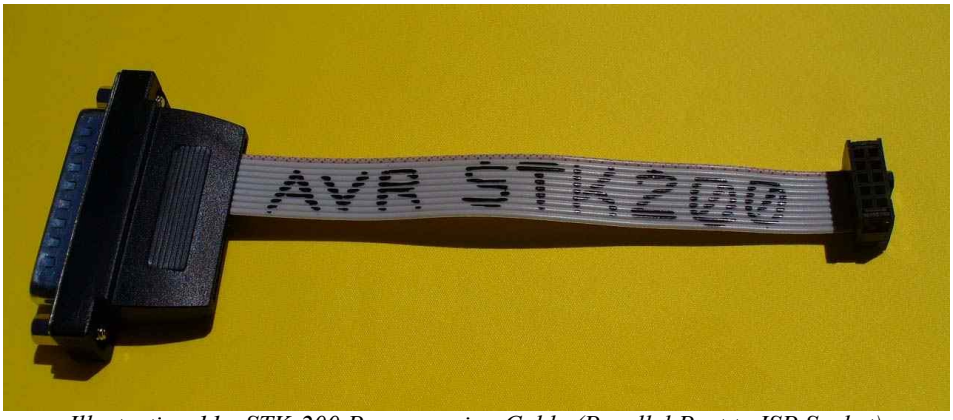


*Illustration 11: STK-200 Programming Cable (Parallel Port to ISP Socket)*

The cable circuit is defined in the document avr_programming_cable.pdf which is on the main web site www.pjradcliffe.wordpress.com and on the live-DVD under /home/user/projects/avr_info/ousb_related. See also the next page for a simplified diagram.

**To use the programming cable-**

- Boot the live-DVD.
- Connect Open-USB-IO to the PC using the USB connector.
- Plug the STK-200 cable between the parallel printer port on the PC and the ISP connector on Open-USB-IO, right next to the USB connector.
- Open up a terminal by double clicking on the terminal icon on the desktop.
- To re-install the standard USB firmware type `ousb_prog` (not ousb_usbprog). This should take about 10 seconds and generate several screens of text.

- To download most of the example programs first open the folder which contains the example. Next hit F4 to open a terminal. Now type `make` to ensure everything is compiled, then `make prog` to download the binary code. The ATMEGA32 program should immediately start executing.

