

# Kolban's Book on Raspberry Pi

**FEBRUARY 2016**

Neil Kolban

## Table of Contents

Introduction – The PI Book.....	14
What is a Raspberry Pi.....	15
Pi Product Models.....	15
Powering the Pi.....	17
Power Consumption.....	17
Pi Operating Systems.....	17
Pi Zero.....	18
Box housings for the Pi.....	18
Powering off the Pi.....	19
Architecture.....	20
The ARM CPU.....	20
Broadcom System On a Chip.....	21
Video subsystem.....	21
Storage subsystem.....	21
Installation.....	21
Installing Raspbian.....	21
Installation tools.....	23
apt-cache.....	23
apt-file.....	24
apt-get.....	24
apt.....	25
dpkg.....	25
synaptic – GUI package management.....	25
Updating.....	26
The Raspbian Environment.....	26
The shell.....	27
The file system.....	29
Mounted file systems.....	30
File permissions.....	31
Editors.....	31
Processes.....	34
Kernel mode vs User mode.....	36
Root privileges and running applications.....	37
Security and userids.....	38
System applications.....	38
Writing your own units.....	39
The systemd-ui.....	41
Kernel modules.....	41
The Device Tree.....	42

CPU - /proc/cpuinfo.....	43
Memory - /proc/meminfo.....	45
System parameters.....	46
Clock frequencies.....	46
Internal temperature.....	47
Memory.....	47
Hardware interfacing.....	47
Pin numbering.....	47
Bread-boarding.....	51
Power.....	52
GPIO.....	52
GPIO theory.....	53
GPIO on the Pi.....	53
SPI.....	56
SPI theory.....	56
SPI on the Pi.....	57
SPI in Python.....	58
I2C.....	59
I2C theory.....	59
I2C on the Pi.....	61
PWM.....	64
PWM theory.....	64
PWM on the Pi.....	65
UART (Serial) Interface.....	67
UART theory.....	67
UART on the Pi.....	67
Using a USB to UART as additional UART.....	69
Analog input.....	70
USB.....	70
Audio.....	71
Speech output.....	72
Ethernet.....	72
Video output.....	73
Camera.....	73
WiringPi.....	74
WiringPi setup.....	75
WiringPi GPIO.....	75
WiringPi SPI.....	75
WiringPi I2C.....	76
WiringPi PWM.....	76
Memory mapped I/O.....	77
Using sysfs to access GPIO.....	77
Using an Arduino as a peripheral controller.....	77
GPIO mapping between the Pi and Arduino.....	79

I2C mapping between the Pi and Arduino.....	80
Integrating the ATMEGA 328P with your Pi.....	82
Programming in general.....	83
ELF structure.....	84
Libraries of code.....	84
Static vs Dynamic libraries.....	86
Building shared libraries.....	87
General development tools.....	87
ar.....	87
ldd.....	88
nm.....	88
objdump.....	88
ranlib.....	88
readelf.....	88
C Programming.....	89
C Programming theory.....	89
C tools for Pi.....	91
Simple compilation.....	91
Editing sources via Eclipse.....	91
Editing sources on the Pi.....	92
Cross compilation.....	93
Makefiles.....	100
Debugging C.....	104
Debugging on a remote Pi.....	105
C++ Programming.....	109
Calling functions.....	110
Install the ARM compiler tool-chain.....	110
Java Programming.....	110
Java Development Environment for Pi.....	111
Using Eclipse for Java Pi development.....	112
Pi4J Project.....	112
Pi4J Installation.....	113
Writing Pi4J applications.....	113
Pre-built Pi4J classes.....	114
The Device I/O API.....	114
UI programming in Java.....	114
JavaFX.....	114
Calling C from Java.....	115
JavaScript Programming.....	117
Node.js.....	117
NPM.....	118
GPIO from Node.js.....	119

Using npm wiring-pi.....	120
Nashorn.....	121
JavaScript bit-twiddling.....	121
Python Programming.....	121
Arduino programming for the Pi.....	124
Overview of the Arduino IDE.....	126
Installing the Arduino IDE.....	128
Installing the Arduino IDE board configuration for the Pi.....	128
Configuring the Pi.....	129
Selecting the Pi as the target for development.....	129
Developing with an NFS export from or to the Pi.....	130
Debugging a sketch.....	131
Differences from a hardware Arduino.....	133
Redirecting Serial output to the Console.....	133
Importing a 3 <sup>rd</sup> party library.....	134
Running the Arduino IDE on the Pi.....	136
Raspbian Environment Programming.....	136
Audio output from a your application.....	136
Multi threading.....	137
Open file descriptors.....	139
Networking.....	141
Networking Devices.....	141
Ethernet.....	141
Setting up a static Ethernet IP Address.....	141
Using Windows Internet Connection Sharing.....	141
WiFi.....	142
WiFi USB devices.....	143
Scanning WiFi networks.....	145
Setting up WiFi connections.....	145
Setting up the Pi as an Access Point.....	146
Wireless Suplicant.....	148
Programming to WiFi.....	149
Using a Windows PC as a WiFi access point / hotspot.....	149
DHCP.....	149
Remote login.....	151
Domain Name Service.....	151
DNS Service Discovery and Multicast DNS.....	153
Web browsers and servers.....	154
Pi as a Web Server.....	154
REST requests.....	156
Using postman for testing.....	156
Writing an App that makes a REST request.....	156
Writing a REST request using Node.js.....	156
Writing an App that listens for REST requests.....	157

Listening for REST requests using Node.js.....	157
WebSocket.....	158
A WebSocket browser hosted application.....	159
Pi as a WebSocket server.....	160
WebSocket Server for C using libwebsocket.....	160
WebSocket Server for C using noPoll.....	160
WebSocket Server of JavaScript.....	161
Sharing Windows files.....	161
Network File System (NFS).....	161
Cloud Systems.....	163
Thingspeak.com.....	163
Sockets API programming.....	169
Finding an IP address from a hostname.....	169
Creating a socket client.....	170
Creating a socket server.....	170
Sending data through a socket.....	171
Receiving data through a socket.....	171
Closing a socket.....	171
UDP programming.....	172
JavaScript socket.io.....	172
Sending an email.....	172
MQTT.....	174
MQTT Protocol.....	175
Mosquitto MQTT.....	177
Building mosquitto from source.....	178
Writing MQTT clients.....	179
Eclipse paho.....	179
C – Mosquitto client library.....	179
Node.js JavaScript – MQTT.....	181
Browser JavaScript – MQTT.....	182
Interacting with Mobile devices.....	185
Blynk.....	185
X-Windows.....	186
Using xming on a Windows PC.....	188
Desktop environments.....	192
Logging.....	193
Performance.....	195
Resource Utilization.....	195
Memory Utilization.....	195
Performance Commands.....	196
top.....	196
lxtask.....	197

vmstat.....	198
Performance characteristics of different programming languages.....	198
Security.....	199
Firewalls ... on and off.....	199
Bare Metal Programming.....	200
Hardware Interfacing Examples.....	201
Simple GPIO Output change.....	202
Using the shell.....	203
Using WiringPi.....	204
Using Pi4J.....	205
Using Node.js.....	205
Using RasPiArduino.....	205
Using Python.....	206
Simple GPIO Input detection.....	206
Using the shell.....	207
Using WiringPi.....	208
Using Pi4J.....	209
Using Node.js.....	209
Using Python.....	210
Interrupt Driven Input detection.....	210
Using the shell.....	210
Using WiringPi.....	211
Using Pi4J.....	212
Using Node.js.....	212
GPIO extenders.....	213
PCF8574.....	213
MCP23017.....	216
NeoPixels.....	220
NeoPixel theory.....	220
NeoPixels on the Pi.....	220
DC Electric Motors.....	220
Servos.....	221
Using the gpio command.....	222
Using WiringPi.....	223
GPS.....	224
The gpsd package.....	225
GPS from your cell phone.....	230
Video / Webcams.....	231
Streaming a webcam video.....	234
Recording a webcam image for a period of time.....	235
Playing video.....	235
Analog to Digital conversion – ADC0832.....	236
Using the shell.....	240
Using WiringPi.....	240

Using Pi4J.....	241
Using Node.js.....	243
Analog to Digital conversion – MCP3208.....	244
Using WiringPi.....	247
Analog to Digital conversion – ADS1015.....	249
Using JavaScript.....	251
Analog to Digital conversion – Arduino.....	252
Joysticks.....	252
Compass – HMC5883L (aka GY-271).....	253
Using the shell.....	258
Using WiringPi.....	258
Using Pi4J.....	259
Using Node.js.....	260
Accelerometer and Gyroscope – MPU-6050 (aka GY-521).....	261
Using the shell.....	266
Using WiringPi.....	266
Using JavaScript.....	267
Using Arduino libraries.....	268
Motion detectors – Passive Infrared Sensor.....	269
Ultrasonics – HC-SR04.....	270
Using the shell.....	272
Using WiringPi.....	273
Using Pi4J.....	273
Using Node.js.....	274
Audio using the ICSH030A.....	274
Buzzers and Piezos.....	274
FM Radio.....	276
Temperature and Humidity – DHT22.....	278
Using WiringPi.....	279
Using the Arduino APIs.....	279
Temperature and pressure – BMP180.....	280
Using the Arduino APIs.....	281
Pressure strips – FSR 402.....	281
Ambient light sensor – BH1750FVI.....	282
Infrared receivers.....	282
LED 7-Segment displays.....	285
MAX7219/MAX7221 – Serial interface, 8-digit, led display drivers.....	285
Using the Arduino APIs.....	288
Using Python.....	288
Using JavaScript.....	289
LCD dot matrix display – HD44780.....	290
Using WiringPi.....	297

Using Java.....	298
LCD display – Nokia 5110 – PCD8544.....	298
Using WiringPi.....	299
Using Pi4J.....	300
OLED 128x64 – SSD1306.....	300
Using the Android API.....	301
TFT 1.44" Color – ILI9163C.....	301
TFT 2.4" 320x240 – TJCTM24024-SPI.....	301
Frame Buffer TFT driver.....	304
Touch screen input.....	309
RFID MFRC522.....	310
MFRC522 – Low levels.....	312
Initialization.....	315
AntennaOn.....	315
Using Python.....	315
Using JavaScript.....	315
Communications – nRF24L01.....	316
Using the Arduino APIs.....	317
Bluetooth – HC-05/HC-06.....	320
Real time clocks.....	323
CD4051 – Digital switches.....	326
Robotics.....	327
Locomotion.....	328
The H-Bridge.....	329
The robot project.....	330
Robot – Network commands.....	333
Balancing wheel power.....	334
Compass bearing.....	334
Windows IoT.....	334
Connecting the dashboard.....	338
Opening a shell to the Windows 10 IoT.....	339
Process management commands.....	340
Installing Visual Studio Community 2015.....	340
Running samples.....	346
Processing.org.....	347
Importing Blender graphics.....	347
Processing and network interfaces.....	349
Electronics.....	350
Drawing circuits and breadboards – Fritzing.....	350
Using a logic analyzer.....	352
Building a prototyping environment for the Pi Zero.....	353
Overcoming trepidation of using Integrated Circuits.....	354
Logic Level Shifting.....	355
Transistors as switches.....	355

Migrating code from the Arduino.....	357
Arduino I2C – Wire class.....	357
Wire.begin.....	357
Wire.beginTransmission.....	357
Wire.write.....	358
Wire.endTransmission.....	358
Wire.read.....	358
Wire.requestFrom.....	358
Scheduling applications.....	358
Setting up Linux on Virtual Box on Windows.....	358
Web programming.....	364
Browser security.....	365
jQuery – JavaScript framework.....	365
Using jQuery.....	365
Making REST requests using jQuery.....	366
Reference Information.....	366
Recommended software add-ons.....	366
Magazines.....	367
Forums.....	367
You Tube.....	367
Chat rooms.....	368
WiringPi Reference information.....	368
gpio command.....	368
gpio clock.....	368
gpio mode.....	369
gpio pwm.....	369
gpio pwmc.....	370
gpio pwmr.....	370
gpio pwm-bal.....	370
gpio pwm-ms.....	370
gpio read.....	370
gpio readall.....	371
gpio wfi.....	371
gpio write.....	371
delay.....	372
delayMicroseconds.....	372
digitalRead.....	372
digitalWrite.....	372
digitalWriteByte.....	373
micros.....	373
millis.....	373
physPinToGpio.....	373

piBoardRev.....	373
piHiPri.....	373
piThreadCreate.....	374
piLock.....	374
pinMode.....	375
piUnlock.....	375
pullUpDnControl.....	375
pwmSetClock.....	376
pwmSetMode.....	376
pwmSetRange.....	377
pwmWrite.....	377
serialClose.....	377
serialDataAvail.....	378
serialFlush.....	378
serialGetchar.....	378
serialOpen.....	378
serialPutchar.....	378
serialPuts.....	378
serialPrintf.....	379
setPadDrive.....	379
shiftIn.....	379
shiftOut.....	379
softPwmCreate.....	379
softPwmWrite.....	379
softToneCreate.....	380
softToneWrite.....	380
wiringPiI2CRead.....	380
wiringPiI2CReadReg8.....	380
wiringPiI2CReadReg16.....	381
wiringPiI2CSetup.....	381
wiringPiI2CWrite.....	381
wiringPiI2CWriteReg8.....	381
wiringPiI2CWriteReg16.....	382
wiringPiISR.....	382
wiringPiSetup.....	382
wiringPiSetupGpio.....	383
wiringPiSetupPhys.....	383
wiringPiSetupSys.....	383
wiringPiSPIDataRW.....	383
wiringPiSPISetup.....	383
wpiPinToGpio.....	383
wiringPiDev – LCD.....	384
lcdInit.....	384
lcdHome.....	384

LcdClear.....	384
LcdPosition.....	384
LcdPutchar.....	385
LcdPuts.....	385
LcdPrintf.....	385
WiringPi Mapping from Arduino.....	385
Python RPi.GPIO reference.....	385
Python SPI reference.....	387
spi.openSPI.....	387
spi.transfer.....	387
spi.closeSPI.....	387
cURL API programming.....	388
Sample CURL application.....	388
Adafruit GFX library.....	389
Interacting directly with the BCM2835 ARM Peripherals.....	390
Auxiliary Peripherals.....	392
AUXIRQ – Check interrupts.....	393
AUX_ENABLES – Enable the modules Mini UART, SPI1 and SPI2.....	393
AUX_MUI_IO_REG – Read and write data.....	393
AUX_MU_IER_REG – Enable interrupts.....	394
AUX_MUIRR_REG – Interrupt status.....	394
AUX_MU_LCR_REG.....	394
AUX_MU_MCR_REG – Modem control.....	394
AUX_MU_LSR_REG – Data status.....	395
AUX_MU_MSR_REG – Modem status.....	395
AUX_MU_CNTL_REG – Extra controls.....	395
AUX_MU_STATE_REG – Extra status bits.....	396
AUX_MU_BAUD_REG – Baudrate counter.....	396
BCM2835 GPIO.....	397
Register addresses.....	398
GPIO function select setting.....	399
GPIO set high level – GPSET0 and GPSET1.....	401
GPIO set low level – GPCLR0 and GPCLR1.....	401
GPIO query levels – GPLEV0 and GPLEV1.....	401
GPIO event detection – GPEDS0 and GPEDS1.....	401
GPIO rising edge detection – GPREN0 and GPREN1.....	401
GPIO rising edge detection – GPFEN0 and GPFEN1.....	401
GPIO high detection – GPHEN0 and GPHEN1.....	402
GPIO low detection – GPLEN0 and GPLEN1.....	402
GPIO Pullup/Pulldown – GPPUD.....	402
BCM2835 UART.....	402
DR – Data Register.....	403

RSRECR – Receive status / error clear Register.....	403
FR – Flag Register.....	404
IBRD – Integer baud rate divisor.....	404
FBRD – Fractional baud rate divisor.....	404
LCRH – Line control register.....	405
CR – Control register.....	405
IFLS – Interrupt FIFO level select register.....	406
IMSC – Interrupt mask set / clear register.....	406
RIS – Raw interrupt status register.....	406
MIS – Masked interrupt status register.....	407
ICR – Interrupt clear register.....	407
BCM2835 PWM.....	407
CTL – PWM Control.....	408
Interacting using mmap().....	409
Common C language includes.....	412
Missing data types.....	412
String functions.....	412
Math functions.....	412
Cheat Sheets.....	412
Wiring Pi Cheat Sheet.....	413
Pi4J Cheat Sheet.....	414
Linux cheat commands.....	415
Compressed/archived files.....	415
Reference documents.....	415
Credits.....	415
Research Areas.....	415

## Introduction – The PI Book

Howdy Folks,

I've been working in the software business for over 30 years but until recently, hadn't been playing directly with Micro Processors. When I bought a Raspberry Pi and then an Arduino, I'm afraid I got hooked. In my house I am surrounded by computers of all shapes, sizes and capacities ... any one of them with orders of magnitude more power than any of these small devices ... however, I still found myself fascinated.

As I studied the devices, I started to make notes and my pages of notes continued to grow and grow.

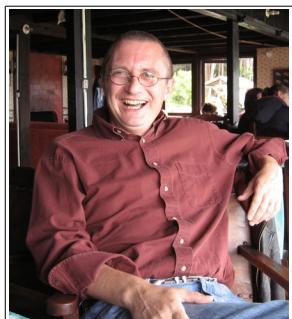
This book is my collated and polished version of those notes. Rather than keep them to myself, I offer them to all of us in the Raspberry Pi community in the hope that they will be of some value. My plan is to continue to update this work as we all learn more and share what we find in the community forums. As such, I will re-release the work at regular intervals so please check back at the book's home page for the latest. I do not plan on ever calling this book "finished" as there will always be more to add. Because of the breadth and depth of material that we want to cover, it is unlikely that you will want to read the book linearly from cover to cover. Instead, I recommend that you familiarize yourself with the table of contents and come back to it as and when you need information.

As you read, make sure that you fully understand that there are undoubtedly inaccuracies, errors in my understanding and errors in my writing. Only by feedback and time will we be able to correct those. Please forgive the grammatical errors and spelling mistakes that my spell checker hasn't caught.

The home page for the book is:

<https://leanpub.com/pi>

Please don't email me directly with technical questions. Instead, let us use the forum and ask and answer the questions as a great community of Raspberry Pi minded enthusiasts, hobbyists and professionals.



**Neil Kolban**  
**Texas, USA**

# What is a Raspberry Pi

Put simply, the Raspberry Pi is a cheap computer that exposes pins for physical computing. This means that you can attach a wide variety of electronic components to it and drive them from software. The Pi primarily runs a version of the Linux operating system called Raspbian but other operating systems (or even none at all) can also be used to host software applications that you write. Designed and distributed as a cheap device for the purpose of educating folks in the skills of computer programming, it is without question that the Pi is more than powerful enough for a wide variety of projects.

## Pi Product Models

Like any product in the market place, one should assume that the Pi devices will evolve. Over the years there have been a number of Pi models released. These models have changed their specifications on number of CPU cores, RAM sizes, number of USB ports and headers exposed. The following is a quick summary of the different models and how they have changed over time:

Model	SoC	Speed	RAM	USB	GPIO
Pi 1 Model A	BCM2835 / ARMv6	700MHz	256MB	1	26
Pi 1 Model B	BCM2835 / ARMv6	700MHz	512MB	2	26
Pi 1 Model A+	BCM2835 / ARMv6	700MHz	256MB	1	40
Pi 1 Model B+	BCM2835 / ARMv6	700Mhz	512MB	4	40
Pi 2 Model B	BCM2836 / ARMv7	4 x 900MHz	1GB	4	40
Pi Zero	BCM2835 / ARMv6	1GHz	512MB	1	40

The Pi 1 and Pi Zero use the ARM v6 processor while the Pi 2 uses the ARM v7 processor.

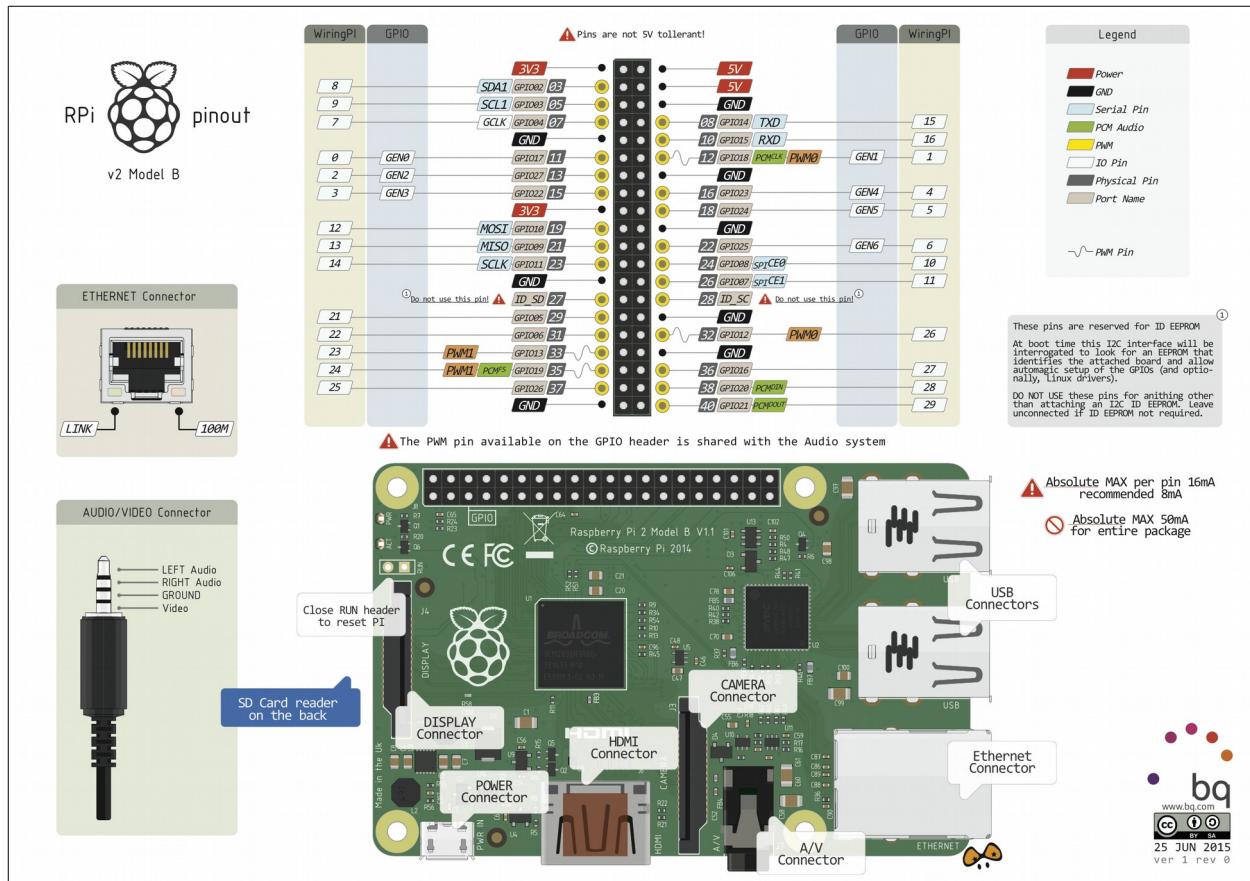
In this book we will focus on the Pi 2 Model B and the Pi Zero. Here is a picture of the Pi 2:



and here is a similar one of the Pi Zero:



Both of these have headers (the Pi 2 with them pre-soldered but for the Pi Zero you have to add them yourself). Onto these headers you can attach a variety of electronics components. On a header are a set of pins to which you can attach your components. In order to do this correctly, you will eventually learn that different pins have different meanings and functions. You will never have to memorize these as there are many references to what pin does what. A fantastic source of pin-outs can be found at [www.pighixxx.com](http://www.pighixxx.com). Here is an example of the Pi 2 Model B which has the same pin configuration as the Pi Zero:



## Powering the Pi

The Pi needs at least a 5V input source capable of delivering 700mA or better. This can come from a USB output or an external battery pack or a wall connected power adapter. For at least the Pi Zero, you can power the Pi Zero by applying a suitable source via the GPIO pins GND and +5V. The power input is physically a micro USB socket.

## Power Consumption

When thinking about embedded devices, power consumption becomes a key consideration. When idle, the following are the believed norms:

Model	Idle consumption	Busy consumption
Pi Zero	~90mA	~150mA
Pi2 Model B	~230mA	~300mA

If we look at the average battery capacities, we can see the approximate lifetime of batteries, we see that:

Type	Capacity	Pi Zero idle	Pi Zero busy
4x AA	2400mAh	26 hours	15 hours
4 x AAA	1000mAh	11 hours	6 hours
4 x C	6000mAh	90 hours	40 hours
4 x D	13000	144 hours	86 hours

Of course, these are approximates and also don't factor in additional current draws such as any devices you may have attached to your Pi.

See also:

- [Raspberry Pi Power Measurements](#)

## Pi Operating Systems

Since the Pi contains a microprocessor, it will thus run some software. The Pi is powerful enough that it is able to run operating systems. An operating system is the system software that provides the environment for user applications to run. There are a number of operating systems available for the Pi with the most common one being called Raspbian. Raspbian is a derivative of the Debian Linux distribution. At the time of writing, the latest version of Raspbian is based on Debian "Jessie" and has a Linux 4.1 kernel. The previous version was called "Wheezy" and had a Linux kernel at 3.18.

A list of alternative installable images can be found here:

<https://www.raspberrypi.org/downloads>

These include:

- Ubuntu Mate

- Snappy Ubuntu Core
- Windows 10 IOT Core

The Pi has no hard-disk as you might find a desk-side or laptop computer but instead uses micro SD cards. These are the storage devices used in cell phones and cameras. An operating system installation image can be downloaded from the Internet and written to a micro SD card using various software packages including the Windows tool called win32diskimager.

## Pi Zero

The Pi Zero is the latest member of the Pi family. It was announced and released at the end of 2015 and took everyone by surprise. Functionally, it is at the low end of the Pi spectrum but what makes it truly special is the price of a mere \$5. This immediately became a game changer as it undercut almost every competitor.



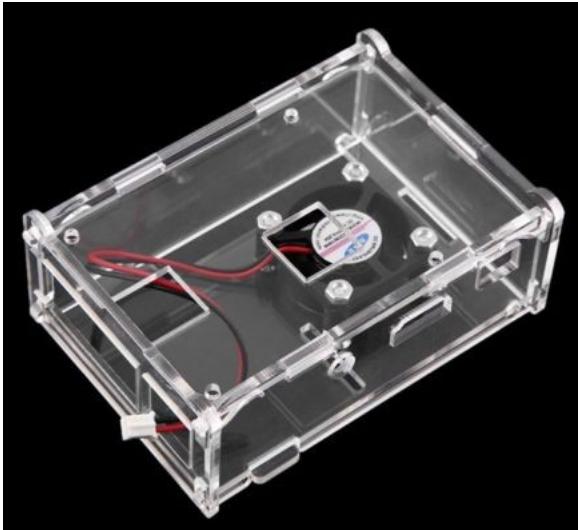
At the time of writing availability of units is very scarce and there is very little empirical evidence upon it. Where will it fit? Does it become an embedded system like the Arduino or is it too power hungry? It does have video, USB, speed, memory and SD storage. As such, it has changed the game in terms of dimensions of applicability.

When you obtain Pi Zeros, also make sure you have the correct adapters and connectors. For example, you may want a converter from the micro HDMI output to the "normal" HDMI output. I also found it very useful to get micro USB converters to female full size USB connectors:



## Box housings for the Pi

When you get a Pi, the chances are that it will come as a simple single circuit board. Obviously you can use it just like this but I recommend buying a box for it. You can get a variety of well formed plastic boxes on the Internet at very low prices (\$3-\$5). The box may prevent a variety of accidental mishaps ... not least of which is placing your Pi on top of some additional circuits you may be working with which will result in inadvertent shorts that may ruin the Pi.



Each board contains 4 mounting holes that can be used to secure it to a box or platform. The diameter of the bolt holes is metric M2.5 (2.5mm). These bolts are not the easiest things to find. However suppliers on eBay will sell 100pcs M2.5 nuts for \$3.50 and 100pcs M2.5 x 10mm bolts for \$4.00. This is likely to be more items than you will ever need but are still likely to be no more expensive than equivalent parts in ones or twos from a high-end DIY store and you will still have enough left over for a couple of dozen friends and colleagues. I recommend the nylon/plastic type.

## Powering off the Pi

It is not uncommon to find instructions on how to start or boot an electronics device but here we have to provide instructions on how to power **off** the Pi. If you are running Raspbian on your Pi then you must assume that at any arbitrary point in time it is writing to the micro SD storage device or performing some other background activity. If you simply switch off the Pi by removing its input source (i.e. unplugging it) then you run a risk. You simply don't know what the Pi might be doing at the moment it is unplugged. It could, for example, be in the middle of re-writing some system file and be exactly half-way through that task. Taking the power away from the system at that point would result in a corrupted file which could quite easily prevent the Pi from re-booting. Obviously not a good situation to end up in. The solution to this is to cleanly shutdown the Pi before powering it off. The command "shutdown -h now" will cleanly shutdown the Pi. When it is complete, you can cleanly remove the power. This command can be run from the shell. However, if you are running "headless" then you might not have the ability to enter commands. One solution to this is to write a demon application which watches for an external signal such as a change in state of a specific GPIO pin. When the signal is detected, the demon can then execute the shutdown command and we will have achieved our task.

See also:

- [Adding an On/Off switch to your Raspberry Pi](#)

# Architecture

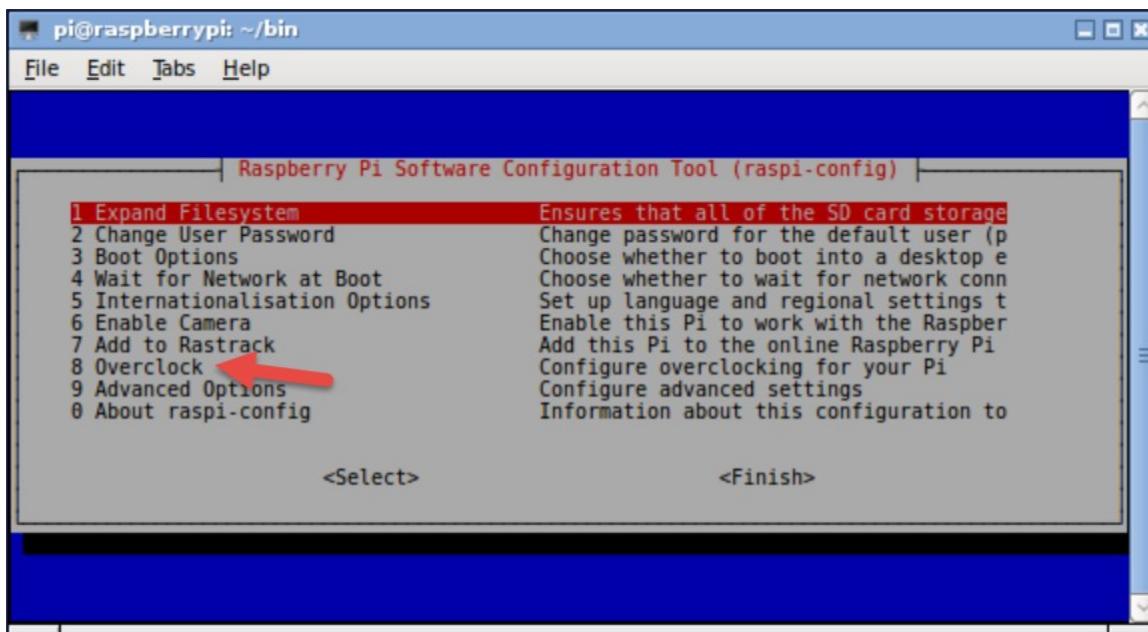
The Pi is composed of a variety of sub-systems that, when combined, provide what we know of as the Raspberry Pi.

## The ARM CPU

The heart of any computer is the Central Processing Unit (CPU). Within the Pi, the CPU is the ARMv6 or the ARMv7 (Pi 2). Take note that the ARMv6 is a model of the ARM11 architecture while the ARMv7 is a model of the ARM Cortex-A7 architecture. This naming can become confusing.

The clock speed on the Pi has a default value but can be overclocked to make it run faster. Before you merrily increase the speed of your Pi "just because you can", pause a moment. If it were just fine for the Pi to run faster than its default speed, wouldn't the vendors already have enabled that higher speed? After all, there is no merit in artificially slowing a processor. The reason that the processor doesn't run at the higher speed is that there is a trade-off between execution speed and life expectancy of the device. If you over-clock it, you are running it faster than it was intended to run under normal operations. My suggestion to you is to remain at the default clock speed unless you absolutely must go faster ... and even then, consult the latest forum posts and other sources of knowledge to read about the negative effects running at the faster speed may cause.

The clock speed on the Pi can be changed in the system configuration tool called "raspi-config":



See also:

- Wikipedia – [ARM architecture](#)
- Wikipedia – [ARM11](#)
- Wikipedia – [ARM Cortex-A7](#)

## Broadcom System On a Chip

The Broadcom BCM2708 is a family of system on a chip devices of which we care about the BCM2835 and BCM2836. These chips provide hardware support for timers, I/O controllers, GPIO, USB, PCM, DMA, I2C, SPI, PWM and UART (more depth on all of these concepts later).

The core reference document for the device is the PDF document called "Broadcom – BCM2835 ARM Peripherals".

See also:

- Error: Reference source not found

## Video subsystem

The Broadcom device also contains a Graphical Processing Unit (GPU) which is capable of producing an HDMI output signal that can be plugged into HDMI compliant monitors and TVs.

## Storage subsystem

The storage subsystem is a FAT32 file system found on SD media. It is the BCM283x chip that contains the logic to interact with the SD media. It is suggested that a fast card be used such as a Class 10 device. Class 10 devices can be identified with a mark on the card that looks as follows:



Realize that SD micro cards are tiny and aren't easily labeled. Consider adding a README file to their root so that you know their history.

The Win32 Disk Imager tool can be used to both copy an image to the card but also can be used to read an image from the card to a single file backup file. If you are concerned about losing work, backup your cards regularly.

See also:

- [Win32 Disk Imager](#)
- Wikipedia – [Secure Digital](#)

## Installation

When you first obtain a Pi, chances are high that it won't come with any pre-supplied software. As such your first task will be to select an operating system and install it on a micro SD card. I suggest working with Raspbian.

## Installing Raspbian

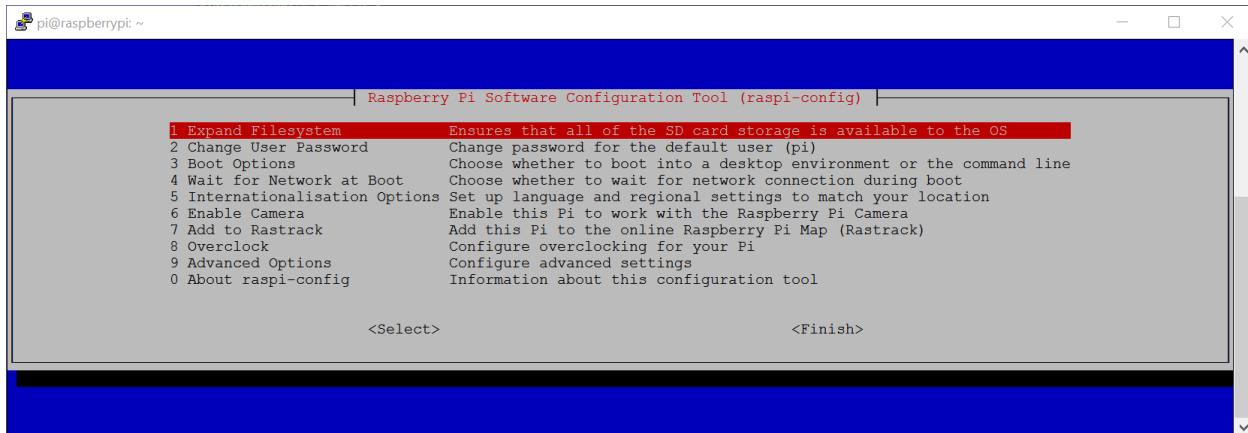
I don't think there has ever been a book written on the Pi that doesn't include the obligatory story on how to install Raspbian and this book will be no exception.

1. Download the Raspbian binary from <https://www.raspberrypi.org/downloads/raspbian/>. At the time of writing, this is Raspbian Jessie which is a version of the Debian v8 Linux distribution. The download size is about 1.3GBytes. The file is a ZIP archive and we must extract the content which will be an ".img" file. This is the image that will be written to the SD card.
2. Download and install Win32DiskImager from <http://sourceforge.net/projects/win32diskimager/>
3. Run Win32DiskImage to copy the Pi OS image to the SD card. This will take a while to run.
4. Insert the SD card in the SD card reader in the Pi and power the Pi on.

After installation of the OS, you are on your own. You are now running a Linux environment with all the pros and cons associated with that. The default userid is called "pi" with a password of "raspberry".

One of the earliest things you will want to do is run the Raspbian configuration tool called `raspi-config`:

```
sudo raspi-config
```



It has options that you will want including:

- **Expand Filesystem** – Expand the file system to use all the available space on the SD card. By default, the image transferred to the SD card exposes only the default image size. If you have an 4GB or 8GB card, the Pi will not see it by default. Expanding the file system causes the complete capacity of the card to be available to the operating system.
- **Boot Options** – Choose whether to boot into a graphics system or console login.
- **Internationalization Options** – Change the language. The default is British English (`en_GB`).

See also:

- [Installing operating system images](#)

## Installation tools

When we think of a Linux operating system, we will find that the state of a system is governed by the existence and versions of files on the file system. If we consider a "package" to be a collection of files necessary to perform some discrete operation, then we can also think of the files on the Linux file system as being members of some package or another (with the obvious exception of application files that you create).

If we wish to install a package, we must obtain a copy of all the files associated with that package and place them in the correct directories where they are expected to be found. If we were attempt to do this by hand, we would end up in all sorts of troubles. Not least of which would be the potentially of making mistakes with manual copies. In addition, some packages are not self contained but instead have prerequisites which are themselves packages. What we need is some registry mechanism where we can learn what packages are installed and a repository where packages can be found and downloaded. This is exactly what the apt technology does for us.

The apt technology is a suite of related applications which coordinate together to perform package management functions. The primary configuration file for apt is the file `/etc/apt/apt.conf`.

To begin with, the apt system needs a list of sources from which packages can be found. This is specified in a file called `/etc/apt/sources.list`. It is from this list of sources that new packages and updates to existing packages can be found. The current repository of Pi packages is from the raspbian.org site. A typical entry in the sources.list file looks like:

```
deb http://mirrordirector.raspbian.org/raspbian/ jessie main contrib non-free rpi
```

Note that the URL for the repository shown above is a mirror re-director which will rewrite the look-up to a "closer" mirror server of the repository.

It is important to note that a package that is downloaded and installed from the Raspbian repository may **not** be the very latest version of that package available. In fact, experience is showing that some packages downloaded from the repository are very old indeed. Let us look at why and how this happens. The Raspbian repository contains the installable images for Raspbian build executables. An executable is built from the source files that comprise that project. These source files are typically open source and found in a variety of locations including Github and Linux. When an author of a package commits a change to their package this does **not** automatically cause a recompilation of their source or a refresh within the Raspbian repository. Instead, it is up to the administrators of the repository to compile and publish and this appears to be performed manually.

Next we will look at some of the primary commands for working with packages. The most important of these is `apt-get`. We will be using `apt-get` extensively to install new packages that aren't present in a default distribution.

See also:

- [apt-rpm.org](http://apt-rpm.org)

### apt-cache

List available packages.

```
$ apt-cache pkgnames
```

Show statistics on the cache. Primarily what we see here is disk space consumed by the cache as well as counts of the known packages.

```
$ apt-cache stats
Total package names: 65836 (1,317 k)
Total package structures: 110459 (6,186 k)
  Normal packages: 76148
  Pure virtual packages: 1130
  Single virtual packages: 9613
  Mixed virtual packages: 2015
  Missing: 21553
Total distinct versions: 79818 (5,747 k)
Total distinct descriptions: 101300 (2,431 k)
Total dependencies: 611151 (17.1 M)
Total ver/file relations: 82964 (1,991 k)
Total Desc/File relations: 101300 (2,431 k)
Total Provides mappings: 17284 (346 k)
Total globbed strings: 172 (2,231 )
Total dependency version space: 2,928 k
Total slack space: 48.6 k
Total space accounted for: 31.4 M
```

See also:

- man(8) – [apt-cache](#)

## apt-file

The apt-file tool allows one to find which package might contain a specific file. Note that `apt-file` needs to be installed with the following command:

```
$ sudo apt-get install apt-file
```

To find which package contains a file:

```
$ apt-file search <filename>
```

To list files contained in a package:

```
$ apt-file list <packagename>
```

## apt-get

The `apt-get` command is a command line interface to the Advanced Packaging Tool (APT) library. It is by far the most commonly used of the package management tools. To install a package we can run:

```
$ sudo apt-get install <pkgnname>
```

To un-install a package run:

```
$ sudo apt-get purge <pkgnname>
```

To download the package without installing it run:

```
$ apt-get download <pkgname>
```

The downloaded package will end in ".deb". We can then install that package using"

```
$ sudo dpkg --install <fileName.deb>
```

See also:

- [25 Useful Basic Commands of APT-GET and APT-CACHE for Package Management](#)

## **apt**

To see which packages are installed, run

```
$ apt --installed list
```

## **dpkg**

The dpkg command is the base command for package installation. If we have a local ".deb" file which contains a package, we can install it using:

```
$ sudo dpkg -i <filename.deb>
```

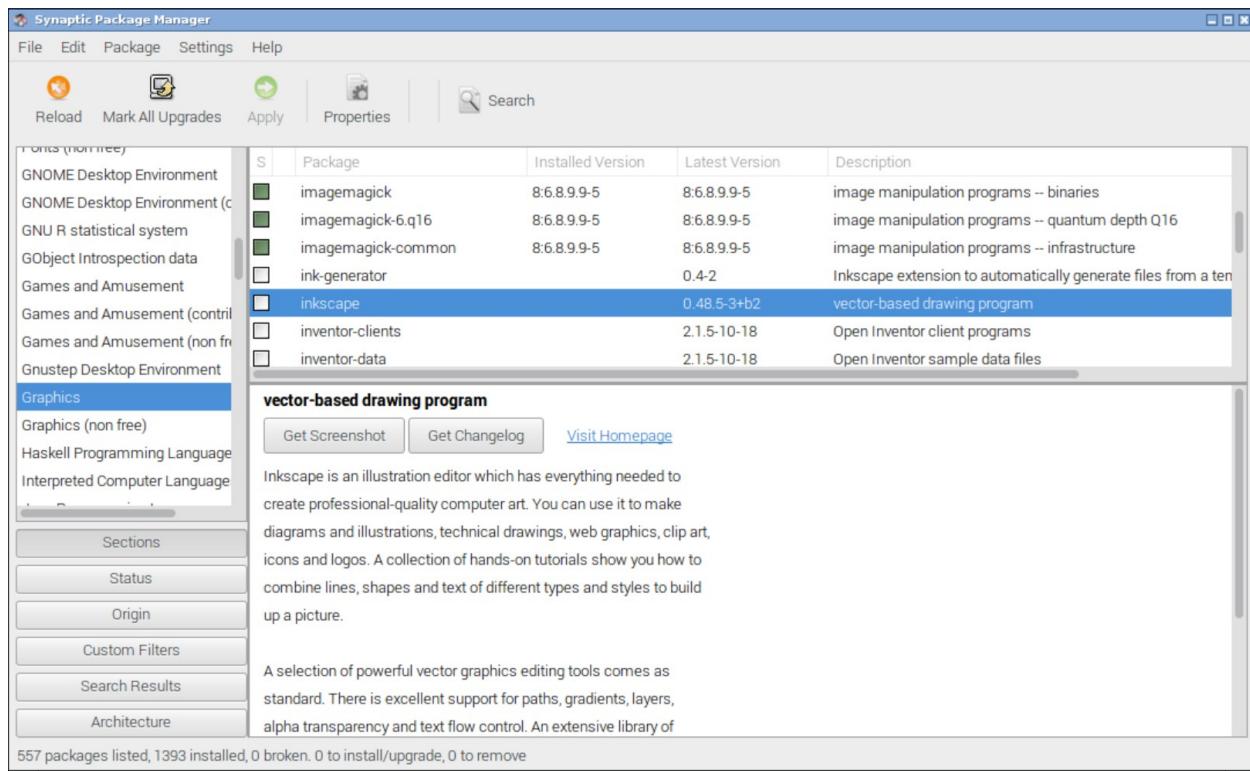
## **synaptic - GUI package management**

A GUI package management interface is available called synaptic. This isn't supplied by default and has to be explicitly installed:

```
$ sudo apt-get install synaptic
```

The tool should be run as root in order to be able to install packages:

```
$ sudo synaptic
```



See also:

- [Synaptic package manager](#)

## Updating

After having installed Raspbian, you will wish to update it to the latest maintenance level. You will also want to perform this task periodically to ensure you have the latest.

To determine the level of the kernel installed, run:

```
$ uname -a
Linux raspberrypi 4.1.13-v7+ #826 SMP PREEMPT Fri Nov 13 20:19:03 GMT 2015 armv7l
GNU/Linux
```

The steps involved are:

```
sudo apt-get update
sudo apt-get upgrade
```

## The Raspbian Environment

This book will not teach you the Linux operating system. There are tons of excellent learning and reference materials on that subject by itself. In fact if a single book were to attempt to cover all aspects of Linux, it would itself be thousands of pages in length. Instead, we will briefly touch upon Linux related

topics only when they are distinct or different for the Pi or else I consider them sufficiently important and summarizable to the Pi concept under discussion.

See also:

- [Raspbian home page](#)

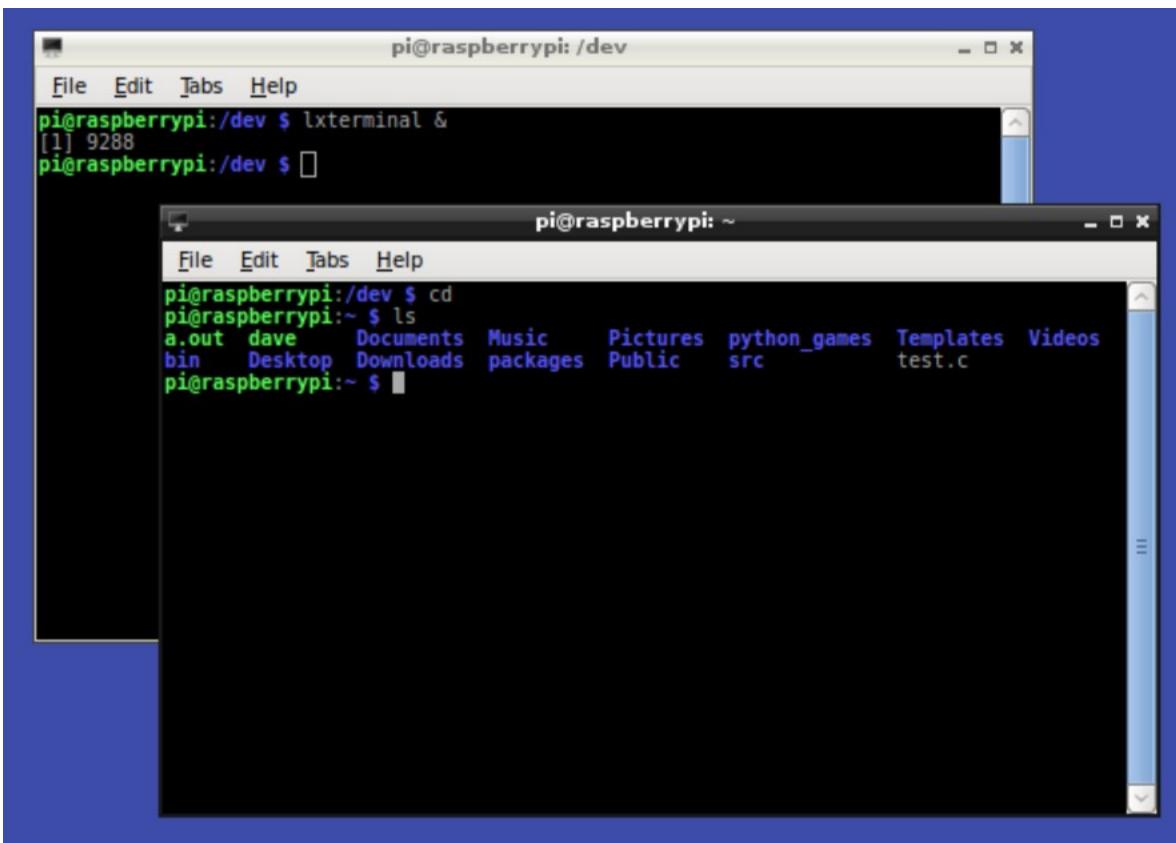
## The shell

The shell is the application that most Linux users see and with which they interact. It is a command line interpreter that prompts the user for input. Normally there is a "\$" prompt waiting for input. When you enter a command at the shell, the shell interprets the command and performs the action requested. There are two primary categories of action. The first is what we call a shell built-in function. This is where when you enter a command, it is the shell's internal implementation (code) that performs some work. However, our core focus is not on shell built-ins ... instead it is on starting executables. By and large, when you enter a command at the shell, the shell looks for a file with the same name within the filesystem and executes the program corresponding to that file. The shell does this by looking in each of the directories specified in the PATH environment variable until it either finds the first one or doesn't find in any in each of the directories. In our story, we will assume that the files are compiled programs. As such, when you enter a command (eg. "xyz"), the shell looks for a file called "xyz" and asks Linux to load and run it. One can also specify an absolute path to a file and that specific file will be run. One particularly useful technique is to prefix the command with "./" which is short-hand for the current directory. This will attempt to launch the command from a file found in the current directory. For example:

```
$ ./xyz
```

will look for a file called "xyz" in the current directory and try and launch it. When we say that we are launching a new executable, what we are actually saying is that we are creating a new instance of a process. Since Linux is a multi tasking operating system, we can run multiple applications simultaneously and the operating system will context switch between them as it needs. Each individual instance of a running program is termed a process.

If you are sitting in front of a Pi, the chances are you are looking at its graphical user interface which is based on X-Windows. Here you can create additional instances of the shell each in its own distinct window. You can start these from the launch bar or by entering "lxterminal &" at an existing shell prompt.



If you are sitting at a PC, then you can remotely access your Pi (assuming it is network connected) by running an "ssh" program. This will start a new instance of the shell on the Pi but redirect the input and output to your local PC. Again, you can open as many shells as you desire. A recommended application that provides "ssh" that is available on most PC platforms is "PuTTY".

Since the shell is where we are going to be entering the vast majority of our Raspbian commands let us look at some ways in which we can use it better. First of all are the cursor keys. Using the up and down cursor allows us to scroll back and forwards through our command history. Using left and right cursor keys allows us to move left and right through the current command where we can modify it before pressing enter to execute it. To see the history of commands that we have executed, enter "history".

```
$ history
2000 cd /var/log
2001 ls
2002 vi syslog
2003 grep -i leanPub *
2004 vi daemon.log
2005 ls
2006 tail kern.log
2007 ls
2008 tail btmp
2009 tail messages
2010 cd
2011 cd bin
```

```
2012  ls
2013  ./mountall.sh
2014  ./startnfs.sh
```

It will return the history of your last set of commands. Notice the number to the left. We can recall that command by entering "`!<Number>`". For frequently used commands, such as "history" (well ... very frequently used by me) you can create a shortened invocation of it. For example:

```
$ alias h=history
```

Now when I enter "h" at the command line, the shell substitutes that for the "history" command.

See also:

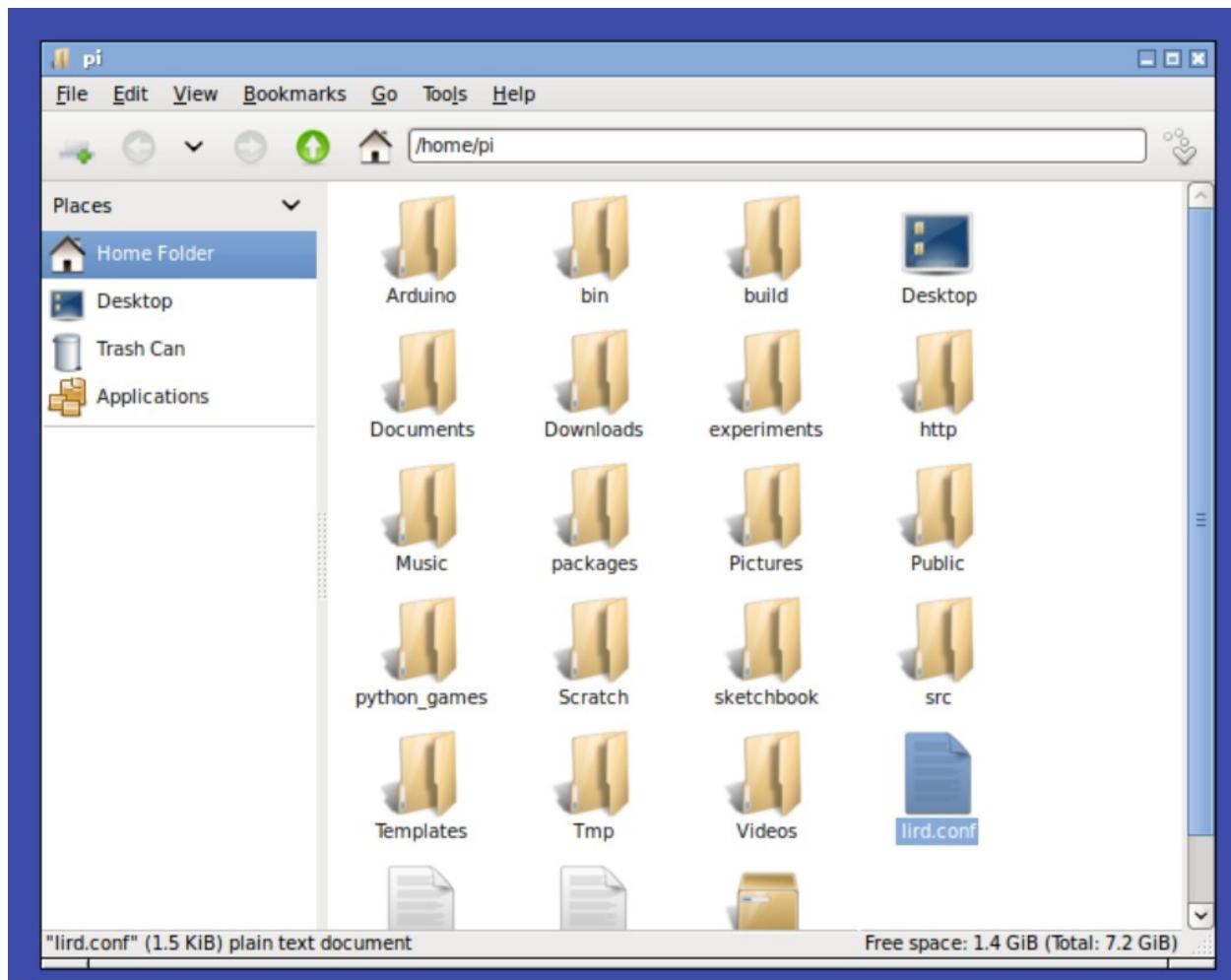
- [PuTTY](#)
- X-Windows

## The file system

The Raspbian file system is a standard Linux file system hosted on a micro SD card. This is a hierarchical file-system comprised of files and directories. It supports all the normal Linux commands for file manipulation:

- cp – copy files
- mv – move files
- rm – remove files
- ls – list files
- ln – link files
- mkdir – make a directory
- rmdir – remove a directory
- chmod – change file and directory permissions
- mount – mount a file system for use.
- umount – un-mount a previously mounted file system.