**No Link needed**: the J9 link that is required for USB bootloading has no function when the STK-200 cable is used.

**The STK-200 programming cable**: the basic version is shown below.

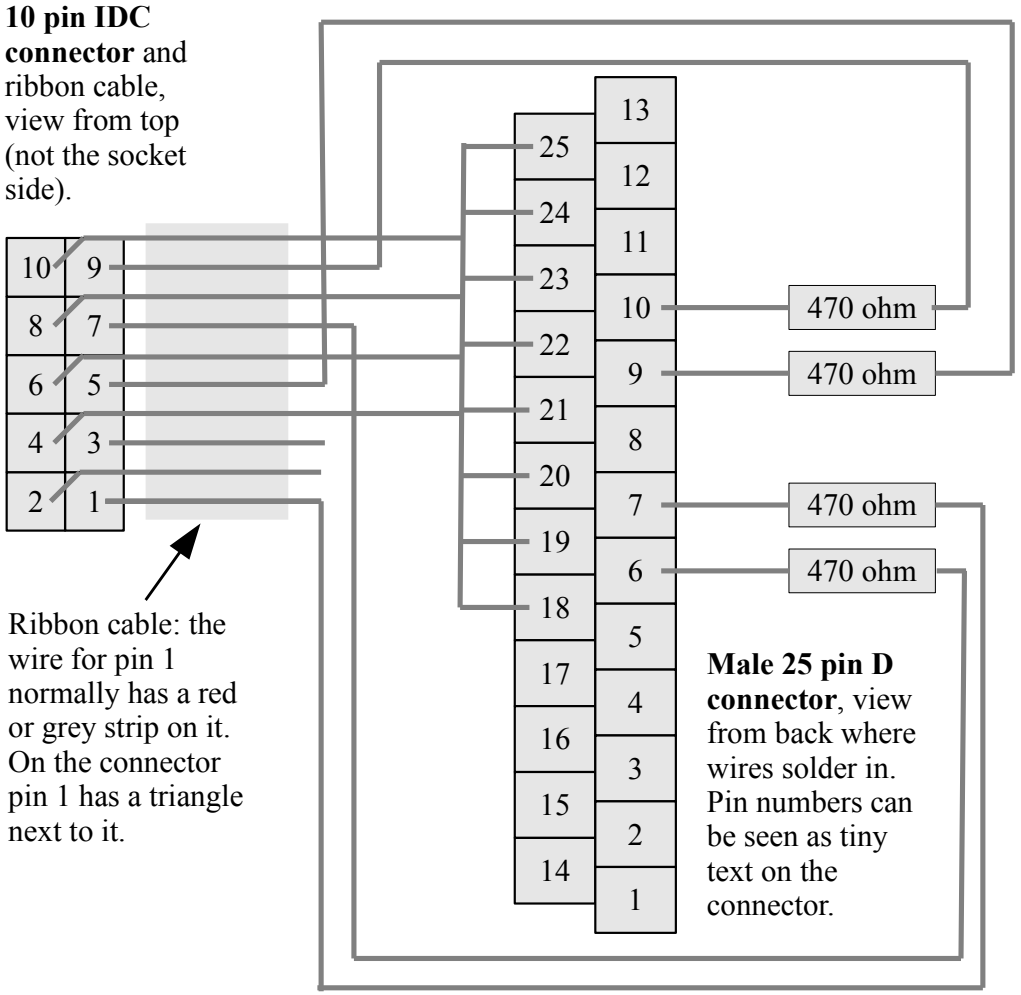


*Illustration 12: STK-200 Programming Cable*

The 10 pin IDC connector is usually available from old PCs where it was used to connect the serial ports on the case the the mother board. The other end, a 9 pin D connector, must be cut off. The male 25 pin D connector will have a plastic shell on the back which can used to house the four resistors.

**Notes**

- If you use the STK-200 cable on Windows it may be necessary to remove the programming cable otherwise the microprocessor may be held in reset.

# 8 Compiling Open-USB-IO Code

This chapter explains how to compile the source code behind the command line Open-USB-IO software. There are two key programs; the PC command line program ousb and the ATMEGA32 microprocessor firmware. The PC program takes command line arguments and sends these to the Open-USB-IO board via USB. The ATMEGA32 firmware acts as a USB client, accepts the commands and takes action, then replies with any result.
You may also use the Open-USB-IO board as a general purpose microprocessor that can have your own code downloaded and run. This is explained in earlier chapters.

Development is much easier under Linux, especially using the live-DVD, but it can also be done on the Mac, Windows XP or Vista. Programming the ATMEGA32 chip can be difficult under Windows compared to Linux.

The PC code must be compiled with gcc/g++ which has been ported to many platforms. The one set of files can work on any platform merely by opening the Makefile and adjusting variables right at the start which define the operating system.

The ATMEGA32 firmware must be compiled with avr-gcc, a variant of gcc developed for Atmel microprocessors from the Tiny26 upward. Like gcc, avr-gcc can be compiled for nearly any platform. This section explains how to get avr-gcc working for Windows and Linux. The Open-USB-IO firmware can be compiled on any platform merely by opening the Makefile and adjusting the variables that define the operating system and the path to the avr-gcc compiler.

## 8.1 Compiling ATMEGA32 Firmware on Linux

The ATMEGA32 firmware can be compiled on Windows or Linux, or the Mac by using the avr-gcc tool-chain. The well known gcc compiler can compile for several targets and the AVR series of microprocessors from the Tiny26 upward are handled. First the tool-chain must be installed, then the firmware compiled, and finally the ATMEGA32 must be programmed.

**Windows tool-chain** : go to http://winavr.sourceforge.net/ and download Win-AVR; the Open Source compiler, programmer, and debugger. Carefully follow the instructions for installation.

Note that if you have cygwin installed then the directory \cywin must be renamed so Win-AVR can work.

**Linux tool-chain**: go to www.pjradcliffe.wordpress.com and download the linux-avr-gcc-kit. Log in as root and place the kit in /usr/local and unzip it. By default it will unzip to /usr/local/avr. Read the readme.html file for detailed instructions.

It is even easier to use the Live-DVD which has the full toolchain installed plus many other useful development tools.