For those who don't want to navigate through the file system from the shell, there is an attractive GUI navigator supplied by the tool called PCManFM.



This can be started from the command line tool "pcmanfm".

See also:

- [man\(1\) – ln](#)
- [PCManFM – File manager](#)

## Mounted file systems

Raspbian supports the standard Unix "mount" command for mounting file systems onto the file structure. An especially useful command is called "findmnt" that displays a visualization of what is mounted and where.

See also:

- [man\(8\) - findmnt](#)

## File permissions

On a Linux OS, files and directories have permissions associated with them. By and large there are three permissions a file can have which are "read", "write" and "execute". These declare whether a file can be read, written to or be executed. A file also has the concept of an owner and a group. When we talk about permissions, we further categorize the permissions into those that are allowed for the owner, those that are allowed for members of the group associated with the file and those that are allowed for everyone else (i.e. neither the owner nor a member of the group).

We thus end up with nine specific permissions:

- owner – read / write / execute
- group member – read / write / execute
- others – read / write / execute

An encoding scheme has been used for visualizing and describing permissions for files using the octal numbering system (base 8).

If we imagine that the three permissions were described in 3 bit binary with an order of read / write / execute then we would have permissions of "rwx" ... or [1/0][1/0][1/0]. So in binary, permissions of 101 would say we have permission to read and execute but not write. If we translate this binary into octal, it would be written as  $4+0+1 = 5$ . Thus we can represent a set of read / write / execute permissions as a single octal digit (0-7). Combining this with the notion that we have three sets of permissions (user, group and other), we can completely describe all the permissions on a file with three octal digits. For example 755 would be a value of 7 for user (read / write / execute) and 5 for both group and other (read / execute).

The Linux command called `chmod` can be used to set the permissions. For example:

```
chmod 755 myFile
```

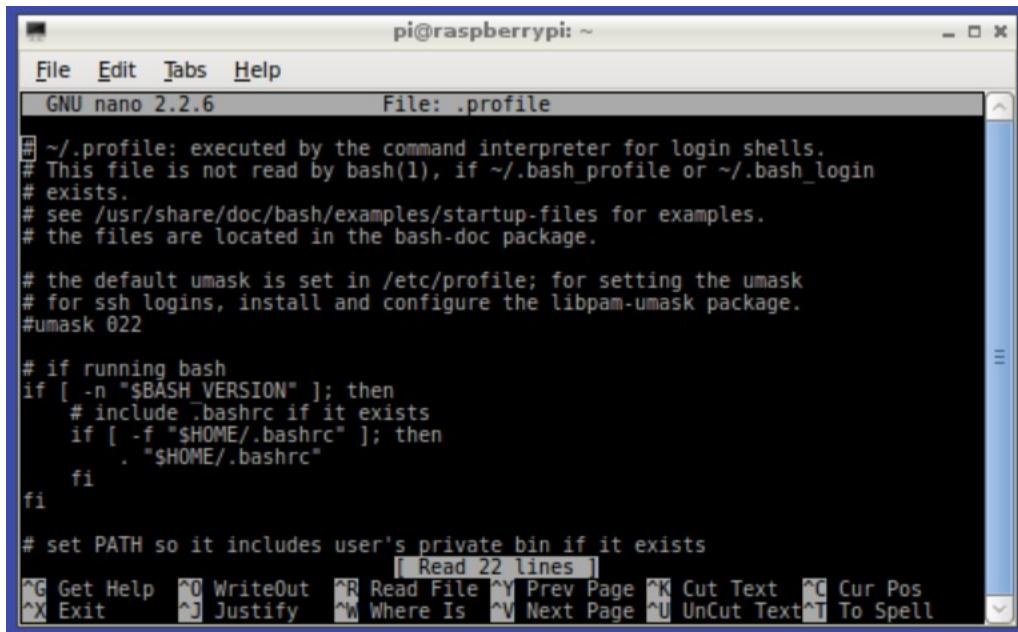
Does this sound complicated? Yes it does ... however it is how Unix has been handled since its inception. There are other mechanisms should you not wish to remember these encodings and those can be read about in the man pages for `chmod`.

## Editors

When developing, it is hard to imagine that you won't be using an editor to examine and change the content of files. On Raspbian there are a wide variety of editors available to you including `vi`, `nano`, `geany`, `gedit`, `leafpad` and more.

The `vi` editor is the historic text based editor that has been around for many decades. Some users love it, and some hate it. It is definitely **not** for the Unix beginner. Its commands are cryptic. If you know `vi` today, awesome ... if you don't know `vi`, then I would not recommend it as the editor to learn. Its strength today is predominantly that it is ubiquitous across every known Unix variant.

The `nano` editor is another text editor but this is significantly easier to use than `vi`. The manual page and/or the editor documentation should be consulted for details ... however the basic model is use the cursor keys and enter/change/delete text.



The screenshot shows a window titled "pi@raspberrypi: ~" containing the contents of the ".profile" file. The file is a Bourne shell script with comments explaining its purpose and how it interacts with other configuration files like .bashrc. It includes logic for determining if it's running bash and setting the umask. The bottom of the window shows a menu bar with "File", "Edit", "Tabs", and "Help". The main area shows the file content, and the bottom right has a status bar with keyboard shortcuts for various functions.

```
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 2.2.6          File: .profile

#!/bin/sh
# ~/.profile: executed by the command interpreter for login shells.
# This file is not read by bash(1), if ~/.bash_profile or ~/.bash_login
# exists.
# see /usr/share/doc/bash/examples/startup-files for examples.
# the files are located in the bash-doc package.

# the default umask is set in /etc/profile; for setting the umask
# for ssh logins, install and configure the libpam-umask package.
#umask 022

# if running bash
if [ -n "$BASH_VERSION" ]; then
    # include .bashrc if it exists
    if [ -f "$HOME/.bashrc" ]; then
        . "$HOME/.bashrc"
    fi
fi

# set PATH so it includes user's private bin if it exists
[ Read 22 lines ]
^C Get Help ^O WriteOut ^R Read File ^Y Prev Page ^K Cut Text ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is ^V Next Page ^U UnCut Text ^T To Spell
```

The geany editor is perfect for editing code in an X-Windows environment. It includes language knowledge as well as build commands. I use geany most of the time for my own editing when editing locally.

The screenshot shows the Geany IDE interface. The main window displays the C source code for `gpsclient.c`. The code includes #include directives for `<stdio.h>` and `<gps.h>`, and a main function that opens a GPS connection, reads data, and prints latitude and longitude. Below the code editor is a terminal-like window showing the output of a make command. The terminal window has tabs for Status, Compiler, and Messages, with the Compiler tab active. It shows that compilation finished successfully and there was nothing to be done for 'all'. The status bar at the bottom indicates the current file is `gpsclient.c`, the line and column are 13 / 26, and the mode is Unix (LF).

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

int main(int argc, char *argv[]) {
    printf("Starting gps client test\n");
    struct gps_data_t gps_data;
    int ret = gps_open("localhost", "2947", &gps_data);
    printf("ret from gps_open=%d\n", ret);
    if (ret != 0)
        return 0;
}
gps_stream(&gps_data, WATCH_ENABLE | WATCH_JSON, NULL);
while(1) {
    if (gps_waiting(&gps_data, 500) != 0) {
        if (gps_read(&gps_data) != -1) {
            printf("fix: %f, %f\n",
                   gps_data.fix.latitude,
```

make (in directory /home/pi/src/projects/gps)  
Compilation finished successfully.  
make: Nothing to be done for 'all'.

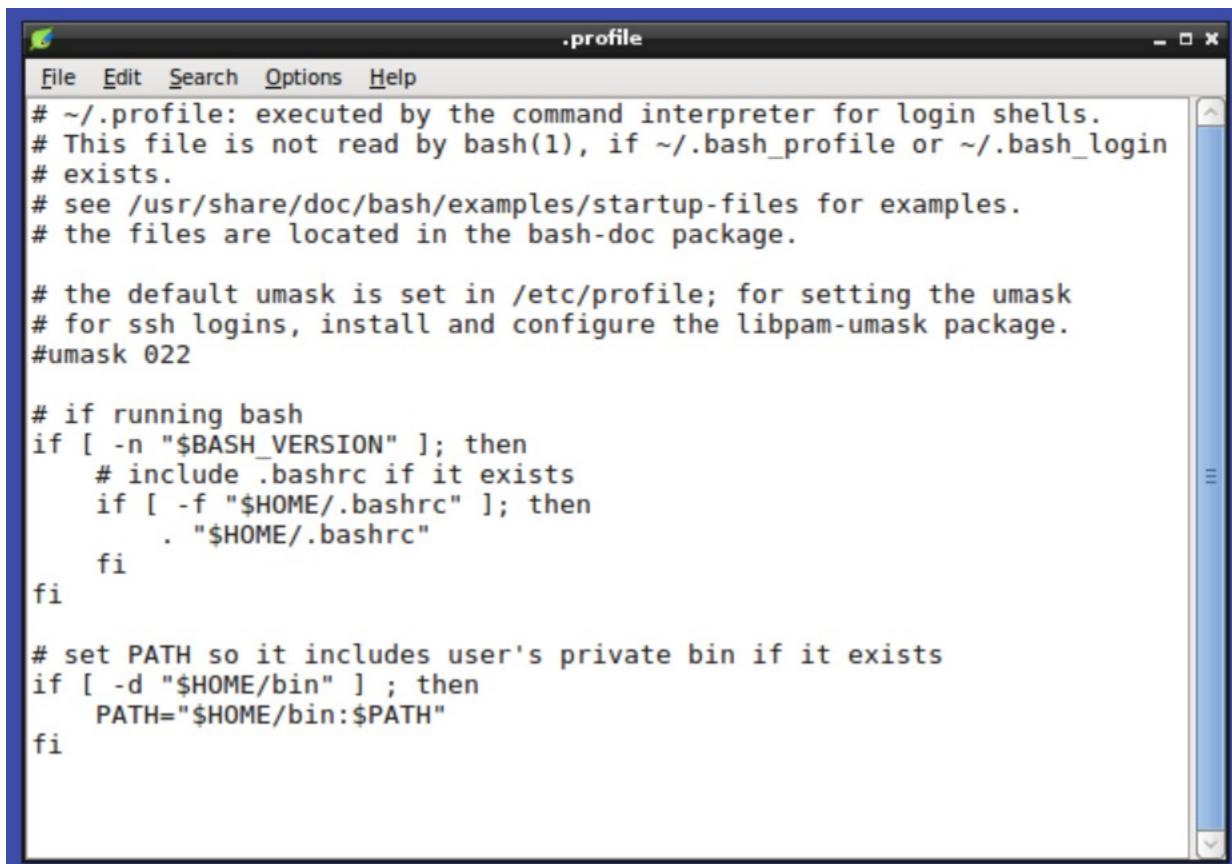
line: 13 / 26 col: 10 sel: 0 INS SP mode: Unix (LF) encoding: UTF-8 filetype: C scope: main

Gedit is another quick to launch general purpose text editor.

The screenshot shows the Gedit text editor window. The title bar indicates the file is `main.c` located in `(~/pi/src/Tests)`. The editor displays a simple C program that includes `<stdio.h>`, defines an external integer `i`, and contains a `main` function that prints "Hello World!" and increments `i`. The status bar at the bottom shows the file is in C mode, tab width is 8, and the current position is Ln 1, Col 1.

```
#include <stdio.h>
extern int i;
char *myStatic = "Yikes!";
int main(int argc, char *argv[]) {
    char *text = "Hello World!";
    printf("%s\n", text);
    i = i + 1;
}
```

Finally, leafpad is another X-Windows based editor which feels like the normal/basic editor one finds on a PC.



The screenshot shows a window titled ".profile" in Leafpad. The menu bar includes File, Edit, Search, Options, and Help. The main text area contains the following shell script code:

```
# ~/.profile: executed by the command interpreter for login shells.
# This file is not read by bash(1), if ~/.bash_profile or ~/.bash_login
# exists.
# see /usr/share/doc/bash/examples/startup-files for examples.
# the files are located in the bash-doc package.

# the default umask is set in /etc/profile; for setting the umask
# for ssh logins, install and configure the libpam-umask package.
#umask 022

# if running bash
if [ -n "$BASH_VERSION" ]; then
    # include .bashrc if it exists
    if [ -f "$HOME/.bashrc" ]; then
        . "$HOME/.bashrc"
    fi
fi

# set PATH so it includes user's private bin if it exists
if [ -d "$HOME/bin" ] ; then
    PATH="$HOME/bin:$PATH"
fi
```

See also:

- [nano editor home page](#)
- Editing sources on the Pi
- Wikipedia: [Leafpad](#)
- Wikipedia: [List of text editors](#)

## Processes

A process is the name we give to a running instance of a program. When we start a program it runs within its own context separate from the context of other programs. The process manages that context and includes the files that program opened, the executable running, the environment variables, the user running the program and more. Each process is unique identified with an integer id call the Process ID or PID. To see which processes are running on our Pi we can issue the "ps" command. This has a ferocious number of options and deep reading of the manual page would be required to gain full appreciation. In its basic form, when we run it it will show the PID and command associated with that pid. For example:

```
$ ps
 PID TTY      TIME CMD
 2508 pts/0    00:00:02 bash
 2622 pts/0    00:00:00 lxsession
 2629 pts/0    00:00:00 dbus-launch
 2647 pts/0    00:00:01 openbox
 2648 pts/0    00:00:00 lxpolkit
 2649 pts/0    00:00:46 lxpanel
 2650 pts/0    00:00:02 pcmanfm
 2722 pts/0    00:00:01 lxterminal
 2723 pts/0    00:00:00 gnome-pty-helpe
 4502 pts/0    00:00:00 dbus-launch
 4823 pts/0    00:01:10 geany
 4824 pts/0    00:00:00 gnome-pty-helpe
 5158 pts/0    00:00:00 ps
```

Notice that we get to see the PID column. Another command available to us is called "pidof". This takes the name of a command and returns us the PIDs for all instances of that command. For example:

```
$ pidof geany
4823
```

Once we know the PID of a running program, we can terminate it by sending it a "kill" signal. These are asynchronously transmitted signals that indicate to the process that some external event has happened. To see the list of possible signals run:

```
$ /bin/kill -L
 1 HUP       2 INT       3 QUIT      4 ILL       5 TRAP      6 ABRT      7 BUS
 8 FPE       9 KILL     10 USR1     11 SEGV     12 USR2     13 PIPE      14 ALRM
15 TERM     16 STKFLT   17 CHLD     18 CONT     19 STOP     20 TSTP      21 TTIN
22 TTOU     23 URG      24 XCPU     25 XFSZ     26 VTALRM   27 PROF      28 WINCH
29 POLL     30 PWR      31 SYS
```

To send a signal we can use one of the following two syntax formats:

```
kill -<number> <PID>
```

or

```
kill -<name> PID
```

For example:

```
kill -1 4823
```

and

```
kill -HUP 4823
```

are identical. Each signal can be trapped by the process that receives it allowing the process to determine how it wishes to handle its arrival. The one exception is (9) KILL ... which is un-trapable and causes the immediate termination of the process.

The way that a new process is created is through a technique called "forking". There is a low level system call named `fork()` that, when executed, causes a brand new instance of the original process to be created. The new instance shares everything from the first one including open files, variable values, environment variables ... everything. The difference between them is that the return code from `fork()` call for the original process is the process id for the new process while the new process receives a value of

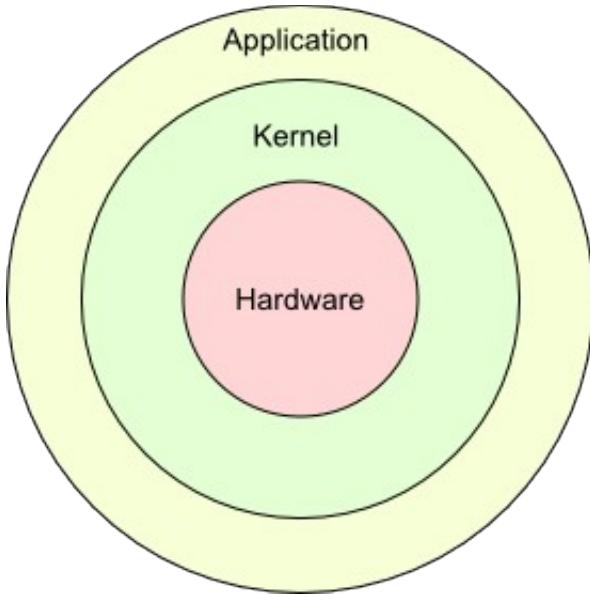
0. In the new forked process, it is then very common for a second system call to be made called `exec()`. What this does is load a new executable into the address space of the existing process (the forked copy) and begin its execution. If this feels a trifle convoluted, I can't argue with that ... however it does provide a mechanism for a newly started process to inherit the environment of the starter which can include aspects such as current userid, group membership, open files and more. And that is a desirable situation.

When we run a new shell command, that also causes a new process to run. The shell starts the process and then waits for it to complete. If we don't wish to wait, we can run it in the background by adding an "&" to the end of the command.

From within a C program, we can also start another process by using the `system()` call. This takes the file system path of an executable and starts it while waiting for it to complete. This is a quick way of performing work by letting sub-programs do complex jobs for us. For example, should we wish to play a sound, we can call `aplay()` to play the sound through the `system()` call.

## **Kernel mode vs User mode**

The Linux operating systems is designed to support both multiple users and multiple processes simultaneously. One of the implications of this is that a badly written or malfunctioning application should not be able to negatively impact the activities of either other users or other processes. The way this is achieved is through the separation of kernel level functions and user code level functions. Each process runs in its own address space with isolation to each of the others. This means that it is difficult for one application to damage the memory of another. However, I/O operations are common between applications. Most have to read and write from disks for example. Somewhere in the operating system there has to be code that interacts at a low level with the disks. If the disks could be accessed from user application code, then that could damage them or compromise security. As such, access to I/O is considered a privileged operation and should only be performed by trusted code. And this is where the Kernel comes into play. Code running on a Linux machine can "run within the Kernel". This means that it is trusted code that runs with the most privileges. It also has full access to all the internal data structures that are used to maintain the operating system as a whole. Kernel code can also run at higher level priorities than other code. In normal Pi programming, you will be unlikely to be writing any new code that will be executing within the Kernel however you will be leveraging many functions that interface with Pi hardware and access to these will be through Pi specific Kernel modules that have been supplied.



## Root privileges and running applications

Linux is a multi-user operating system. This means that multiple distinct users can simultaneously login to a single system and run applications concurrently with each other. It is not desirable for one user to be able to read the files from another and hence there are good permission capabilities associated with each individual user. However, there is a special user in a Linux system that is called "root". This user is also called the "superuser". Root is not constrained by normal security constraints. Root can terminate any process and read/write to any file. As such, root is normally given only to the most trusted system administrators and ordinary users run with their ordinary userids. However, on the Pi, there are times when we wish to access hardware or run powerful commands (anything that affects the system as a whole is usually considered to be under this category). A special command is supplied called "sudo" that takes as a parameter another command to run. That second command is then executed with root permissions. Effectively, it escalates the privileges of the command to root just for the duration of that process.

A question that arises from time to time is that we know that the default userid is "pi" with a password of "raspberry", what then is the default password for "root"? The answer is that there is none and by that we mean that login as root is disabled. If you want to become root, then run:

```
$ sudo -i
```

On occasion there is discussion in the forums on the correctness of having to run some user written Pi applications as root. On a production, multi-user operating system, I wouldn't question that notion but I maintain that the Pi is not that. Since I can't imagine a Pi user not knowing the root password or being allowed to run sudo, the Pi is already compromised from a multi-user perspective. From a system stability point of view, any program that runs as root can potentially destroy the system. As such, take back-ups of your system as needed so that you have a restore point. In my experience, I have yet to make such a serious accidental error that my system has been destroyed.

The sudo command is governed by a "sudoers" file which lists the users that are authorized to elevate their permissions via sudo. To add a user to the set of sudo authorized users, we can run:

```
adduser <username> sudo
```

This will add the user to the "sudo" group and members of that group are authorized to run sudo. After the change, the user must logout and log back in again for the change to take effect.

## Security and userids

When you login to Raspbian, you login with a userid. The default is the user "pi" with password "raspberry". This is an "ordinary" userid and not the same as root or superuser. One can always switch to that user but then all commands executed are performed as root and if a typing mistake is made, you can literally ruin your environment. A solution to the problem is to use the "sudo" command. Sudo takes as a parameter the name and parameters of another command. Sudo then switches to root, and then runs that command as root. However, when the command completes, the switch to root is not maintained. Thus, in effect, we run as root for just the duration of the command.

One of the primary configuration files for sudo is /etc/sudoers. This contains the rules that govern which users may execute sudo and for what commands. On the Pi, the userid called "pi" is fully authorized to run any command and does not have to provide any passwords.

See also:

- [sudo Home Page](#)

## System applications

When Raspbian starts, it needs to know which services it needs to also start. Examples of services would be demons such as web servers, NFS servers and more. The package called `systemd` provides management of such requests.

To see the list of units defined we can run:

```
$ systemctl list-units
```

This not only shows the defined units but their status. The columns shown include:

- UNIT – The name of the unit
- LOAD
- ACTIVE
- SUB
- DESCRIPTION – A description of the unit

We can qualify the list-units command with a `--type=<type>` flag to filter on just specific types of units. For example:

```
$ systemctl list-units --type=service
```

Lists all the units that are services.

If we want to see a graph of the units that are enabled we can run:

```
$ systemctl status
```

To see the status of a specific service we can add the service name:

```
$ systemctl status <service name>
```

This last command will also show the script used to start the service.

To start a service we can run:

```
$ sudo systemctl start <service name>
```

To stop a service we can run:

```
$ sudo systemctl stop <service name>
```

and to restart a service we can run:

```
$ sudo systemctl restart <service name>
```

Some services allow us to reload their configurations with:

```
$ sudo systemctl reload <service name>
```

The details of what a particular unit is going to do when started or stopped can be found by running:

```
$ systemctl cat <service name>
```

This will log the content of the unit configuration file.

See also:

- [systemd System and Service Manager](#)
- [Systemd Essentials: Working with Services, Units, and the Journal](#)
- [Understanding and Using Systemd](#)
- YouTube: [Tutorial – systemd Basics](#)

## Writing your own units

Since `systemd` runs against units and we can write our own units to control our own applications, let us start on that journey. The core locations for units are `/etc/systemd/system/` and `/usr/lib/systemd/system/`.

Let us take apart an existing unit so that we may see what it looks like. Within `/etc/systemd/system/` we will find a file called "`sshd.service`". If we look within we will find the following:

```
[Unit]
Description=OpenBSD Secure Shell server
After=network.target auditd.service
ConditionPathExists=!/etc/ssh/sshd_not_to_be_run

[Service]
EnvironmentFile=-/etc/default/ssh
ExecStart=/usr/sbin/sshd -D $SSHD_OPTS
ExecReload=/bin/kill -HUP $MAINPID
```

```
KillMode=process
Restart=on-failure

[Install]
WantedBy=multi-user.target
Alias=sshd.service
```

At a high level, a configuration file is composed of a series of sections with the section name in square brackets. Following those are a series of `name=value` properties for the corresponding section.

Amongst the sections we have:

- [Unit]
- [Service]
- [Install]

For the [Unit] section we will invariably wish to include a Description line that provides an English description of what the unit means. For example:

```
[Unit]
Description=My service that does XYZ
```

In the [Service] section, here we describe how to start a particular service. There are many possible options here and we will touch on just a few. The first is the Type line. This describes how the service will be started. The value of simple means that `systemd` will simply `fork()` and `exec()` the service. This is commonly just what we want. A value of forking means that again `systemd` will `fork()` and `exec()` the service but that service will itself `fork()` and `exec()` other instances. Next we have the ExecStart line. This is vital. This specifies the path and parameters that will be used to start the service. The WorkingDirectory line (if present) specifies the directory that will be the working directory for the newly started service. The User line is used to name a local Raspbian user that the service will run as.

After having edited a unit file, we must run `sudo systemctl daemon-reload` to cause `systemd` to re-scan its configuration files and unit files.

```
$ sudo systemctl daemon-reload
```

For example if we create `/etc/systemd/system/testService.service` containing:

```
[Service]
Type=simple
User=pi
WorkingDirectory=/home/pi/src/projects/testService
ExecStart=/usr/local/bin/node testService.js
```

Now when we wish to start the service, we can issue:

```
$ sudo systemctl start testService
```

Which will cause the service to be started. Hopefully now you may get an “aha” moment. What this means is that for each service defined, we as administrators of our Pi’s Raspbian OS no longer have to

remember mechanics of how to start daemons. All we need to know is their logical service name and issue a start command.

But what if we want to stop a service? Simply issuing:

```
$ sudo systemctl stop testService
```

will stop it. By default, the way that `systemd` performs that task is to execute a kill against the previously remembered process id of the start request. If we have an alternate shutdown procedure, we can define another statement in our `[Service]` section called `ExecStop`. This command should stop the previously started service in whatever clean manner is possible. A special variable called `$MAINPID` is made available to the command which is the process Id of the previously started service.

We have now touched upon what it takes to start and stop a service manually, but what if we want the service to start automatically? We can again do this with `systemd` but it takes a little more consideration. The first thing to realize is that services typically depend on each other. For example, there is no point in starting the mail processing service until and unless the network services have been started. We can define relationships between units with additional statements in the `[Unit]` section. The first one we will touch upon is the `Requires` statement. This names additional units which should also be started when the current unit is started. We also have the statements `Before` and `After`. These the relative order in which multiple Units should be started. For example if we have:

```
[Unit]
Requires=xyz.service
After=xyz.service
```

then we have declared that when this current unit is started we had better also start the `xyz.service` but it must be started after `xyz.service` has in fact started. There are other statements that are similar to `Requires` but with slightly altered semantics. `Requires` states that if a dependent unit fails to start then so do we. The `Wants` statement again lists units to be started but should they fail, then our unit will still be allowed to start.

See also:

- [man – `systemd.service`](#)

## The `systemd-ui`

A graphical user interface is available for `systemd` operations. It can be installed from the `l` package using:

```
$ sudo apt-get install systemd-ui
```

The launcher command is:

```
$ sudo systemadm
```

## Kernel modules

We have the ability to define modules that run within the kernel environment when Raspbian is loaded. These modules are specified in the `/etc/modules` file. In addition, there is a second file called `/etc/modprobe.d/raspi-blacklist.conf` that explicitly lists the modules that should be excluded from the kernel. When adding a new module, add it to the `/etc/modules` **and** validate that it is not named in `/etc/modprobe.d/raspi-blacklist.conf`.

The concept behind a blacklist is to name kernel modules that would otherwise be loaded that should not be loaded. It is a mystery why when we have some modules that need explicit loading that they are already listed in the blacklist.

To see which modules are installed in the kernel we can `lsmod`.

We can remove a running module with `modprobe -r <module name>`.

There is a configuration directory called `/etc/modprobe.d` into which configuration files will be placed. These are read during start-up and the `modprobe` commands within are executed.

There is also a module load directory called `/etc/modules-load.d`.

See also:

- `man(1) – lsmod`

## The Device Tree

Before the Device Tree, the kernel itself contained the details of the hardware. With Device Tree, this data is kept in a separate set of files called the device tree blob (dtb) files. The source of the Device Tree is composed of Device Tree Source files (dts) and Device Tree Include files (dtsi). A tool called the device tree compile (dtc) builds the device tree blob.

If we think of a single Device Tree as describing **all** the hardware available to the OS, this does not account for the notion of plug-able devices. If we plug in a new USB device into an open USB slot, it doesn't make sense to have all possible drivers for all possible devices baked into the Device Tree. It would become huge and unmanageable. As such what we want is a way to somehow augment the Device Tree via simple configuration. This is possible through a technique called overlays.

One can dump a device tree blob using the command `fdt dump` (part of device-tree-compiler).

One can see the current device tree in use by a running kernel by running:

```
dtc -I fs /proc/device-tree
```

In the `/boot/overlays` are a set of device tree blobs for some common Raspberry hardware. The `README` within gives more details.

The Pi's device tree usage is governed from the content of the file called `/boot/config.txt`. Overlays are added by adding `dtoverlay=<name>` within the `config.txt` file.

Some of the Overlays that will be found include:

ads7846	ADS7846 Touch Controller
at86rf233	ATMEL AT86RF233 8012.15.4 WPAN transceiver
bmp085_i2c-sensor	Configure the BMP085/BMP180 digital thermometer and pressure sensor
dht11	DHT11/DHT21/DHT22 humidity and temperature sensors
enc28j60	ENC28J60 Ethernet Controller

g prio-ir	Infrared receiver
gpio-poweroff	Drive a GPIO pin high or low during a reboot
i2c-gpio	Support for software I2C
i2c-rtc	Add support for a real time clock
i2s-mmap	Configure the bcm2708-I2S memory mapping
lirc-rpi	Configure the Linux Infrared Remote Control
mcp2515-can0	
pps-gpio	Configure pulses per second on GPIO pin

To debug, add `dtdebug=on` and then use

```
$ sudo vcdbg log msg
```

to examine messages.

See also:

- Raspberry Pi – [Device Trees, Overlays and Parameters](#)
- [Device Tree for dummies](#)

## CPU - /proc/cpuinfo

The file at `/proc/cpuinfo` can be read to produce the following output. This contains details about the processors on the Pi.

- processor – <id> A simple integer for each processor starting at 0.
- model name – ARMv7
- BogoMIPS – 38.40
- Features
- CPU implementer – 0x41
- CPU architecture – 7
- CPU variant – 0x0
- CPU part – 0xc07
- CPU revision – 5
- Hardware – BCM2709 – The type of hardware that hosts the CPU.
- Revision – This is a bit encoding value of a variety of flags. It is a 26 bit value that is encoded as follows:

<b>Bits</b>	<b>Size (bits)</b>	<b>Description</b>
3:0	4	PCB revision number
11:4	8	Model id: • 0x00 – A • 0x01 – B • 0x02 – A+ • 0x03 – B+ • 0x04 – Pi 2 model B • 0x05 – Alpha • 0x06 – Compute module • 0x07 – Not used • 0x08 – Not used • 0x09 – Pi Zero
15:12	4	Processor: • 0x00 – BCM2835 • 0x01 – BCM2836
19:16	4	Manufacturer: • 0x00 – Sony • 0x01 – Egoman • 0x02 – Embest • 0x03 – Not used • 0x04 – Embest
22-20	4	Memory size: • 0x00 – 256MB • 0x01 – 512MB • 0x02 – 1024MB
23	1	0 – Revision is not bit encoded 1 – Revision is bit encoded
24	1	Warranty bit
25	1	Warranty bit

To decode your string by sight, you can take its hex value and display in binary on your desktop calendar. For example ... a21041 becomes

1010 0010 0001 0000 0100 0001

in binary. From there it becomes easier to mark off the chunks of encode values.

- Serial – <long hex string> This is presumably the board serial number.

Here is an example from my Pi 2 Model B:

```

processor      : 0
model name    : ARMv7 Processor rev 5 (v7l)
BogoMIPS      : 57.60
Features       : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae evtstrm
CPU implementer: 0x41
CPU architecture: 7
CPU variant   : 0x0
CPU part      : 0xc07
CPU revision   : 5

processor      : 1
model name    : ARMv7 Processor rev 5 (v7l)
BogoMIPS      : 57.60
Features       : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae evtstrm
CPU implementer: 0x41
CPU architecture: 7
CPU variant   : 0x0
CPU part      : 0xc07
CPU revision   : 5

processor      : 2
model name    : ARMv7 Processor rev 5 (v7l)
BogoMIPS      : 57.60
Features       : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae evtstrm
CPU implementer: 0x41
CPU architecture: 7
CPU variant   : 0x0
CPU part      : 0xc07
CPU revision   : 5

processor      : 3
model name    : ARMv7 Processor rev 5 (v7l)
BogoMIPS      : 57.60
Features       : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae evtstrm
CPU implementer: 0x41
CPU architecture: 7
CPU variant   : 0x0
CPU part      : 0xc07
CPU revision   : 5

Hardware      : BCM2709
Revision       : a21041
Serial         : 00000000ee03bac3

```

## Memory - /proc/meminfo

The memory utilization on Raspbian can be found using cat /proc/meminfo.

The peripherals start at address 0x2000 0000 for the Pi 1 and 0x3f00 0000 for the Pi 2. This is defined by the constant `BCM2708_PERI_BASE`.

The system file called /proc/iomem can be read to find the current settings as shown in the following taken from a Pi 2:

```

$ cat /proc/iomem
00000000-3affffff : System RAM
  00008000-00775ccf : Kernel code
  007de000-0092378b : Kernel data
3f006000-3f006ffff : dwc_otg
3f007000-3f007eff : /soc/dma@7e007000
3f00b840-3f00b84e : /soc/vchiq
3f00b880-3f00b8bf : /soc/mailbox@7e00b800
3f200000-3f2000b3 : /soc/gpio@7e200000
3f201000-3f201ffff : /soc/uart@7e201000
  3f201000-3f201fff : uart-p1011
3f204000-3f204ffff : /soc/spi@7e204000

```

```
3f300000-3f3000ff : /soc/mmc@7e300000
3f980000-3f98ffff : dwc_otg
```

The GPIO registers can be found at 0x0020 0000 from the peripheral base and defined with the constant `GPIO_BASE`.

If we look above, we see that the GPIO starts at 0x3f200 0000 which is 0x3f00 0000 + 0x0020 0000.

Since the peripherals are mapped to memory, we can use the `/dev/mem` device driver plus memory mapped files (`mmap`) to read and write the peripherals from our application through memory access.

## System parameters

There are many system parameters that are Pi specific. Most of these can be queried with the powerful `vcgencmd`. We can enter `vcgencmd commands` to get a list of the known commands. This simply produces a list without describing what they do.

See also:

- elinux.org – [vcgencmd usage](#)

## Clock frequencies

We can examine a variety of clock frequencies through `vcgencmd measure_clock <system>`. There are a variety of systems available including:

System	Description	Pi 2
arm		600062000
core		250000000
dpi		0
emmc		250000000
h264		250000000
hdmi		0
isp	Image Sensor Pipeline	250000000
pixel		0
pwm		0
uart		3000000
v3d		250000000
vec		108000000

## Internal temperature

The internal temperature of the device can be determined with `vcgencmd measure_temp` which returns the value in degrees centigrade.

## Memory

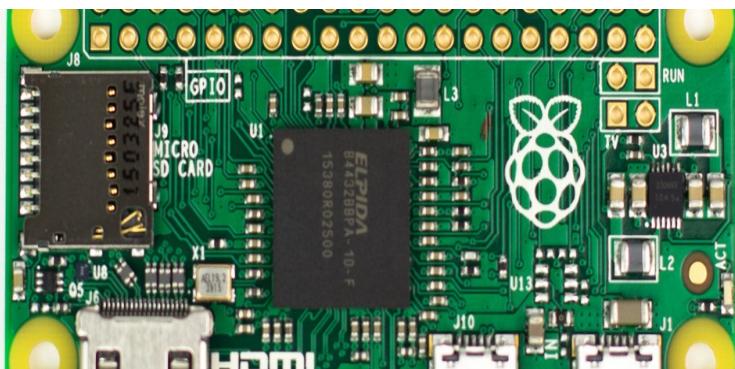
The memory available to each of the major CPU components (ARM and GPU) can be found with `vcgencmd get_mem arm` and `vcgencmd get_mem gpu`.

## Hardware interfacing

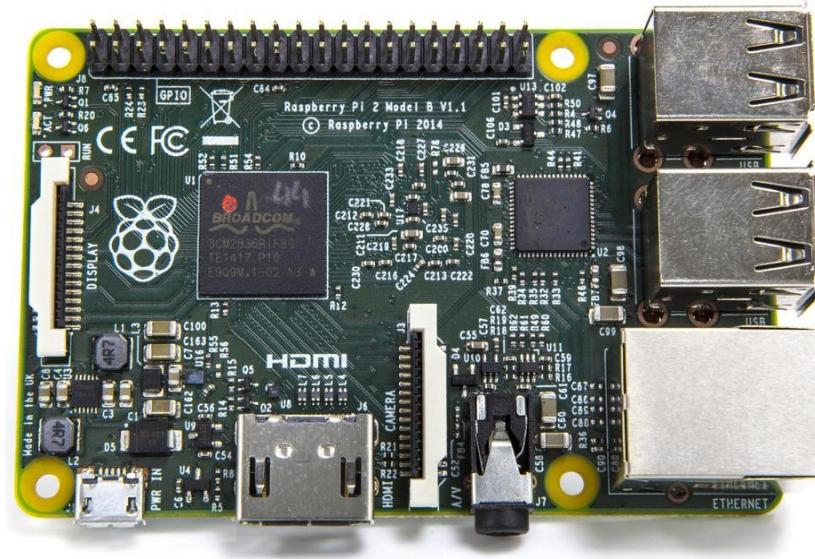
Other than the extremely cheap price of the Pi, the one other attribute that stands out about it is its ability to interface with an arbitrary amount of hardware components through electronic connections. In this section we examine some of the core generic capabilities available to us to interface the Pi with a wide variety of hardware. Discussions on software control of these items comes later and examples of interacting with specific hardware components comes later still.

## Pin numbering

The Pi has a set of pins exposed for connections. What we mean by this is simply wiring points where additional electronic components can be attached. If we look at the following picture of a Pi Zero, we see the pins at the top ( ... strictly speaking we see where the header can be soldered on to the Pi to expose the pins ... the Pi Zero does not come pre-supplied with pin-headers in order to save costs).



In the following, we see a Pi 2, again with the pins at the top of the picture:



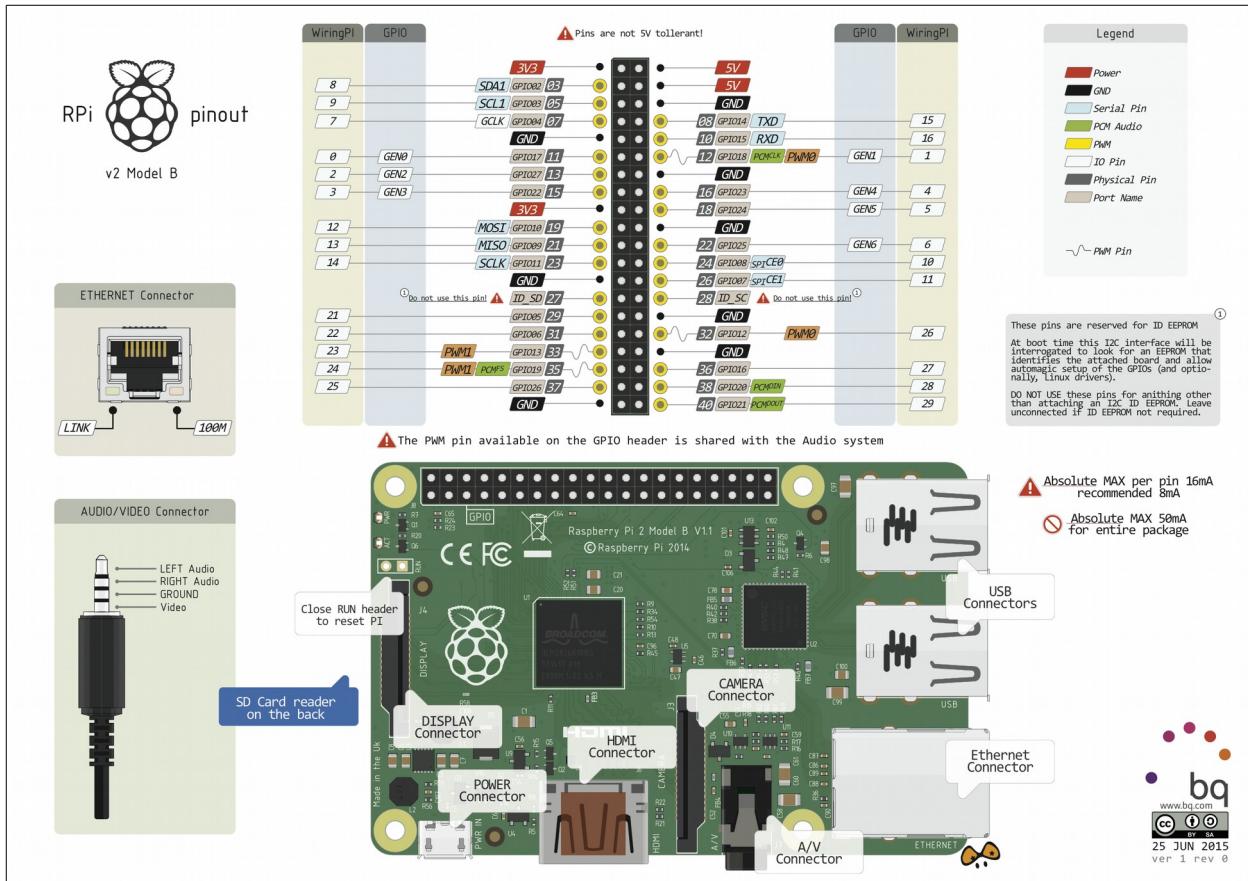
When we think of hardware pins, we immediately consider the notion that we can identify a pin by some common number. For example, if I say "Pin 13", then you should be able to find "Pin 13" on the board. The problem with the Pi is that there are multiple conventions for naming pins. Specifically, there are four stories:

- Physical header numbering
  - Broadcom pin numbering
  - Broadcom GPIO numbering
  - WiringPi numbering

To make matters even more complex, the hardware connectors on different models of the Pi differ from each other. There are basically two hardware connectors. One has 26 pins and the other has 40 pins.

As such, it is absolutely vital that when you think you are reading and writing from a pin, you have researched and understood the identification of that pin. Simply saying "Pin 13" is ambiguous without qualifying the pin identification technique that you are using.

A fantastic source of pin-outs can be found at [www.pighixxx.com](http://www.pighixxx.com). These show the mappings for all common pin numbering schemes.



Another excellent diagram is the one found at [pi4j.com](http://pi4j.com):

Raspberry Pi 2 Model B (J8 Header)								
GPIO#	NAME				NAME			GPIO#
	3.3 VDC Power		1			5.0 VDC Power		2
<b>8</b>	GPIO 8 SDA1 (I2C)		3			5.0 VDC Power		4
<b>9</b>	GPIO 9 SCL1 (I2C)		5			Ground		6
<b>7</b>	GPIO 7 GPCLK0		7			GPIO 15 TxD (UART)	<b>15</b>	8
	Ground		9			GPIO 16 RxD (UART)	<b>16</b>	10
<b>0</b>	GPIO 0		11			GPIO 1 PCM_CLK/PWM0	<b>1</b>	12
<b>2</b>	GPIO 2		13			Ground		14
<b>3</b>	GPIO 3		15			GPIO 4	<b>4</b>	
	3.3 VDC Power		17			GPIO 5	<b>5</b>	
<b>12</b>	GPIO 12 MOSI (SPI)		19			Ground		18
<b>13</b>	GPIO 13 MISO (SPI)		21			GPIO 6	<b>6</b>	20
<b>14</b>	GPIO 14 SCLK (SPI)		23			GPIO 10 CE0 (SPI)	<b>10</b>	22
	Ground		25			GPIO 11 CE1 (SPI)	<b>11</b>	24
	SDA0 (I2C ID EEPROM)		27			SCL0 (I2C ID EEPROM)		28
<b>21</b>	GPIO 21 GPCLK1		29			Ground		30
<b>22</b>	GPIO 22 GPCLK2		31			GPIO 26 PWM0	<b>26</b>	32
<b>23</b>	GPIO 23 PWM1		33			Ground		34
<b>24</b>	GPIO 24 PCM_FS/PWM1		35			GPIO 27	<b>27</b>	36
<b>25</b>	GPIO 25		37			GPIO 28 PCM_DIN	<b>28</b>	38
	Ground		39			GPIO 29 PCM_DOUT	<b>29</b>	40

**Attention!** The GPIO pin numbering used in this diagram is intended for use with WiringPi / Pi4J. This pin numbering is not the raw Broadcom GPIO pin numbers.

<http://www.pi4j.com>

<b>Physical</b>	<b>Broadcom</b>	<b>GPIO</b>	<b>WiringPi / Pi4J</b>	<b>Special</b>
2	03	GPIO 02	GPIO 08	SDA1
3	05	GPIO 03	GPIO 09	SCL1
4	07	GPIO 04	GPIO 07	GPCLK0
6	11	GPIO 17	GPIO 00	
7	13	GPIO 27	GPIO 02	
8	15	GPIO 22	GPIO 03	
10	19	GPIO 10	GPIO 12	MOSI
11	21	GPIO 09	GPIO 13	MISO
12	23	GPIO 11	GPIO 14	SCLK
15	29	GPIO 05	GPIO 21	GPCLK1
16	31	GPIO 06	GPIO 22	GPCLK2
17	33	GPIO 13	GPIO 23	PWM1
18	35	GPIO 19	GPIO 24	PCF_FS / PWM1
19	37	GPIO 26	GPIO 25	
21	40	GPIO 21	GPIO 29	PCM_DOUT
22	38	GPIO 20	GPIO 28	PCM_DIN
23	36	GPIO 16	GPIO 27	
25	32	GPIO 12	GPIO 26	PWM0
28	26	GPIO 07	GPIO 11	CE1
29	24	GPIO 08	GPIO 10	CE0
30	22	GPIO 25	GPIO 06	
32	18	GPIO 24	GPIO 05	
33	16	GPIO 23	GPIO 04	
35	12	GPIO 18	GPIO 01	PCM_CLK / PWM0
36	10	GPIO 15	GPIO 16	RxD
37	08	GPIO 14	GPIO 15	TxD

Another excellent alternative is to print out a paper template for the Pi on a printer. These can be found which are perfectly placed so that they can be cut-out and pushed over the pins:

<https://sander.grids.be/raspberry-pi-b-printable-pinout/>

When printing them out, make sure that they are printed at actual size as opposed to scaled to what is loaded in your printer paper tray. Print out a few of them as you will have to learn how to punch the headers through the paper by trial and error and it may take a few attempts.

See also:

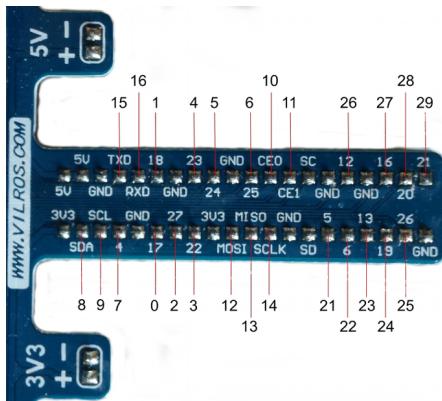
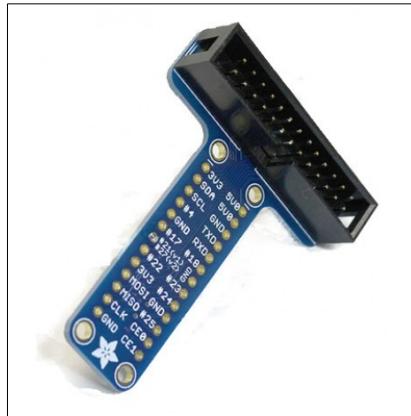
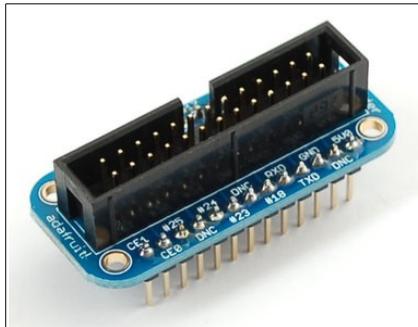
- YouTube – [Learn About The GPIO Connector on Raspberry Pi](#)

## Bread-boarding

When we are building circuits involving the Pi, it is unlikely we will immediately solder everything together. Instead, what we will want to do is prototype our circuits to ensure that they will work. To that end, we will likely use bread-boards which are plug-able boards into which wires can be pressed into

place and also accommodate other components such as ICs, transistors, LEDs, capacitors, resistors and more.

Various companies produce breadboard ready connectors which can accept ribbon cables that plug between the connector and the Pi. These come in a variety of forms such as the following:



I find connectors like these to be incredibly useful and highly recommend them.

## Power

The Pi is powered from a 5V source. On the header, there are a number of sources of voltage. Specifically, there are both 5V and 3.3V outputs. The total amount of current that may safely be sourced from the 5V pin is 250mA while from the 3.3V output it is 50mA.

## GPIO

GPIOs are the General Purpose Input/Outputs. They provide the primary mechanism to read and write electrical signals to and from the Pi.

## GPIO theory

The theory of General Purpose Input/Output (GPIO) is the simplest of all the hardware interfaces to understand. With GPIO a device exposes a set of physical pins. These pins can either be used to output an electrical signal for consumption by a second device connected to the pin or can be used to read the electrical signal present on a pin as set by a second device. At any one time, each pin can either be defined as an output pin or an input pin. Obviously it can't be both at the same time though may switch roles as needed. When a pin is flagged as output, it can source a current that can be consumed by a connected peripheral however care must be taken. GPIOs on MCUs are only able to source a small amount of current and trying to draw too much can irreparably damage the device. Take care to consult the appropriate data sheets for any devices connected to determine their current requirements and ensure that you are not asking for too much current from your MCU.

## GPIO on the Pi

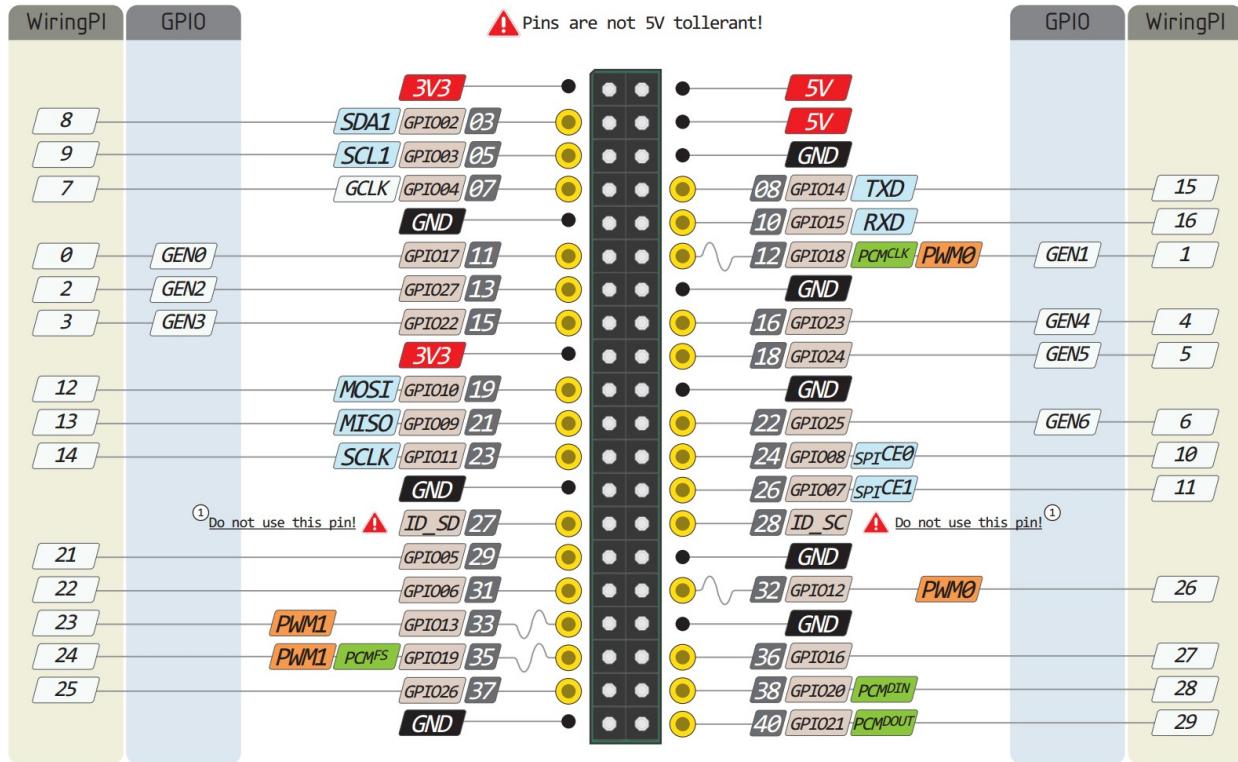
The Pi has GPIO available to it. There are a maximum of 26 pins available for this purpose. The signal levels on the Pi are 3V. What this means is the maximum output signal is 3V and the maximum input signal must also be 3V. Let us be very clear on this. If you attempt to apply an input signal higher than 3V you run a very distinct risk that you will damage or destroy your Pi.

In addition, the maximum output current that may safely be drawn from an output GPIO pin is 3mA. Again, if you try and draw more current than this you risk the destruction of your Pi.

GPIO pins can be configured as input (meaning that we can read signal from them and they don't output signals) or they can be configured as output (meaning that we can write signals to them). On power-up, the majority of the pins are in input mode.

For input mode pins, we also have the notion of pull-up or pull-down resistors that are internal to the device. When we think of an input signal being applied to a pin, we usually consider it high or low ... however there is a third possibility, namely that there is no signal applied to the pin. Think of a pin connected to a button and the button being open. In this scenario, what is the value of the pin should we read it? This is where the pull-up and pull-down resistors come into play. These are resistors that supply sufficient signal that if no alternate signal is supplied, a read of an open circuit pin will result in either a high (pull-up) or a low (pull-down).

Certain pins have overloaded functions and as such the Pi must be placed in a "mode" to designate how the pins are going to be used. For example, GPIO pins GPIO14 and GPIO15 can double as TXD and RXD for UART ... but we can't use them as both GPIO and UART at the same time.



A number of libraries have been written to provide interfaces to the GPIO world from the Pi world. These include:

- [WiringPi](#)
- [Pigpio](#)
- [The Pi4J Project](#)

Another mechanism used to perform I/O against the Pi is to use the file system interface. First let us look at the files in `/sys/class/gpio`:

```
$ ls
export gpiochip0 unexport
```

Of these export and unexport are the ones we want to focus on. If we echo the pin number that we wish to access into export, a new directory appears:

```
$ echo 11 > /sys/class/gpio/export
$ ls
export gpio11 gpiochip0 unexport
```

Notice the new directory called "gpio11". If we enter that directory and list its content what we will find is the following:

```
$ ls
active_low device direction edge subsystem uevent value
```

Let us now look at these special files. If we echo content into these files, that will have the effect of driving function. For example:

```
echo out > direction
```

will set the pin's I/O direction as output.

If we echo 1 or 0 into value, that will set the output value on the pin.