**Open-USB-IO firmware**: from www.pjradcliffe.wordpress.com download the Open-USB-IO_ firmware, unzip, and place in a convenient working directory. Edit the Makefile and

ensure the user adjustable variables at the start are altered to suit your operating system and directory structure.
Note you should download the current command line program as well to ensure compatibility.

Next open a terminal screen to this new directory.
For Windows start->run-> type in cmd then enter. Use the "cd" command to move to working directory.
Type "make help" to see the options available under make. Note "make prog" which can program the ATMEGA32 using just a cable.

Now type just "make" to create the code and check for error messages. The net result should be a *.hex file ready to download to the ATMEGA32.

**STK-200 programmer**: if your computer runs either Linux or Windows XP and the computer has a parallel port, then you can program the ATMEGA32 using nothing more than a cable with 4 resistors and the avr_dude software that comes with the avr-gcc tool-chain.

On www.pjradcliffe.wordpress.com download the "stk-200 programming cable" circuit and build the version with the 10 pin IDC plug. For this you should at least be able to solder a few wires into a cable.

For Windows XP download the "windows_IO_access", unzip it and follow the instructions. This is needed to give the programmer avr_dude access to the parallel port.

**Programming under Linux**:

- Plug a USB cable into the Open-USB-IO board and the computer. This is needed to power the board.
- Plug the stk-200 cable from the parallel printer port to the ISP socket on the Open-USB-IO board.
- From a terminal pointed to the source code directory type "make prog" and check for any error messages.

**Programming under Windows**:

- Plug in the cables as per the first two steps for Linux.
- From a terminal pointed to the source code directory type "make io" to enable IO access.
- From a terminal pointed to the source code directory type "make prog" and check for any error messages.
- Unlike Linux it may be necessary to remove the programming cable before the microprocessor will run.

# 8.2 Compiling ousb on Linux

The command line program ousb is simple to compile on a Linux computer.

- Ensure the gcc and g++ compilers are installed, from a command line try "gcc –version" and "g++ --version" to check they exist.
  Use you package manager to install gcc and g++ if not already installed.
- The library libusb may need to be installed from your package manager if not already installed, install all packages starting with "libusb".
  Alternatively see http://libusb.sourceforge.net
- Download from www.pjradcliffe.wordpress.com the source open-usb-io_command_line and place in a convenient directory. Unzip the file ( right click then extract or Actions->unzip).
  Note you should download the current firmware code as well to ensure compatibility.
- Check the Makefile has LINUX as the target operating system.
- Open a terminal and type "make all" to compile the code. Try "make help" to see other make options.
- Try running ousb as per previous examples.

**Ubuntu & Kbuntu**

Some feedback from users of Ubuntu-9.04 is that the following must be added to the file SRUSB.cpp (thank you Owen)-

```
    #include <string.h>
```

Ubuntu and Kbuntu block access to USB devices for users. Open up a root terminal or file explorer and copy the following file into /etc/udev/rules.d/50-usb.rules

```
#--- PjR 5/3/2010 make all new USB devices user accessible for r/w.
SUBSYSTEM !="usb_device", ACTION !="add", GOTO="usb_rules_end"
SYSFS{idVendor} =="*", SYSFS{idProduct} =="*"
MODE="0666", OWNER="user", GROUP="user"
LABEL="usb_rules_end"
```

It should not be necessary to reboot after adding this file. A new USB device insertion should use this new rule.

## 8.3 Compiling ousb on Windows

The Open-USB-IO command line code can be compiled under Windows or Linux simply by changing one line the the Makefile.

- Get gcc : Open-USB-IO is essentially a gcc/g++ compatible code and so you will need to get gcc/g++ onto your Windows host. The simplest way is to use the MINGW package that you can download from http://www.mingw.org/
  Alternatively download Dev-C++ which provides an excellent C/C++ IDE and the MinGW compiler. See http://sourceforge.net/projects/dev-cpp/
- Get usblib: get the usb library by going to http://libusb-win32.sourceforge.net/ and uploading their Filter Driver. Open-USB-IO was compiled using version 0.1.12.1 though any later version should work. Follow instructions to install the library.
- Download the source open-usb-io_command_line from www.pjradcliffe.wordpress.com and place in a convenient directory. Unzip the file, the free 7-zip from www.7-zip.org works well.
- Check the Makefile has WINDOWS as the target operating system, at most this is a one line chamge. Make sure there is no blank spaces at the end of the line.
- Open a terminal ( start->run type cmd then enter). Change directories to where you have placed the source code. Type “make clean” then "make all" to compile the code.
- Try running ousb as per previous examples.

# 9 Tools & Information for the ATMEGA32

The preferred development environment for the ATMEGA32 is the live-DVD from www.interestingbytes.wordpress.com which has an amazing array of tools plus all the source code for the Open-USB-IO, both the ATMEGA32 and Linux/Windows programs. The manual gives detailed instructions on how to use the live-DVD but this section shows where the basic tool chains were obtained and gives other useful information.

**Windows**:

- AVR-GCC: go to http://winavr.sourceforge.net/ and download the Open Source compiler, programmer, and debugger. Note especially the portable WIN-AVR that can be run entirely from a USB stick. Open-USB-IO firmware has been compiled using this combination.
- AVR Studio from Atmel is a very powerful and free IDE.
  See http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725

**Linux**: the same tools are available but may need to be downloaded individually and compiled. Installation and compilation can be very difficult and so a live-DVD mentioned above is suggested.

- avr-gcc tool-chain: see the linux_avr-gcc-kit at www.pjradcliffe.wordpress.com which makes it easy to install the avr-gcc tool-chain. Open-USB-IO firmware has been compiled using this combination and a live-DVD is available with everything installed.
- http://www.gnu.org/savannah-checkouts/non-gnu/avr-libc/user-manual/install_tools.html is the site that hosts many tools and has up to date information.
  The tools include compilers, programmers, and software for JTAG based In Circuit Emulators (ICE).
- Eclipse, the powerful Open source, multi-purpose IDE, has an AVR plug-in that uses the avr-gcc tool-chain. The documentation is very good, usually a sign that the product is good also. It relies on the GNU tool chain described above.
  See http://avr-eclipse.sourceforge.net/wiki/index.php/The_AVR_Eclipse_Plugin
- Atmel's AVR Studio will run under wine (a Windows emulator) and can even drive a serial port based ICE. See-
  http://appdb.winehq.org/objectManager.php?sClass=version&iId=14589&iTestingId=33793
- kontrollerlab is an Open Source IDE. Its useful but not maintained since 2006.
  See http://www.cadmaniac.org/projectMain.php?projectName=kontrollerlab&section=main
- http://tuxgraphics.org/electronics/200901/avr-gcc-linux.shtml has a useful set of instructions for installing the avr-gcc tool-chain.
- http://avrwiki.com/wiki/index.php/AVR_GCC is old but has useful information.

- http://cdk4avr.sourceforge.net/ is old but also has useful information.

**ICE** ( In Circuit Emulators) are an excellent way to debug microprocessor code and JTAG based hardware is quite cheap. Most ICE devices conform to Atmel's JTAG 10 pin header interface, and either RS-232 or USB on the computer. Computers that lack an RS232 interface may require an RS-232 to USB converter.

- http://www.avrfreaks.net/index.php?module=Freaks%20Tools&func=viewItem&item_id=511 sells a unit for about US$40.
- http://www.ledz.co.kr/avr/main_jtagisp_en.htm make very interesting devices.
- Atmel make their own ICE which is more highly priced. Note there is a Mark 1 and Mark 2. Some software will only work with Mark 1.
  See http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2737

**JUST PROGRAMMERS**: if you are using the built in USB bootloader then you will not need a separate programmer. If you will be working with blank ATMEGA32 chips themn you will need something else. Most ICE will also program the AVR microprocessor but units which just program can be much cheaper, if you have a junk box with the right parts then you can build one for nothing in 10 minutes! The simplest and cheapest programmer is the STK-200 cable which drives from a parallel port on your PC. The wiring for this is shown in the Hardware Schematics appendix. There are other programmers you can buy that run from the serial port or USB.