```
echo 1 > value
```

When we have finished with the pin, remove it from the file system with:

```
echo 11 > /sys/class/gpio/unexport
```

The complete list of parameters is as follows:

- direction – The direction of I/O. Either "in" or "out".
- value – The value on the pin. Either "1" or "0"
- edge – Interrupt edge detection. Either "none", "rising", "falling" or "both".
- active\_low – Unknown.

Device drivers provide the ability to access Pi attached hardware in very efficient and rich ways. Some device drivers need to take control of GPIO pins for their function. Obviously if a device driver is using a pin, your own applications can't. One way to determine which drivers are using which pins is to cat the file /sys/kernel/debug:

```
# cat /sys/kernel/debug
GPIOs 0-53, platform/3f200000.gpio, pinctrl-bcm2835:
gpio-18  (fb_ili9341          ) out hi
gpio-21  (ads7846_pendown     ) in  hi
gpio-23  (fb_ili9341          ) out hi
gpio-24  (fb_ili9341          ) out hi
gpio-35  (?)                 in  hi
gpio-47  (?)                 out lo
```

The output shows pins associated with device drivers.

See also:

- BCM2835 GPIO
- WiringPi
- GPIO mapping between the Pi and Arduino
- Simple GPIO Output change
- [sysfs gpio documentation](#)

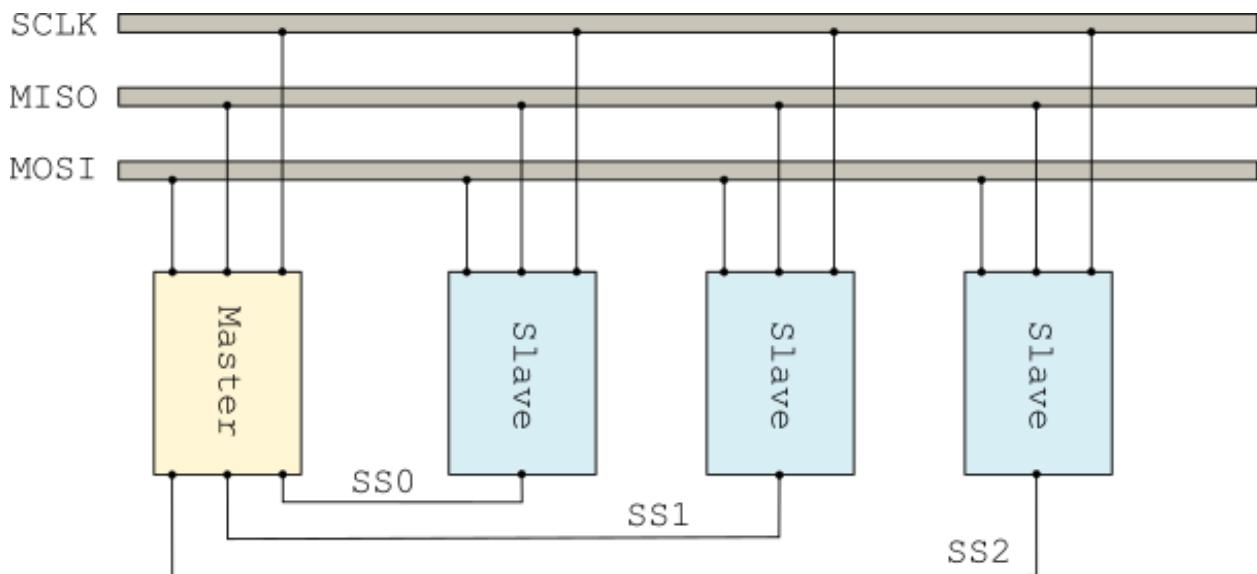
- Simple GPIO Input detection
- gpio command

## SPI

SPI is a parallel to serial bi-directional protocol for connecting a wide variety of devices.

### SPI theory

SPI is a serial protocol used to communicate between masters and slaves. All slaves connect to the same bus but only the slave with its SS driven pin low can transmit. SPI is a full duplex protocol. What this means is that while data is being pushed out from the master to the slave over the MOSI line, the slave is simultaneously sending data back to the master over the MISO line. The MOSI pin contains the serial data from the master to the slave while the MISO pin contains the data from the slave to the master. The master and slaves synchronize their transmissions using a common clock signal.



It is invalid for the master to drive more than one slave select line low at any one time.

Typically three pins:

- MISO – **Master In, Slave Out** – Sending data to the master from the slave
- MOSI – **Master Out, Slave In** – Sending data to the slave from the master
- SCK (SCLK) – Serial Clock – Synchronizes data from the master/slave relationship

There is also an additional signal:

- SS (CSN (Chip Select NOT), NSS) – Slave Select – Used to enable/disable the slave so that there can be multiple slaves. SS low means slave is the active slave.

Since this is a serial protocol and we will receive data in bytes, we need to be cognizant of whether or not data will arrive LSB first or MSB first. There will be an option to control this.

For the clock, we will be latching data and we will need to know what edges and settings are important. There will be a clock mode option to control this. In SPI there are two attributes called phase and polarity. Phase (CPHA) is whether we are latching data on high or low and Polarity (CPOL) is whether high or low means that the clock is idle.

CPOL=0 means clock is default low, CPOL=1 means clock is default high.

When CPOL=0, then the following are the values for CPHA

CPHA=0 means data is captured on clock rising edge, CPHA=1 means data is captured on clock falling edge.

When CPOL=1, then the following are the values for CPHA

CPHA=0 means data is captured on clock falling edge, CPHA=1 means data is captured on clock rising edge.

SPI wraps these two flags into four defined and named modes:

Mode	Clock Polarity – CPOL	Clock Phase – CPHA
SPI_MODE0	0 (Clock default low)	0
SPI_MODE1	0 (Clock default low)	1
SPI_MODE2	1 (Clock default high)	0
SPI_MODE3	1 (Clock default high)	1

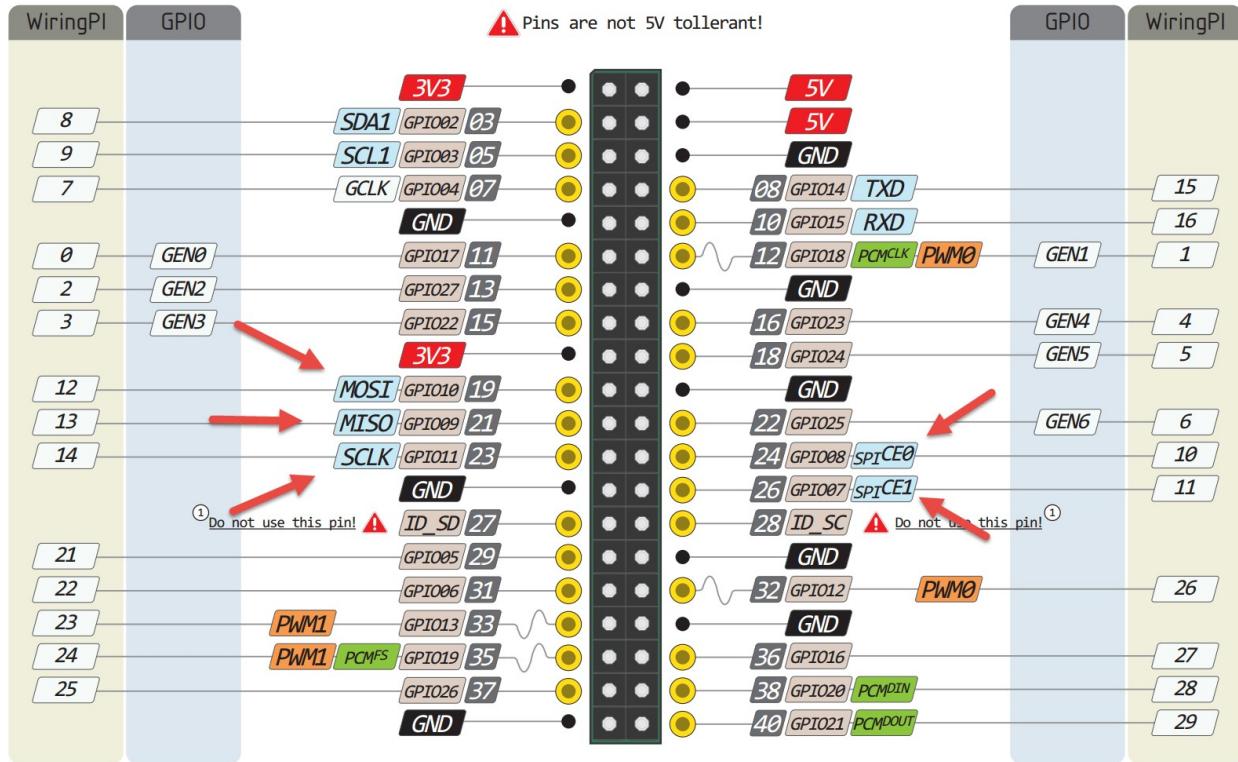
Also for the clock, what speed are we will need to know what speed the data is to be moved. There will be a clock control speed option to control this.

See also:

- Wikipedia – [Serial Peripheral Interface Bus](#)
- [SPI Interface](#)

## SPI on the Pi

The physical pins for SPI are 19 (MOSI), 21 (MISO), 23 (SCLK), 24 (CE0), 26 (CE1).



To validate that SPI is enabled on your Pi, look in the devices directory for files starting `spidev`. For example:

```
$ ls -l /dev/spidev*
crw-rw---- 1 root spi 153, 0 Dec 27 09:31 /dev/spidev0.0
crw-rw---- 1 root spi 153, 1 Dec 27 09:31 /dev/spidev0.1
$
```

In addition, examine the loaded kernel modules with:

```
$ lsmod | grep spi
spi_bcm2835           7216  0
```

The SPI support is loaded at boot time with a line which reads:

`dtparam=spi=on`

See also:

- [WiringPi SPI](#)

## SPI in Python

A Python module is available for the Pi which leverages the SPI interface. It is called "`spidev`". Here is a simple usage flow:

```
import spidev
spi = spidev.SpiDev()
```

```
spi.open(bus, device)
spi.xfer([0x01, 0x02, 0x03])
spi.close()
```

The methods exposed by the object include:

- `open(bus, device)` – Open the device.
- `readbytes(n)` – Read n bytes of data
- `writebytes(values)` – Write the list of values
- `xfer(values, speed_hz, delay_usec, bits_per_word)`
- `xfer2(values, speed_hz, delay_usec, bits_per_word)`
- `close()` - Closes the device

In addition, there are properties on the object:

- `bits_per_word` – How many bits per word, values from 8 – 16.
- `cshigh` – Set to True for CS active high.
- `max_speed_hz` – Maximum transmission speed
- `mode` – 0, 1, 2 or 3

See also:

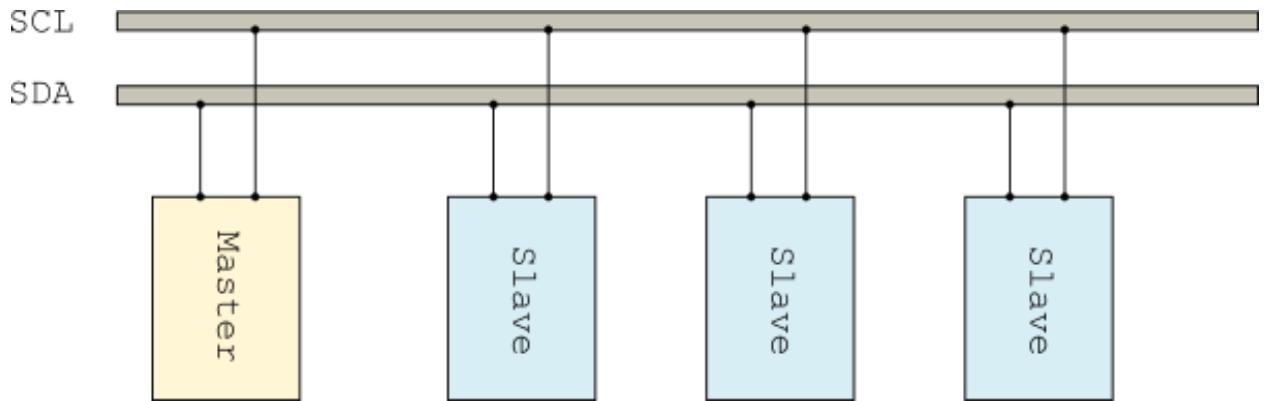
- Github: [py-spidev](#)
- [SpiDev documentation](#)

## I2C

The Inter-IC (I2C) protocol is a protocol for connecting multiple devices together that provides a way to transmit 8 bit bytes over a serial link. Originally developed by Phillips Semiconductors it is now part of NXP Semiconductors.

### I2C theory

The I2C interface is a serial interface technology for accessing devices. The protocol is also called the Two Wire Interface as it uses only two signal bus lines. These signal bus lines are called SDA (Data) and SCL (Clock). I2C allows for bidirectional communication. Attached to these lines are devices that wish to communicate. One device holds a special role and is known as the bus master. It is the master that is the coordinator of all activities and is responsible for generating the clock. All other devices attached to the bus are known as slave devices.



The data transmission rates are up to 100kbits/second in standard mode and 400kbits/second in fast mode.

Each slave device has a distinct and unique address upon the bus. It is the master that always initiates communication which may be either a send request to send data to a specific slave or a read request which asks the slave to respond with specific data. Unless the master explicitly addresses a slave, it must keep silent. The device addresses are commonly 7 bits in size. The master sends the address of the device with which it wishes to communicate. Immediately following the address is a bit that indicates whether this is a read request (1) or a write request (0).

A	A	A	A	A	A	A	R / W
---	---	---	---	---	---	---	-------

Because of this encoding, take care when reading data sheets and looking at the address of a device. Sometimes addresses are written as values from  $0x00 - 0x7f$  describing the device address while sometimes addresses are described as values from  $0x00 - 0xff$  which include the read / write bit.

For example ... the following are all correct:

- The device has address  $0x1e$ .
- The device can be read from  $0x3d$ .
- The device can be written to  $0x3c$ .

The buses are "open collector" with an active low. This means that pull-up resistors should be attached to the lines to ensure that they are logic high by default. A suggested value for a pull-up resistor is 1.8K Ohms however for the Pi these should **not** be added as the Pi hardware has already provided such pull-up resistors.

There are hundreds of devices which are I2C capable. Here is a selection broken out by category:

Accelerometers, Gyroscopes, Compass	
HMC5883L (GY-271)	Compass
MPU-6050 (GY-521)	Accelerometer and Gyroscope

From a signal perspective the SDA line will be read when the clock goes high. The signal on the SDA line must not change while the clock is high (with the exception of the stop bit marker). When the clock goes low, the SDA line can be changed to represent the next bit of data to be transmitted.

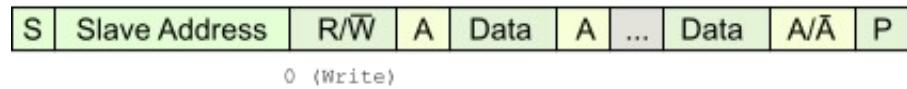
At any one time, the master will be in communication with either no slave or exactly one slave. This sounds very academic but is important to understand. If we think of an initially idle environment, then the master is not communicating with any slaves. When the master now wishes to send data to a slave, it will begin a "transaction". For the life of the transaction, the master may only now communicate with that one slave. The start of a transaction is indicated by a transition from high to low on the SDA line while the clock is high. This is one of only times that the SDA line can change its state while the clock is high and is reserved for a start indication.

A second indicator signal called the stop signal is used to indicate the end of a transaction. This is flagged by an SDA signal line change from low to high while the clock line is high.

After every 8 bits of data are transmitted, the receiver of the data must generate an acknowledgment. The acknowledgment is sent by the receiver and consists of a low value during the high period of the clock. If the receiver detects a situation that it considers an error, it can send a negative acknowledgment by bringing the SDA high low during the high period of the clock.

Data transmission is MSB to LSB.

A summary of the transmission from master to slave and slave back to master for both read and write requests is shown next:

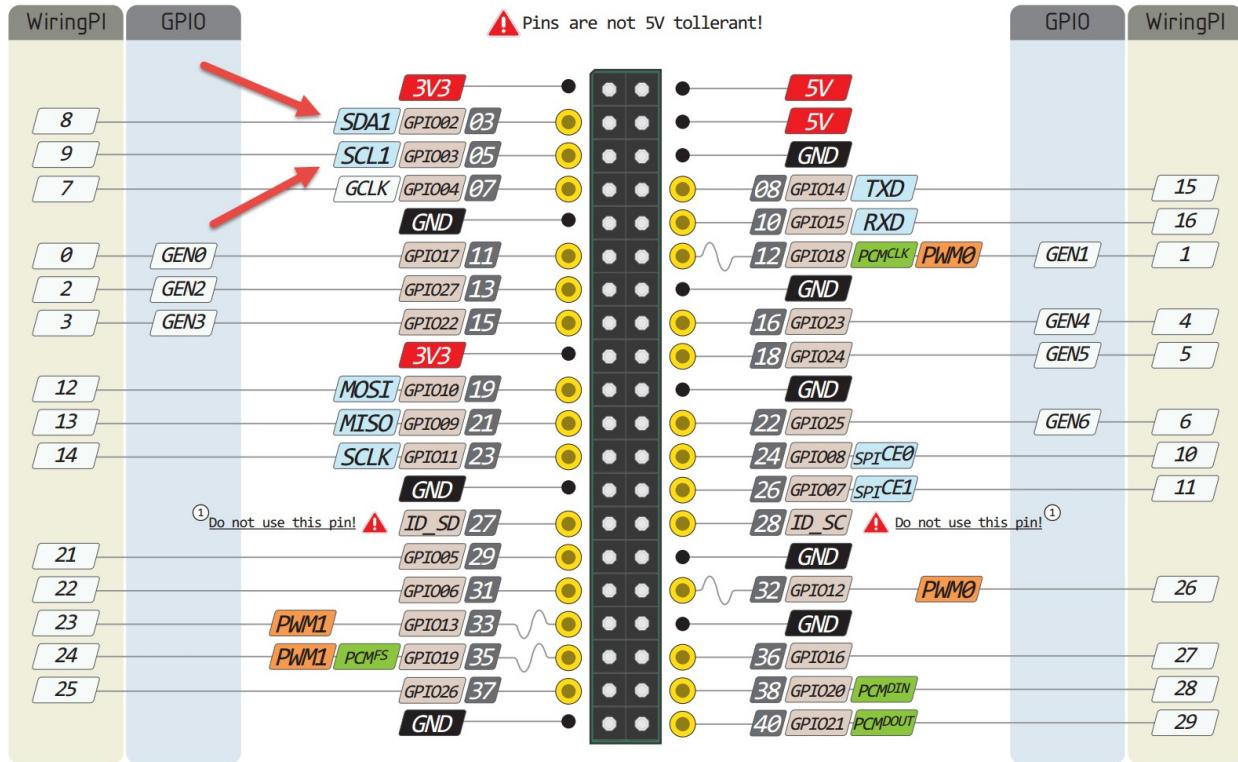


See also:

- Wikipedia – [I2C](#)
- [I2C-bus specification and user manual](#)
- YouTube: [How I2C Communication Works and How To Use It with Arduino](#)

## I2C on the Pi

The physical pins for I2C are fixed and physical pin 03 (SDA) and physical pin 05 (SCL). These are indicated in the following image:



Since I2C is extremely hardware and timing specific, the code that interacts with the I2C bus has to run within the Linux kernel. A decision by the designers was made to allow enabling and disabling of this capability. Presumably this was so that Pi environments that were simply not using I2C need not have any burden at all in I2C processing. As such, the I2C kernel module must be explicitly loaded in order to exploit I2C functions. This can be configured in the Pi setup such that the I2C module is always loaded at Pi start-up. The `raspi-config` command can be used to enable this. It can be found under Advanced Options > I2C.

The result of running this step is to update the `/etc/modules` configuration file which needs the following two entries added for I2C support.

```
i2c-bcm2708
i2c-dev
```

In addition, the setup also edits the `/boot/config.txt` file to add the following:

```
dtparam=i2c1=on
dtparam=i2c_arm=on
```

If you wish to load I2C support manually, you can run:

```
sudo modprobe i2c-dev
```

To determine if the `i2c-dev` is loaded, run the command `lsmod` and look for `i2c-dev` in the resulting list.

The tools for I2C are in the `apt-get` package called `i2c-tools`. If these tools are not installed, further commands such as `i2cdetect` will not be found.

When using I2C we have the notion of a bus with devices attached to this bus. Ideally, we know the identity of the device on the bus but if we aren't sure and/or want to verify that the device is present, we can use the `i2cdetect` command. This command can send the start of an I2C transmission to each of the possible addresses on the bus. By I2C protocol, when we send an address, the device should respond with an acknowledgment that it is ready to proceed. By sending the start of a protocol request to each device, the principle is that those that respond are obviously present and we have thus determined which devices are present on the bus. We can't tell what the device actually is ... only that it is there. Here is an example of `i2cdetect` output:

```
$ i2cdetect -y 1
      0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- --
10:          -- -- -- -- -- -- -- -- -- -- -- --
20:          -- -- -- -- -- -- -- -- -- -- -- --
30:          -- -- -- -- -- -- -- -- -- -- -- --
40:          -- -- -- -- -- -- -- -- -- -- -- --
50:          -- -- -- -- -- -- -- -- -- -- -- --
60:          -- -- -- -- -- 68 -- -- -- -- -- --
70:          -- -- -- -- -- -- -- --
```

Running this command is *potentially* dangerous as we are trusting that each of the devices on the bus won't be confused if it merely receives the start of an I2C request that immediately ends without further interaction. In practice, we know of no real-world circumstances that cause any problems.

Another useful command is `i2cdump`. This takes the bus number and the slave address and returns the values of all registers addressable from the device. Here is an example of output:

```
$ i2cdump -y 1 0x68 b
      0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f  0123456789abcdef
00: 83 01 0d 21 0e ec 24 05 f9 03 f6 28 4c 6e 92  ?????!??$????(Ln?
10: 3a 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 :.....
20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
30: 00 00 00 00 00 00 00 00 00 00 01 fc 00 ff 9c 3b ..... ??.?;.
40: fc ec 20 fe c7 00 4f 00 94 00 00 00 00 00 00 00 ?? ?? .O.?. .....
50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 f0 90 ..... ???
70: 00 00 00 00 00 68 00 00 00 00 00 00 00 00 00 00 ..... h.....
80: 83 01 0d 21 0e ec 24 05 f9 03 f6 28 4c 6e 92  ?????!??$????(Ln?
90: 3a 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 :.....
a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
b0: 00 00 00 00 00 00 00 00 00 00 01 fc 5c ff 34 3b ..... ??\..4;
c0: 90 ec 00 fe c4 00 53 00 9b 00 00 00 00 00 00 00 00 ??..?..S..?.....
d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 f3 50 ..... ??P
f0: 00 00 00 00 00 68 00 00 00 00 00 00 00 00 00 00 ..... h.....
```

In addition, there are two further commands called `i2cget` and `i2cset` that read and write from the I2C bus.

There are potentially multiple I2C buses on a Pi and hence the `i2cdetect` command takes as a parameter the bus to use (a numeric such as 0 or 1). Pi devices with 256MB use bus 0 while Pi devices with more than 256MB use bus 1.

By default, the clock rate is 100kHz. We can change this by stopping the `i2c_bcm2708` module and restarting it with the baudrate flag. For example:

```
sudo rmode i2c_bcm2708
sudo modprobe i2c_bcm2708 baudrate=400000
```

From the shell, we have a couple of commands that can be used to read and write to an I2C slave device. The first is `i2cset`. There are options on it but the simplest is:

```
$ i2cset -y 1 <address> [value]* i
```

When programming to I2C from JavaScript, we can either use the "wiring-pi" or "spi" npm packages.

See also:

- WiringPi I2C
- npm – [i2c](#)
- Adafruit – [Configuring I2C](#)
- man(8) – [i2cdetect](#)
- man(8) – [i2cdump](#)
- man(8) – [i2cset](#)
- man(8) – [i2cget](#)

## PWM

### PWM theory

The idea behind pulse width modulation is that we can think of regular pulses of output signals as encoding information in how long the signal is kept high. Let us imagine that we have a period of 1Hz (one thing per second). Now let us assume that we raise the output voltage to a level of high for  $\frac{1}{2}$  of a second at the start of the period. This would give us a square wave which starts high, lasts for 500 milliseconds and then drops low for the next 500 milliseconds. This repeats on into the future. The duration that the pulse is high relative to the period allows us to encode an analog value onto digital signals. If the pulse is 100% high for the period then the encoded value would be 1.0. If the pulse is 100% low for the period, then the encoded value would be 0.0. If the pulse is on for "n" milliseconds (where n is less than 1000), then the encoded value would be  $n/1000$ .

Typically, the length of a period is not a second but much, much smaller allowing us to output many differing values very quickly. The amount of time that the signal is high relative to the period is called the "duty cycle". This encoding technique is called "Pulse Width Modulation" or "PWM".

There are a variety of purposes for PWM. Some are output data encoders. One commonly seen purpose is to control the brightness of an LED. If we apply maximum voltage to an LED, it is maximally bright. If we apply  $\frac{1}{2}$  the voltage, it is about  $\frac{1}{2}$  the brightness. By applying a fast period PWM signal to the input

of an LED, the duty cycle becomes the brightness of the LED. The way this works is that either full voltage or no voltage is applied to the LED but because the period is so short, the "average" voltage over time follows the duty cycle and even though the LED is flickering on or off, it is so fast that our eyes can't detect it and all we see is the apparent brightness change.

See also:

- Wikipedia – [Pulse-width modulation](#)

## PWM on the Pi

There are two hardware PWM output channels on the BCM283\* chips labeled PWM0 and PWM1. On a 26 pin header, only PWM0 is exposed. If we are willing to accept a little jitter, we can use software PWM.

The Pi has two algorithms for creating PWM signals. One is called Mark/Space and the other is called Balanced. So far, what we have been discussing is the Mark/Space encoding. The other encoding is called Balanced. This is a little trickier to understand so let us take this one slowly.

Consider a period of 1 second and now consider a duty cycle of 25%. This says that the signal will be high for 25% of the period of 250 milliseconds. This means that the signal will be low for 750 milliseconds. In the Mark/Space algorithm, the first 250 milliseconds will be high and the next 750 milliseconds will be low.

Now consider the same story as above with 125 milliseconds high, 375 milliseconds low, 125 milliseconds high and 375 milliseconds low. If we look at any given period of 1 second, again we will find that we have a high for 25% of the time and hence the duty cycle remains at 25%. What has changed is the distribution of the highs and lows. Instead of high at the start and low for the remainder of the period, we have now distributed the highs and lows.

Note that the default mode is Balanced and **not** Mark/Space.

The period of the PWM is a function of the base clock speed which is 19.2MHz. From that we can divide it by a divisor to give us a base frequency. This is known as the clock frequency and can be set by the call to `pwmSetClock()` and then we divide that by a range (default 1024) to create a sequence of pulses.

Here is the working theory on mapping PWM to clock and range.

We start with the base clock rate of 19.2MHz. From there we have a core divisor set by `pwmSetClock`. This divides the base clock by that value. The range can be 1-4095. Let us call the result the divided clock. Next we divide by a second number ... called the range. The result of THAT division is the period.

For example, if we divide 19.2MHz / 1920 we end up with 10KHz. Now if I want a 20msec period, that would be 50Hz which would a range of 200 ( $10\text{KHz}/200 = 50\text{Hz}$ ).

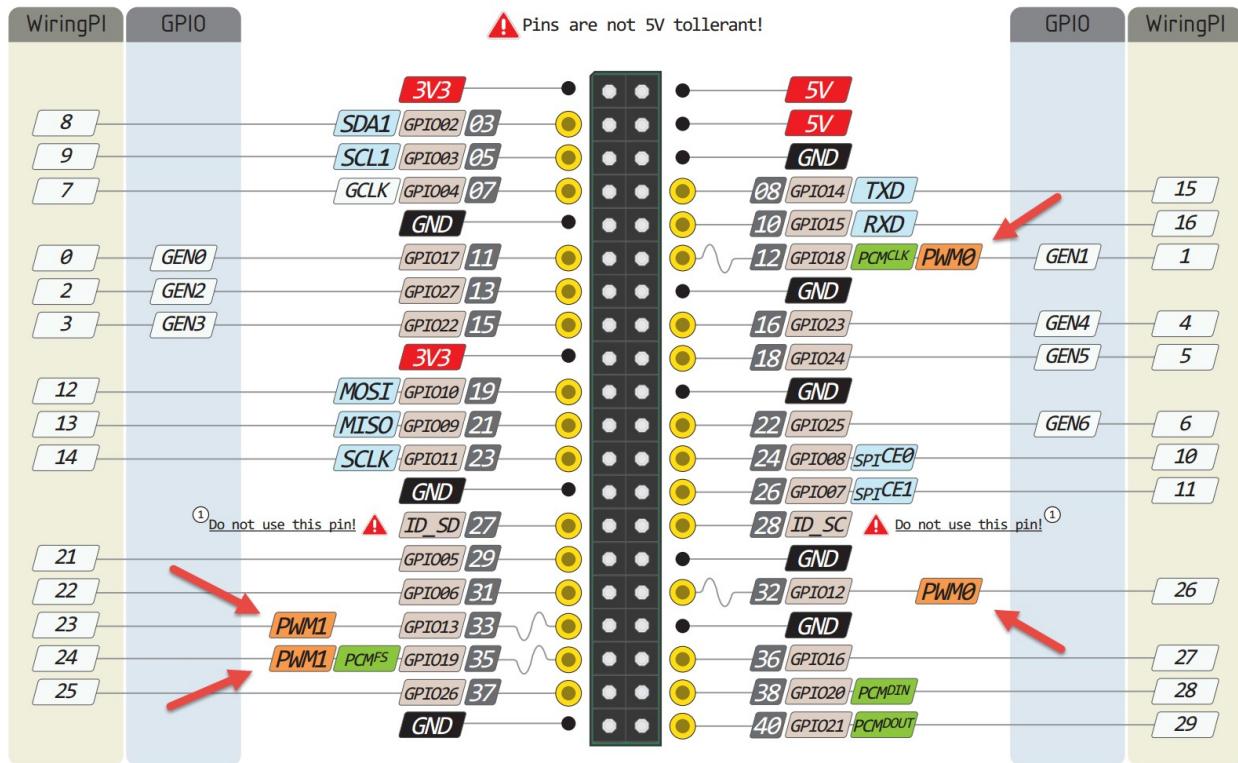
Now for a second example, if we divide 19.2MHz / 192 we end up with 100KHz. Now if we want a 20msec period (50Hz) we would need a range of 2000 ( $100\text{KHz}/2000 = 50\text{Hz}$ ).

By decreasing the clock divisor, we can increase the range and increase the sensitivity of values.

The pins for PWM are:

Physical	Wiring Pi GPIO	Broadcom GPIO	Function
12	GPIO 01	GPIO 18	PWM0
32	GPIO 26	GPIO 12	PWM0
33	GPIO 23	GPIO 13	PWM1
35	GPIO 24	GPIO 19	PWM1

If we look at the physical pin headers, here is where they can be found:



One of the most common usage patterns for PWM is for controlling servo motors. Another function is for controlling the brightness of an LED.

It may be that we have to switch the mode of PWM pins to Alt functions in order to leverage PWM hardware control.

The PWM pins on the Pi are shared with the audio subsystem (audio output on the 3.5mm jack socket). As such, you can have PWM output or audio output, but not both at the same time.

See also:

- [Controlling Servo Motors with Raspberry Pi](#)
- github: [sarfata/pi-blaster](#)
- `pwmWrite`

- gpio pwmr
- MagPi Magazine #30 – [Pulse width modulation motor control](#)

## UART (Serial) Interface

### UART theory

A UART is a mechanism to transmit and receive parallel data over a serial line connection. The data being transmitted is serialized into a sequence of bits and sent down the wire. The receiver receives the bits and assembles them back into the original data. Both the sender and receiver pre-agree on the transmission rate of the data. This is called the baud rate. Before data is transmitted, a start bit is flagged on the wire to indicate that data is about to follow. Then comes the 8 bits of data followed by a single stop bit. It is the start bit that allows the sender and receiver to synchronize on their clocks for the remaining bits. The idle state of a serial line is high. There is also the option of supplying parity bits and other control information.

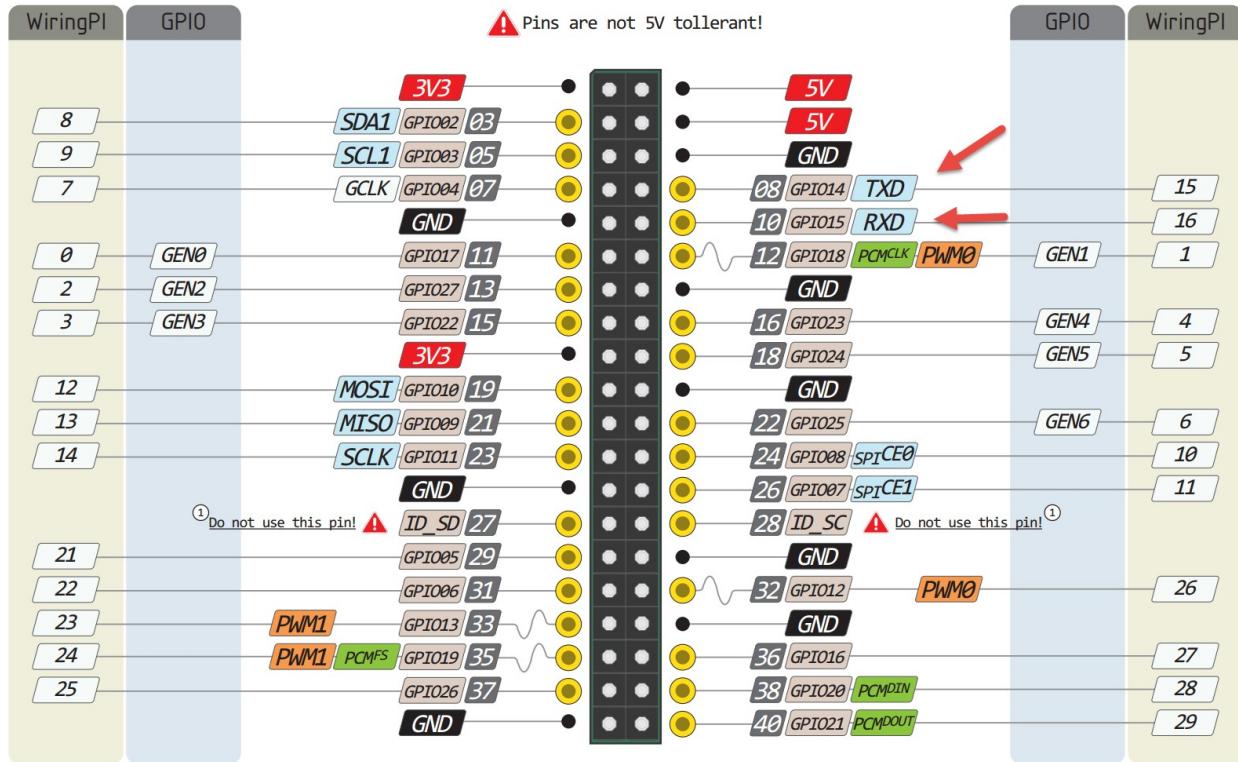
Some baud rates that are common are 9600, 19200, 57600 and 115200. Remember that these are bit transmission rates and not byte rates. Since a single byte will have at least two additional bits (start and stop), this means that there are 10 bits to be transmitted for each byte. As such, a baud rate of 9600 may mean 960 ( $9600 / 10$ ) bytes per second as opposed to 1200 ( $9600 / 8$ ) bytes per second.

See also:

- Wikipedia: [Universal asynchronous receiver/transmitter](#)

### UART on the Pi

The Pi has a UART interface with TX and RX pins. The pins are located at physical locations 8 (TXD) and 10 (RXD).



The logic level outputs are TTL compatible meaning highs of +5V. You need to take care when connecting to UART partners that may need 3.3V inputs as you could fry the partner by applying 5V as a signal.

The serial port on the Pi can be found as the device called `/dev/ttyAMA0`.

By default, the UART is used to provide a terminal for access. This means that there is already an application reading and writing to it. That application is an instance of the Linux `getty` program which is handling terminal access. There are going to be times when we want to use the UART for our own I/O purposes and hence it will be our application that needs access to it. Since we can't have two programs thinking they have control over the UART, we must disable the `getty` program. This is registered as a service which can be stopped with:

```
sudo systemctl stop serial-getty@ttyAMA0.service
```

it can be disabled with

```
sudo systemctl disable serial-getty@ttyAMA0.service
```

Before we finish with the UART as a serial terminal, is there a time we may wish to use that capability? I think the answer is yes.

Many times I find myself on an airplane or in a hotel room with a laptop and a Pi. Unfortunately, I can't play with my Pi for a variety of reasons. Either I have no WiFi environment, no room for a keyboard or am otherwise short something. Now imagine I could attach my Pi to my laptop via a short USB cable. Once connected, I could then open up a terminal emulator on the laptop such as PuTTY which would then

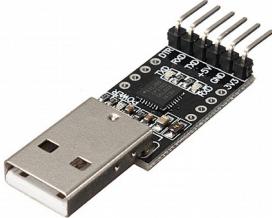
show a login to the Pi. Fortunately, we can do just that using the UART of the Pi connected to a USB→UART adapter.

To determine if the UART is being used as a terminal, run:

```
$ systemctl status serial-getty@ttyAMA0
● serial-getty@ttyAMA0.service - Serial Getty on ttyAMA0
  Loaded: loaded (/lib/systemd/system/serial-getty@.service; disabled)
  Active: active (running) since Sun 2016-01-10 00:15:49 CST; 2s ago
    Docs: man:agetty(8)
          man:systemd-getty-generator(8)
          http://0pointer.de/blog/projects/serial-console.html
   Main PID: 5452 (agetty)
     CGroup: /system.slice/system-serial\x2dgetty.slice/serial-getty@ttyAMA0.service
             └─5452 /sbin/agetty --keep-baud 115200 38400 9600 ttyAMA0 vt102
```

From this we can see that a getty instance is running.

For example, if we take a CP2102:



And connect the TX pin of the USB→UART to the RX pin of the Pi and the RX pin of the USB→UART to the TX pin of the Pi, then if we run Putty against the COM port that shows up in Windows, we get a Raspbian login prompt.

There are a number of good serial terminal applications available, one of the most popular is PuTTY which is available for both Linux and Windows. We can install this on Raspbian by installing the package called "putty". PuTTY is an X-Windows application so will need an X-Server on which to display the output.

See also:

- eLinux – [RPI Serial Connection](#)
- [PuTTY](#)

## Using a USB to UART as additional UART

We have seen that the Pi has built in UART control but what other options do we have? We can plug-in a USB to UART connector into an open USB port on the Pi. Once plugged in, and we run lsusb, we might see a new entry that looks as follows:

```
Bus 001 Device 006: ID 10c4:ea60 Cygnal Integrated Products, Inc. CP210x UART Bridge / myAVR mySmartUSB light
```

This says that a bridge between USB and UART has been detected. If we run `dmesg` to look at new kernel messages generated when the USB was added, we might find messages similar to the following:

```
[126657.073547] usb 1-1.5: new full-speed USB device number 6 using dwc_otg
[126657.180084] usb 1-1.5: New USB device found, idVendor=10c4, idProduct=ea60
[126657.180111] usb 1-1.5: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[126657.180128] usb 1-1.5: Product: CP2102 USB to UART Bridge Controller
[126657.180144] usb 1-1.5: Manufacturer: Silicon Labs
[126657.180158] usb 1-1.5: SerialNumber: 0001
[126657.226244] usbcore: registered new interface driver usbserial
[126657.226345] usbcore: registered new interface driver usbserial_generic
[126657.226423] usbserial: USB Serial support registered for generic
[126657.230607] usbcore: registered new interface driver cp210x
[126657.230748] usbserial: USB Serial support registered for cp210x
[126657.230920] cp210x 1-1.5:1.0: cp210x converter detected
[126657.231385] usb 1-1.5: cp210x converter now attached to ttyUSB0
```

Now if all of that looks like gobble-gook, that is okay, the key message is the last one which states that the USB to UART bridge can be found at `/dev/ttyUSB0`. What that means is that device result in serial data exposed through the TX and RX pins of the USB bridge to which we can connect any number of serial devices. For example, our GPS device or an Arduino.

See also:

- GPS

## Analog input

Simply put, the Pi does not provide any native analog input capability. Instead, you will have to interface with an external analog to digital converter (ADC). This could be a dedicated IC or it could be through the interaction with another MCU such as the Atmel or Arduino.

See also:

- Using an Arduino as a peripheral controller
- Analog to Digital conversion – ADC0832
- Analog to Digital conversion – MCP3208
- Analog to Digital conversion – MCP3208
- Analog to Digital conversion – Arduino

## USB

The nature of USB devices can be found with the `lsusb` command. From there, you can add the `-v` (verbose) option and filter with the `-s devNum` option.

For example, here is an output from `lsusb`:

```
Bus 001 Device 005: ID 18ec:3399 Arkmicro Technologies Inc.
Bus 001 Device 007: ID 0bda:8172 Realtek Semiconductor Corp. RTL8191SU 802.11n WLAN
```

```
Adapter
Bus 001 Device 004: ID 0bda:8176 Realtek Semiconductor Corp. RTL8188CUS 802.11n WLAN
Adapter
Bus 001 Device 003: ID 0424:ec00 Standard Microsystems Corp. SMSC9512/9514 Fast
Ethernet Adapter
Bus 001 Device 002: ID 0424:9514 Standard Microsystems Corp.
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

When we look at the output of a USB listing, we sometimes see pairs of numbers. In the above, we can see 18ec:3399 and 0bda:8172. The first of these numbers is called a vendor id and each product vendor has been allocated a unique code. The second number is a product id and corresponds to a product registered by that vendor. When a USB device connects to a computer, the computer can query these codes and learn what kind of device it is that has attached.

We can use this vendor/product pair as keys in a web look-up to determine more details about that device. A website such as <http://thesz.diecrue.eu/content/usbid.php> will allow us to enter the vendor/product pair and retrieve its details.

See also:

- man(8) – [lsusb](#)

## Audio

The Pi is capable of generating an audio output signal. There are two possible output destinations. The first is to carry the audio signal as part of the HDMI output. The other is to carry it through the 3.5mm output jack (Note: The Pi Zero does **not** have a 3.5mm jack). Only one of these outputs can be used at one time. The default is to attempt to auto-sense the type to be used. This is accomplished by the HDMI driver interrogating the HDMI signals which should query the other end of the HDMI cable typically connected to your TV. If the HDMI signal states that it is capable of handling audio, then HDMI will be used as the audio output, otherwise the 3.5mm jack will be used. If the sensing should not work for you, you can force a mode by running the command:

```
amixer cset numid=3 {0|1|2}
```

where 0 means automatic (the default), 1 means the 3.5mm jack and 2 means HDMI.

There are a number of tools available to play audio. The first is `omxplayer`. If we supply it an MP3 file, it will play its output.

Tools:

- `omxplayer`
- `aplay` – Play a PCM sound from q
- `arecord`
- `speaker-test`
- `alsamixer` – a text but full screen audio mixer
- `amixer` – a command line audio mixer

- mpg123 – MP3 player

Sample WAV and MP3 files can be found pre-supplied on the Pi at:

```
/usr/share/scratch/Media/Sounds
```

One of the easiest ways to play audio from within your application is to spawn a sub-process such as aplay or mpg123 using a system call.

We can also plug in a USB audio output device. We can check that it has been recognized with lsusb:

```
$ lsusb
Bus 001 Device 006: ID 0d8c:000e C-Media Electronics, Inc. Audio Adapter (Planet UP-100, Genius G-Talk)
...

```

To play audio through this device requires us to specify it as the hardware output. For example:

```
$ mpg123 -o alsa -a plughw:1,0 example.mp3
```

See also:

- [Audio Configuration](#)
- YouTube: [Raspberry Pi USB Audio](#)

## Speech output

There is a software package called "eSpeak" that can be installed by:

```
$ sudo apt-get install espeak
```

To speak a phrase we can enter:

```
$ espeak "This is a test"
```

To simply speak phrases, run espeak by itself. Each line entered followed by return speaks the result. The software can include languages. Run:

```
$ espeak --voices
```

to see a list of the available voices. To supply a different voice use the "-v <voice>" command. For example:

```
$ espeak -v en-sc "I can speak with a Scottish accent."
```

See also:

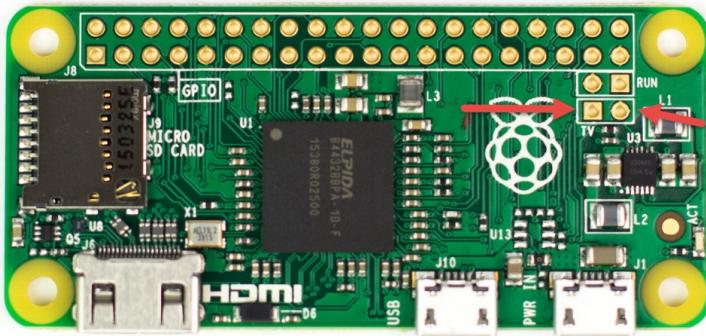
- [eSpeak text to speech](#)

## Ethernet

All the Pi models with the exception of the Pi Zero come with an Ethernet socket (RJ45) and associated controller logic. For a few dollars, one can buy a USB to Ethernet adapter to attach to a Pi Zero.

## Video output

All the Pis have the ability to output HDMI quality video signals. In addition, RCA composite video can also be externalized. On the Pi Zero, there is no supplied connection but there is the ability to wire an RCA connector. On the top of the board there are some solderable holes into which wires can be connected that can join to the RCA connector that can then be plugged into a TV. If the Pi Zero is booted without an HDMI connector, the RCA output will be used.



## Camera

Through USB, many USB attachable webcams can be used. However, the Pi has special hardware support for a dedicated camera module. This is a 5 megapixel device that is attached through a ribbon cable onto a dedicated socket on the Pi board itself. The camera is capable of recording at 1080p video speeds (1920x1080) at 30 frames per second using the H.264 encoding scheme. It needs special applications in order to access it. These include `Raspivid` for video capture and `Raspistill` for still image capture.



- [Raspistill](#)

## WiringPi

The hardware of the Pi can be accessed by low level memory access and bit twiddling. However, this is complex and error prone. It assumes that everyone has spent time and studied the address mapping of the various interfaces and protocols expected. What we really want is a higher level abstraction that provides significantly better ease of use. What we want is a library that encapsulates and hides the low level details from us. We want to call high level APIs and let the implementations of those APIs worry about the gory internal details and leave us with the nice high level illusion of the solution. This is where libraries such as WiringPi come into the story.

Without question, WiringPi is considered to be the gold standard of interfaces between applications and the hardware.

To install WiringPi, the instructions have us download the source, compile it and install the resulting materials. The WiringPi web site has full instructions. As of the time of writing, these are the steps to be followed:

```
git clone git://git.drogon.net/wiringPi
cd wiringPi
git pull origin
cd wiringPi
./build
```

The result will include the creation of the following artifacts:

- /usr/local/bin/gpio
- /usr/local/lib/libwiringPi.so
- /usr/local/lib/libwiringpiDev.so
- /usr/local/include/wiringPi.h
- .... others

Be cautious that there may be pre-existing or alternate WiringPi installations already present so when linking or building, make sure you know which ones you are working against. By default, Raspbian Jessie currently includes a copy of WiringPi at version 2.25.

To use WiringPi we would write a C language application and link our application with the WiringPi library. This library provides the implementation of the functions that the WiringPi interface shields us from. The exposed functions are significantly higher level than those that might otherwise have to be coded by ourselves.

See also:

- [WiringPi home page](#)
- YouTube: [Writing to GPIO pins in C using wiringPi on the Raspberry Pi](#)

## WiringPi setup

Before making the majority of WiringPi API calls, we have to first initialize our environment. We do that by calling **one** of the following APIs ... `wiringPiSetup()`, `wiringPiSetupGpio()`, `wiringPiSetupPhys()`, `wiringPiSetupSys()`. The difference between the various styles are the permissions necessary to run (either root or non-root) and the mechanism used to access the hardware internally and, **most importantly**, the numbering scheme for the pins.

See also:

- Pin numbering

## WiringPi GPIO

WiringPi provides a rich set of APIs for working with GPIO pins. First we have the function called `pinMode()`. This API allows us to define the direction and type of the pin that we wish to use. GPIO pins can be input, output or PWM output. If the pin is defined as input, then we can call `digitalRead()` to read the value of the pin. If the pin is defined as output, we can call `digitalWrite()` to write the value of a pin. At any time we switch the ping to a different mode. If the circuit needs, we can have a pin defined as output, write a value to it and then switch to input to read a value from it. To accommodate the notion of an input pin that might be floating, we can also define pull-up or pull-down resistors that are built into the Pi. We can use the function `pullUpDnControl()` to define an input pins pull-up or pull-down nature (or neither of these).

## WiringPi SPI

WiringPi has the ability to drive the Pi as an SPI master. The hardware of a Pi has two SPI channels called 0 and 1. To use, we first call `wiringPiSPISetup()`. This takes the channel to be used and the clock speed. The allowable clock speeds are between 500,000 (500KHz) and 32,000,000 (32MHz). The return from this calls is a file descriptor that will be used in subsequent calls.

Once configured, the `wiringPiSPIDataRW()` call can be invoked to send and receive data. The input to the call includes a pointer to a memory buffer and the number of bytes to use. The data pointed to by the buffer is written down the bus while data received is used to replace the data in the buffer. Since SPI states that each write of a byte has a simultaneous read from a byte, this works. Be aware that this call replaces what ever was present in the buffer before the call.

In order to use the SPI support of WiringPi we must include the special SPI header files:

```
#include <wiringPiSPI.h>
```

See also:

- `wiringPiSPISetup`
- `wiringPiSPIDataRW`

## WiringPi I2C

WiringPi has the ability to drive the Pi as an I2C master. In order to use the code in our applications, we must include a special I2C header file:

```
#include <wiringPiI2C.h>
```

We start with an API call to `wiringPiI2CSetup(address)`. The address supplied is the address of the I2C slave device with which we wish to communicate. The return from this call is a file descriptor or `-1` if an error was detected. Once we have a file descriptor, we can read and write bytes of data using `wiringPiI2CRead` and `wiringPiI2CWrite`. If we need to specify a register at the device we use the `wiringPiI2CReadReg8/wiringPiI2CReadReg16` and `wiringPiI2CWriteReg8/wiringPiI2CWriteReg16` APIs. For the 16 bit reads and writes, the order is LSB first.

Here is an example:

```
int fd = wiringPiI2CSetup(MY_DEVICE_ADDRESS);
uint8_t myData = wiringPiI2CReadReg8(fd, MY_REGISTER);
```

See also:

- `wiringPiI2CSetup`
- `wiringPiI2CWrite`
- `wiringPiI2CRead`
- `wiringPiI2CReadReg8`
- `wiringPiI2CReadReg16`
- `wiringPiI2CWriteReg8`
- `wiringPiI2CWriteReg16`

## WiringPi PWM

Hardware PWM is exposed through WiringPi. First, we set the pin mode of the desired output pin to be PWM. From there, we can use `pwmWrite()` to set the PWM value. The control of the PWM signal is performed using `pwmSetClock()` and `pwmSetRange()`. The "shape" of the PWM signal can be controlled with `pwmSetMode()`.

See also:

- `pwmSetClock`
- `pwmSetMode`
- `pwmSetRange`
- `pwmWrite`

## Memory mapped I/O

### Using sysfs to access GPIO

If we look in the directory called `/sys/class/gpio` we will find two interesting files. One is called `export` and the other called `unexport`. These files are write only ... which is an unusual thing for a file to be. Writing the character representation of a GPIO pin into the `export` file causes that pin to be exported while writing the same code into the `unexport` file causes that pin to be un-exported. But what does that mean?

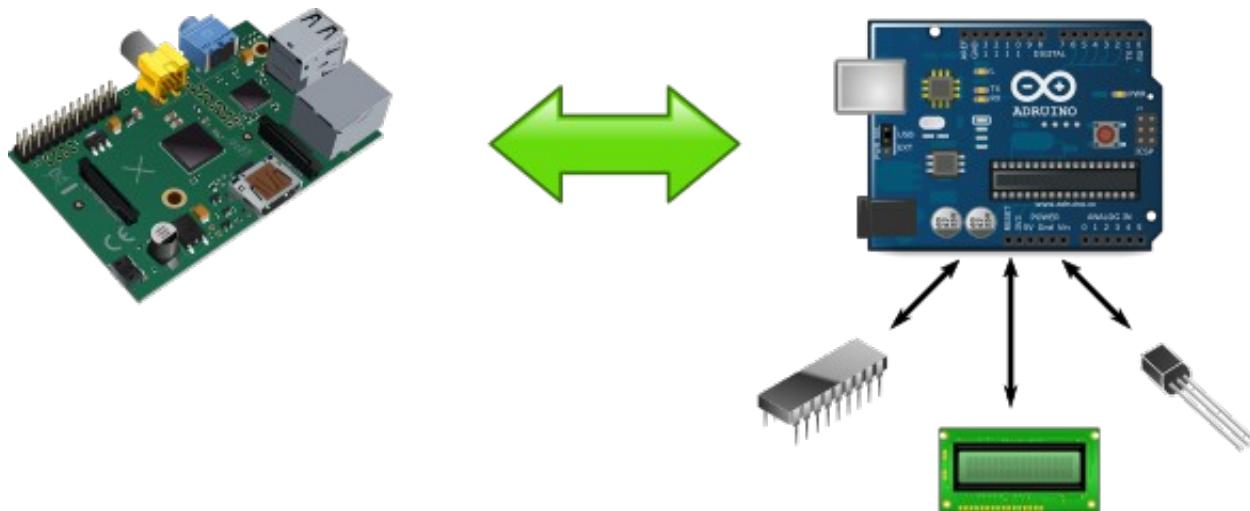
When a pin is exported, it means that we can interact with that pin as though it were itself a file. If we export a pin, then a new entry is created called `/sys/class/gpio/gpio<number>`. If we treat this as a directory and change into it, we find files called `active_low`, `direction`, `edge`, `power`, `subsystem`, `uevent` and `value`.