Note that the new Open-USB-IO firmware includes a boot loader which can load your ATMEGA32 code into memory. If you use this then you probably wont need any of the following programmers. For interests sake some typical programmers include-

- http://www.avrfreaks.net/modules.php?op=modload&name=News&file=article&sid=684
  http://robokits.co.in/shop/index.php?main_page=product_info&cPath=12&products_id=54
- Try a web search on "AVR programmer" and check your nearest hobby shop.
- See the Open-USB-IO downloads section of www.pjradcliffe.wordpress.com where a simple cable with 4 resistors can program the ATMEGA32 using the avr_dude programming software that comes with the avr-gcc toolchain.
- PonyProg at www.lancos.com/prog.html is an excellent programmer with a GUI interface that works on Windows or Linux. It uses any serial port on the PC.
  http://extremeelectronics.co.in/avr-tutorials/part-iii-making-programmer-and-target/ describes a simpler version of the PonyProg that works well. These units may work with USB-serial adaptors but will be very slow.
- USB programmers can be found, but check they come with programming software, or are supported by your existing software.

**ATMEGA32 information**. The Atmel ATMEG32 is a powerful single chip microprocessor with powerful and sophisticated on-chip hardware such as timers, Pulse Width modulators, serial communications, analogue to digital conversion and more. It can be hard to know how to start working with the chip and these peripherals.

- Datasheet: the bible is the well written ATMEGA32 ( and ATMEGA32A) data-sheet from Atmel, see http://www.atmel.com/dyn/products/datasheets.asp?family_id=607#760 This document is big (350+ pages) and will take some time to understand. This is a must read document.
- AVR Freaks is an essential site to visit, see http://www.avrfreaks.net/ This has lots of AVR projects, code, and documents. The web site (as of 2009) has hotlinks on the grey bar near the top, and a set of white tabs under that. These tabs hold a wealth of good information, for example tooltree->App Note->Getting Started has a whole lot of excellent and useful documents. The tools tab has programmers, more useful documents, development boards – a huge array of things. Make time to wander about this site, you are sure to download heaps of useful information and tools.
- Extreme Electronics at http://extremeelectronics.co.in/avr-tutorials/ has some excellent tutorials and small projects.
- Web search: the web has huge numbers of projects published with full circuits and software. For example want to drive a VGA display from the ATMEGA 32 with 6 resistors? Try here: http://www.uelectronics.info/softvga-avr-vga-display .
  How about an iPod interface, try http://dev.emcelettronica.com/category/atmega32

**Where to start**: the ATMEGA32 is so complex that at first glance it can be overwhelming. Try the following approach.

- Get the data-sheet, read introduction section 1 to 7, then section 10 on digital IO. Understand the data direction registers, the inputs and output registers. Use the "ousb io" commands to play around, observe LED output and read switch inputs. Next read any section you find interesting. The timers, the ADC, and PWMs are useful and reasonably easy to follow.
- Look at published projects and the code examples to see how they control the peripherals.

# 10 Common Problems & Solutions

This section outlines common problems that people have found, the next section solves those problems. As a challenge try to solve the problem yourself before looking at the answers. This section is mainly built from problems that users and students have reported. If you have a neat problem and solution please send it to interestingbytes@gmail.com .

**Simple IO** : the following ATMEGA32 code, downloaded into the Open-USB-IO, attempts to do something very simple; take the input from the switches and write that to the LEDs. It doesn't work! See if you can spot the problem and devise a solution. Answers are in the next section.

```
//--- Write the switches to the LEDs (fails).
#include <avr/io.h>

int main()
{ DDRB  = 0xff ; // set all port B pins to output.
  PORTB = 0x00 ; // set port b outputs to zero.
  DDRC  = 0x00 ; // set all port c pins as inputs.
  PORTC = 0x00 ; // set port c to zero.

  for ( ; ; )
   { PORTB = PORTC ;
   }
}
```

**The LEDs don't light** even though the ousb command appears to work.

**Device not found on Linux**. Trying to use ousb on a Linux box results in a message “Device not found”.

**ousb not found**. Attempting to start the ousb program fails even though the binary is on the disk.

**Smoke!** I plugged in the ISP or JTAG cable and the board smoked!

**Board in reset**: I am using the STK-200 programming cable but when I power up my PC the board stays in reset. What's wrong?

**The USB bootloader doesn't work**: I try `make usbprog` but nothing gets downloaded.

**The application wont run** after the USB bootloader runs.

## Not So Common Problems

**Flickering globe**: this is a very tricky problem sent to me by John F. ( thanks John F). A 12 volt incandescent globe was being driven by PWM 2. The globe was powered by an unregulated DC plug pack. At 100% duty cycle the globe showed no flicker. With a PWM of around 50Hz the globe showed a lot of flicker. What is wrong?