- `direction` – The value should be either "in" or "out". An "in" value means we are able to read the value from the pin while "out" means we can write a value to the pin.
- `value` – Reading from this file will return the current value of the pin assuming its direction is "in". If the direction is "out" then writing to this file will change the output value.

`/dev/gpiomem`

### Using an Arduino as a peripheral controller

The Arduino is a low end microprocessor that has a few kilobytes of RAM and a few kilobytes of flash memory. Small applications can be written to it which execute at about 20MHz. For some small embedded projects, the Arduino is a perfect processor for electronic peripheral control. We mention this because an interesting possibility is now available to us. On occasions where we might find it necessary, we can off-load hardware processing to an instance of an Arduino as a co-processor. For example, the Arduino has excellent and clean PWM signal generation. This may not be possible on a Pi as the PWM signal creation can be preempted by more urgent Linux kernel requests. Since an Arduino runs a solitary task, namely the program that we have placed within it, its execution pattern is 100% predictable.



If we use an Arduino as a co-processor, this does make our solution that much more complex but opens up a wealth of possibilities. We will have to choose a design for both the Pi and Arduino to electrically interact with each other. Choices for that could include UART, SPI or I2C. Once chosen, we would then have to design the application that would run natively on the Arduino which receives requests over the communication connection and performs those requests on behalf of the caller (the Pi).

An immediate note of caution on using the Pi and the Arduino together is that they have different voltages on their pins. The Pi uses 3.3V while the Arduino uses 5V. If you wired a 5V output from an Arduino into a 3.3V input on the Pi, you are very likely to damage your Pi. As such, be very, very careful.

When we look at the input and output capabilities of the Pi and Arduino combined, we see that we need to choose a mechanism for electrical connection between the two. Fortunately, we have a rich set of choices available to us including simple GPIO, UART, I2C and SPI. All of these offer strengths and weaknesses.

Behind the Arduino is a processor called the Atmega328p. This processor can be used directly without requiring a full Arduino board. This means that you can build production quality circuits on a PCB without having to mount a full Arduino board. The cost of the Atmega328p at the time of writing is about \$1.60 per unit from eBay. Now contrast that with the cost of the same IC mounted on an Arduino nano form factor (with crystal and micro USB programming port) for a cost of about \$2.30 per unit. Unless you are building production level systems in volume, I recommend working with nanos as sister processors for the Pi as opposed to wiring Atmega328p's directly.

We'll start by looking at simple GPIO. In that story, there is a one-to-one signaling between the Pi and the Arduino. On one side, a pin will be in output while on the other a pin will be in input with a connection between them. When the output pin is changed to be high or low, that can cause a signal to be propagated to the partner. If all we need is a single event indication, then this is likely to be the easiest. However, this is a poor solution if more than the indication of a change is needed as it does not provide any form of broader information propagation.

The next one we will look at is UART. We connect the UART from one side into the UART of the other and they can now send and receive asynchronously between each other.

Finally we will look at an I2C communication between the two.

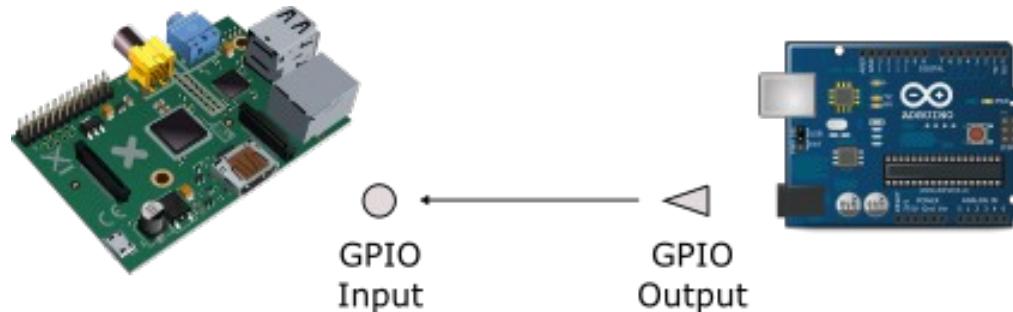
See also:

- [Raspberry Pi and Arduino Connected Using I2C](#)
- [Connect Raspberry Pi and Arduino with Serial USB Cable](#)
- [Connecting an Arduino to Raspberry Pi for the best of both worlds](#)
- [CONNECTING AN ARDUINO TO A RASPBERRY PI USING I2C](#)
- [I2C communication between a RPI and a Arduino](#)
- [raspberry pi to arduino spi communication](#)

## GPIO mapping between the Pi and Arduino

A simple mapping between an Arduino and a Pi can be achieved by wiring a GPIO output pin from the Arduino to an input pin on the Pi. Remember that the output voltage on the Arduino is 5V and the input on a Pi is 3.3V so you **must** use a voltage divider or level shifter between them.

In addition, you must also connect GND to GND between the two device.



On the Arduino side, we now need a program that sets the output signal of the pin as desired. For our test, we will simply toggle a pin from high to low and back to high again using a timer. The default Blink program supplied with the Arduino IDE will do just this for us.

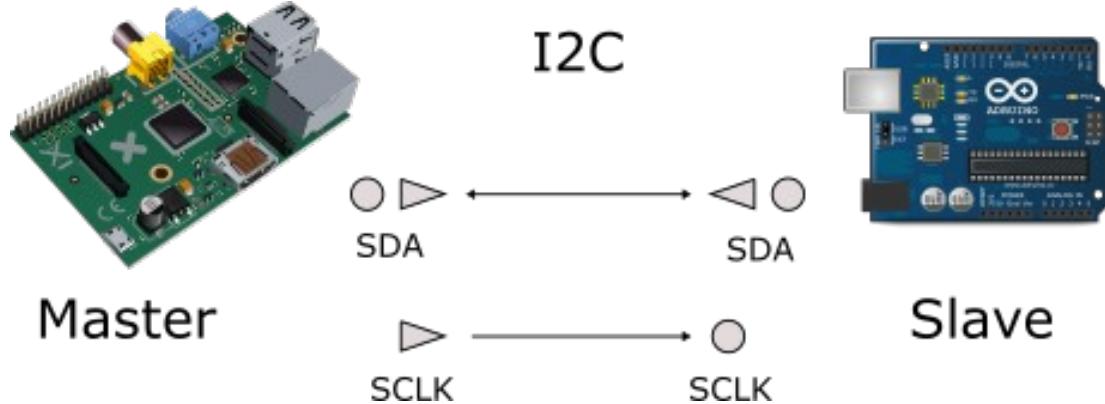
```
void setup() {  
    pinMode(13, OUTPUT);  
}  
  
void loop() {  
    digitalWrite(13, HIGH);  
    delay(1000);  
    digitalWrite(13, LOW);  
    delay(1000);  
}
```

We can test that the program is working by attaching an LED (with a resistor) to Pin 13 of the Arduino. It should blink periodically to show that work is being done.

## I2C mapping between the Pi and Arduino

When moving data between a Pi and an Arduino, we can also use the I2C interface. The Pi can only be an I2C master while the Arduino can be either an I2C master or an I2C slave. We can then write the Pi to the

Arduino. Notice that the SDA line is bi-directional and since the Pi is 3.3V and the Arduino is 5V, it is essential to have a logic level shifter between the two. We don't need this for the SCLK line as it is output only from the Pi.



For the Arduino sketch, we need to set it up as am I2C slave. We can use the supplied Wire library which implements I2C functions. To start, we call:

```
Wire.begin(address)
```

where address is the address that we want the Arduino I2C slave to appear. Choose an address that isn't in use and, ideally, you haven't seen used by other devices. The Pi i2cdetect command can be used on the Pi to show the addresses currently in use. In our example, we will use 0x10. Next we can register a data handler that will be invoked when the Pi transmits some data. This is registered with API called:

```
Wire.onReceive(receiveHandler)
```

The handler is a C function with the following signature:

```
void receiveHandler(int numBytes)
```

The handler function can invoke

```
Wire.read()
```

This will read the next byte which can be repeated for numBytes.

Since the I2C protocol requires the master to request new data, to be read from the I2C slave, we also need to respond to such requests. The API called:

```
Wire.onRequest(requestHandler)
```

registers a C function to be invoked with the Pi wishes to request data. The signature of the function is:

```
void requestHandler()
```

The handler function can invoke one of the:

```
Wire.write()
```

Functions to transmit response data.

Here is a sample sketch that illustrates some basic usage:

```

#include <Wire.h>

#define DEVICE_ADDRESS (0x10)

void receiveHandler(int numBytes) {
    Serial.print("receiveHandler - We have received some data: ");
    Serial.println(numBytes);
    int i;
    for (i=0; i<numBytes; i++) {
        int value = Wire.read();
        Serial.print("read: ");
        Serial.println(value);
    }
}

void requestHandler() {
    Serial.print("requestHandler - We need to return some data\n");
    Wire.write(0x99);
}

void setup() {
    Serial.begin(115200);
    Wire.begin(DEVICE_ADDRESS);
    Wire.onReceive(receiveHandler);
    Wire.onRequest(requestHandler);
    Serial.println("Ready!");
}

void loop() {
}

```

After deploying the sketch and running

```
$ i2cdetect -y 1
```

we see that the Arduino does indeed show up as a slave device:

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00:	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
10:	10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
20:	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
30:	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
40:	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
50:	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
60:	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
70:	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

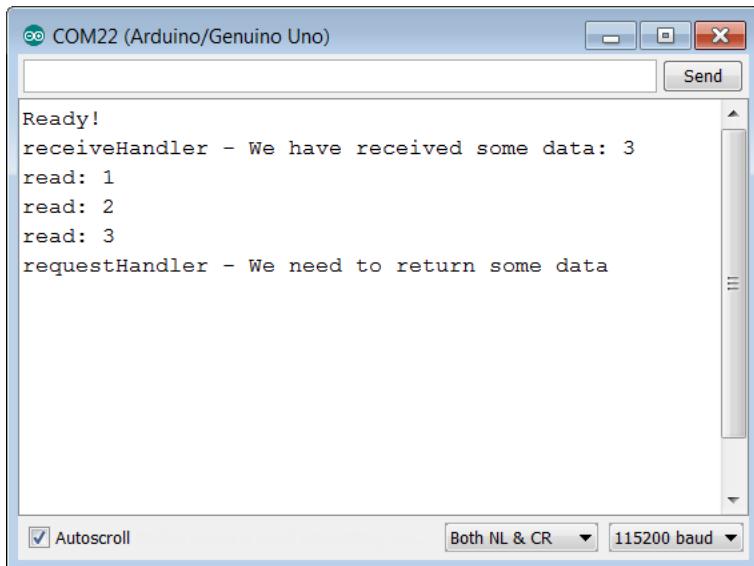
To see it work, open up an Arduino console and run:

```
$ i2cset -y 1 0x10 1 2 3 i
```

followed by

```
$ i2cget -y 1 0x10
```

In the Arduino console we will see:



```
Ready!
receiveHandler - We have received some data: 3
read: 1
read: 2
read: 3
requestHandler - We need to return some data
```

This shows that we received data from the Pi and returned data to the Pi on request.

See also:

- Arduino – [Wire library](#)
- I2C theory

## Integrating the ATMEGA 328P with your Pi

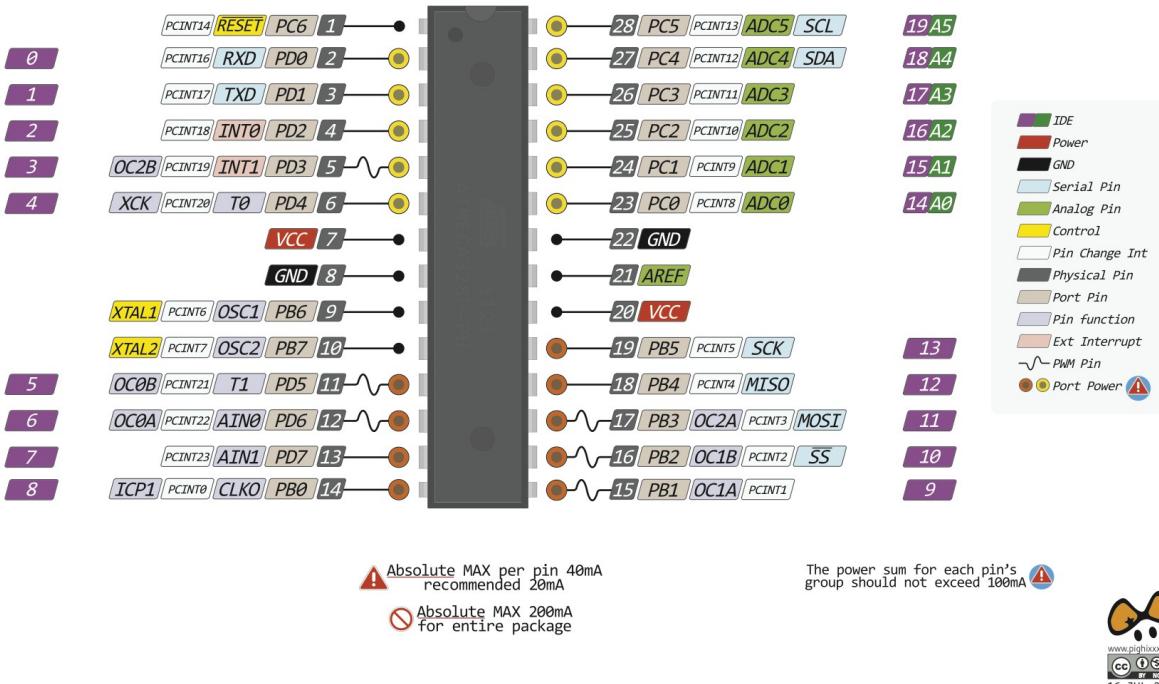
The Arduino comes in all shapes and sizes and we are most familiar with it as a series of boards we can plug into. However, if we dig deeper, we find that it is powered by a processor from Atmel called the ATMEGA 328P. Digging deeper, we find that we can actually obtain these processors very cheaply and include them directly in our circuits without the need for a full Arduino board. Let us now look at how we can go about performing this task.

Parts list:

- ATMEGA328P
- 16MHz crystal
- 2x 22pf capacitors
- LED and resistor

The following is the best pin-out available on the ATmega328 again from <http://www.pighixxx.com/>.

## ATMEGA328 PINOUT



See also:

- [YouTube: 1-Day Project: Build Your Own Arduino Uno for \\$5](#)
- [Arduino: Arduino on a Breadboard](#)

## Programming in general

One of the exciting aspects of the Pi environment is that one is not forced to work in any particular programming language when building Pi applications. One has a wealth of choices available. Languages available include C, C++, Java, JavaScript, Python, C# and more. Some are much more commonly used and prevalent than others and for some languages we even have multiple choices of compiler and environment to use. With choice comes decision. When you wish to undertake a project, which language do you use? Despite what many folks might argue, my vote is to use the one you "want" to use (where possible). The choice of one language over another can quickly break down into a religious battle where personal bias holds as much argument as technical merits.

An early decision that you will make is whether you develop on the Pi itself or on a PC. I am assuming that you have a Windows or Linux PC available to you. It is a safe bet that your PC is going to be much more powerful than your Pi. Your PC will be faster, have more RAM and have disk capacity to hold all the files you need. My recommendation is to develop your applications on a PC and either cross-compile or copy across for compilation the files and artifacts necessary. Cross compilation is the better option of the two. With cross-compilation, you edit and compile your source code on the PC and then test and run

it on the Pi. This gives you the power of a PC for development and insulates you from Pi crashes which are all too possible when working with low-level parts and electronics in general. Accidents can happen and if you short your Pi, then you will be restarting your environment from a fresh boot. The reduction in time that it takes to perform a build on a PC is also very dramatic.

## ELF structure

When we compile an application on a Linux environment for execution, the result is a file containing executable instructions. The format of the file is known as the Executable and Linkable Format (ELF) and contains a program that "thinks" it has an address space of  $0x0000\dots$  to  $2^n$  where n is either 32 or 64. The ELF file contains logical sections which contain data for distinct purposes. Among the sections are:

- `.text` – Instructions for execution
- `.data` – Initialized data. This could be data that can be read-and written to that has an initial value. For example:

```
int myVar = 123;
char *myString = "Hello World";
```

- `.bss` – Uninitialized data. This is data that can be read and written to that has a zero initial value.

```
static int myVar;
```

- `.rdata` – Read-only data. This is data that can be read from but not written to. For example:

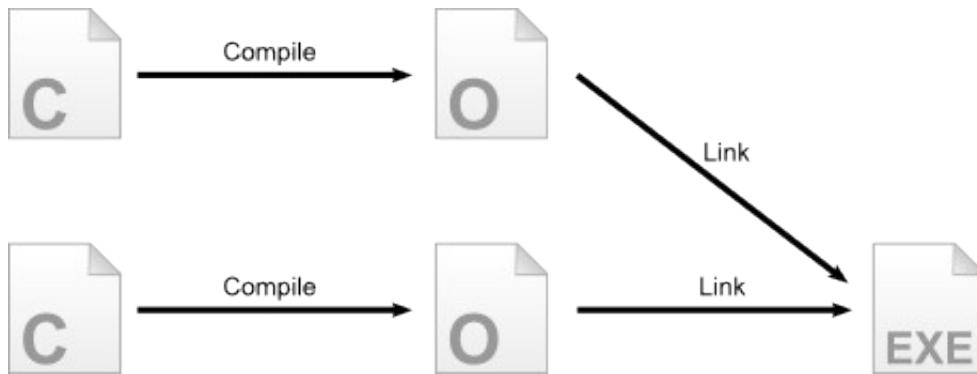
```
const int myVar = 123;
printf("Hello World!\n");
```

It would not be sensible to believe that large applications can be written in just one source file. Instead, applications typically are broken up into multiple source files where the code in one file typically performs some specific function. In order for this to be useful to us, we must also allow code that exists in one source file to invoke code that may exist in a different source file. This brings us to the concept of exposed symbols. When we compile a source file to its object file representation, the resulting object file can expose the symbols of functions and variables. By exposing these symbols, we can then find their memory locations during link time. The flip side of this is the concept that we can declare variables and functions in our source code as being defined elsewhere. This means that during compilation, these variables and functions will not be found but, at linking time, the linker will ensure that somewhere in the aggregate of files being brought together, all undefined symbols are resolved to some defined symbol. We can use the `nm` command to list the defined and undefined symbols within an object file or archive.

## Libraries of code

When you compile a C program, an object file is produced which is the compiled representation of your C source. It consists of instructions ready to be executed by the CPU. Only when the object file is linked with the C run-time does it become an executable. Now imagine that you have authored some code that

you want to use over and over again. Perhaps it is a general logging function or some specialized algorithm for drawing graphics on the display. When ever you want to re-use that code, you could include the source in your project, recompile it and link it with the rest of your application. However this is not an optimal idea for many reasons. First, you have broken your desire for encapsulation. You can go into the source file of the function and tinker with it. This can make maintenance a challenge as you will now have multiple distinct copies of the source floating around. You will also be wasting compilation time re-building the object file each time you compile.

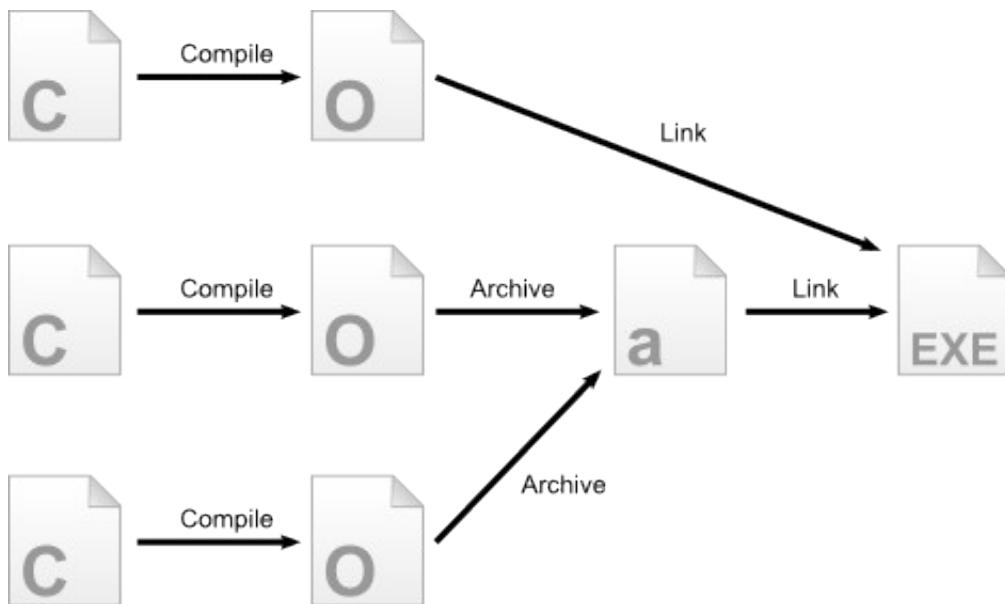


A better solution would be to compile the original source once and re-use the object file directly when needed. Now the object file becomes an encapsulation of the function and can't be contaminated with inadvertent tweaks. It is also likely that you won't just have one re-usable piece of function but instead have many. This means that you may end up with dozens of object files each of which need to be linked with your final application. There may also be dependencies between these object files and trying to remember which object files you need to pass to the linker can become a challenge.

Fortunately, Linux provides an easy and elegant solution in the form of the supplied archiver tool called `ar`. The `ar` tool allows us to take a set of object files and aggregate them together into a single file that we call an archive file. By Linux convention, the name of an archive file begins with "lib" (for library) and ends with the file suffix of ".a" for archive.

If you are familiar with taking multiple files and producing a ZIP file, this is a similar concept (but without compression).

The linker has knowledge of the meaning of an archive file and when you add that to the compilation steps, references to symbols (functions and variables) will be sought within the object files contained within the archive. The end result is that you can group together useful libraries of function into a single archive file and merely link with that "black box" to incorporate the function contained within into your own applications.



Having spoken about building your own libraries, you should also realize that this story also applies to libraries that you may leverage from others. If a new whiz-bang Open Source library becomes available, you can use the functions it exposes in your own applications without having to know how they work.

With C programming, we have the need to define functions and variables which are not locally present within the compiled app. For example, imagine that you want to call a function called `whizBang()` that is supplied in a library. If you simply call `whizBang()` from your code, you should end up with a compile time warning or error that the compiler doesn't know anything about `whizBang()`. The solution is to include a C language header file that contains the definition of the function. For example:

```
void whizBang(char *name, int age);
```

This gives the compiler all the knowledge it needs to validate that a call to the function is as expected. As such, when a library is provided there should be one or more header files that are accompanied with it that provide the relevant definitions to allow you to correctly use it.

## Static vs Dynamic libraries

So far we have been discussing what we will learn is called "static libraries". A static library is merely a short-hand mechanism for distributing object files that are linked into your own application. The code contained within these libraries is copied into your application. This means that your application becomes larger. This should not be a surprise as if you include new function in your application, you would expect it to become larger.

However, there is an alternative story that we need to consider and this called the dynamic library. The notion of a dynamic library is that when you link your application with a library, instead of the code contained within that library being copied into your application, your application merely remembers that before it can execute, the functions contained within that library need to be present. They are not **made** present by copying them into the executable at link time but are instead made present at run-time by

dynamically loading them from the archive file into memory at the start of execution. What this means is that your application is no longer just one executable file but is instead the executable plus the set of archive files that need to be loaded. The benefit comes with the notion that **many** applications may also be using the same library. Now instead of each application having a copy of the same library (and hence consuming more disk space), all the applications will share the same archive file. The result will be an overall saving of disk space when you have two or more applications that need the same function.

There is another massive benefit also available. When an application runs, it needs to have all its code loaded into memory to operate. If two applications are using the same static library, they in effect each have a copy of the same compiled function. With a dynamic library, the operating system knows if the dynamic library has already been loaded. If it has been loaded, it shares the in memory executable code of that library across multiple applications through a technique called memory mapping.

There is yet another benefit to shared libraries. Since the shared library is kept in a file distinct from the executable then if we replace the archive file with a new version then the next time the application is started, it will simply pick up that new version and start using it. What this means is that if our library is changed to incorporate bug fixes, those fixes become available to the new applications. With static libraries, we would have to relink the application as a whole to insert a copy of the new code into the application file.

By convention, shared libraries have the file suffix of ".so" as opposed to ".a" for static libraries.

## Building shared libraries

Building a shared library is not a complex task. When compiling your code, you need to add the additional compiler flag called "-fPIC". To aggregate the object files together into a shared library, we don't use the `ar` tool, instead we use the compiler again adding the "-shared" flag.

Let us look at the "-fPIC" flag. PIC is an acronym for "Position Independent Code". By specifying this flag, we are indicating to the compiler that it should produce code that is not dependent on where it will be loaded in the address space of a process. This allows different applications to load the same code at different memory locations since each application may have its own needs as to where code is found.

## General development tools

### ar

Work with the content of an archive file. An archive file can be thought of as a container for other files. When we are working with C source files and compile those to object files, we could end up with hundreds and hundreds of such objects to be managed. To make our lives easier, we can aggregate these object files together and place in an archive and then treat that archive as a single unit.

To create an archive and populate it, we can use the following:

```
ar crv <archiveName> [<members> ...]
```

See also:

- man(1) – [ar](#)

- [UNIX ar Examples: How To Create, View, Extract, Modify C Archive Files \(\\*.a\)](#)

## l~~d~~

The l~~d~~ tool shows the shared library dependencies upon an executable.

See also:

- man(1) - [l~~d~~](#)

## n~~m~~

The n~~m~~ command can be executed against an object file or an ELF executable to display the symbol tables contained within. Contrast this command with objdump. The objdump command displays memory layouts while the n~~m~~ command displays symbol table layouts.

Some of the more important flags are:

- --undefined-only – Display undefined symbols only.
- --defined-only – Display defined symbols only.
- --print-file-name – Prefix each symbol with the file it comes from.

See also:

- man(1) – [n~~m~~](#)

## objdump

The objdump command can be executed against an object file or an ELF executable. It dumps and disassembles these files with a wide variety of options.

Some of the more important flags are:

- --section-headers – Display a summary of the sections contained in the file.

See also:

- man(1) – [objdump](#)

## r~~a~~nlib

See also:

- man(1) – [r~~a~~nlib](#)

## readelf

See also:

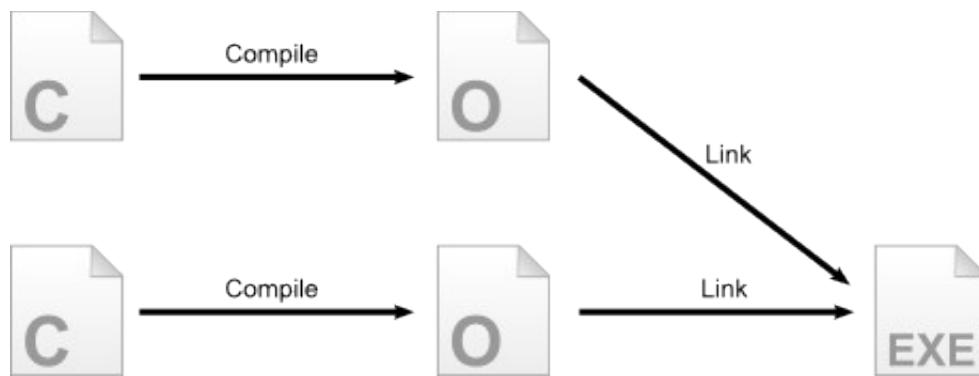
- man(1) – [readelf](#)

## C Programming

To be honest, we are not going to get far in the tasks we are looking to achieve without some skills in C programming and knowledge of the associated development tools. C is one step up from writing assembly language programs ... however it is a big step up which is goodness. C provides us a useful mix between high level programming with concepts such as named variables, looping structures, scopes, function calls and more while at the same time providing us all the capabilities we need to "twiddle" bits and bytes at arbitrary memory locations. Since Linux (and hence Raspbian) are themselves predominantly written in C, the linkage between our own custom applications and the operating system is very close. Also, should we get into writing "bare metal" applications, C and assembler are our primary choices.

## C Programming theory

We'll start with the basics. We use an editor to enter source C code. We save the source to files giving them names ending in ".c" which is by unanimous convention, the file type for a C language program. Since the C source code is not directly executable, we must compile that source into machine instructions the Pi can understand. We use a tool called a C compiler to perform that task. The Raspbian environment provides the popular GNU C Compiler (GCC) to perform that task. The result of compilation is a new file with a file suffix of ".o". This is what is called a relocatable object file. The nature of this is that it contains compiled code but that code is not yet ready to run. The addresses that are used for memory reads and writes as well as branch locations have not yet been fixed. Our next step is to perform a step called "linking" which takes one or more object files plus libraries and link edits them together which results in the final executable. Think of this like putting the pieces of a jigsaw puzzle together. This executable has had its addresses resolved and is ready to be run. The linking can again be performed with the same C compiler. The overall flow is as follows:



The primary value of going through this multi-step process is that we can break our large programs into multiple source files each of which results in its own object file. We can then link these object files together to produce our desired executable. The benefit of this is that if we change just one source file, we only have to recompile the single source file that changed to build its new object file. When we link edit the new object file with the already existing object files from previous compiles, we end up with our

new executable. This can save considerable time for very large applications as we don't have to continually recompile all our code for each small change.

A second and equally important reason for this process is that we can take object files and group them together by adding them to an archive file. An archive file typically starts with "lib" and ends with the file suffix of ".a". When we are linking our application, we now no longer need to name each of the object files but instead supply the archive file. The linker is intelligent enough to know which of the object files within the archive are needed for the executable and which can be ignored. This means that our resulting executables don't simply contain everything in the archive but only what is needed for execution.

This library mechanism serves another purpose that is possibly the secret sauce that makes everything work well. By placing object files in an archive, we can distribute that archive to other programmers and they can make calls to functions that are contained within the object files within the archive without having to know exactly what is contained within the archive or how they work. This provides the powerful concept of reusable code. If I were to write a set of new C functions of high quality and value, I could pack them up in a library and supply only the library and documentation. You could then use those functions merely by linking with the library. The Linux operating system itself provides a huge array of functions contained within libraries supplied by the operating system.

The C compiler does double duty when it comes time to perform the linking. When it is executed with different inputs, it performs the linking. To be clear, the C compiler is capable of both compiling a source file to an object file and is also capable of linking together object files with optional libraries to produce an executable.

If your program should consist of only a few C source files, you can combine these steps into one command which both compiles and links the application in one single step. This, however, is not a common circumstance. Instead what we usually wish to do is compile only the source files that have changed and then link everything together.

The C compiler will compile a source file to an object file when asked but it has no deeper intelligence to know about which files have changed or not changed. To efficiently compile applications together, such that only the changed source files are re-compiled, we need a tool which can detect changes to source files and re-compile only those that need re-compilation. Fortunately a tool called "make" is provided to perform exactly that task. The way make works is that we provide it a set of rules that it should follow to build an executable. To build an executable, we tell it the names of the object files that it should link together. For each object file we tell it the corresponding name of the source file that needs to be compiled to produce that object file. If we think this through for a while, we will see that to build an executable, we depend on the existence of a set of object files and those object files depend on the C source files. An object file is considered out of date if the source file has a time stamp of when it was last changed which is newer than the time stamp of when the object file was created. In addition, an executable is considered out of date if the file containing that executable has a time stamp older than any of the resulting object files.

This chain of thinking and examination of source file, object file and executable file time stamps is what is handled through the make tool. Since make doesn't know by itself which files constitute our project,

we must provide those instructions in a separate file which is called a Makefile. This Makefile contains the rules that make follows when executed. Each rule commonly takes two inputs. The target file we wish to create and the set of one or more files that are the dependencies to create that target. So if an executable is our target, its dependencies will be the set of object files and each object file will itself be a separate target which has the C source file as its dependency. When we ask make to build our executable, it works through the instructions contained within the Makefile to determine which targets are out of date. For each rule that detects that a target is out of date, a Raspbian command is supplied that is responsible for creating the target from the corresponding dependency files.

For example, if make determines that an object file is out of date with respect to its corresponding source file, the command would be a request to compile the source to produce the new object. If an executable is out of date with respect to its object files, the command would be a request to link together the object files.

## C tools for Pi

GNU C is the common C compiler environment. To determine the level of C installed, run `gcc -v` and look for the version indicator. For example, on my current Raspbian, the output ends with:

```
gcc version 4.9.2 (Raspbian 4.9.2-10)
```

## Simple compilation

To compile and run a program on the Pi we can use the gcc compiler. For example, let us look at the following simple C program:

```
<insert hello world.c>
```

If we then compile the program with:

```
gcc hello.c
```

we will find that we have an executable called `a.out`. If we run this with `./a.out`, we will see the message printed for us.

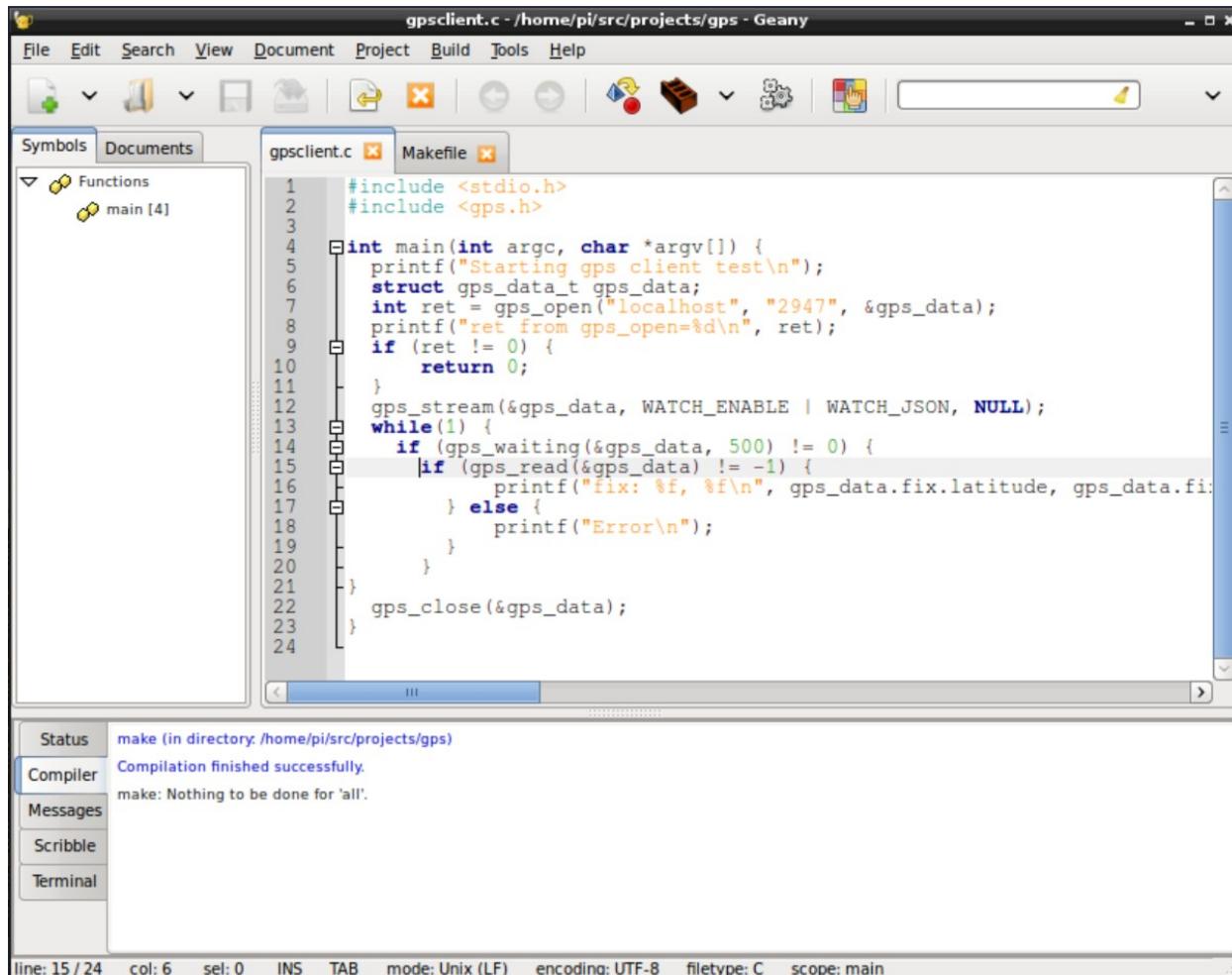
## Editing sources via Eclipse

There are a number of excellent Integrated Development Environments (IDEs) available. My favorite is called Eclipse and is freely available. Eclipse provides excellent editors as well as compilation tools and is available on both Windows and Linux. We can use Eclipse to edit source files for compilation on the Pi. To achieve this, I recommend mounting an exposed Pi file system and mount on your PC operating system. Next, on the PC, we can start an instance of Eclipse and point it to a workspace contained on the mounted Pi file system. What this means is that when we create Eclipse based projects and then create source files within those projects, the files will be actually saved on the Pi. This allows us to work with our source files using the full power of Eclipse. In addition to the source files, we will also want to create a Makefile to be used for building our solution.

If we also install the Terminal application into Eclipse, we can open a terminal window using SSH to our target directory and execute builds using make without ever leaving Eclipse.

## Editing sources on the Pi

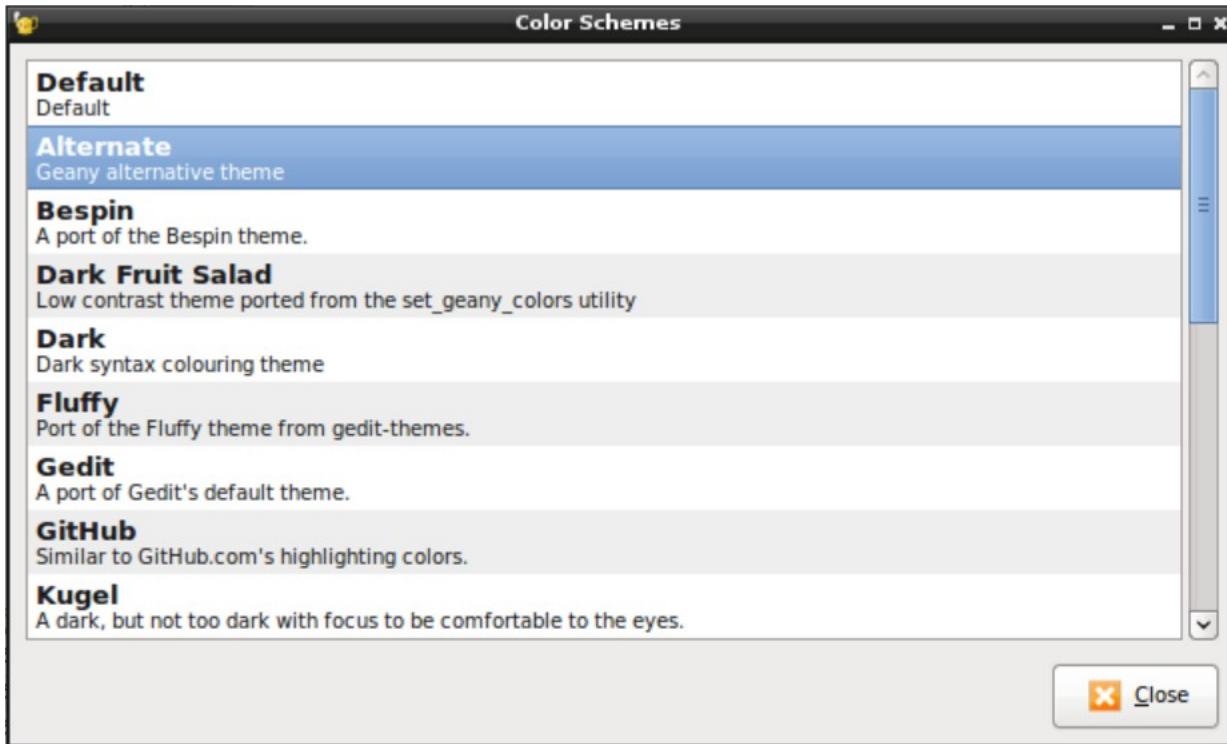
An X-Windows based editor called geany is available which runs on the Pi. From within this editor one can both edit multiple source files and invoke a Make build.



It can be installed with:

```
$ sudo apt-get install geany
```

I found the colors to be not at all to my liking but fortunately they were easily changed. A git hub project called [geany/geany-themes](#) is available. Once the color schemes are download and installed and geany restarted, a menu entry under View can be found called "Change Color Scheme". This brings up a dialog where we can select the theme we would like to have in place:



See also:

- [Geany home page](#)

## Cross compilation

When you wish to build an application that runs on the Pi you will likely write that application in a text editor and then run a compiler against it to produce executable code. The compiler will take the code you entered into the text file and transform it into executable instructions. These instructions will be understandable by the processor on which your executable will run. For example, for a Pi this is an ARM processor.

If you are compiling your code on a Pi then it makes sense that the code that is generated is itself ARM executable code. For example, if you run:

```
gcc myFile.c
```

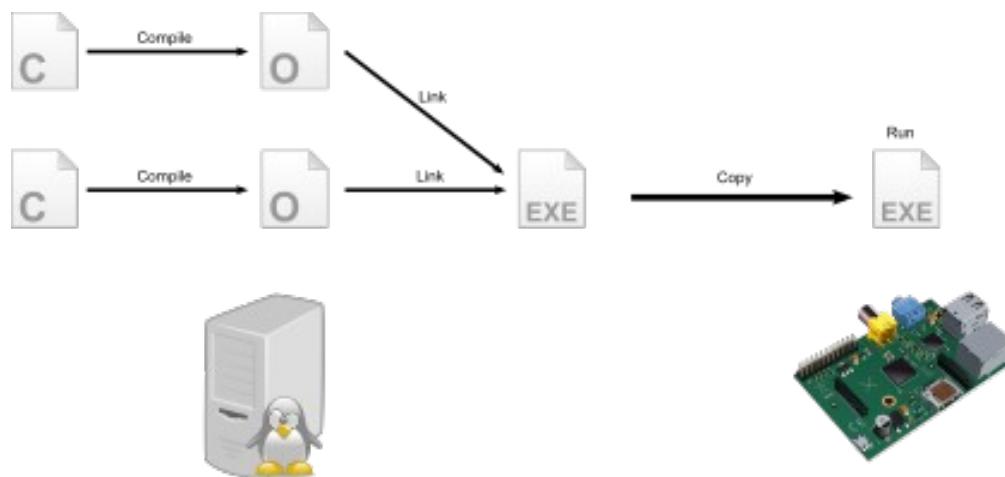
on a Pi, the resulting `a.out` executable file will contain ARM executable code.

Now consider running the same command on a Windows or Linux machine running on a PC. It too will generate an `a.out` file ... however the executable code contained within will be Intel processor instructions. If we copied that generated file to a Pi, it would be invalid and would not run. Conversely, if we copied an `a.out` generated on a Pi to a PC, it also would not run for the same reasons. This seems to say that we must compile our code on the same type of device/machine as that on which we are going to run it. However, this is not always desirable. Although the Pi is a good little computer, it is unlikely going to be anywhere near as powerful as your desk-side or laptop PC. This means that the act of compiling a lot of code on the Pi will take longer (sometimes considerably longer) than if we compiled

the same code on a PC. However, if you have followed the story, compiling the code on a PC produces PC code which is not what we want. What we want is code that will execute on a Pi.

Now we are ready to discuss cross-compilation. Cross compilation is the sequence of steps of running a compiler on one type of machine that generates executable code for another type of machine. Fortunately, we have that very thing available for us. We have access to cross compilers that run on Intel processors on Linux on a PC which generate ARM instructions that can be executed on a Pi. This means that we can compile code on a PC and run the resulting executables on a Pi.

Here is a diagram illustrating our task:



Now let us turn our attention to getting the tools needed to perform compilation of C source on an Intel Linux box to produce ARM executable binaries.

Note: If you are running a Windows PC (like I do) but want to do your Pi development on Linux (like I do) the you can run Linux on top of Windows! There is a free software product from Oracle called Virtual Box. Using this product is a breeze. What it does is allow us to run one operating system as a guest on another. For example, I run a 64 bit version of Ubuntu Linux under Virtual Box running on Windows 10. I am unable to detect any slowness and it recognizes all my USB devices and networks without incident.

To setup such a tool chain, perform the following tasks in a temporary directory:

```
$ git clone https://github.com/raspberrypi/tools
Cloning into 'tools'...
remote: Counting objects: 17851, done.
remote: Total 17851 (delta 0), reused 0 (delta 0), pack-reused 17851
Receiving objects: 100% (17851/17851), 325.16 MiB | 3.30 MiB/s, done.
Resolving deltas: 100% (12185/12185), done.
Checking connectivity... done.
Checking out files: 100% (15867/15867), done.
```

After completing this task, we will find a directory called `tools/arm-bcm2708`. Within there we will find 4 further directories:

- `arm-bcm2708hardfp-linux-gnueabi`

- arm-bcm2708-linux-gnueabi
- gcc-linaro-arm-linux-gnueabihf-raspbian
- gcc-linaro-arm-linux-gnueabihf-raspbian-x64

It is the last two that are of interest to us. Both of those are **intel** processor host to ARM target tool chains. One is for a 32bit Linux and the other for a 64bit Linux.

Note: To figure out which flavor you need, run `uname -a` and look for "**\*64**" in the result. If you have that, you are 64bit otherwise you are 32 bit. For example:

```
Linux kolban-VirtualBox 4.2.0-18-generic #22-Ubuntu SMP Fri Nov 6 18:25:50 UTC 2015
x86_64 x86_64 x86_64 GNU/Linux
```

Within the `gcc-linaro-arm-linux-gnueabihf-raspbian*` directory we will find a `bin` directory which contains the executables we need. These are the standard tool-chain commands prefixed with "arm-linux-gnueabihf-".

I recommend creating a directory called `/usr/local/dev` and copying the newly created `tools` directory into there. I then suggest an environment variable called:

```
RASPI_TOOLS_DIR=/usr/local/dev/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-
raspbian-x64
```

On the Intel Linux machine, create the following directories:

- `/mnt/pi/home` – This will be the mount of the pi user's home directory, mounted read / write.
- `/mnt/pi/root` – This will be the mount point of the root of pi file system, mounted read / write.

On the Pi, export the directories "/" and "/home/pi" using the following entries in `/etc/exports`:

```
/home/pi          *(rw,sync,no_subtree_check,insecure,fsid=1)
/                  *(ro,sync,no_subtree_check,insecure,fsid=2)
```

Note: Since the Pi "/" directory is exported read-only and the "/home/pi" is contained within the "/" directory, the default is to force that child structure to also be read-only. This is of course not what we want. The solution is the addition of the "fsid" parameters which provide a priority for resolution of properties such as read/write or read-only.

Now we have an environment setup, we can get to work.

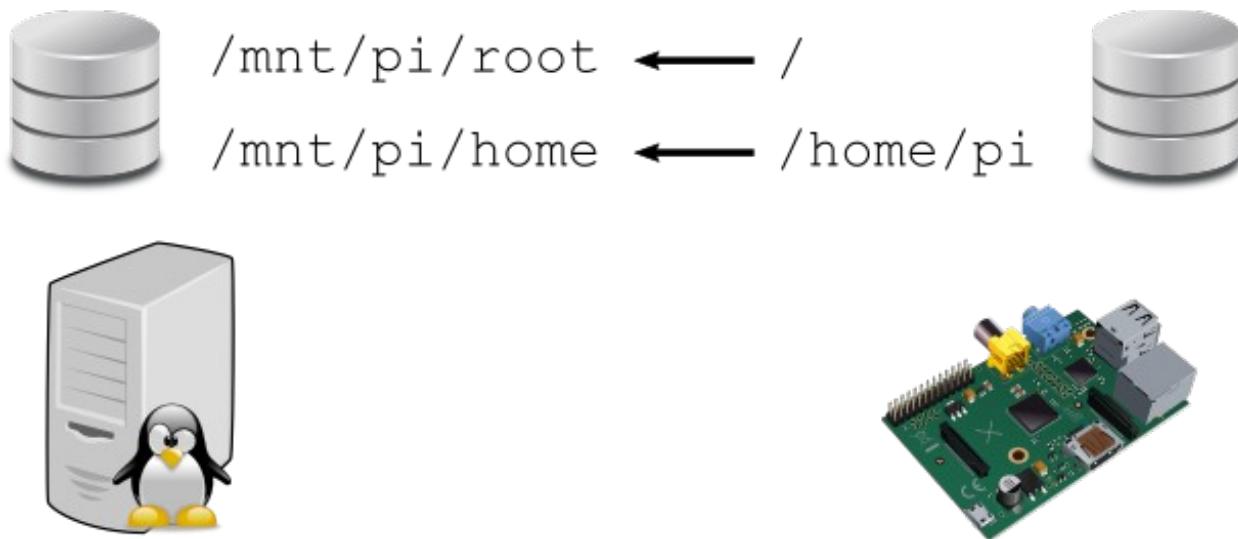
The notion is that we will use the Linux PC to compile our source files but will execute them on the Pi. Let us assume for a moment that we did compile them on the PC, we now need a very convenient way to get the compiled binaries to the Pi for testing. That is where the export of `/home/pi` comes into play. Having exported that directory structure from the Pi and mounted it on the Linux PC, when we have compiled our binary, we need only copy the result to `/mnt/pi/home` (or one of its sub-directories) and then it is immediately present on the Pi for testing. Nice and neat.

The next thing we want to consider is the notion that when we compile C source files, we understand that we take the C source code and the result is executable ARM instructions for execution on the Pi.

However if we think a little deeper, there is more to the story. When we compile a C source file, the chances are high that we will need to pull in C language header files. Either these header files don't exist on the Linux PC or they are simply different from what would really be expected if we were to **actually** compile on the Pi. The header files exist on the Pi in the form we need and not on the Linux PC. The solution to this is to mount the file system from the Pi which contains the needed header files and make those available on the Linux PC during the compilation step.

After compiling code but before having an executable, there comes the link edit step. This needs libraries that, once again, may only be found on the Pi. We can use the same technique that we used for header files for libraries.

On the Linux PC, mount the following:



Pi File System	Linux PC mount point
/	/mnt/pi/root
/home/pi	/mnt/pi/home

When using the GNU C compiler for the Pi on a PC, there will be an expectation that some GNU libraries for the ARM architecture will be found in `/lib` and `/usr/lib`. This is not the case on the PC. What we have to do is fool the PC into thinking that they are present when, in fact, they are to be found on the NFS mounted directories. As such, we need to create some symbolic links:

Target	Source
<code>/usr/lib/arm-linux-gnueabihf</code>	<code>/mnt/pi/root/usr/lib/arm-linux-gnueabihf</code>
<code>/lib/arm-linux-gnueabihf</code>	<code>/mnt/pi/root/lib/arm-linux-gnueabihf</code>

Here is a suitable Makefile:

```

APP=test1
OBJS=test1.o

#----
PI_ROOT=/mnt/pi/root
PRE=arm-linux-gnueabihf-
CC=$(PRE)gcc

# The directory into which the final application will be copied
# during a 'make install'.
OUTPUT_DIR=/mnt/pi/home/src/builds
INCLUDES=-I$(PI_ROOT)/usr/local/include
LIBDIRSS=-L$(PI_ROOT)/usr/local/lib
LIBS=-lwiringPi

all: $(APP)

$(APP): $(OBJS)
    $(CC) $(LIBDIRSS) -o $(APP) $(OBJS) $(LIBS)

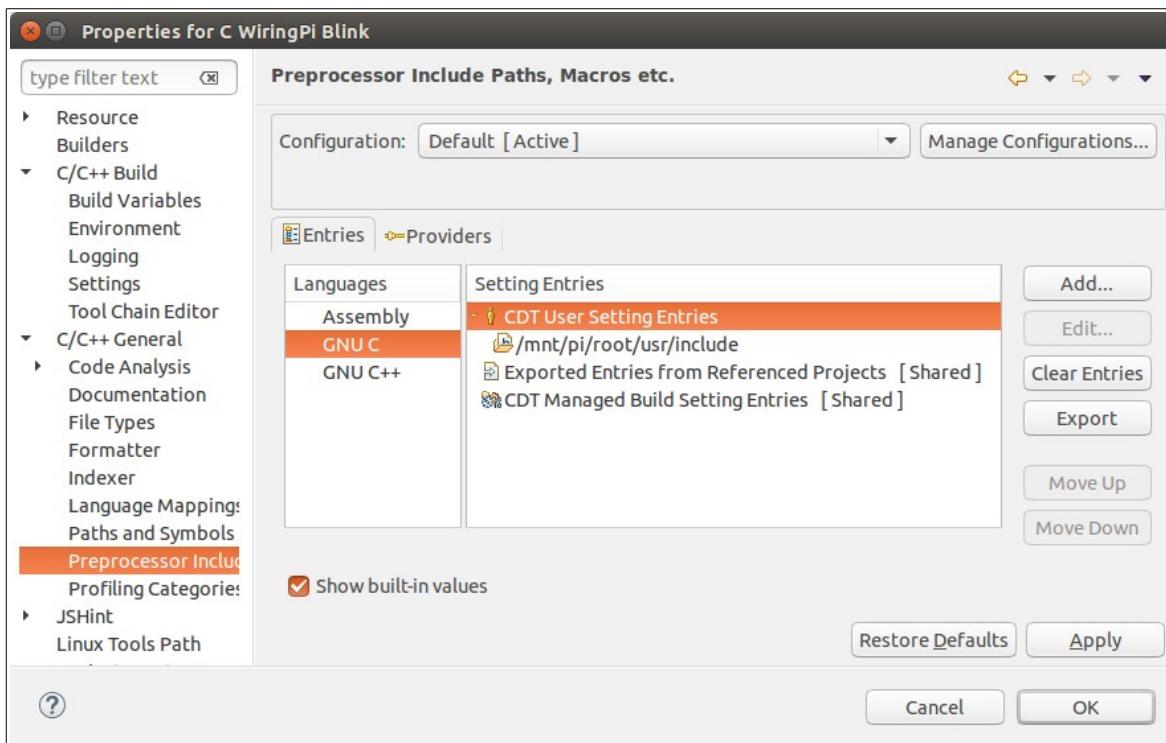
.c.o:
    $(CC) $(INCLUDES) -c $< -o $@

clean:
    rm -f $(APP) $(OBJS)

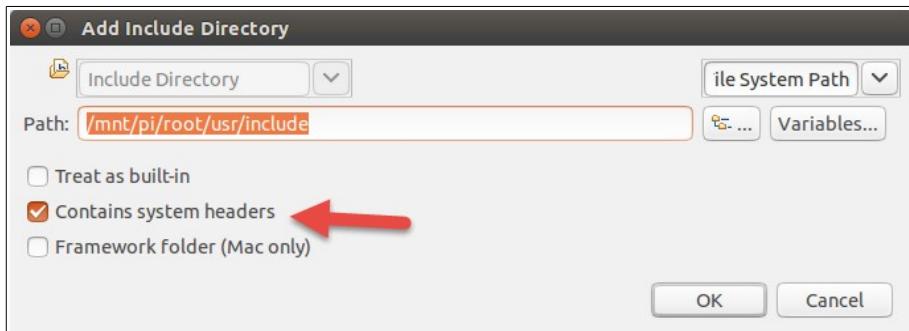
install: all
    cp $(APP) $(OUTPUT_DIR)

```

If we are using the Eclipse C Development Tools (CDT) on the PC to build our environment, we want to add some additional include directories on our C/C++ General → Preprocessor Includes:



For the directories, if they are to be found as system include files (those in <> brackets), check the appropriate box:



The directories that I recommend you add are:

- \${RASPI\_TOOLS\_DIR}/arm-linux-gnueabihf/include/c++/4.8.3
- \${RASPI\_TOOLS\_DIR}/lib/gcc/arm-linux-gnueabihf/4.8.3/include

Turning C code into object files is only part of the story when cross compiling. We also want to link our code with libraries and the chances are that those libraries can only be found on the target machine. Our choice then is to copy the libraries to our compilation machine or mount the file system from the target machine onto our compilation machine. A problem with the later technique is that if we mount directories, they won't be in the "usual" places expected by the compilation machine compiler.

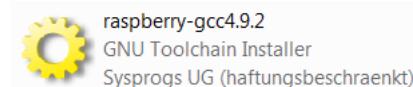
For example, on a Pi, the directory called `/usr/lib/arm-linux-gnueabihf` contains a variety of libraries we might need to link against. If we mount that on our compilation machine as `/mnt/pi/root/usr/lib/arm-linux-gnueabihf` then the compiler/linker doesn't know to look for them there. The solution is to add the linker flag called "`-rpath-link`" which takes an additional directory to search. For example, during linking we can add:

```
-Wl,-rpath-link /mnt/pi/root/usr/lib/arm-linux-gnueabihf
```

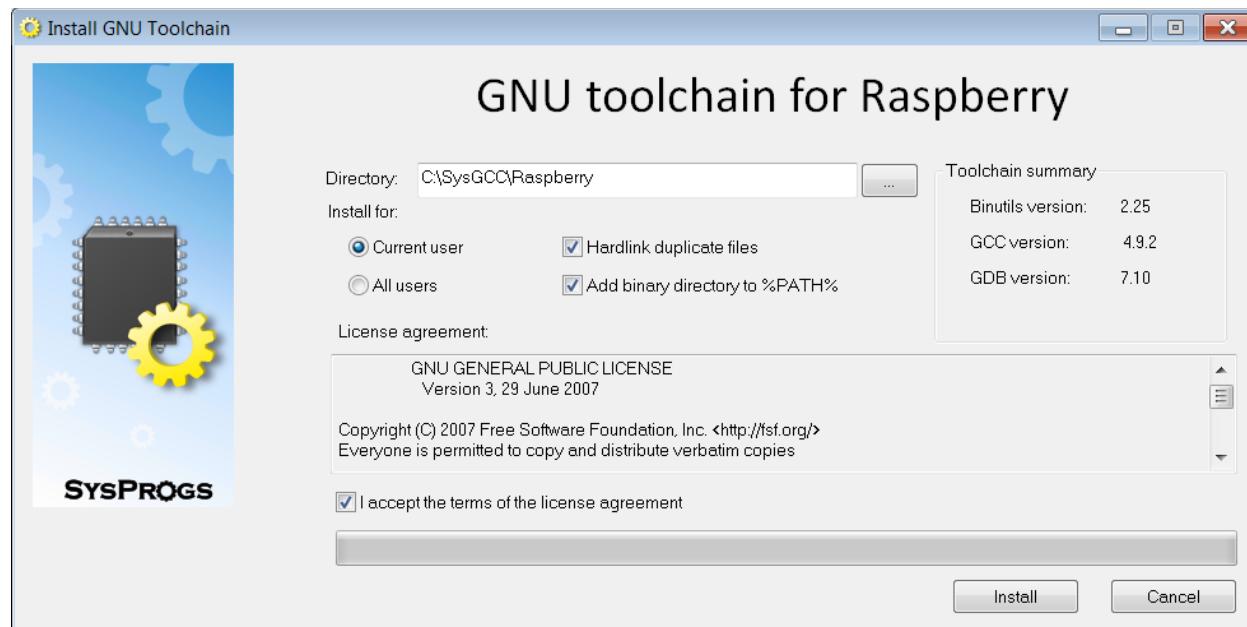
If we wish to install a tool chain for Windows, there is a fantastic website called <http://gnutoolchains.com/raspberry> that provides pre-built images. Visit the site and download the version we need for the Pi. Currently, the download page shows the following:

GCC	Compatible Raspbian image	Toolchain download link
4.9.2	2015-08-30-jessie-raspbian	<a href="#">raspberry-gcc-4.9.2.exe (320 MB)</a> 
4.9.1	(Manual update to Jessie distro)	<a href="#">raspberry-gcc4.9.1-r3.exe (244 MB)</a>
4.6.3	Universal (sysroot update required)	<a href="#">raspberry-gcc4.6.3-nosysroot.exe (18 MB)</a>
4.6.3	2013-07-26-wheezy-raspbian	<a href="#">raspberry-gcc4.6.3.exe (111 MB)</a>

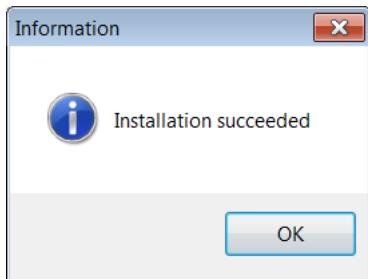
Once downloaded, we can run the installer.



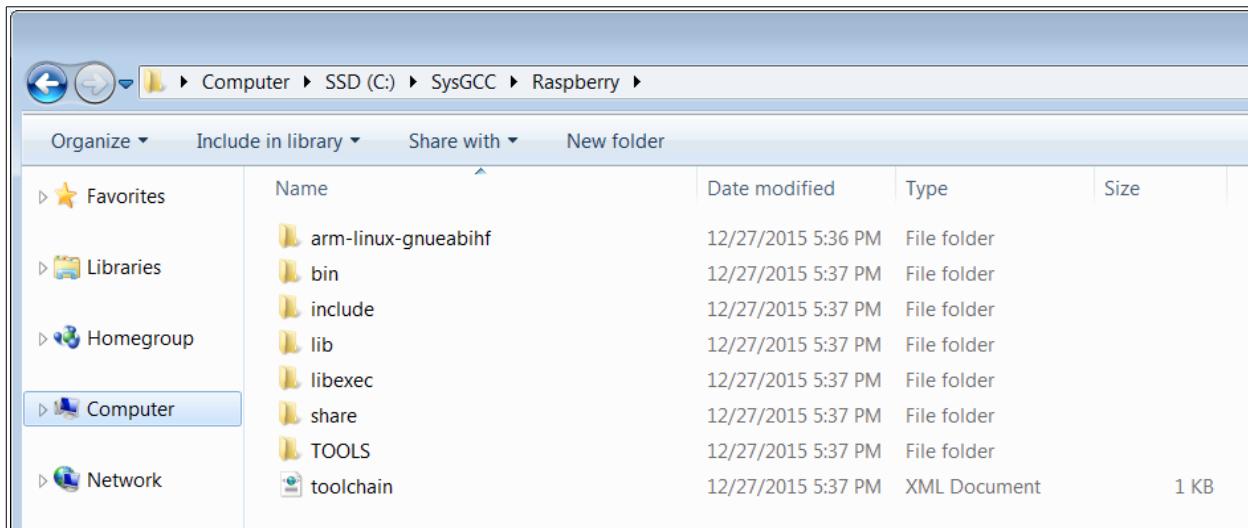
When run, an attractive installer is displayed:



On completion we get a success dialog:



At the conclusion we have the following directory structure:



with the `bin` folder of this directory added to our windows path.

See also:

- raspberrypi.org: [Kernel building](#)
- elinux.org: [Raspberry Pi Kernel Compilation](#)
- github: [raspberrypi/tools](#)
- [gnutoolchains.com](#) – Source of tool chains for GNU for windows

## Makefiles

Books have been written on the language and use of Makefiles and our goal is not to attempt to rewrite those books. Rather, here is a cheaters guide to beginning to understand how to read them.

The core notion here is that an application you may be building is built from a variety of source files. These source files can be compiled together to form the resulting executable. For example, if you have source files `a.c`, `b.c` and `c.c` you could compile them using:

```
cc -o myApp a.c b.c c.c
```

That was easy. Now imagine that you stepped away for a couple of weeks and came back and you had forgotten what had to be compiled or which libraries needed to be included in the linking. To solve this, you could capture your compilation commands in a shell script and re-run the script when needed. At a basic level, you can think of the Makefile system as performing that task for you. It remembers what you have to do in order to build your application.

However, there is a deeper value. Imagine you now edit one of the source files, for example b.c. Your recipe for rebuilding the application is to run the script ... and the script says to recompile **all** the source files. As such you are wasting time recompiling source files that were previously compiled. Here is where the Makefile story comes into its own. With a `Makefile`, you declare the application you wish to build, and the object files that need to be linked together to build that application. You also provide rules that declare how to convert a source file to an object file (by compilation). The `Makefile` then tracks what is needed to be done to build your application and only re-compiles source files if they have been changed since last time you compiled them.

Here is an example `Makefile`:

```
myApp: a.o b.o c.o
    cc -o myApp a.o b.o c.o

.c.o:
    cc -c $<
```

The way to read this is "myApp is made up from a.o, b.o and c.o. Once those are built, execute the compile statement to link them together". There is also a rule that says "If you find a file called ?.o, look for a file called ?.c and if it exists and it is newer than the .o file then compile it to produce a new .o file.". As you may be starting to see, the Make system is incredibly powerful ... but it is also full of features and functions ... some of which can get tricky if you haven't spent a lot of time studying them.

A general rule in a make file has the form:

```
target: prereqs ...
       recipe ...
```

Variables are defined in the form:

```
name=value
```

We can use the value of a variable with either `$(name)` or `$(name)`.

Another form of definition is:

```
name:=value
```

Here, the value is locked to its value at the time of definition and will not be recursively expanded.

Some variables have well defined meanings:

Variable	Meaning
CC	C compiler command
CXX	C++ compiler
AR	Archiver command
LD	Linker command

OBJCOPY	Object copy command
OBJDUMP	Object dump command

We can use the value of a previously defined variable in other variable definitions. For example:

```
XTENSA_TOOLS_ROOT ?= c:/Espressif/xtensa-lx106-elf/bin
CC           := $(XTENSA_TOOLS_ROOT)/xtensa-lx106-elf-gcc
```

defines the C compiler as an absolute path based on the value of a previous variable.

Special expansions are:

- \$@ - The name of the target
- \$< - The first prereq

Comments are lines that start with an "#" character.

Wildcards are:

- \* - All characters
- ? - One character
- [...] - A set of characters

Make can be invoked recursively using

```
make -C <directoryName>
```

Imagine we wanted to build a list of source files by naming directories and the list of source files then becomes all the ".c" files, in those directories? How can we achieve that?

```
SRC_DIR = dir1 dir2
SRC := $(foreach sdir, $(SRC_DIR), $(wildcard $(sdir)/*.c))
OBJ := $(patsubst %.c, $(BUILD_BASE)/%.o, $(SRC))
```

The puzzle

Imagine a directory structure with

```
a
    a1.c
    a2.c
b
    b1.c
    b2.c
```

goal is to compile these to

```
build
    a
        a1.o
        a2.o
    b
```

```
b1.o  
b2.o
```

We know how to compile x.c → x.o

```
MODULES=a b  
BUILD_BASE=build  
BUILD_DIRS=$(addprefix $(BUILD_BASE) /,$(MODULES))  
SRC=$(foreach dir, $(MODULES), $(wildcard $(dir)/*.c))  
# Replace all x.c with x.o  
OBJS=$(patsubst %.c,%.o,$(SRC))  
  
all:  
    echo $(OBJS)  
    echo $(wildcard $(OBJS)/*.c)  
    echo $(foreach dir, $(OBJS), $(wildcard $(dir)/*.c))  
    echo "SRC: " $(SRC)  
  
test: checkdirs $(OBJS)  
    echo "Compiled " $(SRC)  
  
.c.o:  
    echo "Compiling $(basename $<)"  
    $(CC) -c $< -o build/$(addsuffix .o, $(basename $<))  
  
checkdirs: $(BUILD_DIRS)  
  
$(BUILD_DIRS):  
    mkdir -p $@  
  
clean:  
    rm -f $(BUILD_DIRS)
```

Makefiles also have interesting commands:

- \$(shell <shell command>) - Run a shell commands
- \$(info "text"), \$(error "text"), \$(warning "text") – Generate output from make

To find a set of files and delete them given their suffixes we can use

```
find . -type f \(-name "*.o" -o -name "*.c"\) -delete
```

See also:

- [GNU make](#)
- [Makefile cheat sheet](#)

## Debugging C

When you write an application, you should always test it to make sure it works. However, if it doesn't work as expected or crashes, then you need to perform debugging. One way to debug is to include

`printf()` statements within your code and explicitly log data as sections of your code are reached, however that is not very efficient and, once you have resolved your problem, you then have the issue of removing or commenting out the explicit debug statements you added. A better way would be to use a debugging tool. Fortunately, we have one at our disposal.

The tool is called `gdb` which is short for the GNU Debugger.

The `gdb` tool is a command line debugger for C and other native compiled applications. It will not work with interpreted or JVM based programs such as JavaScript, Python or Java. Command line based debuggers are not everyone's desire so there are also GUI wrappers for `gdb` that provide all the GUI environment you may want. There is a split screen window/text version built into `gdb` when invoked with the `"-tui"` option. Richer debuggers come in the form of `ddd` and `Eclipse`.

There are a number of commands in `gdb` that are of very high importance, these are:

- `break` – set a break-point.
- `clear` – remove a break-point.
- `next` – step to the next statement stepping over function calls.
- `step` – step to the next statement stepping into function calls.
- `continue` – resume execution until the next break-point is reached.
- `print` – display the value of a variable.
- `watch` – break when the value of a variable changes or an expression becomes true.
- `frame` – switch to a particular stack frame.
- `up` – move up a stack frame.
- `down` – move down a stack frame.
- `backtrace` – display the stack frames.
- `run` – run the program with optional parameters.
- `help` – display help information.
- `finish` – continue till we end the current nested function.

In order to use `gdb`, you must prepare your application for such a task. This is done by compiling with the `-g` flag. This causes the symbol table to be retained or generated for the application. Without this information, `gdb` has no means of mapping source files and line numbers to the compiled code.

There are a rich set of options that can be entered within a `gdb` session. You can save these options in a text file and have them applied when `gdb` starts by supplying the `"-command <file>"` parameters when starting the debugger.

A GUI wrapper is available for `gdb` called `ddd`. When we run this in an X-Windows environment, a new UI appears from which we can run our `gdb` session while also looking at the source.

Beyond using debuggers, we have other tools at our disposal. The first one I'd like to talk about is the notion of assertions. An assertion is code within your logic that tests the validity of an expression. If the expression is true, then the program can continue. If the expression is false, then we have failed in an assertion that we believed to be true at that point in the program. The use of assertions can be very powerful and can quickly tell us if something we thought was correct wasn't. To use assertions, include "assert.h" and then insert:

```
assert(expression)
```

into your code. When reached, the expression is evaluated and we only get to move forward if it is true.

See also:

- [GDB: The GNU Project Debugger](#)
- DDD – [Data Display Debugger](#)
- [Remote Debugging of Raspberry Pi Applications From Eclipse](#)

## Debugging on a remote Pi

Debugging an application on a desktop is an easy proposition. We have a mouse, keyboard and a monitor. There isn't much of a challenge to attaching a debugger to see what is going on. However, debugging on a Pi can at times be more of a challenge. There are times when our Pi is running "headless" with no keyboard or screen. How then can we debug such a device? The answer is that we can use a network connection to access the application remotely through gdb. We run gdb on our PC and it connects via a network to the remote Pi and the application.

On the Pi, we run a program called `gdbserver`.

Let us work through an example.

On the PC, we will have a simple C program that looks as follows:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Starting Test debug program\n");
    int i;
    for (i=0; i<10; i++) {
        printf("Loop = %d\n", i);
    }
    printf("Ending the program!");
}
```

We compile it with:

```
arm-linux-gnueabihf-gcc -g -o test1 test1.c
```

This generates an executable for the Pi called `test1`. We next copy this file over to the Pi machine.

On the Pi, we then run:

```
gdbserver localhost:10000 ./test1
```

This starts the `gdbserver` listening on port 10000 ready to run the `test1` application.

We will be told:

```
Process ./test1 created; pid = 3187
Listening on port 10000
```

Back on the PC, we are now ready to run gdb. We run it with:

```
arm-linux-gnueabihf-gdb ./test1
```

This loads gdb as shown next:

```
GNU gdb (crosstool-NG linaro-1.13.1+bzr2650 - Linaro GCC 2014.03) 7.6.1-2013.10
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=x86_64-build_unknown-linux-gnu --target=arm-linux-
gnueabihf".
For bug reporting instructions, please see:
<https://bugs.launchpad.net/gcc-linaro>...
Reading symbols from /home/kolban/pi/src/Workspace/C_Test_Debug/test1...done.
(gdb)
```

Now we connect to the remote gdbserver running on the Pi:

```
(gdb) target remote raspi:10000
Remote debugging using raspi:10000
Reading symbols from /usr/local/dev/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-
raspbian-x64/arm-linux-gnueabihf/libc/lib/ld-linux-armhf.so.3...(no debugging symbols
found)...done.
Loaded symbols for /usr/local/dev/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-
raspbian-x64/arm-linux-gnueabihf/libc/lib/ld-linux-armhf.so.3
0x76fcfd60 in ?? ()
    from /usr/local/dev/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian-
x64/arm-linux-gnueabihf/libc/lib/ld-linux-armhf.so.3
(gdb)
```

On the Pi gdbserver will report a new connection:

```
Remote debugging from host 192.168.1.6
```

Back on the PC we can set a break point in main with:

```
(gdb) b main
Cannot access memory at address 0x0
Breakpoint 1 at 0x8458: file test1.c, line 4.
(gdb)
```

And now we are ready to step through our source code using standard gdb commands. Entering cont to continue will cause the debugger to stop in main.

```
(gdb) cont
Continuing.

Breakpoint 1, main (argc=1, argv=0x7efff7d4) at test1.c:4
```

```
4      printf("Starting Test debug program\n");
(gdb)
```

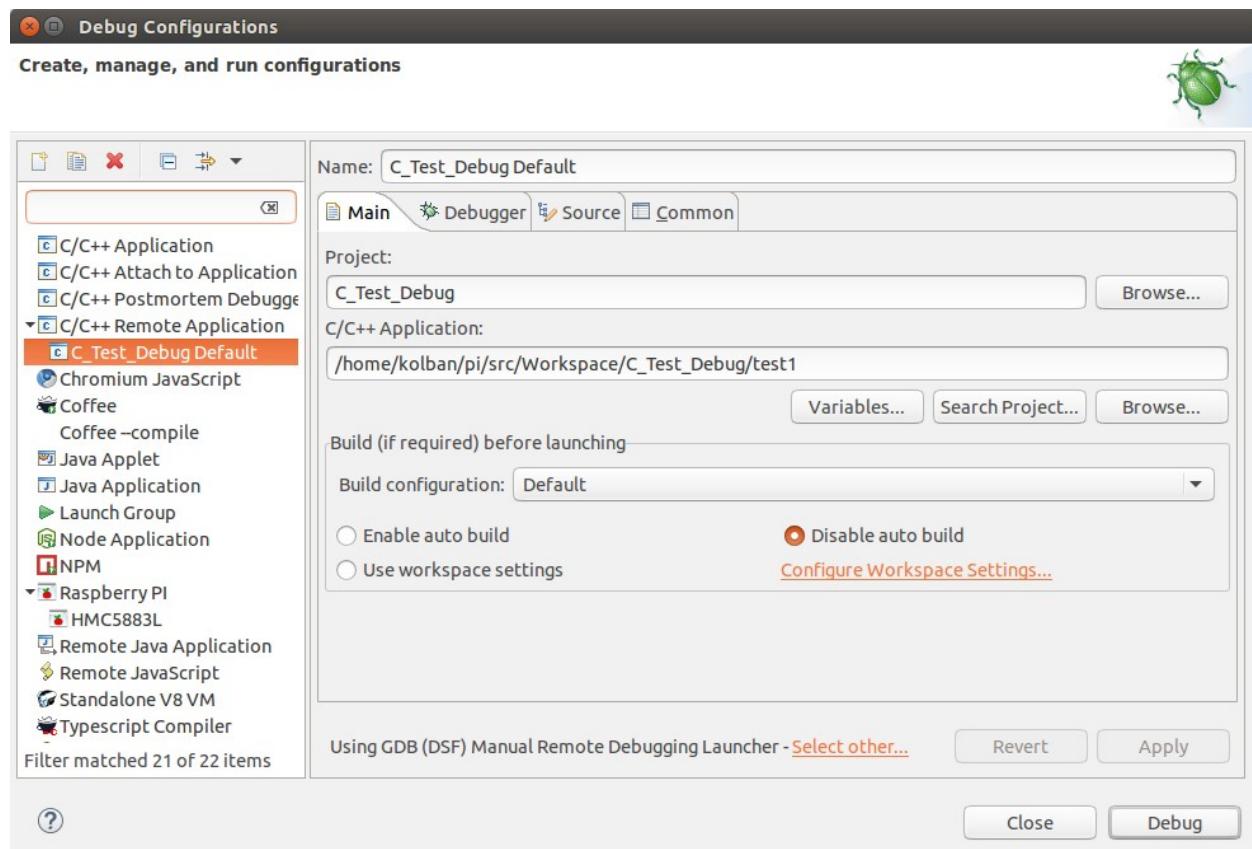
When we quit the PC debugging session, that also ends the remote gdbserver.

We can restart the debugging session without quitting the debugger by running the command:

```
jump _start
```

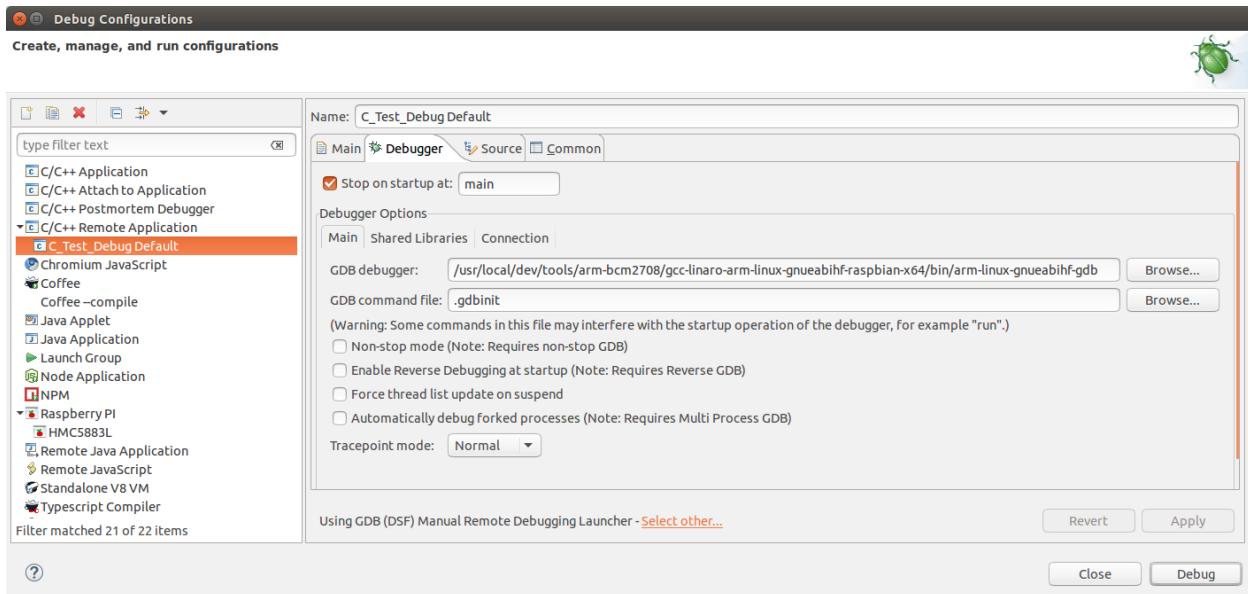
In theory, we can also use Eclipse for remote debugging.

1. Use Eclipse to host your project.
2. Add a new Debug Configuration

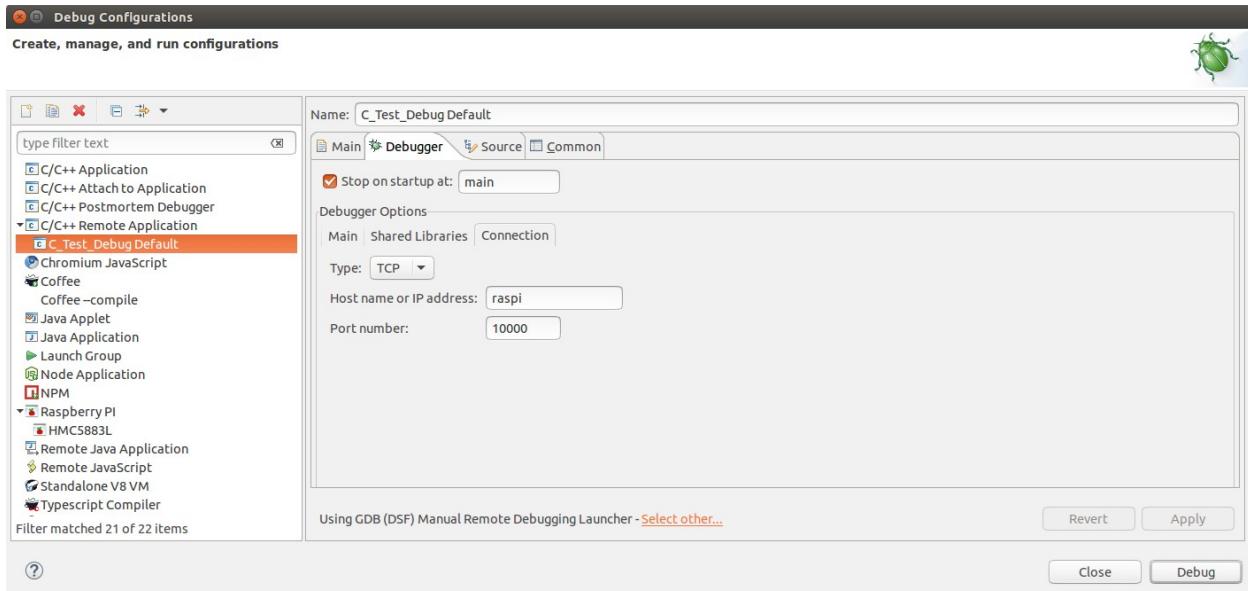


Remember to set the local binary for the application and select Manual Remote Debugging Launcher.

3. Set the debugger to use:



and



#### 4. Run the debugger.

Unfortunately, as of the time of writing, it fails.

On the Pi which is running gdbserver, we get:

```
*** Error in `gdbserver': munmap_chunk(): invalid pointer: 0x001d4708 ***
```

An alternative GUI wrapper for gdb is called ddd. We can specify the executable for the gdb debugger using the --debugger <path> command line flags. This is important because if we are running ddd on a PC and we are remotely debugging an application running on the Pi, we will need to use the gdb version that understands ARM instructions.

See also:

- [How to Debug Programs on Remote Server using GDBServer Example](#)

## C++ Programming

The C language is very powerful but it lacks some of the more modern paradigms we expect from modern languages such as objects. This is where the C++ language comes into play. C++ programming is very similar to C with extensions that allow us to leverage classes.

We introduce a class with the syntax:

```
class <className> {  
}
```

within the declaration, we define members and functions which are either public or private. Items which are public are prefixed with the `public` keyword while items which are private are prefixed with the `private` keyword. For example:

```
class MyClass {  
    public:  
        void func1();  
        int a1;  
  
    private:  
        void func2();  
        int a2;  
}
```

The function `func1()` and member `a1` are public while `func2()` and `a2` are private.

The class is usually declared in a header file while the actual implementation of member functions are included in C++ source files with the convention of ".cpp" as the file suffix.

Members are defined to be for the class by prefixing them with the class name. For example:

```
void MyClass::func1() {  
    // body here  
}
```

We can also define constructor functions which are called when an instance of the class is created. A constructor has the same name as the class and no return type and also defined in the public section.

```
class MyClass {  
    public:  
        MyClass();  
}
```

and in the implementation as:

```
MyClass::MyClass() {  
    // body here  
}
```

## Calling functions

Should we need to call a global function, we can prefix the name of the function with "::" to ensure that it is scope correctly. For example:

```
// some code  
::printf("Hello World\n");  
//
```

## Install the ARM compiler tool-chain

A standard C/C++ compiler on an Intel based Linux PC doesn't know how to compile to generate ARM executable code that will run on a Pi. As such, we need to download the set of tools that run on an Intel Linux machine but generate ARM code. Fortunately, the good folks over at Raspberry Pi have packaged these tools for us. The repository for the tools can be found on [Github](#). We recommend creating a directory at `/user/local/dev` and cloning the repository there.

```
$ sudo mkdir /usr/local/dev  
$ cd /usr/local/dev  
$ git clone https://github.com/raspberrypi/tools  
...  
$
```

Our last step on the PC is to associate the new tool-chain with the tools to be used by the Arduino IDE. We do this by creating a link between the Arduino IDE directories and the newly extracted tools.

Change directory to:

```
$ cd /usr/local/arduino-1.6.7/hardware/RaspberryPi/piduino/tools
```

Now create a symbolic link from arm-linux-gnueabihf to /usr/local/dev/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian-x64

```
$ sudo ln -s /usr/local/dev/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian-x64 arm-linux-gnueabihf  
$
```

Note: For windows the command will be:

```
C:> mklink /d arm-linux-gnueabihf c:\SysGCC\Raspberry
```

Now we can move on to the Pi configuration.

## Java Programming

To install a Java JDK on the Pi, we must first download an installation. The latest installation is jdk-8u65. When downloading for the Pi, realize that the Pi processor is an ARM v7. You may find downloads for other processors at the Oracle Java download page and care should be taken to obtain the correct version. At the time of writing, the following was the correct version:

Java SE Development Kit 8u65		
You must accept the Oracle Binary Code License Agreement for Java SE to download this software.		
Thank you for accepting the Oracle Binary Code License Agreement for Java SE; you may now download this software.		
Product / File Description	File Size	Download
Linux ARM v6/v7 Hard Float ABI	77.69 MB	<a href="#">jdk-8u65-linux-arm32-vfp-hflt.tar.gz</a>
Linux ARM v8 Hard Float ABI	74.66 MB	<a href="#">jdk-8u65-linux-arm64-vfp-hflt.tar.gz</a>
Linux x86	154.67 MB	<a href="#">jdk-8u65-linux-i586.rpm</a>
Linux x86	174.84 MB	<a href="#">jdk-8u65-linux-i586.tar.gz</a>
Linux x64	152.69 MB	<a href="#">jdk-8u65-linux-x64.rpm</a>

The file distributed is a gzip file. Assuming we want to extract to /usr/lib/jvm we can run:

```
sudo tar -zxvf jdk-8u65-linux-arm32-vfp-hflt.tar.gz -C /usr/lib/jvm
```

After installation, we can set the following environment variables:

```
JAVA_HOME=/usr/lib/jvm/jdk1.8.0_65
PATH=$JAVA_HOME/bin:$PATH
```

See also:

- [Java Embedded](#)
- [Getting Started with Java SE Embedded on the Raspberry Pi](#)

## Java Development Environment for Pi

Now that we realize that we have the ability to run Java applications on the Pi, the next question is how do we go about building such? Obviously, one can use the native editors, compilers and make tools running natively on the Pi. However, that is no where near as productive as we can possibly achieve. There are a number of fine and free IDEs for Java including Eclipse and Netbeans. Because I am familiar with Eclipse and it is my weapon of choice, I am going to focus on that.

The core nature of Java is the write-once-run-anywhere metaphor. What this means is that we edit our Java source files using an editor and when we compile them with the Java compiler, the result are files called Java class files. The content of these files is executable code for a "virtual machine". This instruction code is also known as byte-code. The code is not natively executable by a CPU such as Intel or ARM but must instead be supplied to an application called a Java Virtual Machine (JVM) for execution. The JVM was installed as part of the Java install and is launched through the command called `java`. Since the class files generated by the Java compiler are targeted at the JVM and the JVM is a native binary executable available for the Pi (as well as Windows, Mac and Linux), the JVM insulates us from machine dependencies of running those class files. What this means is that we can compile the Java source files on a Windows or Linux machine and simply copy the resulting Java class files to our Pi for execution. Since we are assuming that our desktop PCs are far more powerful than the Pi, we now have an excellent story for Java development.

We edit and compile our Java source on the PC and ship the resulting class files to the Pi for testing and final execution. Java architects the notion of zipping up a set of class files into a ZIP file with the file suffix of `.jar`. These files are referred to as jar files. What this means is that we don't have to ship over a whole slew of files with the challenges that might entail but instead can compile our individual Java

source files to class files and then archive those class files into one jar file which can then be copied to the Pi as a single unit for execution.

## Using Eclipse for Java Pi development

The latest Eclipse is known as "Mars". A nice application plugin for Eclipse is called "LaunchPI". What this plugin does is allow us to develop Java applications on a PC and then run them on a Pi using the standard Eclipse launch mechanism. The way it does this is that when we wish to run the application, the Jars are packaged together and copied to the Pi and then a JVM on the Pi is launched. The console output of the JVM is redirected to the console output on Eclipse. All in all, very easy and seems to work as advertised. It was tested on the latest Mars Eclipse release and worked without issue. The source on Github hasn't been touched for a while though so it is possible the maintainer has ended support.

See also:

- [LaunchPI](#)
- [CodeRaspIDE](#)

## Pi4J Project

The Java libraries provide an absurdly rich set of capabilities we can use to write applications but what they don't contain are I/O functions for our embedded devices. Fortunately, a solution has been provided in the shape of the Pi4J project. This is a set of Java classes and other components that provide a set of rich Java interfaces for interacting with all of our hardware components including GPIO, I2C, SPI and more.

Let us spend a few moments talking about how Pi4J works. When programming and running code in a Java environment, we are running inside a Java Virtual Machine (JVM) which is interpreting Java byte-code. This code runs completely within the JVM environment and insulates the Java code from any environment specific distinctions. There are times when we want to step outside of the JVM environment and this is where the concept of the Java Native Interface (JNI) comes into play. JNI is the notion that we can create and call a C function which contains native code compiled for the platform on which the JVM runs. In our case this will be the ARM/Pi environment. If we also consider that there is an existing library called WiringPi that allows us to invoke hardware functions from C programs by calling the WiringPi library capabilities. If we marry these ideas together, then we have the idea that we can write Java code that invokes WiringPi hardware access functions through the JNI interface. Although extremely powerful, working at the raw JNI level is usually similar to mixing oil and water. In a Java environment we want to think about classes and objects while working on JNI calls we are back to the procedural C interfaces. What Pi4J has done is encapsulate / wrap the low level C functions found in WiringPi with Java friendly classes and interfaces.

See also:

- [The Pi4J project](#)

## Pi4J Installation

The installation documentation of Pi4J should be studied at the Pi4J website. As of writing, it was the simple instruction:

```
curl -s get.pi4j.com | sudo bash
```

Following installation, the parts of Pi4J can be found in /opt/pi4j.

## Writing Pi4J applications

The classes for Pi4J are contained in the `com.pi4j.*` package structures. Be **aware** that the Pi4J pin numbering scheme is the same as that for WiringPi. There is no option for Broadcom pins.

To work with GPIO one must create an instance of a `GpioController`. This should be only attempted once within the application and the instance created should be shared across the application.

```
final GpioController gpio = GpioFactory.getInstance();
```

When we work with GPIO pins, we usually associate them as either being for input or output. There are two distinct objects for pins. One object is used to represent an input while the other is used to represent an output. To create an input pin one would call `gpio.provisionDigitalInputPin` while to create an output pin, one would call `gpio.provisionDigitalOutputPin`.

For example:

```
GpioPinDigitalInput myInput = gpio.provisionDigitalInputPin(RaspiPin.GPIO_02);
GpioPinDigitalOutput myOutput = gpio.provisionDigitalOutputPin(RaspiPin.GPIO_04);
```

If we think of an output pin as having a state of either high or low, we have a number of methods available to us to change or set this state:

```
myOutput.setState(myValue); // Set the state to the given value.
myOutput.low();           // Set the state to low
myOutput.high();          // Set the state to high
myOutput.toggle();         // Set the state to the opposite of what it is now
myOutput.pulse(1000);
```

For input we have the functions called `gpio.getState()`, `gpio.isHigh()` and `gpio.isLow()`. For input pins, we can also set the pull up state. This defines what happens when the pin is left dangling. We can use the `GpioPin` method called `setPullResistance()` to set a value of up, down or off.

If we don't wish to poll the value of a pin, we can register a state change listener which will be invoked when the state of the pin changes. We can use the `GPIO pin addListener()` method to perform this task.

For I2C from Pi4J, we use the `I2CFactory.getInstance()` method to retrieve an `I2CBus` object. From the `I2CBus`, we can call `getDevice()` to get an instance of an object representing a specific `I2CDevice`. Finally, from `I2CDevice`, we can call read and write methods to read and write data.

See also:

- [Pi4J – JavaDoc](#)

## Pre-built Pi4J classes

Through the nature of Java being an object oriented language, we have the notion of reusable classes of function being created once and then made available for others to use over and over again. Part of the Pi4J project includes a rich set of pre-built classes that provide access to electronics components and devices. Without having to know how they work in detail, we can start to leverage their capabilities through their exposed interfaces. These can be found within the Jar called `pi4j-device.jar` and within the github source tree at `pi4j/pi4jdevice/*`.

## The Device I/O API

One of the primary values of Java is the write once and run-anywhere mantra. This means that if I write a Java application for System A it should run un-changed on System B. The way this is achieved is through standardization. A specification for a standard is produced that achieves some form of function such as database access (JDBC), queue access (JMS), naming & directory access (JNDI) and more. Each of these specifications provides a set of interfaces that have to be implemented. How they are implemented by the designers is not important to the users of the APIs ... only that they conform to the specification and semantics.

When it comes to hardware / device interactions, there had been no such specification. This has meant that implementations such as Pi4J and pigpio, while excellent for the Pi, are simply not present on other platforms where Java might run. This means that if I write an application in Java for the Pi, I will likely have lost the ability to simply move my Java class files to a new target platform and run them.

This is where a new Oracle sponsored specification called "Java Device I/O" has entered the picture. This specification prescribes a common interface that, if implemented on each target platform, would result in commonality across those platforms for developer written Java applications.

An implementation of Device I/O is available for the Pi at the Open JDK web site.

See also:

- Voxxed – [The Device I/O API](#)
- JavaDoc – [Device I/O API 1.0](#)
- OpenJDK – [Device I/O](#)

## UI programming in Java

Java is a good language for UI programming. Here we will examine a number of options for building UI apps in Java.

### JavaFX

JavaFX is the standard UI for working with screens within a Java environment. JavaFX is supplied as standard with Java 8 and beyond. On a Pi, JavaFX works well however it will not inter-operate with X-Windows. What this means is that JavaFX on the Pi interacts directly with the video GPU / frame buffer and the application owns the whole of the display. This obviously means that you need to connect your Pi

via HDMI to a display. We tried to use the `fbtft` driver that creates a new frame buffer at `/dev/fb1` but it seems that JavaFX wants to use the frame buffer at `/dev/fb0` which is connected to the GPU. We tried copying the image using `fbcp` but there was flicker and lost screen area and eventually abandoned this area of research.

There is an excellent graphic designer for JavaFX called "scenecreator" which can be downloaded and run on either Linux or Windows. The resulting output are FXML files that describe how the screen will appear. To use this, one can download a Debian image (.deb file) and install with `dpkg -i <packageName>`.

See also:

- [Using a Raspberry Pi to Deploy JavaFX Applications](#)

## Calling C from Java

When working in Java, we commonly have access to every function we might ever need as the Java packages supplied with the JVM are deep and broad. When we combine those with the rich set of existing packages already available for free download from the Internet, we might be tempted to think we have everything we need. For most larger scale computers, this is indeed the case. However for low-end devices like the Pi, there are times when we might need to step out of the comfort of the Java platform. When we are working on the Pi, there may be times when we need to access low-level aspects of the Pi hardware. Fortunately, Pi4J and other libraries have accomplished the majority of that work for us without us having to go deeper. But there are still edge cases where Java just won't do it for us.

Here is an example. The popular HC-SR04 device is an ultrasonic distance measurement device. It detects how far away an object is and returns that distance encoded in the duration that an output pin is set high. Typically, our high level algorithm for measuring this value would be:

```
while(pin == low) {  
    // do nothing  
}  
timeStart = timeNow();  
while (pin == high) {  
    // do nothing  
}  
timeEnd = timeNow();  
timeDiff = timeEnd - timeStart;
```

Now imagine that the duration that the pin might be high is measured in microseconds. Remember that is an extremely short interval of time. A millisecond is one thousandth of a second while a microsecond is one millionth of a second. Looking back at our algorithm, we see that we have a number of statements such as calculating the times and performing the loops. Unfortunately, Java is much slower than assembler or C code. In practice, it can take 10s of microseconds just to perform a very simple Java statement. When we need precision timing at such a low level, Java simply isn't going to work for us.

Fortunately there is a solution, we can create applications which are a mixture of Java code and C code. The Java statements can run the majority of the solution but for those few occurrences that require ultra fine timing resolution, we can write those as C functions and pass control to those from within our Java context.

Other reasons to interact with C would be to call OS functions that aren't exposed through Java APIs or to invoke C libraries that have been supplied with no Java wrappings pre-defined.

Here is a recipe to create an interaction between Java and C.

In Java, define the interfaces to the C function you want to call in a class:

```
public class MyClass {  
    static {  
        System.loadLibrary("CLibrary");  
    }  
    public static native <returnType> functionName();  
}
```

The `loadLibrary` call will look for a Linux library such as `libCLibrary.so`. Next we create a C language header file by processing the class with the `javah` tool. The generated header will typically look like:

```
jint JNICALL Java_<functionnam>(JNIEnv *env, jclass class, <parameters>)
```

or

```
jint JNICALL Java_<functionnam>(JNIEnv *env, jobject obj, <parameters>)
```

The difference is whether or not the Java method is defined static or is a member function. To call a method "myFunc" in class "cls" we must first retrieve an "id" for that method:

```
jmethodID methodId = (*env)->GetMethodID(cls, "myFunc", "<Signature>");
```

If we don't know the class but have the jobject, we can use `GetObjectClass()` we can then invoke the method with:

```
jdouble result = (*env)->CallDoubleMethod(obj, methodId, <parms>);
```

To compile a C file into a library, consider the following Makefile

```
JAVACLASSDIR=/mnt/share/WorkSpace/TestJNI/bin  
TARGETLIB=libjnicode.so  
TARGETS=kolban_TestJNI.h $(TARGETLIB)  
OBJECTS=jnicode.o  
OPTLIB=/mnt/share/opt/lib  
CC=gcc  
CFLAGS=-c -Wall -fPIC -I $(JAVA_HOME)/include -I $(JAVA_HOME)/include/linux  
  
all: $(TARGETS)  
    cp $(TARGETLIB) $(OPTLIB)  
    echo "Built"  
  
kolban_TestJNI.h:  
    javah -cp $(JAVACLASSDIR) kolban.TestJNI  
  
$(TARGETLIB): $(OBJECTS)  
    gcc -shared -Wl,-soname,$@ -o $@ $^
```

```
.c.o:
$(CC) $(CFLAGS) $<
clean:
rm $(TARGETS) *.o
```

To create a new object, we can use:

```
jclass xClass = (*env)->FindClass(env, "kolban/X");
jmethodID xConstructorId = (*env)->GetMethodID(env, xClass, "<init>", "()V");
jobject newObj = (*env)->NewObject(env, xClass, xConstructorId);
```

To look at a class and see its signatures, use “javap -s <className>”.

See also:

- <http://jonisalonen.com/2012/calling-c-from-java-is-easy/>
- [JNI Specification](#)
- [JNI Tutorial](#)

## JavaScript Programming

JavaScript is a language that was originally invented as a scripting language running in browsers. Even today that is still predominantly the case. It is an interpreted language that has leanings towards Java but is sufficiently different that it is very much in a class of its own. There are many schools of thought and opinions on JavaScript. It is a loosely typed language which means that one doesn't have to declare variables before using them. This can result in tricky puzzles to track down at run-time that would have been caught with other languages at development time. For example:

```
var sale = {
    name: "widget",
    price: 12.34
};

sale.cost = sale.cost + tax;
```

would not catch the mistake that we originally named the property in sale as `price` but referred to it as `cost`. Only at run-time when the statement was reached would we catch it. However, discussions along those lines tend to result in religious views and we will leave those alone. From the perspective of the Pi, there are also non-browser variants of the language including Node.js and Nashorn that are at our disposal.

## Node.js

The current distribution of Raspbian comes with Node.js installed ... however, it is present only to support the technology known as node-red and doesn't appear to be usable as a Node.js environment by

itself. My suggestion is to remove the pre-installed Node.js and install one of the latest distributions. For example:

```
$ sudo apt-get purge nodejs
$ cd /tmp

# Pi 2
$ wget https://nodejs.org/dist/v4.2.4/node-v4.2.4-linux-armv7l.tar.xz
# Pi Zero
$ wget https://nodejs.org/dist/v4.2.4/node-v4.2.4-linux-armv6l.tar.xz
$ tar -xvf node-<version>.tar.xz
$ cd node-<version>
$ sudo cp -R * /usr/local/
```

See also:

- [Installing Node.js v4.0.0 on a Raspberry Pi \(All Models\)](#)
- [node-arm](#)
- Adafruit – [Why node.js?](#)
- Node.js – [home page](#)

## NPM

NPM is the Node Package Manager which is a tool and environment for packaging JavaScript libraries for distribution. In simple terms, an NPM package is a directory which contains JavaScript plus a control/configuration file called `package.json`. Contained within `package.json` is the meta-data about the package itself. Even if we are creating a local application, it is still very useful to create a `package.json`. The content of the file is a JSON object:

```
{
  "name": "mypackage",
  "version": "1.0.0",
  "dependencies": {
    "<SomePackage>": "<SomeVersion>"
  }
}
```

We can update the current packages. For example, running

```
$ npm outdated
```

will list the latest versions.

```
$ npm update
```

will update the packages to the latest versions.

```
$ npm ls
```

will list the packages installed for your application.

```
$ npm uninstall <package>
```

will remove a package.

```
$ npm prune
```

will remove packages that are no longer referenced by your package.json.

To install a package globally:

```
$ npm install <package> -g
```

When we are building a module that will be included be `require()`, we should realize that `require()` will return an object. In our JavaScript that implements the module, the object called "exports" is the one that will be returned. So if we create a JavaScript source file called "index.js" and in that source file code:

```
exports.myFunction = function() {
  console.log("Hello world!");
}
```

Then the object returned from a `require()` of our package will contain a function called "myFunction()" which, when called, will print to the console.

Semantic versioning is used to indicate the nature of changes. New projects should start with 1.0.0.

<major>.<minor>.<patch>

- major changes are not backwards compatible
- minor changes add new function
- patch changes fix problems

See also

- [NPM home page](#)

## GPIO from Node.js

Since Node.js is platform agnostic, one wouldn't expect Pi GPIO capabilities built into the environment. Fortunately, there are NPM hosted modules that provide interfaces to the hardware subsystems from a Node.js platform. Unfortunately (and some may consider this a good thing) there are a bewildering number of such packages to choose from with none apparently a clear stand-out.

Package	I2C	SPI	PWM	UART
rpi-gpio	No	No	No	No
pi-gpio	No	No	No	No
onoff	No	No	No	No
rpio	Yes	Yes	Yes	No
raspi-io	Yes	No	No	No
wiring-pi	Yes	Yes	Yes	Yes

Adafruit seems to have adopted `onoff` for their selection of GPIO.

For I2C, there is a popular npm package called "i2c".

See also:

- npm: [rpi-gpio](#)
- npm: [pi-gpio](#)
- npm: [onoff](#)
- npm: [rpio](#), github: [jperkin/node-rpio](#)
- npm: [raspi-io](#)
- npm: [wiring-pi](#)
- npm: [i2c](#)

### Using npm wiring-pi

My current personal selection for GPIO, I2C, SPI and UART integration with Node.js is to use the `wiring-pi` package. It is available within the npm repository. To install, run the following:

```
$ npm install wiring-pi
```

I recommend installing `wiring-pi` globally for two reasons. The first is that it is so commonly used, we should just "have it" and the second is that it compiles `wiring-pi` and that takes time. To install the package globally, add the "-g" flag:

```
$ sudo npm install wiring-pi -g
```

This will download and build `wiring-pi`. It will take a few minutes during which time there will be log messages.

Within a node application, require `wiring-pi`. Next, we need to initialize it. Both can be seen here:

```
var wpi = require('wiring-pi');
wpi.setup('gpio');
```

To use I2C, we will find that it is very similar to Wiring Pi.

When using SPI, it is essential to call `wiringPiSPISetup()` to initialize the SPI environment.

When using I2C, a call to `wiringPiI2CSetup(address)` will return a file handle. This can then be used with the other read/write operations to read and write data.

See also:

- Node.js – [wiring-pi module documentation](#)
- npm: [wiring-pi](#), Github: [eugeneware/wiring-pi](#)

## Nashorn

Contained within the latest Java 8 environment is a full JavaScript run-time engine called "Nashorn". This JavaScript engine runs within the Java JVM and allows one to mix together Java and JavaScript. However, if all one wants to do is write and run JavaScript applications, that is trivial using the `jjs` command supplied with the Java environment.

If we create a file containing a JavaScript script, we can run that file using `jjs`. For example:

```
$ jjs myFile.js
```

There may not be an existent Pi specific library for GPIO access from Nashorn but in principle, we have a powerful alternative. Since Nashorn is implemented in Java and is able to interact with the complete Java run-time environment, this then embraces all the classes available to Java. Since we have Java interfaces to the Pi GPIO, this then gives us the ability to interact with Pi GPIO through the Java exposed classes.

See also:

- [14Pi4J Project](#)
- [Oracle Nashorn: A Next-Generation JavaScript Engine for the JVM](#)

## JavaScript bit-twiddling

Since JavaScript doesn't have strong typing, there is no concept of creating 8 bit, 16 bit or 32 bit signed or unsigned integers. This means that we have to do some bit-twiddling ourselves. For example, imagine we read in two bytes ... MSB and LSB that we want to treat as a signed 16 bit number. In C, we might code:

```
signed short value = msb << 8 | lsb;
```

however, in JavaScript, this takes a little more work. One algorithm is:

```
var value = msb << 8 | lsb;
if (value > 0x8000) {
    value = value - 0x10000;
}
```

## Python Programming

Python is a good general purpose language that is mature and stable. It offers many of the features we expect to find in other languages such as objects, procedural programming and more. Similar to Java, the Python language is distributed with a large set of pre-supplied extension modules that are themselves written in Python. As such, not only should one be studying the Python language but also the existence of the various modules. There are three important releases of Python for 3 distinct areas. There is the standard version of Python implemented in C, there is the Java implementation of Python called Jython and there is an implementation for Windows that uses the .NET framework. We will only be looking at the C implementation of Python.

It may be that Python is already installed on your Pi. At the shell enter:

```
$ python -V
```

There are a variety of different python versions out there including 2.6, 2.7, 3.0 and 3.4.

We can run python just by entering the command "python". Running it with the "-h" flag lists us the command options available to us. The most common way to invoke a Python application is to place it in a file and run:

```
$ python <fileName>
```

however, running python by itself puts us into an interactive interpreter.

Python is a cross between an interpreted language and a compiled language. When we run a python program it is compiled then and there and then executed. For modules, the compilation output is saved.

If one wants to write a shell script that contains Python, we can begin the script with:

```
#!/usr/bin/python
```

A more common form is to start the script with

```
#!/usr/bin/env python
```

This will run the "python" command found in the PATH environment variable.

In C and Java, we can group multiple statements together with curly braces. For example:

```
if (expression) {  
    statement1;  
    statement2;  
    statement3;  
}
```

The statements between the braces are a grouping of statements. In Python, the designers have chosen to use indentation to signify grouping.