**Can only access ousb as root**. Attempts to access ousb from user level fail as the user does not have the permissions necessary to access USB devices.

# 10.1 Solutions to Problems

**Simple IO solution** : the code will fail for two reasons.

In the for loop, the switches are not being read! PORTC is the register that holds the output, PINC must be used to read the input. If the PORTC is changed to PINC the code will start to work – well sort of. A switch set to on (up) will always turn an LED off, but a switch set to down will result in an LED that is sometimes on and sometimes off. The LED may even flicker as you run you fingers across the PCB.

Even though port C is all set to inputs by setting DDRC to zero, the output register PORTC can still effect the inputs. If a PORTC bit is high then the corresponding pin is connected to an internal pull-up resistor of 20 kohm to 50 kohm connected to Vcc. If the PORTC bit is low the input has no pull-up and has an input resistance of at least 1 megaohm.

Looking at the circuit for the board, the switches have a 4.7 kohm resistor to zero volts so a closed switch will always pull the microprocessor pin to a logic low. If a switch is open circuit then the microprocessor pin is not pulled high or low, it just floats and takes on random values.

The solution is to set PORTC to 0xFF so when the switch is open the internal pullup resistor pulls the microprocessor input to a valid high. The fixed code is now-

```
//--- Write the switches to the LEDs (works).
#include <avr/io.h>

int main()
{ DDRB  = 0xff ; // set all port B pins to output.
  PORTB = 0x00 ; // set portb outputs to zero.
  DDRC  = 0x00 ; // set all port pins as inputs.
  PORTC = 0xff ; // enable internal pullup resistor.

  for ( ; ; )
   { PORTB = PINC ;
   }
}
```

**The LEDs don't light** even though the ousb command appears to work.

*Answers*: 1) Check the strap just above the LEDs is in place.
2) Try hitting the reset button then trying again.

**Device not found on Linux**. Trying to use ousb on a Linux box results in a message "Device not found".

*Answers*: 1) The USB device is not available to this user. Try do this as root.
Your security set up appears to block casual access to USB.
The quick solution is to login as root and try again.

**ousb not found**. Attempting to start the ousb program fails even though the binary is on the disk.

Answers: 1) This is probably a path issue, ousb is not in the path or the current directory.
2) For Linux, to start ousb in the current directory type `./ousb` not just ousb.

**Smoke!** I plugged in the ISP or JTAG cable and the board smoked!

*Answer*: The JTAG and ISP cable have the same shaped socket BUT the wiring is very different. Each has +5v and 0v in opposite positions so if you put a JTAG cable into the ISP hole, or the ISP cable into the JTAG hole, then you short +5v and 0v.

First pull out the cable quickly! If the board still works then all is probably OK. If the power LED does not light, or the commands fail, then probably the blackened inductor L1 or L2 needs to be replaced.

**Board in reset**: I am using the STK-200 programming cable but when I power up my PC the board stays in reset. What's wrong?

*Answer*: when the PC powers up it sets output values on the parallel port. Unfortunately its sets the line used for reset to low which causes the ATMEGA32 to halt and do nothing.

There are several solutions, the simplest is to pull out the STK-200 cable. A better solution is to use the built in command `ousb_reset` which will reset the ATMEGA32 and then un-reset it so the application code can run.

**The USB bootloader doesn't work**: I try `make usbprog` but nothing gets downloaded.

*Answer*: the boot loader requires that the link on J10 be moved to J9 and the reset button be pushed.

**The application wont run** after the USB bootloader does a download.

*Answer*: remember to take move the J9 link back to J10.

 

**Flickering globe**: this is a very tricky problem sent to me by John F. ( thanks John F). A 12 volt incandescent globe was being driven by PWM 2. The globe was powered by an unregulated DC plug pack. At 100% duty cycle the globe showed no flicker. With a PWM of around 50Hz the globe showed a lot of flicker. What is wrong?

*Answer*: the DC plug pack is unregulated and has a high level of 50 Hz ripple when loaded by the incandescent globe. The mains 50 Hz and the PWM 50 Hz are slightly different in frequency and so there is a beat frequency effect that produces a low frequency flicker in the globe.