```
if (expression)  
    statement1;  
    statement2;  
    statement3;  
statement4;
```

In the above, statements 1,2 and 3 are executed if the expression is true. Statement 4 falls outside the grouping and will always be executed. If you are familiar with other languages, this is a significant difference.

Python provides some interesting data structures:

- A list – [1, 3.141, "elephant"]
- A tuple – (1,2,3)
- A dictionary - {'lng': 35.123, 'lat': -97.45}

Every variable in Python is an object with the special object called "None" referring to null.

With expressions we have boolean values called True and False.

Variables do not have to be declared prior to use. The act of assigning a value to a variable causes it to be created. Objects can have attributes associated with them. We refer to an attribute of an object using the `."` operator. For example `x.y` refers to the attribute called `y` of the object called `x`.

When working on the Pi, we commonly wish to manipulate bits of data and we have bitwise operations including:

Operator	Description
<code>~</code>	Bitwise inversion (not X)
<code>&lt;&lt;</code>	Left bit shift
<code>&gt;&gt;</code>	Right bit shift
<code>&amp;</code>	Bitwise AND
<code> </code>	Bitwise OR
<code>^</code>	Bitwise Exclusive OR (XOR)

The core language constructs are also present:

```
if <expression>:
    statement1
    statement2
elif <expression>:
    statement3
else:
    statement4

while <expression>:
    statement1
    statement2

for <var> in <iterable>:
    statement1
    statement2
```

The special function `range()` can be used to produce iterables. For example:

- `range(x)` – produces  $[0, 1, 2, \dots, x-1]$
- `range(x, y)` – produces  $[x, x+1, x+2, \dots, y-1]$

Functions can be defined using the `def` keyword:

```
def <functionName>(<parameters>):
    statement1
    statement2
```

Variables defined in a function are local in name scope to that function. To access the global variable, we need to define that we wish to do such with `global <variableName>`.

Object oriented classes can be defined with:

```
class <className>(<baseClass>):
    statement1
    statement2
```

methods defined in classes always have a first parameter that by convention is called self.

Variables which are private to a class should start with two underscores.

Instances of classes are created by invoking the class name. For example:

```
class C:
    statement1
    statement2
myInstance = C()
```

If a function called `__init__` is defined in the class then this is invoked as the constructor.

We can import modules using the import statement:

```
import <ModuleName>
```

If you are going to be doing Python development, you must install the package called `python-dev`:

```
$ sudo apt-get install python-dev
```

See also:

- Python RPi.GPIO reference
- [RPi.GPIO Python Module](#)

## Arduino programming for the Pi

Here is where we are potentially going to get controversial. Don't we just love that. Let us take a few minutes to consider Arduino programming. The Arduino is without question a fantastic board. There are a number of models available but I'm going to focus on the Uno for the sake of choosing one. The Arduino board is based upon the ATmega328P processor which is an 8 bit device running at 16MHz. The device has 2K of RAM and an in built 32K of flash memory. It was an early adopter of Open Source hardware and its full schematics are freely available. The device has 16 GPIO pins and 6 analog input pins. In addition, many of the pins can be multiplexed to provide I2C, SPI, PWM and UART. All in all an impressive device. The first release of the Arduino was back in 2005 and its growth and success has continued.

Part of what made the Arduino so incredibly successful was its low barrier to entry. The IDE for the Arduino is also free and incredibly simple to use. It basically looks like a text editor for editing C/C++ code with big friendly buttons for compile and deploy. The Arduino connects to a PC where the IDE runs using a USB→UART cable. It only takes minutes to get a blinky going. In addition to providing a coding environment, the IDE also comes pre-supplied with a rich set of examples and samples immediately ready to run.

When one is programming the Arduino, the language is basically C and C++. However there is a well defined set of documented APIs that provide the interface between a programmer's code and the Arduino model of the hardware. These APIs are easy to use and well trodden. As such, there is a large magnitude

of existing Arduino based code out in the wild that has been used to integrate with about every conceivable piece of hardware imaginable.

Now let us look at price. The retail price of an Arduino Uno from Arduino themselves is just under \$25. However, if one performs a quick search on eBay, we can find Arduino clones from \$4-\$6 that are functionally identical. If we choose the latter price and compare that to a Raspberry Pi which ranges from \$20-\$40 depending on the model, we see that there is a dramatic price difference. If what we want to use an Arduino for is some simple embedded electronics, we may not want to spend Pi prices. This then means that we sometimes have to work around Arduino based limitations such as memory, speed and hard storage. But now with the release of the Pi Zero, suddenly we find the cost of an entry level Pi to be exactly comparable in price to an Arduino clone ... and hence choosing an Arduino over a Pi on the basis of price is no longer such a consideration.

Since the Pi is itself a fully fledged "computer", one would think it would be easier to build applications for the Pi than for the Arduino. This is not necessarily a true statement. There has been so much work done over the last decade on the Arduino that there are samples, blogs, books and videos on just about any consideration of Arduino programming. Wouldn't it be great to leverage the existing work of the Arduino in a Pi environment?

Obviously one can take an Arduino program (called a sketch) and hand port it to C or C++ on the Pi or even translate the algorithms to alternate languages such as Python or JavaScript but there is one more alternative. Simply treat the Pi **as** an Arduino.

Imagine that we could install the Arduino IDE on our PCs and write or re-use our Arduino sketches just as we would for a real Arduino ... but instead of deploying to an Arduino, we deploy to the Pi. This is exactly what an Open Source project that can be found on Github offers to do. The project is called RasPiArduino and can be found:

<https://github.com/me-no-dev/RasPiArduino>

It provides two core features that together provide the ability to build and run Arduino sketches on the Pi.

The first is an efficient implementation of the Arduino API libraries written for the Raspbian environment. These libraries expose upwards all of the APIs that are found in the core of an Arduino. For example, on an Arduino I might code `digitalWrite()` to write to a GPIO pin, the Pi library provides exactly the same function with the same signature. Thus if one takes the original sketch and compiles it against this Pi library, the sketch will perform the same task. This story has been repeated for all the other Arduino exposed APIs.

The second component of the project is the definition of a new "board" for the Arduino IDE. The IDE has always been extensible. The anticipation and reality was that additional Arduino boards would become available (and they did) and the steps necessary to compile and deploy those boards might change. The IDE was thus architected to provide a level of separation between the core functions of the IDE (editing, libraries, source management ...) and the board specific functions (compilation and deployment). The Pi project has implemented the components necessary for a new Arduino IDE board. Once plugged into the IDE, the user writes their Arduino sketches just as they always would but now at compilation time, the board selected is the Raspberry Pi and the deployment is to the Pi over a network connection.

As we shall shortly see, the combination of the Arduino IDE and the Pi make a great combination for hosting Arduino sketches ... however there is yet another consideration. The development tools including debuggers and profilers are far richer on the Pi. We can single step through our sketches using native Pi debuggers and much more. If what we are developing is an Arduino sketch that is still intended to be run on an Arduino, it is a consideration to use the Arduino IDE for development and the Pi for testing and debugging. If one can iron out the majority of logical software kinks on the Pi one should then be able to trivially "port" (if any work is even needed) the same sketch to a real Arduino device. As such, for dyed-in-the-wool Arduino consumers, the use of the Pi is still a tool in their toolbox for more sophisticated debugging and diagnostics.

## Overview of the Arduino IDE

The Arduino Integrated Development Environment (IDE) was designed for the Arduino devices. Its goal was to balance ease of use with enough features to make it useful. Although it only has a fraction of the capabilities of other IDEs lacking even basic features such as find and replace, it still manages to do its job very well. Rather than call the programs you create with it "programs", it has chosen to call them "sketches" ... thus if you hear the term sketch you can mentally translate that into a program.

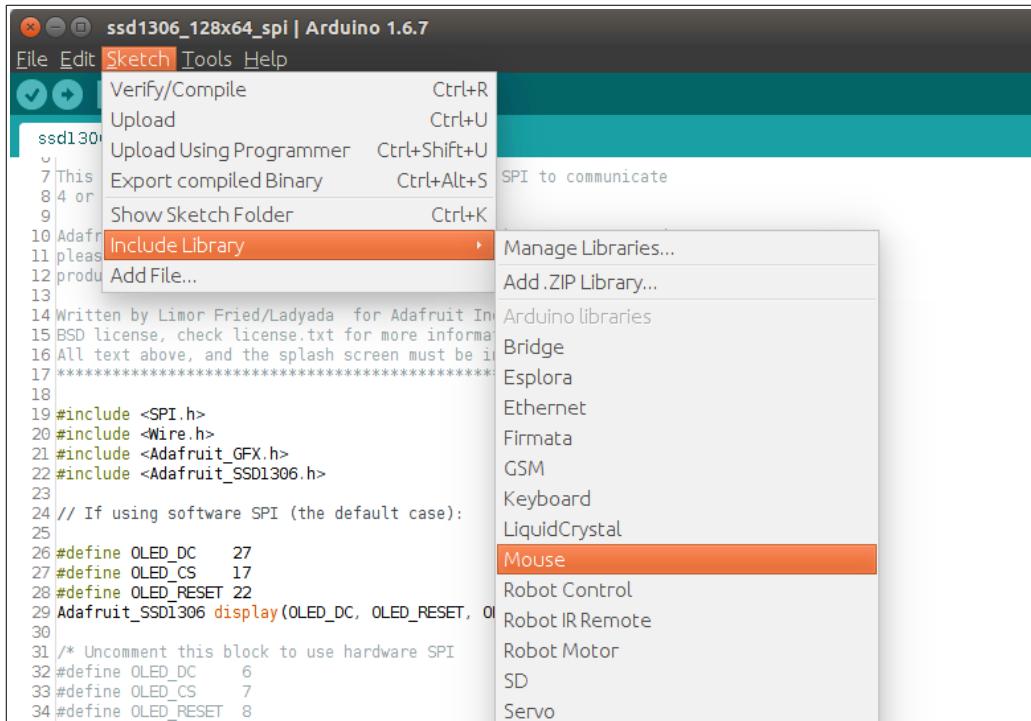
At its most basic level, when you create a sketch, you are given a single editor window into which you can write your code. Following the Arduino architecture, there are two functions you must implement. These are called `setup()` and `loop()`. The `setup()` function is called once and once only at the start of execution. When `setup()` completes, control is passed to `loop()`. Should `loop()` end, control will again return back to `loop()`. There is no requirement that you actually return from `loop()` but, if you do, realize that you will start at the beginning of `loop()` again.

When you save your project, what gets created is a directory with the name you choose. Within that directory you will also find a file also named after your sketch with the file suffix ".ino". I don't know why it was called that ... but in effect, this is the source of the sketch you edited.

The Arduino IDE contains all of the logic to build your code. There are no makefiles or other compiled flags. One simply clicks on the verify button to build the code. A text window at the bottom logs any error messages that are produced if the code won't compile. Remember, this is an editor environment and even the best of us make typing errors that cause the code not to compile.

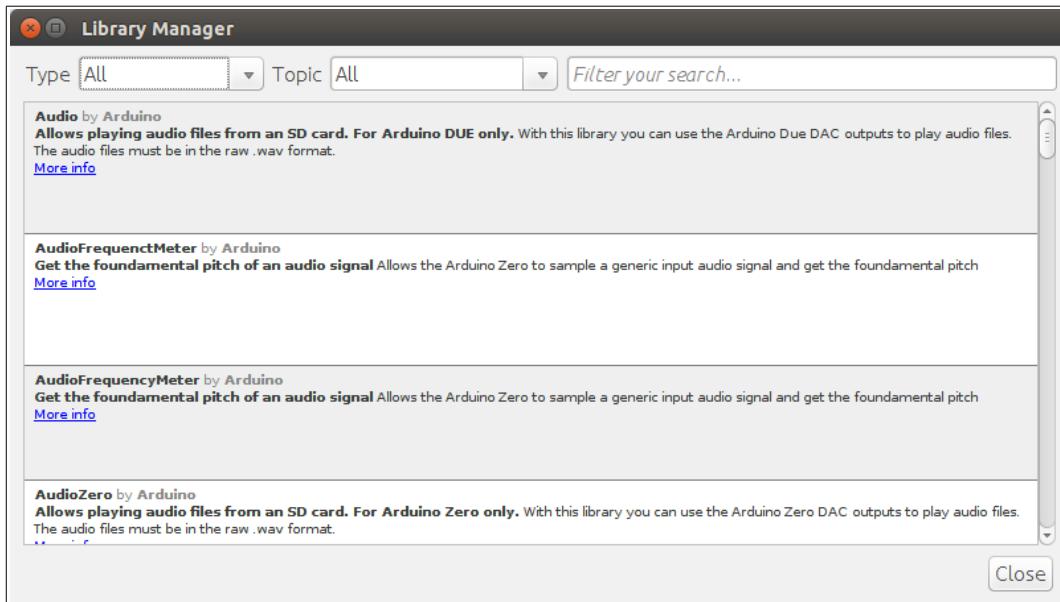
One of the most elegant part of the Arduino IDE is its library inclusion system. When working with embedded devices, there are lots and lots of details that need to be considered when interacting with electronics modules. All those details have been carefully considered by others who have produced libraries that encapsulate these function. Instead of us having to pour through complex data sheets, we can instead simply leverage the work that has been built by those who have traveled the same path before us. The IDE allows us to search for existing libraries and include them in our environment.

From the Sketch → Include Library menu options, we are presented with a list of libraries that are already known to the IDE.



If we need additional libraries, we can select the "Manage Libraries..." menu entry.

We will then be presented with a Library Manager dialog from which we can sort, search, install and manage libraries that we can re-use:



See also:

- arduino.cc – [Arduino Software \(IDE\)](#)

## Installing the Arduino IDE

First you must install the Arduino IDE on your PC. This can be freely downloaded from the [Arduino](#) website. Since we are illustrating Linux here, download one of either the 32 bit or 64 bit Linux variants. I am going to assume the 64 bit version. At the time of writing, the download arrived as a single file called `arduino-1.6.7-linux64.tar.xz`. Our goal is to install our Arduino IDE into the directory `/usr/local`. Now we execute the command:

```
$ sudo tar -xvf arduino-1.6.7-linux64.tar.xz --directory /usr/local
```

The result will be a new directory called `/usr/local/arduino-1.6.7`. This is the Arduino install home directory. It is within there that you will find the program called `arduino` which, if started, would launch the Arduino IDE. However, we have more work yet to do.

## Installing the Arduino IDE board configuration for the Pi

Change to the Arduino IDE home directory (`/usr/local/arduino-1.6.7`) and then into the hardware sub-directory. It is here that information about the different platforms that the IDE can target is located. From there, create a new folder called `RaspberryPi`.

```
$ sudo mkdir RaspberryPi  
Change into that directory.  
$ cd RaspberryPi  
$ pwd  
/usr/local/arduino-1.6.7/hardware/RaspberryPi  
$
```

Now we need to install the files necessary for building Pi applications. We will use the `git` command to download the latest materials from the Internet.

```
$ sudo git clone https://github.com/me-no-dev/RasPiArduino piduino
```

The result will be a new directory called `piduino`. We want to change into that directory.

```
$ cd piduino
```

We will find a number of files that start with `platform.txt`.

```
$ ls platform.txt*  
platform.txt  platform.txt.arm-linux-gnueabihf  platform.txt.arm-none-linux-gnueabi  
$
```

Care is needed here. The `platform.txt` is the Arduino IDE configuration file used to define how the IDE will build the application. Our eventual goal is to automate this step but for now, we need to choose **one** of these files and make it the one that will actually be used. The one for our story is called `platform.txt.arm-linux-gnueabihf`. We will rename this to be `platform.txt`.

```
$ sudo mv platform.txt.arm-linux-gnueabihf platform.txt
```

## Configuring the Pi

First we open a terminal window onto the Pi.

Next we need to download a temporary copy of the RasPiArduino project. We will do this in /tmp.

```
$ cd /tmp  
$ sudo git clone https://github.com/me-no-dev/RasPiArduino  
...  
$
```

Next we change into the extracted project's 'tools' directory:

```
$ cd /tmp/RasPiArduino/tools
```

We are now going to configure the avahi subsystem to know about a new Arduino based service. This subsystem knows how to publish information over the LAN via a broadcast mechanism. This will allow the Arduino IDE to dynamically locate the Pis on which the Arduino sketches can run.

```
$ sudo cp arduino.service /etc/avahi/services/arduino.service  
$ sudo service avahi-daemon restart
```

Next we need to install some tools into the path:

```
$ sudo cp /tmp/RasPiArduino/tools/arpi_bins/* /usr/local/bin  
$ sudo ln -s /usr/local/bin/run-avrdude /usr/bin/run-avrdude
```

And finally, the telnet package is also a pre-requisite.

```
$ sudo apt-get install telnet
```

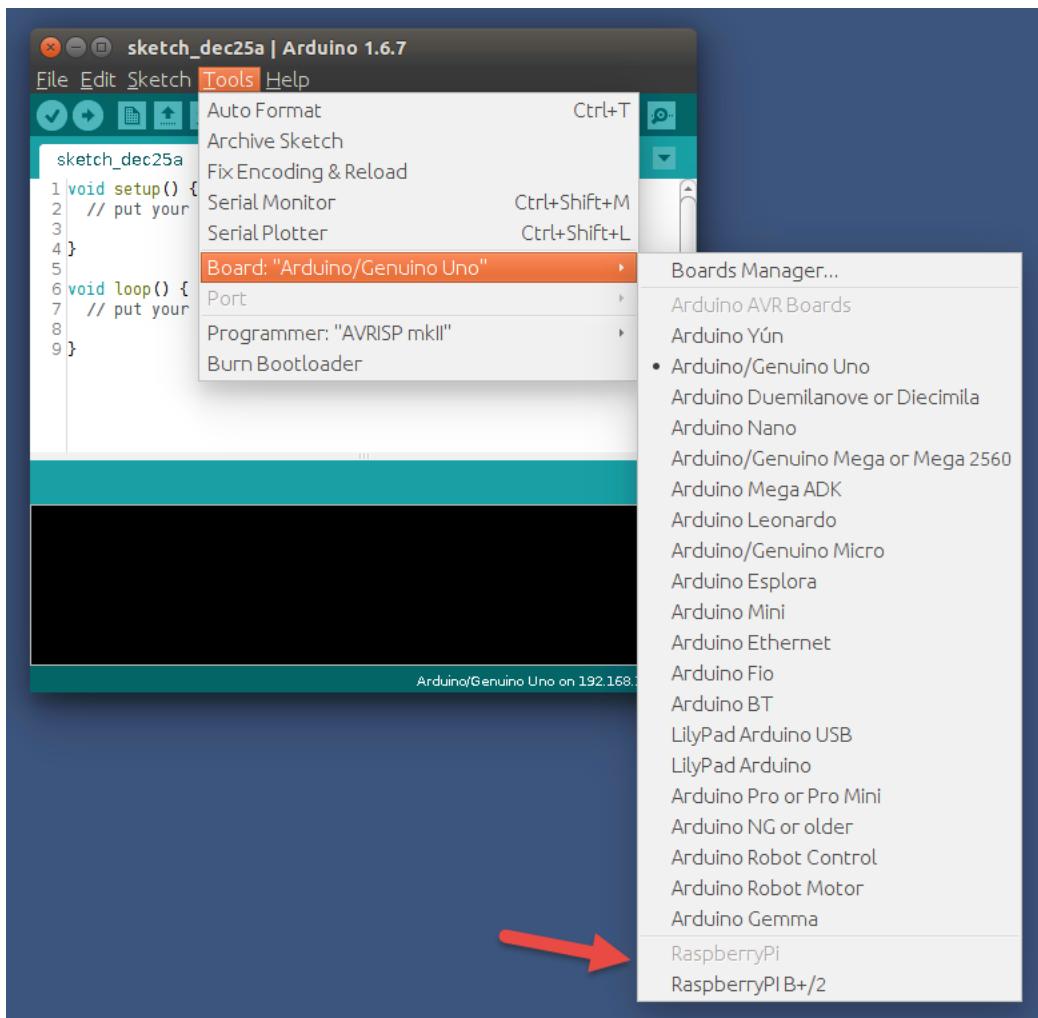
When the deployment from the Arduino IDE executes, we will be prompted for the root password of the root user on the Pi. If you don't know that password, reset it now.

We are now finished with the temporary extraction of the RasPiArduino project and can delete it.

```
$ sudo rm -rf /tmp/RasPiArduino
```

## Selecting the Pi as the target for development

When the Arduino IDE is opened for the first time, we must instruct it on what target we wish to select for compilation and deployment. From the Tools menu, we should select the "Board" option. From there we will be presented with a large list of each of the different possible targets that the IDE knows how to target.



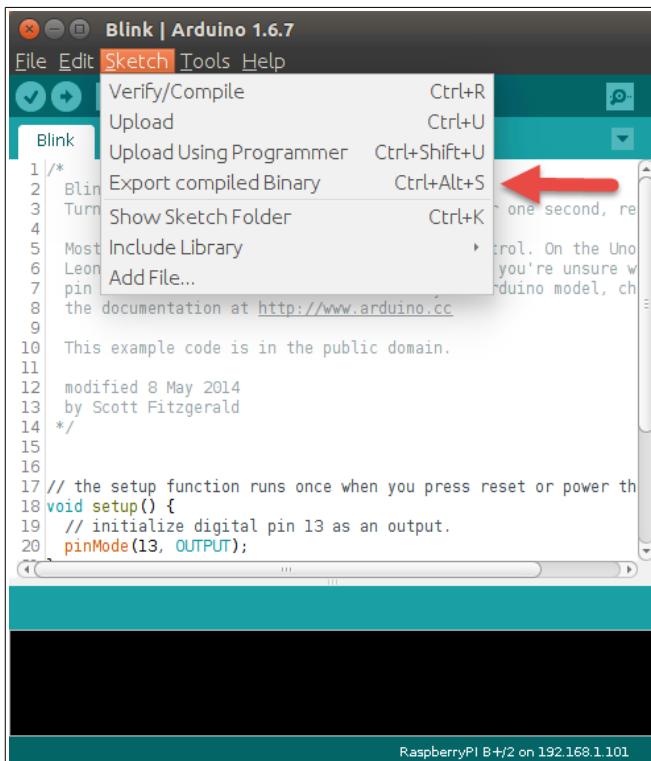
Since we have extended the Arduino IDE with knowledge about the Raspberry Pi at the bottom of the list we will see the Pi specific entries. One of these should be selected.

## Developing with an NFS export from or to the Pi

An NFS export allows us to specify a directory hosted on one Linux system as being exported and available to be mounted on a different Linux system. If we then read or write to that mounted file system, the data will be copied over the network and be available on the machine that hosted it. When working with RasPiArduino there is a neat feature at work here that we can leverage. Imagine we export the PC directory we are using for saving our Arduino sketches. If we do this, then we can mount that directory on the Pi. If we traverse into that directory we will see the source of those sketches. However, that by itself isn't useful. It is the Arduino IDE on the PC that is compiling our binary applications. However, there is an interesting technique we can now employ. In the Sketch menu, we will find a selection called "Export compiled Binary" which has a short-cut key combination of Ctrl+Alt+S. If we execute that command, the Arduino IDE will compile our sketch and, once done, will write the binary in the same

directory as the saved Sketch. If that is an NFS exported directory, what that means is that the binary will appear on the NFS import on the Pi ... and we can then run it from that directory. Putting this in a classic development frame of mind, we can edit, compile, save ... and then run on a window on the Pi. Since the binary is local on the Pi, we are back in the known world of classic Linux programming with the ability to run `gdb` and other Linux programming commands.

The following image shows what the menu entry for a binary file save looks like:



Let us work through a scenario. Imagine I have a PC called "PC" and a Pi called "RASPI". On the PC we export the file system called `/home/kolban/Arduino` which is where my Arduino IDE sketches are kept. On the Pi, I create a directory called `/home/pi/Arduino/remote`. Finally, I NFS mount the PC `:/home/kolban/Arduino` over the NFS mount point at `/home/pi/Arduino/remote`.

Back on the PC, I create a new Arduino sketch and save it to `/home/kolban/Arduino/MySketch`. I then export the compiled binary for the sketch which places it in the same directory where the sketch itself was saved.

Going over to the Pi, if we go to `/remote/home/pi/Arduino`, we will now find a new directory called `MySketch`. This was created on the PC when we saved the sketch. In that directory we will find an executable binary file corresponding to the compiled sketch. If we run this Raspbian binary, it will run the sketch.

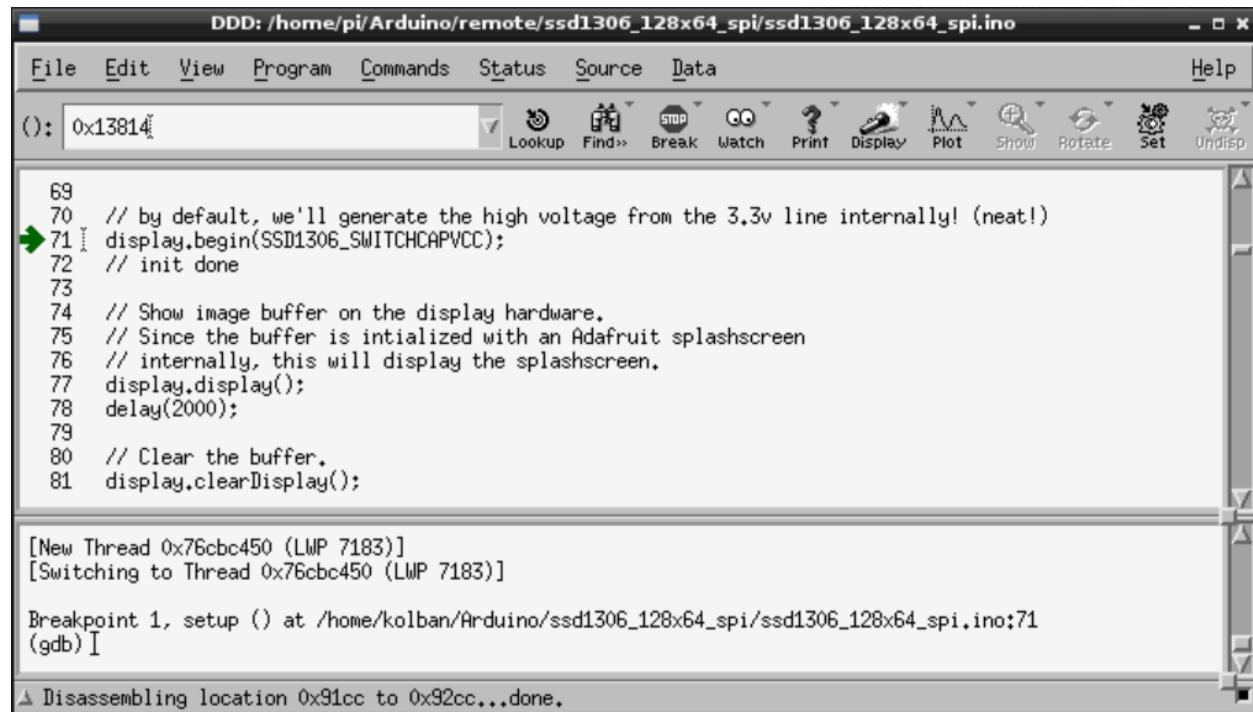
## Debugging a sketch

After we have compiled the sketch and transferred it to the Pi for execution, the next step is to test it. We can run the executable and if all goes well, it will behave just as we had hoped. Unfortunately, things

rarely happen as easily as that. Instead, the sketch may crash or simply not produce the desired results. At this point we enter the debugging phase of our work. Fortunately, because we are running on Raspbian and the compiled sketch is really no more than a compiled C++ application, we can use all the debugging tools already present. Chief amongst these is gdb. If we fire up gdb against our executable, we can set break points and step through our code. One problem though is that gdb expects to be able to find the source code and in our story, that will be residing on the PC and not on the Pi. In order to debug properly, we will need to bring in a copy of our source from the PC to the Pi so that gdb can find it. However, think back to the previous section where we discussed NFS exporting the PC directory containing our executable **and** the source sketch. If we follow that recipe then we will have the source and the executable already available on the Pi for testing.

If we don't have the source in our directories but the source is still accessible somewhere else on the Pi file system, we can use the gdb option called "directory" to add one or more directories to the search path where gdb looks for source files.

For those who don't care for the command line style of debugging and expect something a little less 1980s, we have the "ddd" UI wrapper on top of gdb. This provides us an easier to use and more appealing mechanism to examine the source of our application as well as see current steps, set breakpoints and see the values of our variables.



The screenshot shows the DDD (DejaVu Debugger) interface. The title bar reads "DDD: /home/pi/Arduino/remote/ssd1306\_128x64\_spi/ssd1306\_128x64\_spi.ino". The menu bar includes File, Edit, View, Program, Commands, Status, Source, Data, and Help. Below the menu is a toolbar with icons for Lookup, Find, Break, Watch, Print, Display, Plot, Show, Rotate, Set, and Undisp. A status bar at the bottom shows "Disassembling location 0x91cc to 0x92cc...done."

```
69
70 // by default, we'll generate the high voltage from the 3.3v line internally! (neat!)
71 display.begin(SSD1306_SWITCHCAPVCC);
72 // init done
73
74 // Show image buffer on the display hardware.
75 // Since the buffer is initialized with an Adafruit splashscreen
76 // internally, this will display the splashscreen.
77 display.display();
78 delay(2000);
79
80 // Clear the buffer.
81 display.clearDisplay();
```

[New Thread 0x76cbc450 (LWP 7183)]  
[Switching to Thread 0x76cbc450 (LWP 7183)]

Breakpoint 1, setup () at /home/kolban/Arduino/ssd1306\_128x64\_spi/ssd1306\_128x64\_spi.ino:71  
(gdb)

See also:

- Debugging C

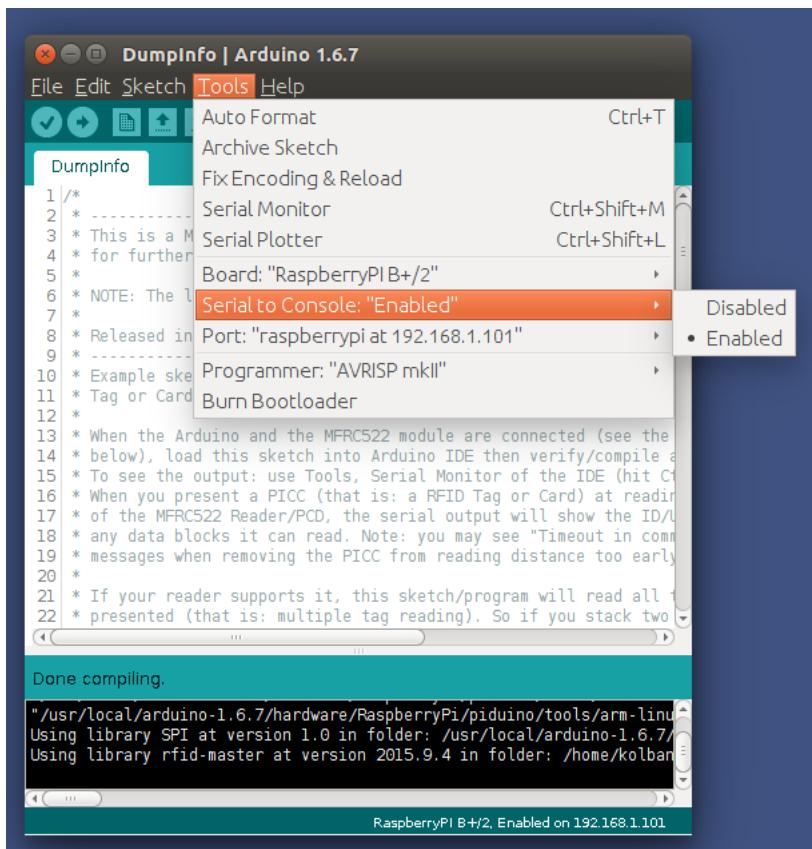
## Differences from a hardware Arduino

There are differences between what is supported by the RaspPiArduino and what is available to a real hardware Arduino. Although the APIs to the Arduino are clearly defined from the programming perspective, some advanced users have figured out how to take advantage of some extremely low level capabilities. These include "bit banging" data into and out of registers on the device. Obviously if a sketch contains a poke to a particular memory location that does something on an ATmega CPU it will have no effect on the Pi and, in most cases, will actually cause the program to crash. There is no easy circumvention to this. If you try and use a pre-defined sketch or library which uses these low level interfaces, you have no real option but to recode to use the valid APIs. The primary reason that these tricks were performed on the Arduino was for raw performance and since the Pi is many times faster than the Arduino, we should be able to achieve the original result by going through the higher level interfaces without loss of function.

## Redirecting Serial output to the Console

On sketches which were authored for the Arduino, it is likely that they will write to the Serial device. This is because the Arduino has no screen output or keyboard input. The primary way of generating output from the Arduino device is to send data through the UART. On the Pi, we have a Console device which can serve that same purpose and can be accessed by coding `Console.print()` or `Console.println()`. However, this would mean changing existing sketches. An option is available for the Pi Arduino support which allows re-direction of the Serial output to the console. To enable this, we have a new option in the Tools menu called "Serial to Console" which can either be "Enabled" or "Disabled".

Setting enabled causes the next compiled version to map calls to "Serial" to "Console".

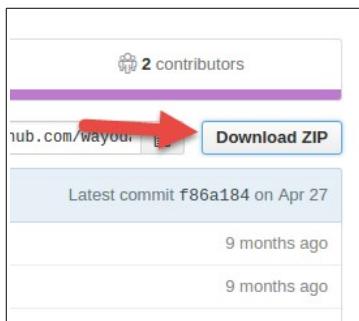


## Importing a 3<sup>rd</sup> party library

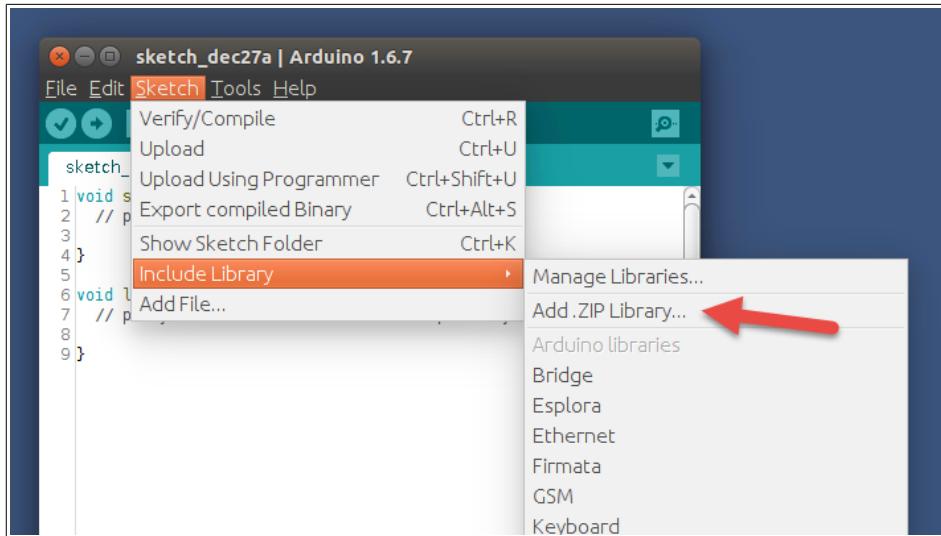
Imagine you have obtained some interesting new piece of hardware and you want to see it work straight away. You can't find any obvious libraries for the Pi but you find one for the Arduino on the net. You know that given a bit of time you can port this to the Pi but you want to see it run first. Now we get into the concept of downloading and installing a 3<sup>rd</sup> party library. We will use an example here of an LED controller library that can be found on Github here:

<https://github.com/wayoda/LedControl>

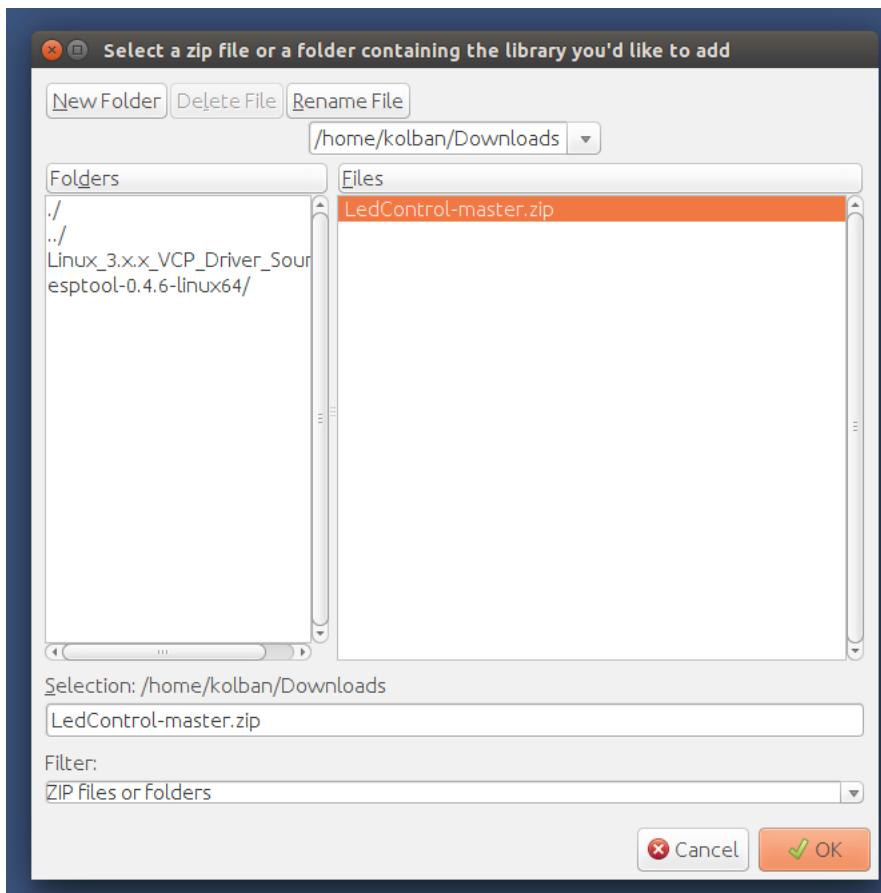
If we visit that page we will find the option to download the project as a ZIP:



Once downloaded, we can import it into our IDE. From the Sketch menu we can select Include Library → Add .ZIP Library.



We can then navigate to the folder where the ZIP file is contained and select the file.



Clicking OK will import the library and it will now be present for use in our applications. If we look in the examples directory, we will also find any examples it may have supplied.

You may wonder "Where did the library actually go?". The answer is that libraries you have installed are placed in the "Arduino" folder of your home directory under the "libraries" folder. If you imported a library called "XYZ" then there will be a new folder at "~/Arduino/libraries/XYZ". The IDE looks for libraries in the "~/Arduino/libraries" folder. If you wish to delete no longer needed libraries, simply delete the folders from there.

## Running the Arduino IDE on the Pi

If one installs the default Arduino IDE on the Pi we will find that it is very old indeed. I believe it is version 1.0.5. This is too old to be useful. Thankfully, there is a solution. We can download and install the source code of the Arduino IDE and build it ourselves. This would be a painful task if it weren't made simple by the existence of an excellent Github project called [ShorTie8/Arduino\\_IDE](#). This package automates the download and installation of everything we need.

It requires some pre-req packages, specifically the following should be installed:

```
apt-get install -y mercurial subversion build-essential gperf bison ant texinfo zip  
automake flex libusb-dev libusb-1.0-0-dev libtinfo-dev pkg-config libwxbase3.0-dev  
libtool
```

Once installed, we run the command:

```
sudo ./Arduino_IDE_Builder.bash
```

and then go away for a while. It will take hours to complete.

See also:

- Github – [ShorTie8/Arduino\\_IDE](#)

## Raspbian Environment Programming

Raspbian is a flavor of the Linux operating system and it would be impossible to even scratch the surface of programming to its huge number of features, supplied libraries and third party libraries. Instead what we will catch here are areas that might be of interest to us when writing Pi based applications. What do we define as a Pi application? I define that as something you would want to run on a Pi as opposed to on a PC. These are typically embeddable functions.

Think of this section as the software portion of the interfacing examples.

### Audio output from a your application

Since the PI has the ability to output an audio signal for playing through speakers, that may a feature you wish to add to your applications. For example, we can imagine a scenario where you want to play a sound when an attached sensor is triggered. Maybe it is a doorbell sound when someone approaches your front door? Maybe it is a cheer when you throw a bean bag through a corn-hole game? Maybe it is an alarm sound when the carbon monoxide sensor registers too high a concentration?

The simplest way to output a sound from your program is to execute one of the existing programs found on the file system such as `aplay`. Executing a `system()` function call and pointing to the `aplay` program and the path to the file you wish to play may be all that is needed.

An alternative is to use one of the pre-supplied libraries such as `libao`. This library takes a buffer of data and sends it to your audio device. The data is expected to be in a PCM format. The library knows nothing about data formats or encodings so if your data comes from a file, it is likely that you will need to decode that data before passing it to `libao` functions. The core architecture is that first we decide where we are going to send the audio. We have two options ... to a device for playing or to a file for saving. There are two APIs for this. The first is `ao_open_live()` which connects with the audio playing subsystem and returns a handle. The second is `ao_open_file()` which opens a new file for writing and returns a handle. In either case, the `ao_play()` call can then be made to "play" the audio into the previously opened destination. In order to develop `libao` applications we must install the `libao-dev` package.

In order to play data from a file, we will need some decoder library such as `libmpg123`. To develop with `libmpg123` we must install the `libmpg123-dev` package.

A fantastic set of samples illustrating playing an MP3 file can be found at the following web site:

<http://hzqtc.github.io/2012/05/play-mp3-with-libmpg123-and-libao.html>

From a Node.js environment, there is an `npm` package called `play-sound`. This assumes the existence of a media player application such as `mpg123`. Testing on the Pi it works exactly as advertised.

See also:

- [libao](#)
- [mpg123](#)

## Multi threading

When we write a C program we assume that it starts at main and runs till completion in a single path. This is easy to understand but may not make optimum use of the resources available on the Pi. For example if we write a Pi application that wishes to write to files then while the file operation is happening, the application isn't doing anything else. This is also true for network operations. It may be that the operations only takes a few tens of milliseconds but at the level of the CPU, that is a lot of instructions. When I make a cup of coffee in my kitchen, I may fill the pot with water and place it on the stove. However, since I know it will take time to boil, I can do other tasks in the meantime. I can open my mail or clean the sink or other chores. If I simply stared at the pot, I wouldn't be as productive or useful as I could be. Another consideration is that if my goal is to wash the windows in my house, if there is just me cleaning then it will take a certain amount of times but if I enlist the help of some friends, then it will take far less time as we can divide the same amount of work between us. In both of these stories, what would help us is the notion of parallelism. Parallelism is the notion that we can break up our tasks into parallel activities that may not have relationships with each other. Then when I am otherwise waiting for something to complete, I can switch over to another task ... or if I have multiple workers, I can distribute the tasks amongst them.

Imagine that we wanted to perform multiple steps in parallel within our code. A harsh way to achieve this would be to `fork()` our process and somehow try and have these peer processes communicate with each other. This was in fact the way it used to be. However there is a better way and it is called "threading". The notion of threading is that we can write a C application that runs in a single process but has multiple threads of control. Think of it as being able to perform multiple tasks simultaneously. The key to make this work is the in-built library called `pthreads`. The core function is called `pthread_create()` which causes the creation of a new thread and specifies which function should be called when the thread is started. That function is invoked within the context of the new thread.

After compiling an application which includes `pthreads`, we must link with the `-lpthreads` flag.

Writing programs that employ multi-threading can take a different mind set. Each of the threads has read/write access to all the global variables and memory. What this means is that they could easily step on each other. For example, if thread #1 is reading a variable and thread #2 is writing to it, we may run into problems. For example:

thread#1

```
if (myVar != 0) {  
    int x = 100/myVar;  
    ...  
}
```

This looks like sensible code. It checks that the variable isn't zero before dividing by it.

Now consider

thread#2

```
myVar = 0;
```

If this logic is executed in thread#2 **after** the `if(expression)` but **before** the division statement then we will have a problem. Threads within the `pthreads` environment can be assumed to execute at any time.

The solution to the puzzle would be to use a condition variable. The notion behind this is that when one wishes to access some shared resource (eg. the `myVar` variable), all participants would have to first secure access to it. Only once they have access will they be allowed to read or write to it. If someone else already has access when they request it, then the requester will block waiting for the current holder to release access.

On a machine with only a single processor, multi-threading takes advantage of blocking calls waiting on some external activity to complete such as file or network. However on a machine with multiple processors, then we can better leverage the system by dividing our work into logical items that can be farmed off amongst those processors.

See also:

- [Audio](#)
- [man\(7\) – `pthreads`](#)
- [POSIX thread \(`pthread`\) libraries](#)

## Open file descriptors

Almost everything in Raspbian is based around the notion of a file. Whether that file is a real file on the file system or a simply a file descriptor of an opened physical device, from the Raspbian process perspective, it is a file descriptor. Since these are owned and managed by the kernel itself, there is the possibility of using a tool to show us what file descriptors a process has open or, conversely, look at a device or file and ask what processes have it open.

The tool in question is called "lsof". It is not installed by default so it must be manually installed from the "lsof" package:

```
$ sudo apt-get install lsof
```

This command has a deep richness of options and great study of the man page would be needed to understand all the possibilities. However, there are some common recipes that one can use without knowing too much additional details.

Here is a recipe that shows which processes have a particular file open.

```
$ sudo lsof /dev/tty1
COMMAND PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
agetty  634 root    0u  CHR    4,1      0t0 1042 tty1
agetty  634 root    1u  CHR    4,1      0t0 1042 tty1
agetty  634 root    2u  CHR    4,1      0t0 1042 tty1
```

When it comes to networking, lsof is again extremely valuable. The primary TCP/IP networking API is called sockets and uses file descriptors as the means of maintaining context for a connection. Running "sudo lsof -i" lists the socket domain protocols.

```
$ sudo lsof -i
COMMAND PID   USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
systemd  1     root    33u  IPv6  8309      0t0  TCP localhost:gpsd (LISTEN)
systemd  1     root    34u  IPv4  8310      0t0  TCP localhost:gpsd (LISTEN)
avahi-dae 386 avahi   12u  IPv4  7073      0t0  UDP *:mdns
avahi-dae 386 avahi   13u  IPv6  7074      0t0  UDP *:mdns
avahi-dae 386 avahi   14u  IPv4  7075      0t0  UDP *:48241
avahi-dae 386 avahi   15u  IPv6  7076      0t0  UDP *:57837
dhcpcd   415 root    8u  IPv4  11289     0t0  UDP *:bootpc
sshd     518 root    3u  IPv4  10553     0t0  TCP *:ssh (LISTEN)
sshd     518 root    4u  IPv6  10555     0t0  TCP *:ssh (LISTEN)
ntpd     580 ntp    16u  IPv4  8699      0t0  UDP *:ntp
ntpd     580 ntp    17u  IPv6  8700      0t0  UDP *:ntp
ntpd     580 ntp    18u  IPv4  8707      0t0  UDP localhost:ntp
ntpd     580 ntp    19u  IPv6  8708      0t0  UDP localhost:ntp
ntpd     580 ntp    21u  IPv4  10688     0t0  UDP 192.168.1.101:ntp
ntpd     580 ntp    22u  IPv6  10689     0t0  UDP
[fe80::6773:4a57:bd35:7872]:ntp
apache2  641 root    3u  IPv6  8729      0t0  TCP *:http (LISTEN)
sshd     874 root    3u  IPv4  10694     0t0  TCP 192.168.1.101:ssh->pc:52557
(ESTABLISHED)
sshd     882 root    3u  IPv4  10704     0t0  TCP 192.168.1.101:ssh->pc:52558
(ESTABLISHED)
rpcbind  900 root    6u  IPv4  9548      0t0  UDP *:sunrpc
```

```

rpcbind 900 root 7u IPv4 9551 0t0 UDP *:651
rpcbind 900 root 8u IPv4 9552 0t0 TCP *:sunrpc (LISTEN)
rpcbind 900 root 9u IPv6 9553 0t0 UDP *:sunrpc
rpcbind 900 root 10u IPv6 9554 0t0 UDP *:651
rpcbind 900 root 11u IPv6 9555 0t0 TCP *:sunrpc (LISTEN)
sshd 905 pi 3u IPv4 10694 0t0 TCP 192.168.1.101:ssh->pc:52557
(ESTABLISHED)
sshd 905 pi 10u IPv6 10804 0t0 TCP localhost:6010 (LISTEN)
sshd 905 pi 11u IPv4 10805 0t0 TCP localhost:6010 (LISTEN)
sshd 905 pi 13u IPv6 23063 0t0 TCP localhost:6010-
>localhost:37031 (ESTABLISHED)
sshd 917 pi 3u IPv4 10704 0t0 TCP 192.168.1.101:ssh->pc:52558
(ESTABLISHED)
geany 2577 pi 3u IPv6 21788 0t0 TCP localhost:37031-
>localhost:6010 (ESTABLISHED)
sshd 2593 root 3u IPv4 23078 0t0 TCP 192.168.1.101:ssh->pc:60047
(ESTABLISHED)
sshd 2601 root 3u IPv4 23095 0t0 TCP 192.168.1.101:ssh->pc:60048
(ESTABLISHED)
sshd 2604 pi 3u IPv4 23078 0t0 TCP 192.168.1.101:ssh->pc:60047
(ESTABLISHED)
sshd 2604 pi 10u IPv6 23117 0t0 TCP localhost:6011 (LISTEN)
sshd 2604 pi 11u IPv4 23118 0t0 TCP localhost:6011 (LISTEN)
sshd 2616 pi 3u IPv4 23095 0t0 TCP 192.168.1.101:ssh->pc:60048
(ESTABLISHED)
apache2 3426 www-data 3u IPv6 8729 0t0 TCP *:http (LISTEN)
apache2 3427 www-data 3u IPv6 8729 0t0 TCP *:http (LISTEN)

```

This command is particularly useful if you are writing server applications and some other process already has the port you want to listen upon open. You can use "lsof" to determine which process that might be.

To filter the results we can use the "-i<filter spec>" where the filter specification is:

[46] [protocol] [@hostname|@IP Address] [:service|:port]

- 46 – Either 4 for TCP/IP v4 or 6 for TCP/IP v6
- @hostname – The hostname of the partner
- @IP Address – The IP address of the partner
- :service – The TCP/IP service name
- :port – The TCP/IP port number

For example, to see what processes are using the web server port:

```
$ lsof -i:80
COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME
apache2 641 root 3u IPv6 8729 0t0 TCP *:http (LISTEN)
apache2 3426 www-data 3u IPv6 8729 0t0 TCP *:http (LISTEN)
apache2 3427 www-data 3u IPv6 8729 0t0 TCP *:http (LISTEN)
```

See also:

- man(8) – [lsof](#)

## Networking

The story of networking on a Linux platform is deep and wide. Here are some of the key facts that will be necessary for you to learn.

To determine the Pi's IP address, use the command `ifconfig` which lists all the network interfaces known to Linux.

If your PC is a windows PC, get, install and learn the `PuTTY` tool. `PuTTY` provides a remote terminal client that includes both Telnet and SSH support. See here:

<http://www.putty.org/>

## Networking Devices

Depending on the model of Pi you are using you may or may not have an Ethernet port. If you don't, then you will likely be adding a WiFi USB dongle. In fact, even if you do have an Ethernet port, it is still likely you will be adding a dongle. Let us now look at some of the considerations when using these physical interfaces.

### Ethernet

If your Pi has an Ethernet port then it is based on the LAN9512 chip set with a 10/100Mb capacity.

#### Setting up a static Ethernet IP Address

There are many times that I find myself on the road with only a Pi and a laptop. As such, I have no WiFi nor HDMI video (TV). To be productive, I plug a regular Ethernet cable into my Pi 2 and the other end into my laptop. My laptop has a static IP address for its end of the Ethernet at 192.168.2.2 and I want my Pi to also have a static IP address of 192.168.2.3. When set up in that fashion, I can then reach one from the other. However, running the `ifconfig` command at the console may not be an option so I needed to find a way to configure such that the Ethernet static IP address was present at boot. The solution is quite simple. We edit the file called `/etc/dhcpcd.conf` and add the following lines at the end:

```
# Static IP configuration
interface eth0
static ip_address=192.168.2.3/24
```

Following a restart, the Pi is ready to be connected.

#### Using Windows Internet Connection Sharing

Imagine you are in a hotel room and you have your Windows laptop PC which is connected to the hotel WiFi. You now connect your Pi to the laptop via Ethernet and you can SSH into the Pi. Great. You then realize that you want to connect to the Internet from the Pi ... but ... huh ... it doesn't work. That is correct ... if we follow the story, the Windows PC is connected to the Internet and the Pi is connected to the PC but that doesn't mean that the Pi is connected to the Internet. A few new things have to happen. First of all, we need to enable "Internet Connection Sharing" on the WiFi adapter and name the Ethernet

adapter as the connection to be shared. Next, on the Pi, we need to add a default route statement that will send all traffic not on the Pi's local LAN to the Ethernet adapter on the PC which will THEN be routed to the Internet. Assuming that the PC's Ethernet address is 192.168.2.2, then on the Pi we would run:

```
$ sudo route add -net default gw 192.168.2.2
```

We can check that this has taken effect by running:

```
$ route
Kernel IP routing table
Destination      Gateway          Genmask         Flags Metric Ref    Use Iface
default          192.168.2.2      0.0.0.0        UG     0      0        0 eth0
192.168.2.0      *               255.255.255.0  U       202    0        0 eth0
```

We see that the default gateway is now 192.168.2.2.

Finally, we need to edit `/etc/resolv.conf` to add some default nameservers:

```
# Google IPv4 nameservers
nameserver 8.8.8.8
nameserver 8.8.4.4
```

And that should be it ... we can test by pinging some hostnames from the Pi which should now resolve. Note that if we are using DHCP client or some other networking components, the `resolv.conf` can be generated for us and will undo any hand editing we may have made. As an alternative, consider editing `dhcpcd.conf` and adding:

```
interface eth0
static ip_address=192.168.2.3/24
static domain_name_servers=8.8.8.8 8.8.4.4
```

The "static domain\_name\_servers" adds the entries we need.

## WiFi

Some of the more common WiFi dongles out in the market are based on the Realtek 8192cu chip. We can configure some of the lower level properties of the device driver for this chip by editing the file called `/etc/modprobe.d/8192cu.conf`. When these chips power up, they may have a power saving mode associated with them. What that means is that after a period of no network activity, they can suspend themselves. In principle, this sounds great, but in practice, it can be a pain. We find that network applications such as NFS start to hang when accessed as the network to the Pi has been put to sleep and can take time to come back on-line. It is not uncommon for us to want to suspend the power suspension capabilities of the device. To do that, we can edit the `8192cu.conf` file and add the following:

```
options 8192cu rtw_power_mgnt=0 rtw_enusbss=0 rtw_ips_mode=1
```

After adding and rebooting, the device will no longer suspend itself. To see the current values of the device configuration, examine the files at `/sys/module/8192cu/*`. The ones in the `parameters` sub-folder are especially useful.

The command `iwconfig` can be used to display and change a variety of parameters associated with WiFi devices. Running `iwconfig` by itself produces a good summary of the qualities of the attached WiFi devices:

```
$ iwconfig
wlan0      IEEE 802.11bgn  ESSID:"kolban"  Nickname:"<WIFI@REALTEK>"
            Mode:Managed  Frequency:2.442 GHz  Access Point: A4:2B:8C:81:47:95
            Bit Rate:72.2 Mb/s  Sensitivity:0/0
            Retry:off    RTS thr:off    Fragment thr:off
            Power Management:off
            Link Quality=100/100  Signal level=98/100  Noise level=0/100
            Rx invalid nwid:0  Rx invalid crypt:0  Rx invalid frag:0
            Tx excessive retries:0  Invalid misc:0  Missed beacon:0

lo        no wireless extensions.

eth0      no wireless extensions.

wlan1      IEEE 802.11bgn  ESSID:"kolban"  Nickname:"rtl_wifi"
            Mode:Managed  Frequency:2.442 GHz  Access Point: A4:2B:8C:81:47:95
            Bit Rate:150 Mb/s  Sensitivity:0/0
            Retry:off    RTS thr:off    Fragment thr:off
            Power Management:off
            Link Quality=100/100  Signal level=100/100  Noise level=0/100
            Rx invalid nwid:0  Rx invalid crypt:0  Rx invalid frag:0
            Tx excessive retries:0  Invalid misc:0  Missed beacon:0
```

See also:

- `man(8) – iwconfig`
- YouTube: [Raspberry Pi – WiFi Tutorial!](#)

## WiFi USB devices

There appears to be a limitless number of USB based WiFi connectors available. If you get one, it is difficult to know exactly what you have. One of the things to realize is that the physical unit you have is likely to be built using standard components and, if all is well, well behave based upon the nature of those standard components. If we plug the device into the USB port of our Pi and run the `lsusb` and `lsusb -v -s <device>` we will be presented with a list of details. Here is a portion of output from my execution:

```
$ sudo lsusb -v -s 7
...
idVendor      0x0bda Realtek Semiconductor Corp.
idProduct     0x8172 RTL8191SU 802.11n WLAN Adapter
bcdDevice     2.00
iManufacturer 1 Manufacturer Realtek
iProduct      2 RTL8191S WLAN Adapter
...
```

and here is another run with a different USB:

```
$ sudo lsusb -v -s 4
...
```

```

idVendor          0x0bda Realtek Semiconductor Corp.
idProduct         0x8176 RTL8188CUS 802.11n WLAN Adapter
bcdDevice         2.00
iManufacturer     1 Realtek
iProduct          2 802.11n WLAN Adapter
...

```

and yet another:

```

idVendor          0x148f Ralink Technology, Corp.
idProduct         0x7601
bcdDevice         0.00
iManufacturer     1 MediaTek
iProduct          2 802.11 n WLAN

```

Despite being from different vendors, the first two dongles have internals from the same underlying WiFi technology provider (Realtek). However, they are both based on different chip-sets.

- [RTL8191SU](#)
- [RTL8188CUS](#)

Here is one more entry:

```

idVendor          0x0bda Realtek Semiconductor Corp.
idProduct         0x8179
bcdDevice         0.00
iManufacturer     1
iProduct          2

```

Notice in this case there isn't a description of the product (idProduct). However, knowing the vendor id (0bda) and the product id (8179), we can look it up in the USB database of registrations. For example visiting the site:

<http://thesz.diecru.eu/content/usbid.php>

We can enter a vendor and product id and be given a description of what it is:

### USB ID Database

Search for USB devices with Vendor ID, Product ID and/or Name:

Vendor ID:	0x0BDA
Product ID:	0x8179
Name:	Name
<input type="button" value="🔍 Search"/>	

Search Results:

Vendor ID	Product ID	Name	Comment
0x0BDA		<a href="#">Realtek Semiconductor Corp.</a>	<a href="http://www.realtek.com.tw/">http://www.realtek.com.tw/</a>
0x0BDA	0x8179	<a href="#">Realtek Semiconductor Corp.</a>	RTL8188EUS 802.11n Wireless Network Adapter

In this case another Realtek device based on the RTL8188EUS.

Vendor id	Product id	Description
148f	7601	MT7601U – Ralink – iw list works – no AP mode
0bda	8179	RTL8188EUS (TP-LINK)
0bda	8172	RTL8191SU (Antenna)
0bda	8176	RTL8188CUS (Vilros)
148f	3070	RT3070 – Ralink

See also:

- USB
- [Realtek home page](#)
- Wikipedia – [Ralink](#)

### Scanning WiFi networks

From the shell, the following command will list the available WiFi networks:

```
$ sudo iwlist wlan0 scan
```

### Setting up WiFi connections

Assuming you know the network ID (SSID) of your WiFi network and your password, you can have Raspbian remember these such that when your Pi boots, it will automatically attempt to connect to the network. The configuration file can be found at `/etc/wpa_supplicant/wpa_supplicant.conf`. Within this file, we can add a stanza that reads:

```
network={  
    ssid="Your_SSID"  
    psk="Your_WiFi_password"  
    key_mgmt=WPA-PSK  
}
```

If the change is not automatically detected, restart your WiFi adapter with the following commands:

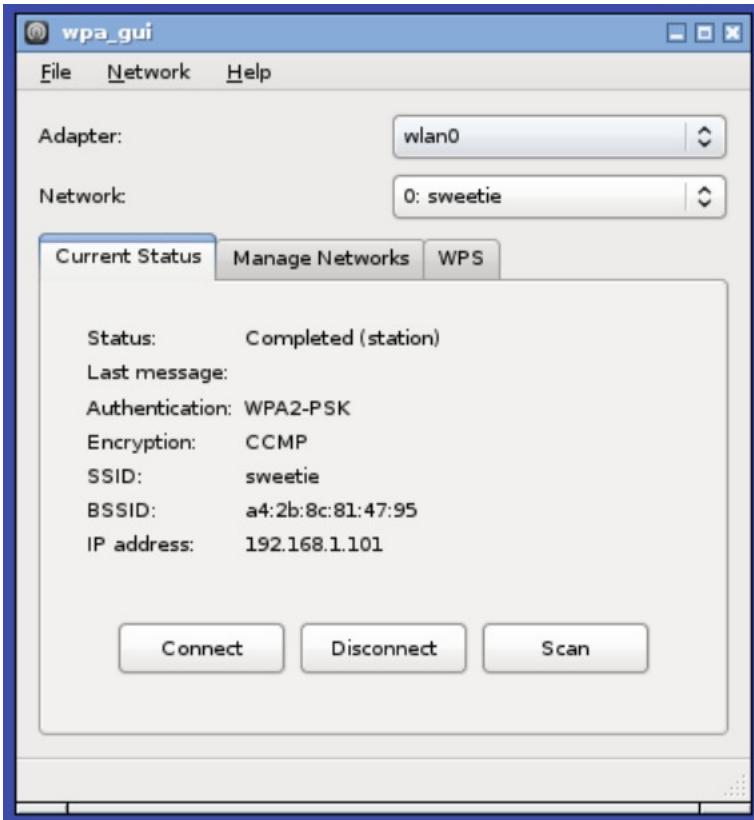
```
$ sudo ifdown wlan0  
$ sudo ifup wlan0
```

As an alternative to these command line options, we can also choose to use a GUI tool. By default, the tool is not installed but can be by running:

```
$ sudo apt-get install wpagui
```

The problem with this is that it assumes you already have network connectivity in order to download the tool. Unless you have transiently connected with an Ethernet cable, this is unlikely to be the case as the reason you need the tool in the first place is to set up WiFi connections.

However, assuming you do have the tool installed, running `wpa_gui` will launch the tool:



To setup a static WiFi address, it depends on whether or not we are being a client of a DHCP server. If we are, which is common ... then we need to edit the /etc/dhcpcd.conf file. At the end, we want to add a stanza that reads:

```
ssid <ssid name>
inform <chosen IP address>
static routers=192.168.1.1
static domain_name_servers=8.8.8.8 8.8.4.4
```

### Setting up the Pi as an Access Point

Normally when we think of a WiFi device we think of it as connecting to our hub at home that is connected to the Internet via DSL or cable or some other broadband mechanism. What is really happening here is that the WiFi device you are connecting from is acting as a WiFi Station and the hub is acting as a WiFi Access Point.

It is also possible for the Pi to behave as an Access Point and have other WiFi stations connect to it.

The core to setting up an access point is the package called hostapd. Once installed, its configuration file must be setup. The file can be found at /etc/hostapd/hostapd.conf. By default, it doesn't exist so one will have to be created. A sample set of configurations are:

```

interface=wlan0
driver=nl80211
ssid=My_AP
hw_mode=g
channel=6
macaddr_acl=0
auth_algs=1
ignore_broadcast_ssid=0
wpa=2
wpa_passphrase=My_Passphrase
wpa_key_mgmt=WPA-PSK
wpa_pairwise=TKIP
rsn_pairwise=CCMP

```

Despite all this theory, the reality is that it is finicky. So far the only permutation I have had to work is with a Vilros WiFi dongle that reports as USB vendor/device pair as 0bda:8176 and even then using a version of hostapd that I know very little about (<http://www.daveconroy.com/wp3/wp-content/uploads/2013/07/hostapd.zip>) and using the following hostapd.conf:

```

interface=wlan1
#driver=nl80211
driver=rtl871xdrv
ssid=test
macaddr_acl=0
auth_algs=1
ignore_broadcast_ssid=0
channel=7
hw_mode=g

```

There is a lot of voodoo in this one. When a station connects to our Pi which is running as an access point, it will need to allocate an IP address. This means we need to be running a DHCP server. There appear to be a few variants available including:

- udhcpd
- isc-dhcp-server

For me, when I run:

```
$ sudo hostapd ./hostapd.conf
```

Using the standard distribution of hostapd, the error I continually receive is:

```

Configuration file: ./hostapd.conf
nl80211: Driver does not support authentication/association or connect commands
nl80211 driver initialization failed.
hostapd_free_hapd_data: Interface wlan0 wasn't started

```

We can build hostapd from source. First, make a working directory in which we are going to work. In that directory, download the latest source package from <http://w1.fi/hostapd/>. As of writing this is the file called hostapd-2.5.tar.gz. Extract the contents.

From there, change into relative directory hostapd

If you don't have the package called "libnl-dev" installed, install it.

If you don't have the package called "libssl-dev" installed, install it.

Next copy the file `defconfig` to `.config`.

Run make to build the executable. When done, if we then run "`hostapd -v`" we will see that we are running v2.5 from the following output:

```
$ ./hostapd -v
hostapd v2.5
User space daemon for IEEE 802.11 AP management,
IEEE 802.1X/WPA/WPA2/EAP/RADIUS Authenticator
Copyright (c) 2002-2015, Jouni Malinen <j@w1.fi> and contributors
```

When running hostapd for a WiFi interfaces, it is essential that the named interface is **not** also being managed as a WiFi station via `wpa_supplicant`.

Once we have setup the WiFi dongle as an access point, we are not yet finished.

See also:

- Adafruit – [Setting up a Raspberry Pi as a WiFi access point](#)
- elinux – [RPI Wireless Hotspot](#)
- Instructables – [WiFi – Access Point out of a Raspberry Pi \(Repeater\)](#)
- YouTube – [Raspberry Pi – Wireless Access Point](#)
- [How-To: Turn a Raspberry Pi into a WiFi router](#)
- Wikipedia – [hostapd](#)
- [Hostapd home page](#)
- [hostapd configuration file](#)

### Wireless Supplicant

Before working with the Pi, I had heard of the English word "supplicant" but could not have quoted its definitions. Looking it up in a dictionary, we find it means "a person who asks for something in a respectful way from a powerful person or God". What an interesting word.

The purpose of a Wireless supplicant is to allow a computer to ask the WiFi access point for permission to join the WiFi network. On the Pi, this is implemented by the `wpa_supplicant` implementation. This is the service that is responsible for authenticating with an access point.

See also:

- `wpa_supplicant` home page
- Wikipedia – [Wireless supplicant](#)
- Wikipedia – [wpa\\_supplicant](#)
- Wikipedia – [WiFi Protected Access](#)

## Programming to WiFi

We have seen that there are a variety of commands for working with WiFi but what if we wish to configure or work with WiFi from an API layer ... is that possible? The answer is yes ... but it isn't the easiest thing in the world. Raspbian uses a kernel level interface driver called "cfg80211" and that has a library called "nl80211" that can be used by applications to act as a user space bridge into the kernel functions. Tools such as "iw" and "hostapd" use this API and since these are Linux open source, one can look at how they work to learn how to leverage this API yourself.

See also:

- [nl80211](#)

## Using a Windows PC as a WiFi access point / hotspot

Imagine you are out in the wilderness and you wish to connect to a Pi that you have brought with you. You also have a Windows based PC. The problem is that there is no WiFi network nearby ... what is a guy to do? Fortunately, there is a rather elegant solution. Windows has the capability to configure a suitable WiFi network device such as a dongle to be an Access Point. What that means is that a Pi that also has a network dongle can join the network that the Windows PC is advertising ... and now we have a network. The key to making this all work is the "netsh" command on Windows.

After having plugged in the WiFi dongle, open a DOS command prompt and run the command:

```
C:> netsh wlan show drivers
```

This will produce a lot of information but the key entry we are looking for is the line which reads:

```
Hosted network supported : Yes
```

If the answer is "No" then we can't make any further progress with this device. If the answer is "Yes" then we can move on. To create a hot-spot, we run the command:

```
C:> netsh wlan set hostednetwork mode=allow ssid=<networkName> key=<password>
The hosted network mode has been set to allow.
The SSID of the hosted network has been successfully changed.
The user key passphrase of the hosted network has been successfully changed.
```

Now that you have defined the parameters, we can run the command:

```
C:> netsh wlan start hostednetwork
```

Now the Pi can connect to the network.

See also:

- [About the Wireless Hosted Network](#)
- [How to Create WiFi HotSpot in Windows 8 & Windows 8.1 – Share Laptop Internet Connection](#)

## DHCP

Consider plugging your Pi into an Ethernet Lan or starting up a WiFi device ... somehow your Pi gets an IP address ... but where from? The answer is a technology called DHCP. Rather than juggle multiple devices, we'll think exclusively just about WiFi for now. When you power up your Pi and it has a WiFi

dongle, it connects to your local WiFi access point. When it connects, it asks the access point for an IP address. The access point allocates one for it and then the WiFi device in the Pi uses that address. The protocol exchanged between the Pi and the access point is DHCP.

From a Pis perspective, it can be either a DHCP server in which case it owns addresses to allocate to other who connect to the Pi or else, and this is by far the most common scenario, it is a DHCP client that interacts with another DHCP server on the network. When the Pi is being a DHCP client, it is running a daemon called "dhcpcd". The dhcpcd daemon is configured by the configuration file called /etc/dhcpcd.conf. To say that there is a lot to consider is an understatement. Researching the internet, there aren't a lot of tutorials on it either. However, what we can learn are certain cheats that will get the job done. If we look inside our default configuration file, we will find many of the following entries:

- clientid – Send a clientid. If no clientid is present then a default one will be sent.
- hostname – Send the hostname to the DHCP server.
- interface <interface name> - Subsequent options are for this specific interface only.
- ssid <SSID> - Subsequent options are for this SSID only.
- static <value> - Use the values supplied here as opposed to asking the DHCP server for them. Common entries are:
  - static ip\_address=<IP> - The IP address to assign
  - static routers=<IP> - The default route through this address
  - static domain\_name\_servers=<IP> - Domain name servers to use

To setup a static IP address for WiFi, the following can be used.

In the /etc/network/interfaces file, ensure there is an entry that reads:

```
allow-hotplug wlan0
iface wlan0 inet manual
    wpa-conf /etc/wpa_supplicant/wpa_supplicant.conf
```

In the /etc/wpa\_supplicant/wpa\_supplicant.conf, file ensure there is an entry that reads:

```
network={
    ssid=""
    psk=""
    key_mgmt=WPA-PSK
}
```

In the /etc/dhcpcd.conf file, ensure there is an entry that reads:

```
ssid <Your SSID>
inform <Your static IP>
static routers=192.168.1.1
static domain_name_servers=8.8.8.8 8.8.4.4
```

See also:

- WikiPedia: [Dynamic Host Configuration Protocol](#)
- man(8) – [dhcpcd](#)
- man(5) – [dhcpcd.conf](#)

## Remote login

To login from one machine to another we can use a variety of techniques. It is likely the case you want to login from your PC into your Pi, however the reverse is also possible depending on what is running on your PC. We will assume for just now that the goal is to get from the PC to the Pi.

One of the most common remote login tools is called `ssh`. We can use the command:

```
$ ssh <userid>@<host>
```

to open a new session to the remote host and login with the supplied userid. At this point, we will be prompted for the password of the userid. It is common in our development environment that we will be doing lots of remote ssh commands and we want our local PC userid to be a trusted partner to the remote. If our PC is running a Linux flavor, the following recipe can be followed to set up trust between the two. In our story we will assume that the PC has a hostname of "host-pc" and that the Pi has the hostname of "host-pi". We will also assume that the userid on the PC is "pcuser" while on the Pi it will be "pi".

On the PC, in the user's home directory, run:

```
$ ssh-keygen -t rsa
```

When prompted for a pass phrase, simply hit enter and not provide one.

On the Pi, login as "pi" and make a directory in the home directory called ".ssh".

Back on the PC, in the user's home directory, we want to copy the public key into the newly created directory on the Pi appending to a new file called "authorized\_keys". We can do this with:

```
$ cat .ssh/id_rsa.pub | ssh pi@host-pi 'cat >> .ssh/authorized_keys'
```

Following this, if we now run

```
$ ssh pi@pi-host
```

on the PC, we will immediately be logged into the Pi without password challenge.

There are alternative commands for remote login such as `telnet` and `rlogin` but `ssh` is considered the current best practice.

## Domain Name Service

The internet (or other TCP/IP network) uses IP addresses to identify machines. An internet address is a 32 bit address commonly written as 4 decimal bytes of data separated by periods... eg. 1.2.3.4. Note that this is known as an IPv4 address and there are now also IPv6 addresses which are 128 bit addresses.

Trying to refer to a machine by its IP address would be a big challenge to say the least. To solve that problem, a resolution mechanism was created which maps names to IP addresses. This is called the

Domain Name Service (DNS). When we know the name of a machine, we can supply its name and DNS will return the IP address corresponding to that name. Once the software knows the IP address, it can then communicate with the target directly. As you can imagine the database of IP addresses to names could be incredibly large ... and it is. In fact it is what is known as a distributed database meaning that the mapping is itself partitioned down into lower and lower levels. So instead of one massive database, there are smaller and smaller databases that have administrative control over subsections. If we were to think of it in a hierarchical fashion, we have databases for each of the "top level domains". These are:

- com
- org
- net
- gov
- etc etc etc

For each entry in those databases, they then point to machines which may then host further databases ... for example "google.com" would look up the database entry at "com" for the entry for "google". If we further ask for "play.google.com", this would then look up the entry for "google.com" which would then ask a DNS server there for the address of the machine called "play". Thankfully, we don't have concern ourselves with these levels of details when actually resolving host names to IP addresses when using the Pi. Instead, we tell the Pi about a single DNS server that is part of the hierarchy and we send it requests. It in turn knows the locations of top level domain servers and the requests get forwarded up the chain until we achieve resolution. There are other optimizations at play including caching which make this more than acceptably fast. The question though ... is how does the Pi find the address of the original DNS server that it should use to begin with? The answer to this depends on how your Pi connects to your network. If you are using DHCP, then when the network starts your Pi will contact the DHCP server on your own network and will be boot-strapped with all the information it needs from the DHCP server. This obviously moves the problem further to the right ... but the chances are high that your DHCP server is built into your modem supplied by your internet service provider (ISP) such as your cable or DSL provider. When they supplied the modem, they configured the up-stream services and your modem "by magic" gets the information it needs.

However ... there is also the concept of your LAN. Addresses in the range 192.168.1.x are defined to be local addresses. This means that the network traffic on that LAN lives only locally and doesn't propagate into the Internet. For example, in my house, my Pi might have the IP address 192.168.1.2 while in your house, your Pi might have exactly the same address. This means that you can connect to other devices in your own house that have IP addresses in the range 192.168.1.1 to 192.168.1.254 with impunity while I can do the same and none of us will tread on each other. This too is great ... but what now if I don't want to remember that 192.168.1.2 is my Pi while 192.168.1.3 is Windows Pc while 192.168.1.4 is a Linux box. What we want is the ability to resolve host names just as we do with DNS. For example, I might want to ping "mypi" or "mywindows" or "mylinux" by name rather than remember the addresses.

This is where the system file called `/etc/hosts` comes into play. This is a text file which we can edit. Within the file we can add lines of the form:

```
<IP Address> <hostname>
```

for example:

```
192.168.1.2 mypi  
192.168.1.3 mywindows  
192.168.1.4 mylinux
```

Once added to the file, any attempt to resolve these host names would succeed. What happens is that the name resolution subsystem of Raspbian first looks for a match of host name in `/etc/hosts` and, if not resolved there, **then** makes contact with a DNS server for Internet resolution.

If we know the host name of a machine and want to see its IP address, we use the command:

```
$ host <hostname>
```

for example:

```
$ host www.google.com  
www.google.com has address 216.58.217.4  
www.google.com has IPv6 address 2607:f8b0:400f:802::2004
```

Note that the order of resolution for a host name is governed by the content of the file called `/etc/nsswitch.conf`.

## DNS Service Discovery and Multicast DNS

Avahi runs as the `systemd` daemon called "avahi-daemon". We can determine whether or not it is running with:

```
$ systemctl status avahi-daemon  
● avahi-daemon.service - Avahi mDNS/DNS-SD Stack  
   Loaded: loaded (/lib/systemd/system/avahi-daemon.service; enabled)  
   Active: active (running) since Wed 2016-01-20 22:13:35 CST; 1 day 13h ago  
     Main PID: 384 (avahi-daemon)  
       Status: "avahi-daemon 0.6.31 starting up."  
      CGroup: /system.slice/avahi-daemon.service  
              ├─384 avahi-daemon: running [raspberrypi.local]  
              └─426 avahi-daemon: chroot helper
```

The `avahi-daemon` utilizes a configuration file found at `/etc/avahi/avahi-daemon.conf`. The default name that `avahi` advertizes itself as is the local hostname.

When hostname resolution is performed, the system file called `/etc/nsswitch.conf` is used to determine the order of resolution. Specifically the hosts entry contains the name resolution. An example would be:

```
hosts:          files mdns4_minimal [NOTFOUND=return] dns
```

Which says "first look in `/etc/hosts`, then consult mDNS and then use full DNS". What this means is that a device which advertizes itself with mDNS can be found via a lookup of "`<hostname>.local`". For example, if I boot up a Pi Zero which gets a dynamic IP address through DHCP and the hostname of that machine is "pizero", then I can reach it with a domain name address of "pizero.local". If the IP

address of the device changes, subsequent resolutions of the domain name will continue to correctly resolve.

Avahi tools are not installed by default but can be installed using the "avahi-utils" package:

```
$ sudo apt-get install avahi-utils
```

To see the list of mDNS devices in your network, we can use the avahi-browse command. For example:

```
$ avahi-browse -at
+ wlan0 IPv6 raspberrypi-2                               Remote Disk Management
local
+ wlan0 IPv6 raspberrypi                                Remote Disk Management
local
+ wlan0 IPv4 raspberrypi-2                               Remote Disk Management
local
+ wlan0 IPv4 raspberrypi                                Remote Disk Management
local
+ wlan0 IPv6 raspberrypi                                _arduino._tcp      local
+ wlan0 IPv4 raspberrypi                                _arduino._tcp      local
+ wlan0 IPv6 raspberrypi-2 [00:e0:4d:04:fd:83]          Workstation       local
+ wlan0 IPv6 raspberrypi [60:e3:27:11:14:58]            Workstation       local
+ wlan0 IPv6 kolban-VirtualBox [08:00:27:eb:48:37]       Workstation       local
+ wlan0 IPv4 raspberrypi-2 [00:e0:4d:04:fd:83]          Workstation       local
+ wlan0 IPv4 raspberrypi [60:e3:27:11:14:58]            Workstation       local
+ wlan0 IPv4 kolban-VirtualBox [08:00:27:eb:48:37]       Workstation       local
+ wlan0 IPv4 Living Room                                _googlecast._tcp   local
```

See also:

- [avahi home page](#)
- [man\(1\) – avahi-browse](#)
- [man\(5\) – avahi-daemon.conf](#)

## Web browsers and servers

Where would the internet be without browsers and web servers? When we think about a Pi, it isn't immediately obvious that there might be value in the Pi in this Web space but I think we will find that there actually is quite a bit of function we can use here.

### Pi as a Web Server

Let us start with the notion of the Pi as a Web Server. One of the most prevalent web server software packages available is the Apache web server. This web server runs just fine on a Linux environment and Raspbian is no exception. The installation of Apache can be performed via the apt-get command.

```
$ sudo apt-get install apache2
```

After installation, the root of the web site can be found at /var/www/html. To test, we can point a browser at the URL to the IP address on which Pi is running and the default web page will appear:

The screenshot shows a web browser window titled "Apache2 Debian Default Page". The address bar displays "192.168.1.101". The main content area features the Debian logo and the title "Apache2 Debian Default Page". A red banner at the top right contains the text "It works!". Below this, a message states: "This is the default welcome page used to test the correct operation of the Apache2 server after installation on Debian systems. If you can read this page, it means that the Apache HTTP server installed at this site is working properly. You should **replace this file** (located at /var/www/html/index.html) before continuing to operate your HTTP server." Another message below it says: "If you are a normal user of this web site and don't know what this page is about, this probably means that the site is currently unavailable due to maintenance. If the problem persists, please contact the site's administrator." A section titled "Configuration Overview" discusses the configuration layout for an Apache2 web server installation on Debian systems, mentioning files like apache2.conf, ports.conf, and mods-enabled. A code block shows the directory structure of /etc/apache2.

```
/etc/apache2/
|-- apache2.conf
|   '-- ports.conf
|-- mods-enabled
```

The apache server is registered with `systemd` for operations. This means that we can query its status with

```
$ sudo systemctl status apache2
● apache2.service - LSB: Apache2 web server
  Loaded: loaded (/etc/init.d/apache2)
  Active: active (running) since Fri 2016-01-22 12:09:54 CST; 29s ago
    Process: 1426 ExecStart=/etc/init.d/apache2 start (code=exited, status=0/SUCCESS)
   CGroup: /system.slice/apache2.service
           ├─1441 /usr/sbin/apache2 -k start
           ├─1444 /usr/sbin/apache2 -k start
           └─1445 /usr/sbin/apache2 -k start
```

```
Jan 22 12:09:53 raspi apache2[1426]: Starting web server: apache2AH00558: apache2:
Could ...age
Jan 22 12:09:54 raspi apache2[1426]: .
Jan 22 12:09:54 raspi systemd[1]: Started LSB: Apache2 web server.
```

We can also start and stop server using the `systemctl` command. If we wish to disable the server from starting at boot time we can use the "disable" option of `systemd`.

See also:

- [Apache HTTP Server Project](#)

## **REST requests**

In the field of computers, the moment that there was more than one in the world, we had the notion that they could collaborate to get work done. As networked devices have become the norm, various protocols and techniques have been designed to allow applications to work with each other. These distributed computing technologies have been widely varied and include such members as Sockets, DCE RPC, Web Services (SOAP/HTTP), JMS and more. Here we look at a specific technique called "REST". REST is the usage of the HTTP protocol to transmit information from a caller to be processed by a server. As such, it is client/server oriented. The client formulates a REST request by building and sending an HTTP protocol request over a TCP/IP network. This request is commonly received by a server application that is listening at a well known port. On receipt of the request, the server analyzes the incoming data and performs some corresponding task using the data supplied. If needed, the server can then send back a response message to the original caller.

REST calls are stateless. When a call is made from the client to the server and the call completes, there is no context maintained about that previous call.

See also:

- Web programming
- cURL API programming
- Thingspeak.com

## **Using postman for testing**

There are a wide variety of tools available to test and debug REST based services. My favorite is called "postman". This is a Google Chrome based application that allows one to enter REST requests of all types with all kinds of parameters. These can be saved for later re-use. Once the data has been entered, postman can send the REST request and show you the response.

See also:

- [Postman home page](#)

## **Writing an App that makes a REST request**

For C programming, the REST interface library I recommend is called `libcurl`. This can be read about in more detail at the cURL section of the book.

## **Writing a REST request using Node.js**

Baked into the Node.js supplied libraries are packages called "`http`" and "`https`". The first makes un-encrypted HTTP calls while the second makes SSL encrypted HTTP calls. By using the `request()` method of classes we can make REST calls.

The general syntax of the command is:

```
http.request(options, [callback])
```

The options parameter is a structure which contains the details of the request to be made. The callback parameter is a JavaScript function that will be invoked when a response is returned.

The full details of the function can be found in the documentation so need to repeat them here. Let us look at a sample scenario. Imagine there is a web service at `https://example.com` that when passed a request to `/sales` with an `accessKey` query parameter returns a sales record in JSON. For example:

```
https://example.com/sales?accessKey=1234
```

the response would be:

```
{
  "unitsSold": 100,
  "revenueEarned": 456.78
}
```

To achieve this in Node.js, we would use logic similar to the following:

```
var http = require('http');
http.request({
  host: "example.com",
  path: "/sales?accessKey=1234",
  method: "GET"
}, function(res) {
  res.setEncoding("utf8");
  res.on("data", function(data) {
    var parseData = JSON.parse(data);
    revenueEarned = parseData.revenueEarned;
    // ... process here
  });
});
```

See also:

- Node.js – [http.request\(\) API](#)

## Writing an App that listens for REST requests

A full-blown Web Server can listen for Web requests and service them but the core notion of a REST service provider is that it is usually an application that is passively waiting for client requests. For the Pi, this would mean that we want to listen for incoming requests and, when one arrives, process it. There are solutions for this puzzle for most of the languages that we may wish to want to choose.

### Listening for REST requests using Node.js

Node.js has a pre-supplied Web Server class. This makes it trivial to start listening for incoming client requests. Here is an example of use:

```
var http = require('http');

const PORT=8080;

function handleRequest(request, response) {
```

```

        response.end('Client request arrived: ' + request.url);
    }

var server = http.createServer(handleRequest);

server.listen(PORT, function() {
    console.log("Server listening on: http://localhost:%s", PORT);
});

```

For an HTTP GET request, it may carry query parameters of the form:

<name>=<value>&<name>=<value>...

We can parse these out using the Node.js class called "url". For example:

```
var params = require('url').parse(request.url, true);
```

In addition, from the parsed URL, we can also find the pathname property which, in REST, is equivalent to the name of the service being requested.

See also:

- [Build your first HTTP server in Node.js](#)
- Node.js – [HTTP class](#)
- Node.js – [URL class](#)

## WebSocket

WebSocket is both an API and a protocol introduced in HTML5. Simply put, if we imagine an HTTP server sitting waiting for incoming HTTP requests, we can convert a current request into a socket connection between the server and the browser such that either end can send data to be received by its partner.

Here we see a raw request to upgrade an HTTP connection to a WebSocket connection:

```

GET / HTTP/1.1
Host: 192.168.1.10
Connection: Upgrade
Pragma: no-cache
Cache-Control: no-cache
Upgrade: websocket
Origin: file://
Sec-WebSocket-Version: 13
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/46.0.2490.86 Safari/537.36
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
Sec-WebSocket-Key: saim6TzFH+zVb4qY2nrh0Q==
Sec-WebSocket-Extensions: permessage-deflate; client_max_window_bits

```

See also:

- html5rocks – [Introducing WebSockets: Bringing Sockets to the Web](#)
- [The WebSocket protocol – RFC6455](#)
- [The WebSocket API](#)
- [websocket.org](#)
- Wikipedia – [WebSocket](#)

## A WebSocket browser hosted application

The highest likelihood is that you will be running your Pi as a WebSocket server. This would imply that you are going to have browser hosted applications that will be connecting to a WebSocket server as clients and from there, you will likely be writing some WebSocket client code ... if nothing else then for unit testing your server. Because of that, we will now spend some time talking about what is involved in writing a WebSocket client application.

Let us assume that we will be writing JavaScript hosted in the browser. We start by creating an instance of a WebSocket object passing in the URL to the WebSocket server:

```
var ws = new WebSocket("ws://<somehost>[:<someport>]");
```

The WebSocket API is mostly event driven and there are a number event types of interest to us:

- `open` – Invoked when the connection to the WebSocket server has been established and we are now ready to send or receive data. We can define this with the "onopen" property of the WebSocket as a function reference.
- `message` – Receive a message from the server. We can define this with the "onmessage" property of the WebSocket as a function reference.
- `error` – Receive an indication that an error was detected. We can define this with the "onerror" property of the WebSocket as a function reference.
- `close` – Receive an indication that a request to close the connection was detected. We can define this with the "onclose" property of the WebSocket as a function reference.

Event handlers can be registered either with an "on<Event>" mechanism or with an `addEventListener()` call.

There are two methods defined on a WebSocket object. Those are:

- `send` – Send data to a WebSocket server
- `close` – Close the connection to a WebSocket server. The `close()` method takes two parameters:
  - `close code` – An integer close code describing the reason for the close.
    - 1000 – `CLOSE_NORMAL`
    - 1001 – `CLOSE_GOING_AWAY`
  - `status message` – A string describing the close reason.

Finally, there are a few attributes:

- `readyState` – The state of the WebSocket connection. Values include:

- WebSocket.CONNECTING
- WebSocket.OPEN
- WebSocket.CLOSING
- WebSocket.CLOSED
- bufferedAmount – Amount of data that is buffered pending transmission to the WebSocket server
- protocol – The WebSocket server selected protocol being used.

## Pi as a WebSocket server

When we start to think of the Pi as processing WebSocket requests, we now have options available to us. When the browser requests that a WebSocket request is made to the Pi, there will be a server side (Pi) application that owns the server side half of the socket connection. That application is one that you will be writing. As such, the nature of that application should be under your control and, of course, the core nature of the application is the language in which it is written. As we have seen, we have a rich assortment of languages available to us and now we need to look at the different technologies we could use to leverage a WebSocket from that language.

### WebSocket Server for C using libwebsocket

To build `libwebsocket`, we must install the pre-req packages called `cmake` and `libssl-dev`. After that, one need only follow the instructions defined in the `libwebsocket` web site and the build completes cleanly on a Pi environment.

See also

- [libwebsockets.org](http://libwebsockets.org)

### WebSocket Server for C using noPoll

The library called "noPoll" is an open source implementation of WebSocket for C. It has been found to compile cleanly for the Pi on Raspbian. After compilation and installation, the headers can be found in `/usr/local/include/nopoll` and the linkable libraries in `/usr/local/lib`.

To use noPoll in your app, you must include `"nopoll.h"`.

The context is very important and should be created at the start and disposed of at the end. For example:

```
noPollCtx *ctx = nopoll_ctx_new();
// Do something ...
nopoll_ctx_unref();
```

Once a context has been created, we can then register ourselves as a server:

```
noPollCon *listener = nopoll_listener_new(ctx, "0.0.0.0", "1234");
nopoll_ctx_set_on_msg(ctx, listener_on_message_handler, NULL);
nopoll_loop_wait(ctx, 0);
```

When a message is received, the registered function (`listener_on_message_handler`) will be invoked:

```
void listener_on_message_handler(
    noPollCtx *ctx,
    noPollConn *conn,
    noPollMsg *msg,
    noPollPtr userData) {
    // Do something
    nopoll_conn_send_text(conn, "Thanks", 5);
}
```

See also:

- [noPoll home page](#)
- [noPoll core library manual](#)
- [noPoll modules](#)

## WebSocket Server of JavaScript

See also:

- [socket.io](#)

## Sharing Windows files

It is likely we will want to access and share files between the Pi and either a Windows or Linux desktop. This means that we will want to expose file systems from the Pi and access them on the remote machines.

Let us assume that we wish to share file from the Pi so that they are accessible from a Windows machine.

To start we need to add a couple of packages to our Pi environment. These are called `samba` and `samba-common-bin`.

```
sudo apt-get install samba samba-common-bin
```

## Network File System (NFS)

The Network File System (NFS) allows us to mount directories hosted by one Linux machine on the file system of another. For example, imagine the Pi exposed a local file system and our development Linux system mounted that exposed file system. We could then edit and compile files on our PC desktop and the files would immediately be present on the Pi.

To setup NFS on a Linux machine, we need the following packages installed:

- `nfs-common` (already installed on Jessie)
- `nfs-server`

- portmap (already installed as rpcbind on Jessie)

```
sudo apt-get install nfs-common
```

Mount a remote file system

```
$ mount -t nfs <IP address>:<remote directory> <local directory>
```

To see what has been exported from a remote NFS server, run:

```
$ sudo showmount -e <host>
```

to see what has been exported from our local NFS server run:

```
$ sudo showmount -e
```

To configure an NFS server:

1. Make sure that the directory you wish to share exists.
2. Add it to /etc/exports ... for example

```
/home/pi *(rw,sync)
```

3. Reload the NFS configuration by running

```
$ sudo exportfs -r
```

4. Validate that the file systems are not exported by running

```
$ showmount -e
```

5. We may have to enable rpcbind with:

```
$ sudo update-rc.d rpcbind enable
$ sudo rpcbind start
```

To force a restart of the NFS subsystem on the server run:

```
/etc/init.d/nfs-kernel-server restart
```

or through the service command with

```
sudo service nfs-kernel-server restart
```

To un-mount an NFS mounted file system, we can use the `umount` command. If we try and access an NFS mounted file system and the network is not present or down, we will block the caller. If we want to force an un-mount an NFS file system while the network is down, we must use the `umount -l` command which performs a lazy un-mount.

## Cloud Systems

The "Cloud" has become one of those catch-all phrases that has as much hype associated with it as detail. Broadly speaking, Cloud is the notion that some resource that you want to use is hosted for you on the

Internet. Classically, this would have been a service such as a Web Service (SOAP/HTTP) or a REST service. In more recent days it now encompasses virtual machines and disk storage.

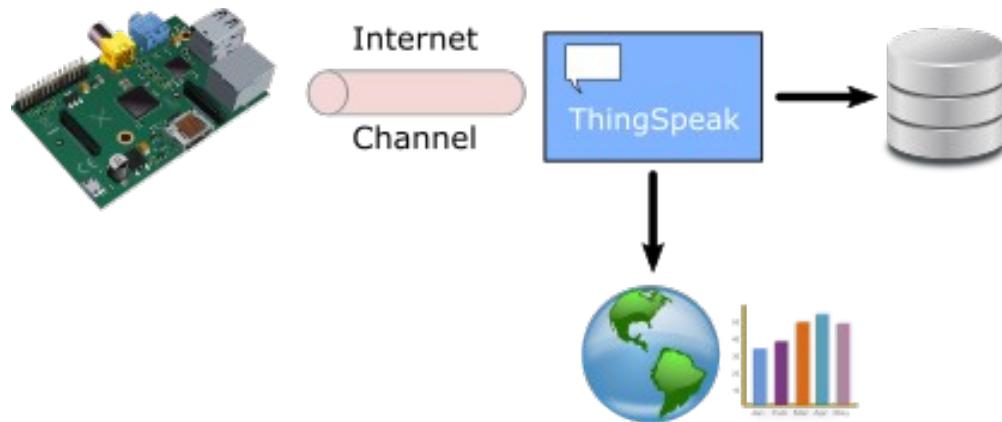
## Thingspeak.com

Thingspeak is a free service that allows you to record data into an Internet based simple database. The data that is expected to be recorded there is sensor data from devices attached to MCUs such as the Pi.

One can register for a free account. Once you have an account, you can create one or more channels. Think of a channel as a recording of data for a specific purpose. I may want to record the temperature outside of my house or I may want to record every time I open a bottle of beer. These sensor recordings could be directed to different channels. As such, think of the channel as the collection of sensor reading that I want to record as a group ... most likely because they are related.

Thingspeak allows us to record the data over the Internet to servers maintained by Thingspeak. To record data, the caller needs to know the channel identity as well as the security key needed for writing. The caller then makes a REST call passing in these settings as well as the data to be recorded.

Once the data has been recorded by a channel, it can be displayed in charts and graphs also supplied by the Thingspeak website.



As we see, the channel is the core concept here. When we define a channel to Thingspeak, it is allocated a unique ID that we can subsequently use to refer to it. Also associated with the channel are a variety of properties. These include:

<b>Property</b>	<b>Description</b>
Name	The human readable name we wish to give to this channel. For example, "Temperature in Dallas".
Description	A human readable description of the data. For example, "The outside temperature in Texas at ZIP code 76182".
Field#	The fields enabled for the channel. There can be a total of 8 distinct fields.
Metadata	Meta data associated with the channel.
Tags	Comma separated tags used to search for data. For example, "temperature, Texas"
Latitude	The global latitude of where the data originated.
Longitude	The global longitude of where the data originated.
Elevation	The altitude above sea-level (in meters) of where the data originated.
Make Public	Can the data be read by anyone knowing the channel id. If private, the read key is needed.
URL	URL to where one can study further on this data.

Here is an example of creating the data for a new channel at the Thingspeak web site:

**ThingSpeak** Channels Apps Blog Support Account Sign Out

## New Channel

Name: Temp in Texas

Description: The outside temperature at 76182.

Field 1: temp

Field 2:

Field 3:

Field 4:

Field 5:

Field 6:

Field 7:

Field 8:

Metadata:

Tags: temp, Texas   
(Tags are comma separated)

Latitude: 32.868

Longitude: -97.188

Elevation: 184

Make Public?

URL:

Video ID:   YouTube  Vimeo

**Save Channel**

## Help

### ThingSpeak Channel

Channels store all the data that a ThingSpeak application collects. Each channel includes eight fields that can hold any type of data, plus three fields for location data and one for status data. Once you collect data in a channel, you can use ThingSpeak apps to analyze and visualize it.

### Channel Settings

- Channel Name:** Enter a unique name for the ThingSpeak channel.
- Description:** Enter a description of the ThingSpeak channel.
- Field#:** Check the box to enable the field, and enter a field name. Each ThingSpeak channel can have up to 8 fields.
- Metadata:** Enter information about channel data, including JSON, XML, or CSV data.
- Tags:** Enter keywords that identify the channel. Separate tags with commas.
- Latitude:** Specify the position of the sensor or thing that collects data in decimal degrees. For example, the latitude of the city of London is 51.5072.
- Longitude:** Specify the position of the sensor or thing that collects data in decimal degrees. For example, the longitude of the city of London is -0.1275.
- Elevation:** Specify the position of the sensor or thing that collects data in meters. For example, the elevation of the city of London is 35.052.
- Make Public:** If you want to make the channel publicly available, check this box.
- URL:** If you have a website that contains information about your ThingSpeak channel, specify the URL.
- Video ID:** If you have a YouTube or Vimeo video that displays your channel information, specify the full path of the video URL.

### Using the Channel

You can get data into a channel from a device, website, or another ThingSpeak channel. You can then visualize data and transform it using [ThingSpeak Apps](#).

See [Tutorial: ThingSpeak and MATLAB](#) for an example of measuring dew point from a weather station that acquires data from an Arduino device.

[Learn More](#)

Once created, we are presented with a simple control panel:

ThingSpeak

Channels Apps Blog Support Account Sign Out

## Temp in Texas

Channel ID: 73922  
Author: kolban  
Access: Private

The outside temperature at 76182.  
temp, texas

Private View Public View Channel Settings API Keys Data Import / Export

Add Visualizations Data Export MATLAB Analysis MATLAB Visualization More Apps

### Channel Stats

Created less than a minute ago  
Updated less than a minute ago  
0 Entries

**Field 1 Chart**

Temp in Texas

temp

Date

ThingSpeak.com

**Channel Location**

A map of the United States with a red marker pointing to the state of Texas. The map includes state names and major cities like Houston, Dallas, and Austin. Labels for the Gulf of California and the Gulf of Mexico are also visible.

With the channel defined at Thingspeak, the next thing to do is to submit some data. This is achieved through REST APIs. From a security perspective, we might now want anyone / anything other than our chosen devices to write records to the channel. As such, when one writes to a channel, a channel write key is needed. This is a string which can be found at the channel page under the heading "API Keys". Similarly, we don't want just anyone to be able to read the data so we also have the concept of a read key to allow reading back.

The outside temperature at 76182.  
temp, texas

Private View   Public View   Channel Settings   API Keys   Data Import / Export

## Write API Key

Key

[Generate New Write API Key](#)

## Read API Keys

Key

Note

[Save Note](#) [Delete API Key](#)

[Generate New Read API Key](#)

## Help

API keys enable you to write data to a channel or read data from a private channel. API keys are auto-generated when you create a new channel.

### API Keys Settings

- **Write API Key:** Use this key to write data to a channel. If you feel your key has been compromised, click [Generate New Write API Key](#).
- **Read API Keys:** Use this key to allow other people to view your private channel feeds and charts. Click [Generate New Read API Key](#) to generate an additional read key for the channel.
- **Note:** Use this field to enter information about channel read keys. For example, add notes to keep track of users with access to your channel.

**Create a Channel**  
POST <https://api.thingspeak.com/channels.json>  
api\_key=LXLKD7LMLZIWXHV  
name=My New Channel

**Update a Channel**  
PUT <https://api.thingspeak.com/channels/73922>  
api\_key=LXLKD7LMLZIWXHV  
name=Updated Channel

**Clear a Channel**  
DELETE <https://api.thingspeak.com/channels/73922/feeds.json>  
api\_key=LXLKD7LMLZIWXHV

**Delete a Channel**  
DELETE <https://api.thingspeak.com/channels/73922>  
api\_key=LXLKD7LMLZIWXHV

[Learn More](#)

Because Thingspeak is a free service and should not be abused, data writes are limited to no more than one every 15 seconds. If you need to record data for public consumption at a rate faster than 4 records a minute, look elsewhere. The practical implication of this is that your applications should not attempt to send new records faster than this throttle.

To write a record to Thingspeak, we can use an HTTP POST command targeted at <http://api.thingspeak.com/update>. Here is an example HTTP transmission:

```
POST /update HTTP/1.1
Host: api.thingspeak.com
Connection: close
X-THINGSPEAKAPIKEY: (Write API Key)
Content-Type: application/x-www-form-urlencoded
Content-Length: (number of characters in message)
```

```
field1=(Field 1 Data)&field2=(Field 2 Data)&field3=(Field 3 Data)&field4=(Field 4  
Data)&field5=(Field 5 Data)&field6=(Field 6 Data)&field7=(Field 7 Data)&field8=(Field  
8 Data)&lat=(Latitude in Decimal Degrees)&long=(Longitude in Decimal  
Degrees)&elevation=(Elevation in meters)&status=(140 Character Message)
```

We can also use the HTTP GET command and pass the parameters through an HTTP query string. For example:

```
http://api.thingspeak.com/update?key=<Write API Key>&field1=<value>&...
```

The only mandatory parameter is the `key` field. The possible optional parameters for both POST and GET are `field1`, `field2`, `field3`, `field4`, `field5`, `field6`, `field7`, `field8`, `lat`, `long`, `elevation` and `status`.

Each time a new record is submitted to a Thingspeak channel, an entry-id is generated for the record. This is a value starting at 1 that keeps incrementing by 1 for each new record. The return value from a REST request is the entry-id for the new record. If 0 is returned, that means that the record was not accepted. Note that this is different from the HTTP status return code which remains at 200 whether or not the record was accepted.

Here is a sample C program that writes data to Thingspeak:

```
#include <stddef.h>  
#include <unistd.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <curl.h>  
  
static void checkRC(CURLcode rc, char *message) {  
    if (rc != CURLE_OK) {  
        printf("Error: %d %s\n", rc, message);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("About to start\n");  
    char urlText[100];  
  
    CURLcode rc;  
    CURL *curl = curl_easy_init();  
    if (curl == NULL) {  
        printf("Failed to setup curl\n");  
        exit(-1);  
    }  
    while(1) {  
        int value = rand() % 20;  
        printf("Setting value: %d\n", value);  
        sprintf(urlText, "https://api.thingspeak.com/update?  
api_key=D7YJKGFR1WV5D0UD&field1=%d", value);  
        curl_easy_setopt(curl, CURLOPT_URL, urlText);  
        curl_easy_setopt(curl, CURLOPT_HTTPGET, 1L);
```

```

    rc = curl_easy_perform(curl);
    checkRC(rc, "perform");
    sleep(20);
}
curl_easy_cleanup(curl);
printf("Ending\n");
return 0;
}

```

The value written is just one field with a random integer value but I hope you can see that any value you read from a sensor of any description will also work just fine.

See also:

- REST requests
- cURL API programming

## Sockets API programming

Assuming your Pi has access to a TCP/IP network (your local LAN or the Internet) then you can write C language applications to send and receive data. The classic Unix level API for TCP/IP programming is called "sockets". At a high level, one makes a call to `socket()` which returns a handle to an endpoint of communication. From here, the program will decide whether it is going to be a socket client (a program that calls outbound to initiate work) or a server (a program that passively listens for incoming connections from other clients). To be a client, the next API that is usually invoked is `connect()`. This provides the IP address and port number to which the client wishes to connect. Once that succeeds, the client can then send and receive arbitrary amounts of data through the socket using `send()` and `recv()` to send and receive data. When done, it can call `close()` to terminate the connection.

Should the application wish to be a server, it will associate itself with a port on the local machine using the `bind()` call and then execute a `listen()` call to begin listening for incoming client requests. Finally, it will execute an `accept()` call which causes the server to block and wait for new incoming client requests. When a client does connect, the `accept()` call will unblock and return a new socket that represents the connection to the client.

Return codes from the sockets APIs are usually integers. If the value -1 is returned, this indicates an error and the global `errno` is further set to indicate the nature of the error.

See also:

- [TCP/IP Sockets in C: Practical guide for programmers](#)

## Finding an IP address from a hostname

Imagine we are given the hostname of a server and we wish to connect with it. When we look at the sockets API, we will find that they expect 32bit IP addresses. What we need is a mechanism to convert a hostname into its corresponding IP address. Fortunately there is a function that will do just that for us. It is called "gethostbyname". It and its related data structures are defined in the header called "netdb.h". To use, we invoke `gethostbyname()` passing in a string representation of the hostname or a string representation of the IPv4 dotted decimal address. What we get back is a pointer to a `struct`

`hostent`. This structure has been populated by the `gethostbyname()` call and we can examine its content to obtain the results. Specifically, to find the IP address we can use:

```
hostent.h_addr
```

The return type of this is a "char \*" which will point to the IP address. For example:

```
struct hostent *myHost = gethostbyname("www.kolban.com");
char *ipAddress = myHost->h_addr;
```

If we can't resolve the hostname or some other error has occurred, the return from `gethostbyname()` is NULL.

The addresses returned are already in network byte order so no conversion from host byte order is needed nor should be performed.

Having just explained `gethostbyname()` we are now going to say that it has been mostly superseded by the `getaddrinfo()` call.

See also:

- [man\(3\) – gethostbyname](#)

## Creating a socket client

Let us now look at some practical examples. First we will set up the creation of a socket client connection. This will allow us to connect from a C program using sockets to an already existing Socket server running else where. In order to form a connection we will assume that we know the hostname or IP address of the target server as well as the port number on which it is listening. The first thing we will do is create a socket:

```
int rc; // Return code for checking
int s = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
struct sockaddr_in serverAddr;
serverAddr.sin_family = AF_INET;
serverAddr.sin_addr.s_addr = ...;
serverAddr.sin_port = htons(port);
rc = connect(s, (struct sockaddr *)&serverAddr, sizeof(serverAddr));
// We are now connected to the target
```

## Creating a socket server

The socket server begins with the creation of a socket in exactly the same manner as a client. However, once the socket is created we bind it to local IP port and then ask it to start listening for incoming connections requests from clients. Since the server doesn't know when a client will connect, it executes an `accept()` call which normally blocks waiting for a client connection to arrive. When a client does connect, `accept()` returns with a new socket connection that is ready to be used and constitutes the connection to the client.

```
int s = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
struct sockaddr_in serverAddress;
```

```

serverAddress.sin_family = AF_INET;
serverAddress.sin_addr.s_addr = htonl(INADDR_ANY);
serverAddress.sin_port = htons(port);

rc = bind(s, (struct sockaddr *)&serverAddress, sizeof(serverAddress));
listen(s, 5);
struct sockaddr_in clientAddress;
int length = sizeof(clientAddress);
int clientSocket = accept(s, (struct sockaddr *)&clientAddress, &length);

```

If we don't care about the address information of the client then the address structure and length parameters can be supplied as NULL.

See also:

- [man\(2\) – accept](#)
- [man\(2\) – bind](#)
- [man\(2\) – listen](#)

## Sending data through a socket

The API to send data through a socket is called `send()`. It takes as parameters the socket descriptor to use, a pointer to the raw memory to send, the length of the data to send and some optional flags.