**Can only access ousb as root**. Attempts to access ousb from user level fail as the user does not have the permissions necessary to access USB devices.

*Answer*: if you are using ousb on some Linux distributions you will find that the security settings bloc user access to USB devices. This is a good idea in general as handling USB should be left to the operating system. Its a bad idea for engineers and programmers as they will want to tinker with USB devices.

The solution is outlined in the section 8.2 Compiling ousb on Linux.

# 11 Appendix 1 : Hardware Schematics

The full hardware schematics are available from www.pjradcliffe.wordpress.com. This section has a few other useful diagrams.

**J4 and J5 connectors** are shown below for convenience. Orient the board with the 8 switch block and the reset switch at the top. Note pin 1 has a small triangle marked on the board. Looking down at each connector the pin functions are as follows-

**J4**

| Function | Pin # | Pin # | Function |
|---|---|---|---|
| PC0 | 1 | 2 | 0v |
| PC1 | 3 | 4 | 0v |
| PC2 | 5 | 6 | 0v |
| PC3 | 7 | 8 | 0v |
| PC4 | 9 | 10 | 0v |
| PC5 | 11 | 12 | 0v |
| PC6 | 13 | 14 | 0v |
| PC7 | 15 | 16 | 0v |
| Vcc | 17 | 18 | 0v |
| 0v | 19 | 20 | 0v |
| PB0 | 21 | 22 | 0v |
| PB1 | 23 | 24 | 0v |
| PB2 | 25 | 26 | 0v |
| PB3 | 27 | 28 | 0v |
| PB4 | 29 | 30 | 0v |
| PB5 | 31 | 32 | 0v |
| PB6 | 33 | 34 | 0v |
| PB7 | 35 | 36 | 0v |
| Vcc | 37 | 38 | 0v |
| 0v | 39 | 40 | 0v |

**J5**

| Function | Pin # | Pin # | Function |
|---|---|---|---|
| PA0 | 1 | 2 | 0v |
| PA1 | 3 | 4 | 0v |
| PA2 | 5 | 6 | 0v |
| PA3 | 7 | 8 | 0v |
| PA4 | 9 | 10 | 0v |
| PA5 | 11 | 12 | 0v |
| AVcc | 13 | 14 | 0v |
| PA7 | 15 | 16 | 0v |
| PD3 | 17 | 18 | 0v |
| PD6 | 19 | 20 | 0v |
| OC-PD4 | 21 | 22 | 0v |
| OC-PD5 | 23 | 24 | 0v |
| OC-PB4 | 25 | 26 | 0v |
| OC-PB3 | 27 | 28 | 0v |
| OC-PB2 | 29 | 30 | 0v |
| OC-PB1 | 31 | 32 | 0v |
| OC-PB0 | 33 | 34 | 0v |
| Vcc | 35 | 36 | 0v |
| CC | 37 | 38 | 0v |
| Vcc | 39 | 40 | 0v |

Vcc = +5v,
PB drives LEDs ( 10 mA load),
PC senses switches ( 4.7kohm pulldown)

PA, PB, PC, PD correspond to the ATMEGA32 IO ports.

OC- are Open Collector drives for motors.
CC = external power source, for OC devices.
AVcc = low noise +5v for analogue work.

**J7 (ISP: In-circuit Serial Programmer) and J8 (JTAG) connectors** are shown below for convenience. Again orient the board with the 8 switch block and the reset switch at the top. Looking down at each connector the pin functions are as follows-

## J7 ISP

| Function | Pin # | Pin # | Function |
|---|---|---|---|
| 0v | 10 | 9 | PB6 (MISO) |
| 0v | 8 | 7 | PB7 (SCK) |
| 0v | 6 | 5 | /RST |
| 0v | 4 | 3 | NC |
| Vcc | 2 | 1 | PB5 (MOSI) |

NC = Not Connected.

## J8 JTAG

| Function | Pin # | Pin # | Function |
|---|---|---|---|
| 0v | 10 | 9 | PC5 (TDI) |
| NC | 8 | 7 | NC * |
| /RST | 6 | 5 | PC3 (TMS) |
| Vcc | 4 | 3 | PC4 (TDO) |
| 0v | 2 | 1 | PC2 (TCK) |

* Note that some JTAG devices need Vcc on pin 7. In this case solder a link from pin 4 to pin 7.