```
rc = send(s, buffer, size, 0);
```

See also:

- [man\(2\) – send](#)

## Receiving data through a socket

The API to receive data through a socket is called `recv()`. When called, it expects the socket descriptor through which to receive the new data. It also takes a buffer point into which the data received will be placed. The size of the buffer is also supplied. Lastly there will be some optional flags. The return from this function is the size of the data actually received which could be less than the buffer size supplied which is the maximum amount of data that can be received in one request. The call can block if there is no data available. It can block the caller until such time as there is data. The blocking can be over-ridden to provide alternate options. An error is indicated by the return value of -1.

See also:

- [man\(2\) – recv](#)

## Closing a socket

When either the server side or the client side of the socket connection wishes to indicate that it has finished, it should close the socket with a call to `close()`.

See also:

- [man\(2\) – close](#)

## UDP programming

Where TCP socket programming is stream based and data pushed in one side appears on the partner side, UDP socket programming works on the concept of datagrams. A datagram is a packet of data that is transmitted from one end to another as a unit. The maximum size of a datagram is 64K. Unlike TCP, there is no assured delivery. What this means is that if a sender sends a datagram it is possible for the datagram to never reach the intended recipient and neither end will know it was lost. UDP is a fire and forget protocol. With these apparent limitations, what then is the value of UDP? The first is that because it is datagram oriented, it has a smaller overhead in terms of transmission. There is no receipt of delivery and hence no extra chatter. Since a datagram is delivered as a unit, it also means that there is a logical record boundary which is not present in TCP. However the most compelling reason for UDP is that there needs to be no pre-acquired connection between a sender and a receiver. This means that a sender can send a datagram to a recipient without first having to form a connection. A significant value for this is that a sender can broadcast a datagram to a set of endpoints and **all** the recipients will receive a copy. Contrast this with TCP where if I want to send data to a set of machines, I have to send a distinct copy of that data to each partner.

We use UDP by creating a socket but flag it as a datagram:

```
int s = socket(AF_INET, SOCK_DGRAM, 0);
```

Next, on the server, we bind the socket to a local port to act as the endpoint using the `bind()` API.

Finally, on the server, we can block waiting for an incoming datagram using the `recvfrom()` API.

On the client, we perform a `socket()` call to create a socket and then execute a `sendto()` API call to send a datagram to the target server.

See also:

- [man\(2\) – sendto](#)
- [man\(2\) – recvfrom](#)

## JavaScript socket.io

A package for JavaScript called "socket.io" provides a wealth of network programming capabilities. To install, we can use npm:

```
$ npm install socket.io
```

## Sending an email

There may be times where you wish to send an email from your Pi. Hopefully it isn't because a sensor had detected that your house is on fire (although you may actually want that). Maybe you want to receive an email when Johnny comes home from school. In this case you could build a motion sensor outside the door that, when triggered, takes a photo of the scene and sends it to your camera.

The way mail works on a Linux system is that a shell program called mail communicates with a local program called sendmail. The mail is handed off to sendmail and then sendmail relays it onwards out onto the Internet. The sendmail application also handles incoming emails and it can become quite complex. If all we want to do is send emails from Raspbian, there is an easier solution through the package called ssmtp.

First we want to install ssmtp using:

```
$ sudo apt-get install ssmtp
```

Here is a sample configuration file for a GoDaddy hosted email account:

```
#  
# Config file for sSMTP sendmail  
#  
# The person who gets all mail for userids < 1000  
# Make this empty to disable rewriting.  
root=pi@kolban.com  
  
# The place where the mail goes. The actual machine name is required no  
# MX records are consulted. Commonly mailhosts are named mail.domain.com  
mailhub=smtpout.secureserver.net  
  
AuthUser=pi@kolban.com  
AuthPass=*****  
UseTLS=NO  
UseSTARTTLS=NO  
# Where will the mail seem to come from?  
rewriteDomain=kolban.com  
  
# The full hostname  
hostname=smtpout.secureserver.net  
  
# Are users allowed to set their own From: address?  
# YES - Allow the user to specify their own From: address  
# NO - Use the system generated From: address  
FromLineOverride=YES
```

After installation we are ready to send emails, however the mail sending tools are also not yet installed. To install those, we need to install the package called mailutils:

```
$ sudo apt-get install mailutils
```

Once installed we have the classic mail command. To send an email from the command line we can use:

```
$ echo "Here is your mail" | mail -s "Your Mail" -A /etc/passwd pi@kolban.com
```

With this in place, we can now send an email from almost any app. For example, when a sensor attached to the Pi detects something un-toward, we can send an email to alert someone.

One of the things to consider about emails is that they are "people interrupters". When an email arrives we usually want to attend to it. Receiving irrelevant emails or emails that don't tell us anything we didn't already know can be a strong negative. As such, be very cautious when writing applications that send emails. At a minimum, think through the frequency with which emails are sent. If you have a sensor

which, when it triggers can trigger again repeatedly, design your solution so that only one email is sent and not hundreds of emails that will merely clutter (and annoy) the recipient.

## MQTT

The MQ Telemetry Transport (MQTT) is a protocol for publish and subscribe style messaging. It was originally invented by IBM as part of the MQSeries family of products but since has become an industry standard governed by the Oassis standards group. The latest specification version is 3.1.1.

Being Pub/Sub, this means that there is a broker (an MQTT Broker) to which subscribers can register their subscriptions and publishers can submit their publications. Publications and subscriptions agree on the topics to be used to link the messages together. A client can be a publisher, a subscriber or both.

The value of MQTT is that it can be used to deliver data from an application running on one machine to an application running on another. Immediately we seem to see an overlap between MQTT and REST calls but there are some major differences. In a REST environment, when you form a connection from a client to a server, the server must be available in order for the client to deliver the data. With MQTT that is not necessarily the case. The client can publish a message which can then be held by the broker until such time as the receiving application comes on-line to retrieve it. This is a store and forward mechanism.

Every message must have a topic associated with it.

Topics are broken into topic levels. There are wild-cards.

- + - Single topic level wild-card

eg. a/+ /c

would subscribe to a/b/c and a/x/c.

- # - Multi topic level widlcard

eg. a/#

would subscribe to a/<anything>.

MQTT is built on top of TCP/IP. Clients connect to the broker (not to each other) over a TCP connection.

There is a quality of service requested by a client. This is encoded in the QoS field:

- QoS=0 – Send at most once. This can lose messages. At most once means perhaps never.
- QoS=1 – Send at least once. This means that the message will be delivered. Saying this another way, a message will not be discarded or lost. However, duplicates can arrive ... i.e. the message can be delivered twice or more.
- QoS=2 – Send exactly once. This means that the message will not be lost and will be delivered once and once only.

MQTT also has the capability to buffer messages for subsequent delivery. For example, if a client subscriber is not currently connected, a message can be queued or stored for delivery to the client when it eventually re-connects. We call a client that is not connected an off-line client. For a subscription, we have the choice to deliver all the queued messages for a client or just the last message. To understand the difference, we can imagine a published message that says "I will be attending the party" ... we want all such messages sent to the client because they are all of interest. However if we think of a published message of "Todays forecast is sunny and warm" then there may be no need for old messages and only the current weather forecast is of interest to us. The last message published on a topic is called the "retained message". When a client subscribes, it can ask to receive the last message immediately ... so even if a subscription takes place after a previous publication, it can still receive data immediately.

Clients make their status known to the broker so the broker can tell if a client is connected. This is achieved via a keep-alive/heartbeat.

If a client connection is lost because of a network disconnection, the broker can detect that occurrence. This is where we get morbid. We define this as the client having "died". In the real world, when someone dies, there may be a last "will and testament" which are the desired of what the person wanted to happen when they die. MQTT has a similar concept. A client can register a message to be published in the event of the clients death. This is remembered by the broker and in the event of the client dieing, the broker will perform the role of the attorney and publish the last registered "will and testament" message on behalf of the deceased client.

See also:

MQTT Hive

Mosquitto

- [MQTT.org](#)
- [Oasis MQTT spec – 3.1.1](#)
- [Mosquitto.org](#)
- YouTube: [Internet of Things – Why You Need MQTT](#)

## MQTT Protocol

Protocol

Client sends Connect

Broker replies with Connack

Connect MQTT packet

- clientId
- cleanSession
- username
- password
- lastWillTopic

- lastWillQos
- lastWillMessage
- keepAlive

Connack

- sessionPresent
- returnCode

Publish

- packetId
- topicName
- qos
- retainFlag
- payload
- dupFlag

Puback

??

Pubrec

- packetId

Pubrel

- packetId

Pubcomp

- packetId

Subscribe

- packetId
- qos
- topic1
- qos2
- topic2
- ...

Suback

Disconnect

## Mosquitto MQTT

One of the most prevalent implementations of MQTT is called `Mosquitto` and is available as an open source implementation. On a Linux system we would install with:

```
$ sudo apt-get install mosquitto
```

Unfortunately, the version of `mosquitto` supplied by the repository is 1.3.4 which is quite old and pre-dates many important functions such as websocket support. To resolve this, you may have to download the latest version and build yourself.

Where `systemd` is installed, `mosquitto` is controlled by it. To see if it is running execute

```
$ systemctl status mosquitto
● mosquitto.service - LSB: mosquitto MQTT v3.1 message broker
  Loaded: loaded (/etc/init.d/mosquitto)
  Active: active (running) since Thu 2016-01-21 21:32:26 CST; 6min ago
    Docs: man:systemd-sysv-generator(8)
   CGroup: /system.slice/mosquitto.service
           └─12871 /usr/sbin/mosquitto -c /etc/mosquitto/mosquitto.conf

Jan 21 21:32:26 kolban-VirtualBox systemd[1]: Starting LSB: mosquitto MQTT v3.1
message broker...
Jan 21 21:32:26 kolban-VirtualBox mosquitto[12862]: * Starting network daemon:
mosquitto
Jan 21 21:32:26 kolban-VirtualBox mosquitto[12862]: ...done.
Jan 21 21:32:26 kolban-VirtualBox systemd[1]: Started LSB: mosquitto MQTT v3.1 message
broker.
```

The default configuration file for `mosquitto` is `/etc/mosquitto/mosquitto.conf`. Messages from `mosquitto` are logged to `/var/log/mosquitto/mosquitto.log`.

Two sample applications are provided that perform subscriptions and publications. These are `mosquitto_sub` and `mosquitto_pub`. These are distributed as part of the `mosquitto-clients` package.

As a simple test, open two terminal sessions. In one run:

```
$ mosquitto_sub -t greeting
```

In the other run:

```
$ mosquitto_pub -t greeting -m "Hello World"
```

You will see the published message appear in the subscriber window.

Let us now dig a little further into a `mosquitto` configuration file. Within there, we can specify multiple "listener" entries. These can bind to a port or a port and interface. Within a listener section, we can configure the protocol that the listener should use. By default, it will be plain TCP for normal client connections but we can specify that the protocol should be "websockets". For example, adding the following into the `mosquitto.conf` file:

```
listener 8081
protocol websockets
```

will cause `mosquitto` to listen for websocket clients on port 8081. If you want to listen on **both** standard mqtt protocol and websocket, then you will need multiple listener definitions.

```
listener 1883
protocol mqtt

listener 8081
protocol websockets
```

See also:

- [Mosquitto.org](#)
- [man\(1\) – mosquitto\\_sub](#)
- [man\(1\) – mosquitto\\_pub](#)
- [man\(5\) – mosquitto.conf](#)
- [MQTT Community Wiki](#)

### [Building mosquitto from source](#)

Because the current distribution of `mosquitto` is quite old, we may want to download, build and install `mosquitto` from source.

To obtain the source, visit:

<http://mosquitto.org/download/>

Download the GZip file and extract. At the time of writing this was `mosquitto-1.4.7.tar.gz`. In the resulting directory, we will find a `Makefile` which can be run with `make`. I recommend editing `config.mk` and changing:

```
WITH_UUID:=yes
to
WITH_UUID:=no
and
WITH_SRV:=yes
to
WITH_SRV:=no
and
WITH_WEBSOCKETS:=no
to
WITH_WEBSOCKETS:=yes
```

If we are enabling websockets, we must also install the websockets libraries. See the following for instructions:

<https://libwebsockets.org/index.html>

In summary, we clone a git project, within the project directory create a directory called "build", in the build directory execute "cmake .." and when that completes, execute a "make" against the Makefile that was just built for us.

This install needs cmake which can be installed via:

```
$ sudo apt-get install cmake
```

## Writing MQTT clients

Having an MQTT environment available is nice but without clients doesn't add much value. Your clients will be the applications that will register subscriptions and publish messages. The subscriber will name the topic on which messages published will be received.

While the MQTT protocol may be standardized, the client APIs appear to not be. As such there are multiple APIs for any given language.

### Eclipse paho

See also:

- [Paho home page](#)
- [Practical MQTT with Paho](#)

### C - Mosquitto client library

The library called libmosquitto is available for linking with C applications. You will need to install the package called libmosquitto-dev.

- mosquitto\_lib\_version – Determine the version of the library in use
- mosquitto\_lib\_init – Initialize the library
- mosquitto\_lib\_cleanup – Conclude the use of the library
- mosquitto\_new – Create a client
- mosquitto\_destroy – Destroy a client
- mosquitto\_reinitialize – Destroy a client and then create a new one
- mosquitto\_username\_pw\_set – Set the userid and password for authentication
- mosquitto\_will\_set – Set the last will of the client for estate planning purposes
- mosquitto\_will\_clear – Revoke the last will
- mosquitto\_connect – Connect a client to a broker

- `mosquitto_connect_bind` – Same as `mosquitto_connect` but constrains interface to bind with
- `mosquitto_connect_async` – Asynchronous connection to the broker
- `mosquitto_reconnect` – Reconnect to a broker after a lost connection
- `mosquitto_reconnect_async` – Same as `mosquitto_reconnect` but asynchronous
- `mosquitto_disconnect` – Disconnect a client from a broker
- `mosquitto_publish` – Publish a message on a given topic
- `mosquitto_subscribe` – Subscribe to messages on a topic
- `mosquitto_unsubscribe` – Unsubscribe to messages on a topic
- `mosquitto_loop` – Perform processing for the `mosquitto` client. It is here that incoming message from the broker are received or previously unsent publications transmitted.
- `mosquitto_loop_read`
- `mosquitto_loop_write`
- `mosquitto_loop_misc`
- `mosquitto_loop_forever` – Same as `mosquitto_loop` but does not return and keeps processing until the client is disconnected
- `mosquitto_socket` – Retrieve the low level TCP/IP socket that the `mosquitto` client is using
- `mosquitto_want_write` – Return true if there is pending data to be sent to the broker
- `mosquitto_loop_start` – Start a thread to process a `mosquitto_loop` in the background
- `mosquitto_loop_end` – Stop a previously started loop that was created using `mosquitto_loop_start`

Here is an example publishing client:

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

int main(int argc, char *argv[]) {
    char *host="pc9100";
    int port = 1883;
    char *message = "hello world!";
    char *topic = "greeting";

    mosquitto_lib_init();
    struct mosquitto *mosq = mosquitto_new(
        NULL, // Generate an id
        true, // Create a clean session
```

```

        NULL); // No callback param
    int rc = mosquitto_connect(
        mosq,      // Client handle
        host,      // Host of the broker
        port,      // Port of the broker
        false);   // No keepalive
    if (rc != MOSQ_ERR_SUCCESS) {
        printf("Error with connect: %d\n", rc);
        return(rc);
    }
    mosquitto_publish(
        mosq,          // Client handle
        NULL,          // Message id
        topic,         // Topic
        strlen(message)+1, // Length of message
        (const void *)message, // Message to be sent
        0, // QoS = 0
        false); // Not retained
    mosquitto_destroy(mosq);
    mosquitto_lib_cleanup();
}

```

See also:

- [man\(3\) – libmosquitto](#)
- [mosquitto.h](#)

## Node.js JavaScript - MQTT

There is Node.js package called `mqtt` that provides MQTT functions for JavaScript applications. To install this package we should run:

```
$ npm install mqtt
```

Here is a sample JavaScript application that acts as a publisher:

```

var mqtt = require("mqtt");
var client = mqtt.connect("mqtt://pc9100");
client.on('connect', function() {
    console.log("Connected ... now publishing");
    client.publish("greeting", "Hello from JavaScript");
    client.end();
});

```

And here is an application fragment that acts as a subscriber:

```

var mqtt = require("mqtt");
var client = mqtt.connect("mqtt://pc9100");
client.on('connect', function() {
    client.subscribe("greeting");
});
client.on('message', function(topic, message) {
    console.log("Received message for topic=" + topic + ", message=" + message);
});

```

Error handling can be accommodated by registering a client handler for an error event:

```

client.on("error", function(error) {
  console.log("We detected an error: " + error);
});

```

The full details of the API can be found on the NPM home page for the MQTT package. At a high level, the functions are:

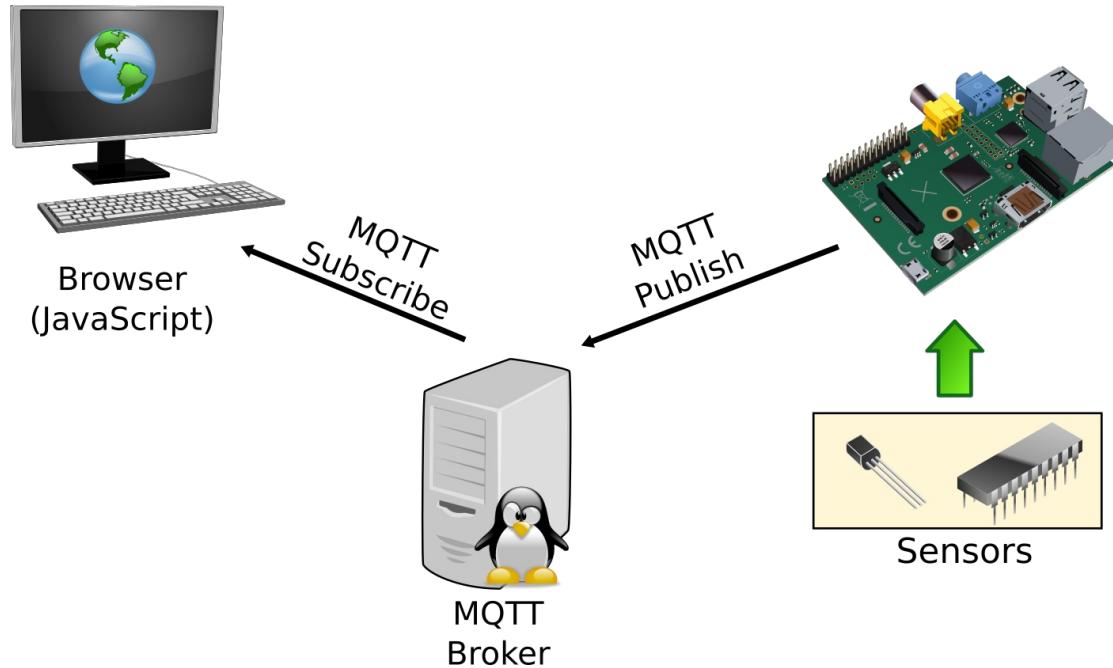
- `connect` – Connect to a broker.
- `Client.publish` – Publish a message.
- `Client.subscribe` – Subscribe to a topic.
- `Client.unsubscribe` – Unsubscribe from a topic.
- `Client.end` – Close the client connection to the broker.
- `Client.handleMessage` – A mechanism for handling messages.

See also:

- [npm – mqtt](#)

### Browser JavaScript - MQTT

The Eclipse Paho project includes a JavaScript client API that is suitable for running within the context of a browser. This client API uses the HTML5 WebSocket API to communicate with an MQTT broker.



The MQTT broker chosen must support MQTT protocol through WebSocket. Popular brokers such as Mosquitto are able to perform that role. From a security standpoint, browsers are only allowed to make WebSocket requests back to the HTTP server from which the page running in the browser was originally

loaded. As such, the browser can only make WebSocket requests to the MQTT broker if the MQTT broker served up the page. We can alleviate that issue by introducing full function Web Servers and proxies into the story but for simple purposes, brokers such as Mosquitto can serve up static web pages directly. This means that we can point our browser to Mosquitto which will then look for an HTML file on the local file system to Mosquitto and send that back to the browser. The browser will now see the MQTT broker as the server of the web page and when the JavaScript in that web page asks for an MQTT subscription, it will succeed. Within the Mosquitto configuration file, each listener definition that wishes to also provide HTTP files must supply an "http\_dir" option which names the directory that will be searched for HTML file requests from the browser.

We are not limited running an MQTT broker on a separate server machine, we can run the MQTT broker directly on the Pi should we desire.

Now let us look and see what is needed to run an MQTT client in the browser.

First we need to download the Paho JavaScript client. If we visit the Paho downloads page:

<https://projects.eclipse.org/projects/technology.paho/downloads>

we will find an entry for "JavaScript client 1.0.1". Follow through that link and download the ZIP file. At the time of writing the result will be:

paho.javascript-1.0.1.zip

we can then unzip the file using unzip. Within we will find a file called mqttws31.js. This is the JavaScript source of the client and needs to be included in the HTML file that will be using MQTT. Here is an example HTML with embedded JavaScript illustrating us connecting to a broker and registering a subscription. When a message is published to the matching topic, it is logged to the browser console.

```
<html>
  <head>
    <script src="mqttws31.js"></script>
    <script>
      var client = new Paho.MQTT.Client("192.168.1.101", 8081, "/", "CLIENT1");
      client.onMessageArrived = function(message) {
        console.log("Message arrived ...: " + message.payloadString);
      };
      client.onMessageDelivered = function(message) {
        console.log("Message delivered ...");
      };
      client.connect({
        onSuccess: function() {
          console.log("On success ... we are now client connected to the broker!");
          client.subscribe("greeting", {
            onSuccess: function() {
              console.log("Subscription success ...");
            },
            onFailure: function(context, errorCode) {
              console.log("Failed to make a subscription ... code=" + errorCode);
            }
          });
        },
        onFailure: function(context, errorCode, errorMessage) {
          console.log("On failure ... code=" + errorCode + ", message=" + errorMessage);
        }
      });
    </script>
  </head>
  <body>
  </body>
</html>
```

```

        }
    }) ;
    </script>
</head>
<body>
</body>
</html>

```

Within the `mosquitto.conf` file, we will have an entry that looks like:

```

listener 1883
protocol mqtt

listener 8081
protocol websockets
http_dir /mnt/pc/projects/robot

```

This defines that we will listen on port 1883 for standard MQTT protocol while we will also listen on port 8081 for websocket HTTP requests and if a request to load a page arrives, we will serve it from a given directory.

The API to publish a message is called "send" and takes as a parameter an MQTT message object.

See also:

- [Eclipse Paho JavaScript client](#)
- [Paho API JavaScript documentation](#)
- [The Mosquitto MQTT broker gets Websockets support](#)

## Interacting with Mobile devices

I define a mobile device as your cell phone or tablet running Android, IOS or Windows mobile. Let us now look and see how these can be leveraged in conjunction with a Pi?

### Blynk

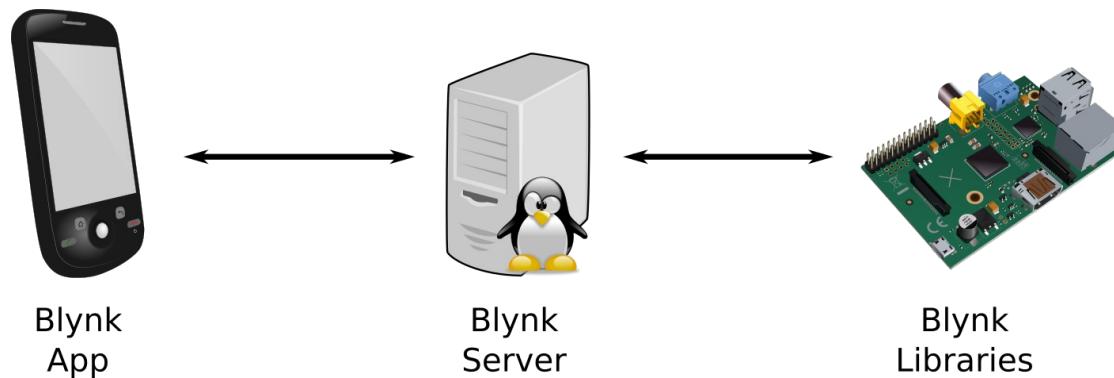
Blynk is a kick-starter project that provides an environment for building mobile apps that interact with the Pi, Arduino or ESP8266 based system. Within the app, we can dynamically build the user interface (UI) that we wish to use to interact with our Pi. The UI is composed from a set of pre-supplied building blocks called widgets. The catalog of widgets currently available include:

- Button – Pressing the button changes the value of a pin on the Pi. The button can be either press/release or switch based.
- Slider – Set a value between a defined min and max.
- Joystick – Either a 1 dimensional or 2 dimensional joystick.

From an architectural perspective, think of Blynk as being composed of three primary components. The first is the "Blynk App". This is the application you install in your mobile device. It is also where you design your screens and also run those screens to interact with your Pi.

The second component is the "Blynk Server". This is a software server that is either hosted by Blynk on the Internet or you run your own private version. Your Pi connects to the Blynk Server and also instances of the Blynk App connect to the Blynk Server. Since both those components route through the Blynk Server, it acts as the gateway connection for both devices.

Finally there is the "Blynk Libraries". These are the code libraries that you use and run on the Raspberry Pi itself. These libraries are responsible for connecting the Pi to the Blynk Server as well as receiving data and commands from the Blynk Server.



Blynk supplies a Node.js library called "blynk-library". In order to use this we need to require it ... for example:

```
var BlynkLib = require("blynk-library");
```

From there, we can create a connection to our Blynk APP with:

```
var blynkApp = new BlynkLib.Blynk(<APP Id>);
```

Now we can create variables that represent the "virtual pins" that the App can write to. For example:

```
var v1 = new blynkApp.VirtualPin(1);
```

When the widget on in the App writes to the virtual pin, this will cause a "write" event to manifest within the JavaScript app running on the Pi which can be caught with:

```
v1.on("write", function(data) {  
    // Do something here ...  
});
```

If a widget on the App explicitly wishes to read from a virtual pin, this will cause a "read" event to occur on the JavaScript app which can be caught with:

```
v1.on("read", function() {  
    // Explicit read request ...  
});
```

Should we wish to push data from the Pi to the App, we can call the `write()` method of the virtual pin:

```
v1.write(1);
```

See also:

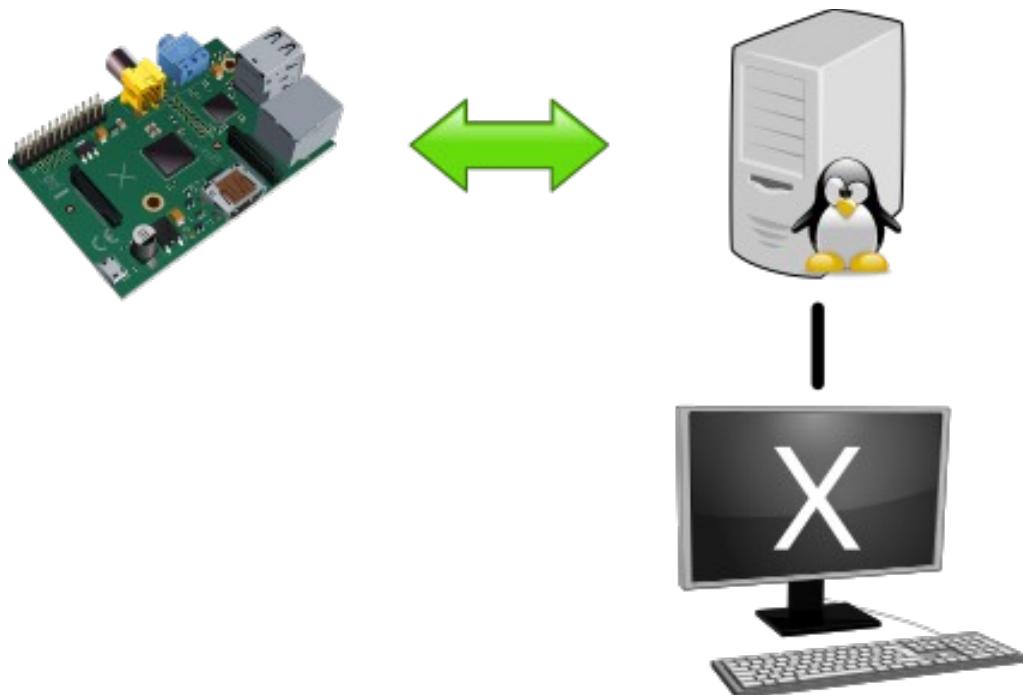
- [Blynk home page](#)
- Github: [Blynkk](#)
- npm: [blynk-library](#)

- Instructables: [Blynk + JavaScript in 20 minutes](#)
- Instructables: [Raspberry Pi + Node.js + Blynk App + DHT11/DHT22/AM2302](#)

## X-Windows

Linux has a graphical user interface called X-Windows. When you start Raspbian and connect your Pi to a TV or other HDMI display device, it is X-Windows that provides the graphical environment into which you can open windows and run other graphics oriented applications.

Unlike other graphics systems, X-Windows was designed to operate in a client/server mode. Since Unix is a multi-user environment, it isn't reasonable to think that all users could sit around the same graphics terminal so a mechanism to run one's applications on Unix while having the graphical results displayed on a local terminal was needed. The X-Windows environment provides a client/server architecture. In that model, the client is the application that wishes to display graphics and the server is the software or hardware that the client draws upon. It is the server that an end user sits in front off.



It is likely you have a PC as your normal workstation environment that includes a monitor, keyboard and mouse. I like to use this as my environment for working with the Pi. In order to do such, I run an X-Windows server on my windows PC, open an SSH connection to my Pi, point the X-Windows output back to my PC and then start the Pi X-Windows desktop. The result of this is a window on my PC which is the desktop of the Pi. This allows me to sit in one seat and switch between working on the Pi and working on the PC. Although the Pi is fantastic, it will be unlikely to be anywhere near as powerful as your PC which is likely loaded with your favorite browser and other goodies.

On my Windows PC, I use the free version of xming (<http://sourceforge.net/projects/xming/>) to run a local X-Server.

To start the Raspberry desktop on the HDMI output, use `startlxde`.

When the device running an application wishes to display its output on an X-Windows server, the application has to know which X-Windows server to use. After all, an X-Windows server is merely an

application sitting at the end of TCP/IP connection. The way to tell the application where to send the graphics request is through an environment variable called `DISPLAY`. The format of the variable is `<HOST or IP>:<Screen>`. Since an X-Windows server can display multiple screens, we specify which screen to use. 99 times out of 100 we want to use 0.

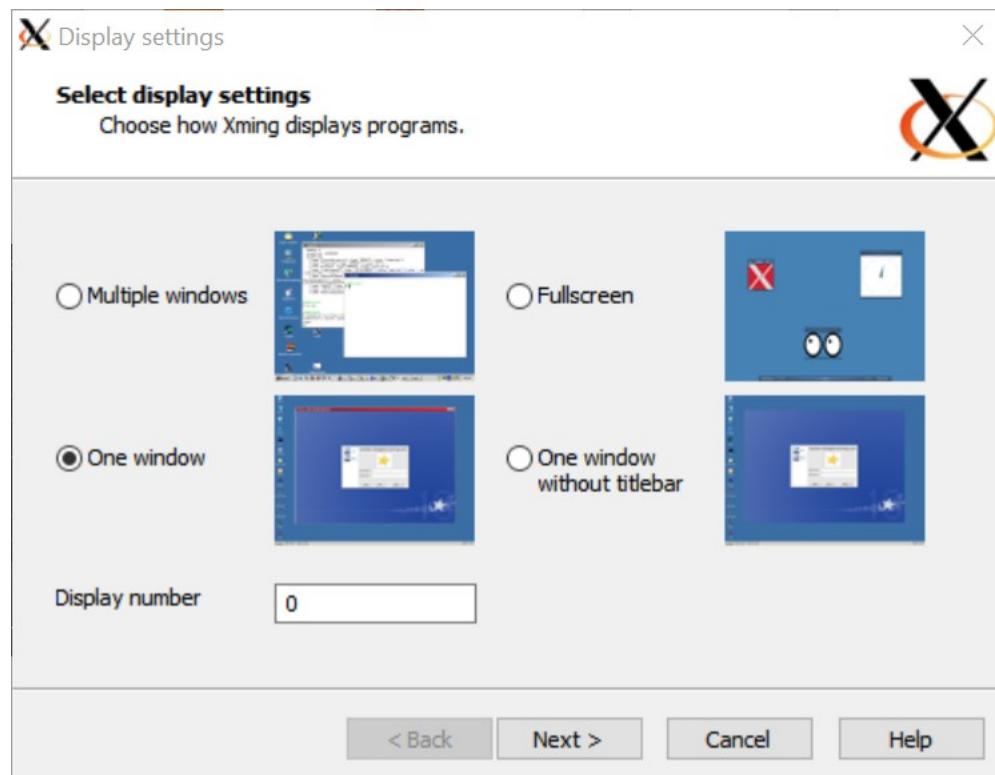
Here is an example of use:

```
$ export DISPLAY=192.168.1.2:0  
$ lxterminal
```

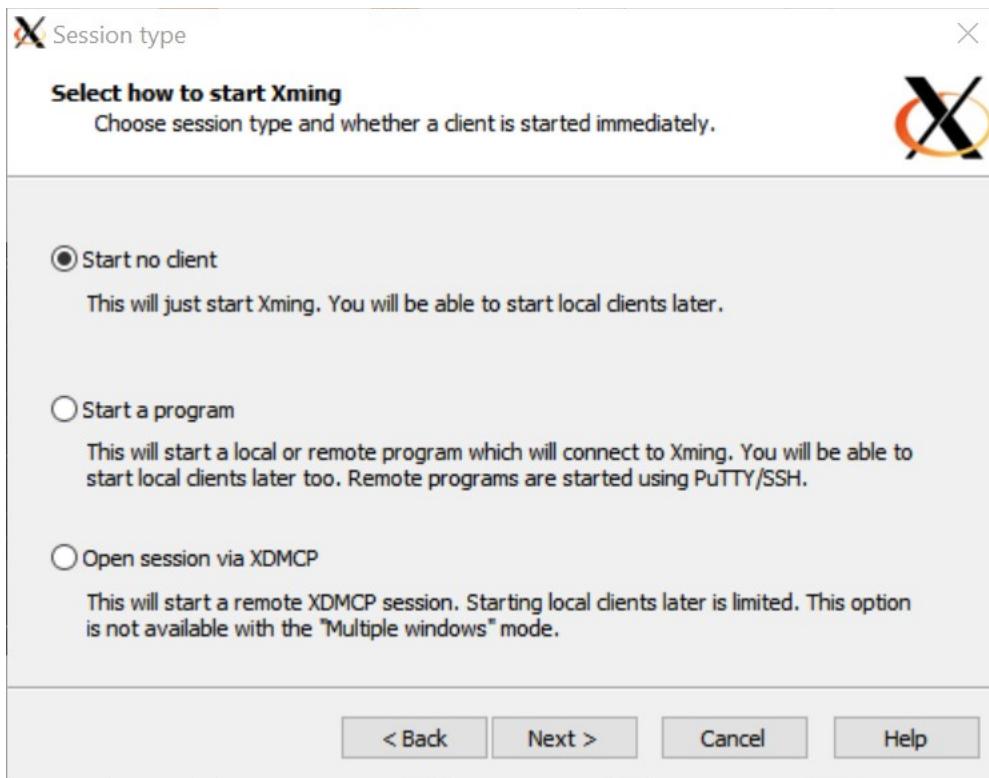
X-Windows provides a level of security configuration to prevent any arbitrary computer from accessing your screen (note that this means reading your screen as well as writing to it). On your X-Windows server environment, you can control this by running the "`xhost`" command. I typically run "`xhost +`" which allows **any** machine on my local network to access the X-Windows server. This should normally not pose an undue security risk as my X-Windows servers are on my local LAN inside my house and there is no inbound route to my X-Windows servers through my router / firewall from the Internet.

## Using xming on a Windows PC

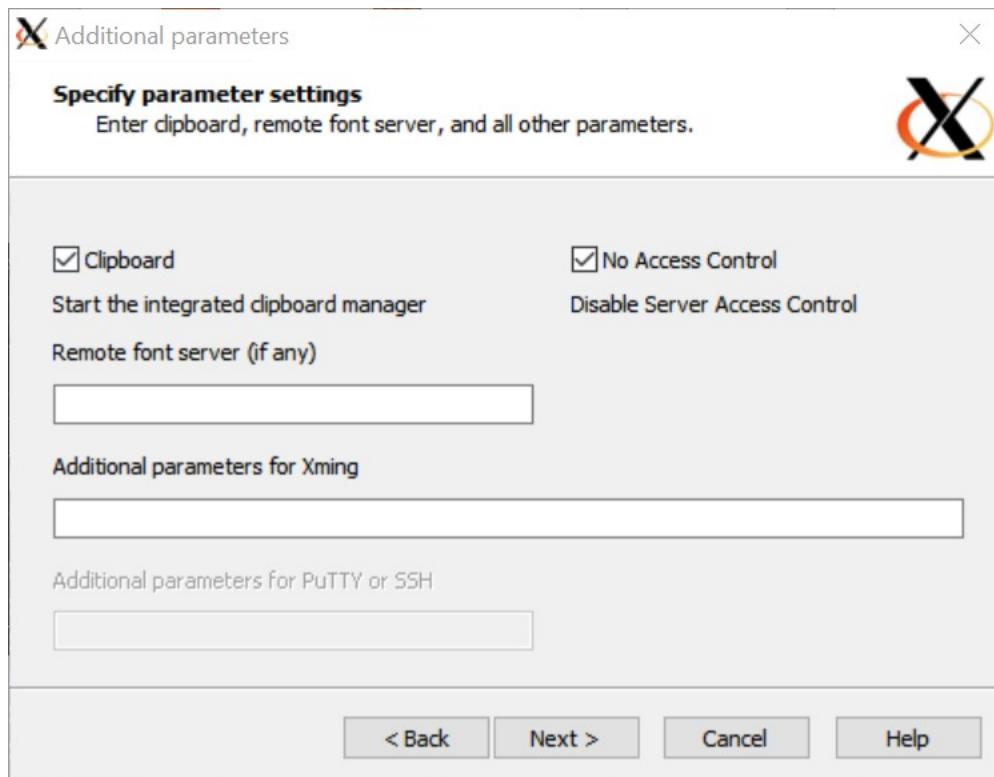
On my Windows 10 PC, I run the free application called `xming`. This is an X-Windows server that performs all the tasks I might need. To start it I run `XLaunch` which then presents me with a series of options. The following image shows the first screen:



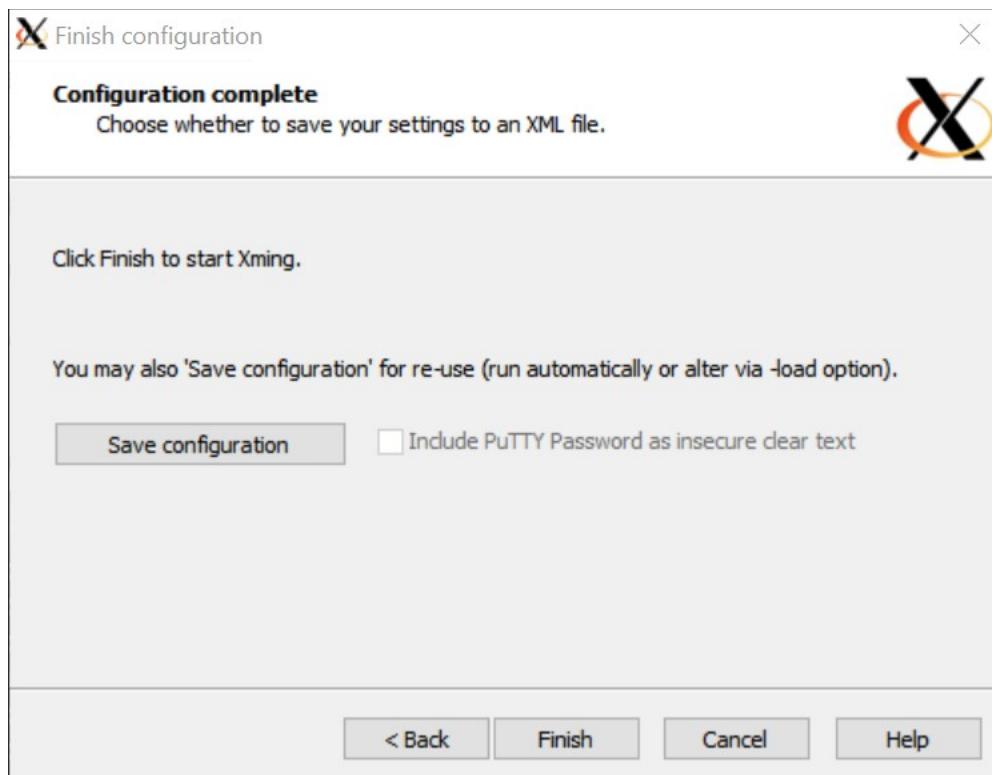
Since X-Windows is itself a windowing environment and we are already running on a windowing environment (Microsoft windows itself), we have the option of how to map the two together. The choice here is a matter of taste. I prefer the one window flavor.



The next screen asks us whether or not we want to start a remote application when `xming` starts. I usually select no.



The next screen is about server settings. I want clipboard copying enabled and I also disable access control.



The final screen allows us to start the server and optionally save the configuration.

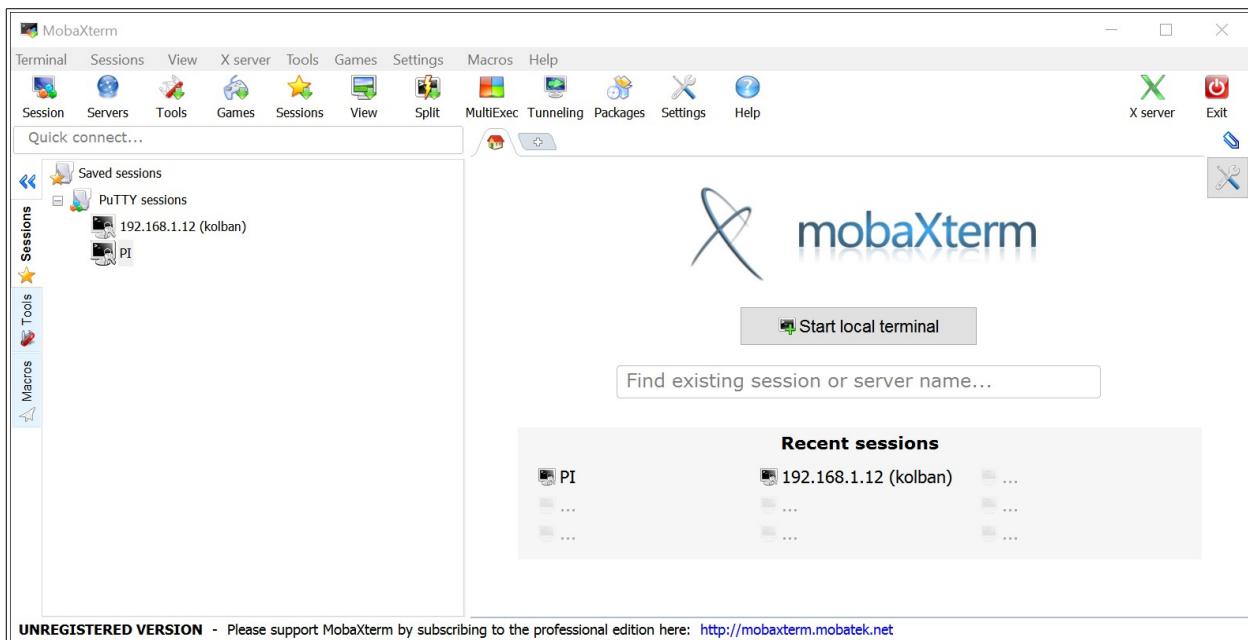
Once done you will see the X-Windows server background which is a dull checkerboard like appearance:



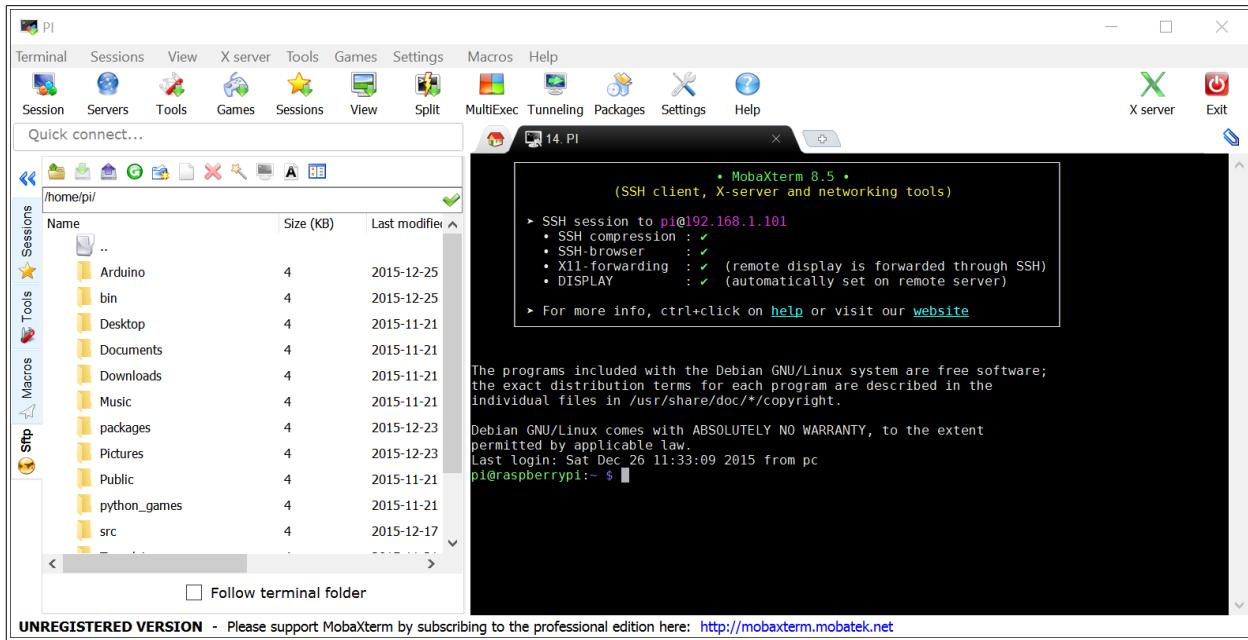
What remains for you to do now is connect into your Pi and run `startlxde`.

An alternative to xming is the Windows based mobaXterm. This has a personal edition which is free of charge. This tool provides an elegant combination of X-Server plus shell login plus file browsing and a host of other features.

When started, we can select to attach to a remote Linux system.



Once selected, we are presented with a very nice terminal emulator which is set up for X clients and provides a file browser:



If we launch an X application, it opens in its own Window window.

## Desktop environments

We need to remember that X-Windows is a raw technology used to display graphics on the screen. It knows nothing about windows or menus or task bars that we now associate with a desktop environment.

Many Linux flavors provide their own desktop environments and the Pi is no exception. It provides a desktop called LXDE.

Among the commands supplied by LXDE are:

- `lxterminal` – An X terminal
- `lxappearance` – Changing the look and feel.

LXDE is not the only possible desktops. Others include [MATE](#).

See also:

- [lxde.org](#)
- [MATE Desktop Environment](#)

## Logging

Messages from various systems and subsystems are logged into files in the `/var/log` directory. There are many log files from all different types of sources here. Some of the more important are:

- `kern.log` – The kernel messages log
- `user.log` – User application messages
- `auth.log` – Authorization messages
- `daemon.log` – Daemon messages
- `kern.log` – Kernel messages

The `dmesg` command displays the kernel logs from a memory ring buffer (a memory buffer that loops around on itself).

Now let us look and see how logging is actually implemented on the Pi. It is based on the `rsyslogd` demon. It is started by default and we can validate that it is running with:

```
$ systemctl status rsyslog
● rsyslog.service - System Logging Service
  Loaded: loaded (/lib/systemd/system/rsyslog.service; enabled)
  Active: active (running) since Thu 2015-12-31 15:37:16 CST; 1 day 3h ago
    Docs: man:rsyslogd(8)
          http://www.rsyslog.com/doc/
 Main PID: 489 (rsyslogd)
   CGroup: /system.slice/rsyslog.service
```

The configuration file for `rsyslogd` can be found at `/etc/rsyslog.conf`. This is a rich file and I don't recommend editing it without first studying in detail the documentation on `rsyslogd`.

The high level notion is that when a message is generated, it is generated by a particular "type" of application. The `rsyslogd` calls this a "facility". In addition, when a message is issued it is issued for a reason. Obviously the reason is to log something for future examination but here we take the reason to mean why the message was issued. For example, is it a warning? Is it debug? The `rsyslogd` calls this the "severity" of the message. Now we have a richer context for considering a message. When a message

is issued it contains within both the facility and severity. The configuration of `rsyslogd` can then use this additional meta information to route the message to the appropriate destination.

To log a message, we use the `syslog` function. We must also include `syslog.h`. The signature of `syslog` is:

```
void syslog(int priority, const char *format, ...)
```

The `priority` is the encoding of both the facility and severity. The `format` parameter is a `printf()` style format string with optional parameters following.

There are definitions provided for the facility:

- `LOG_AUTH` – Authorization messages
- `LOG_AUTHPRIV` – Private authorization messages
- `LOG_CRON` – Messages related to the cron daemon
- `LOG_DAEMON` – Messages related to daemon processes
- `LOG_FTP` – Messages related to file transfer through FTP
- `LOG_KERN` – Messages from the kernel
- `LOG_LOCAL0` – `LOG_LOCAL7` – Messages sources for local use
- `LOG_LPR` – Messages related to interaction with printers
- `LOG_MAIL` – Messages related to the receipt or delivery of mail
- `LOG_NEWS` – Messages related to the Usenet news subsystem
- `LOG_USER` – Generic messages
- `LOG_UUCP` – Messages related to the Unix to Unix copying program

And there are corresponding definitions for the severity:

- `LOG_EMERG` – Messages that indicate a disaster scenario
- `LOG_ALERT` – Messages that indicate immediate attention
- `LOG_CRIT` – Messages that indicate a critical condition that needs addressed
- `LOG_ERR` – Messages that indicate an error
- `LOG_WARNING` – Messages that indicate a warning
- `LOG_NOTICE` – Normal messages but considered important
- `LOG_INFO` – Informational messages
- `LOG_DEBUG` – Debugging or diagnostic messages

Now lets us see what happens when we log a message.

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

int main(int argc, char *argv[]) {
    syslog(LOG_USER | LOG_DEBUG, "Hello World!");
}
```

Since we are logging to `LOG_USER`, once run, we can look at the end of the `/var/log/user.log` file and we will see:

```
Jan  1 19:23:31 raspberrypi a.out: Hello World!
```

Of course, the destination of the message is governed by the configuration of `rsyslogd` and this log file is only the default.

See also:

- [rsyslog home page](#)
- [Raspberry Pi System Logging and Loggly](#)
- man(8) – [dmesg](#)

## Performance

When running your applications, if they are not running fast enough, or stutter or run out of memory, you need to perform some performance analysis. Fortunately, Raspbian has many tools and utilities at your disposal to do just that.

### Resource Utilization

If we have performance issues, the chances are that we may be constrained on resources. As such, we want to determine how to measure the resources we are using and have available.

#### Memory Utilization

Running `vmstat` produces a quick summary of virtual memory availability:

```
$ vmstat
procs -----memory----- ---swap-- -----io---- -system-- -----cpu-----
 r b   swpd   free   buff   cache   si   so   bi   bo   in   cs us sy id wa st
 0 0      0 856032  10488  44484     0     0    17     0 249  132  0  0 99  0  0
```

The column called `free` shows us the number of bytes unused RAM for our applications. In this example, we see 856032 bytes. Since we have a large amount of free ram, we would not expect to be using any swap space and this is backed up by the `swpd` column.

A similar command is the first few lines of the `top` command.

```
$ top
top - 14:13:19 up 14 min,  1 user,  load average: 0.00, 0.03, 0.05
Tasks:  86 total,   1 running,  85 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.2 us,  0.2 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:  948108 total,   91968 used,   856140 free,    10488 buffers
KiB Swap: 102396 total,       0 used, 102396 free.  44484 cached Mem
```

In this line (taken from above)

```
KiB Mem: 948108 total, 91968 used, 856140 free, 10488 buffers
```

We see that we have a total amount of RAM of 948108 bytes of which 91968 is used and 856140 is free. The output here contains more details than `vmstat` as it shows us the total amount of RAM and used RAM as well as unused RAM.

Again, looking at the following line, we see that we have no swap space being used.

```
KiB Swap: 102396 total, 0 used, 102396 free. 44484 cached Mem
```

Since Raspbian supplies virtual memory, this means that each process sees its own address space as though it were the only application in the machine. Since the operating system itself is managing memory, if the accumulation of memory usage across all the programs running on your Pi starts to get close to the total amount of RAM available, the Pi can start paging out RAM to disk. By writing pages of RAM to disk, the operating system can free up those pages for use by other applications. The thinking here is that there will be some applications that are holding on to RAM but are themselves not actively using it. Perhaps these applications sleep for long periods of time or are demon programs. Either way, the chances of them accessing the page of RAM that has been swapped out to disk is believed to be lower than that of other applications. Should those applications later need to access the pages of RAM that have been swapped out, some other application will have some of its pages swapped out and the RAM released by that swapped used to hold the original program's swapped out pages. The algorithms in play here that decide which pages get swapped out is beyond our discussion.

From a performance perspective, it is not necessarily a bad thing to see swap space being used. It simply means that our combined applications are needing more physical RAM than we actually have and the swap space is being used to provide an illusion that we have enough. However, we can enter a situation where we get so low on available RAM relative to the needs of the applications that we spend more and more time swapping in and out pages of RAM than doing productive work. Such a circumstance is called thrashing. Your Pi as a whole will start to feel sluggish at this point.

Should your applications consume more and more RAM, you can imagine that more and more of their pages will be swapped out. Since the amount of disk space allocated to holding swapped out pages (called the swap space) is of a finite and fixed size, we can reach a state where we have filled all our real RAM and also filled the disk space allocated for swap space. At that point your Pi is in trouble. It can no longer allocate extra space for memory requests and applications will start to fail. Raspbian will start to terminate applications to free up their resources.

## Performance Commands

### `top`

The `top` command is a real-time text/full-screen application that shows the current system state as well as the processes currently running and their resource consumption:

Here is an example screen capture:

```

pi@raspberrypi: ~
top - 22:38:01 up 5:29, 3 users, load average: 0.70, 0.72, 0.37
Tasks: 105 total, 1 running, 104 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.2 us, 0.2 sy, 0.0 ni, 99.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem: 948108 total, 246480 used, 701628 free, 23252 buffers
KiB Swap: 102396 total, 0 used, 102396 free. 165800 cached Mem

      PID USER      PR  NI    VIRT    RES   SHR S %CPU %MEM TIME+ COMMAND
 2568 pi        20   0   5092   2500  2144 R  1.0  0.3  0:00.08 top
 820 pi        20   0  11772   3248  2576 S  0.3  0.3  0:02.07 sshd
  1 root       20   0   5384   3896  2728 S  0.0  0.4  0:06.11 systemd
  2 root       20   0     0     0   0 S  0.0  0.0  0:00.01 kthreadd
  3 root       20   0     0     0   0 S  0.0  0.0  0:00.09 ksoftirqd/0
  5 root       0 -20     0     0   0 S  0.0  0.0  0:00.00 kworker/0:0H
  6 root       20   0     0     0   0 S  0.0  0.0  0:01.36 kworker/u8:0
  7 root       20   0     0     0   0 S  0.0  0.0  0:00.50 rcu_preempt
  8 root       20   0     0     0   0 S  0.0  0.0  0:00.00 rcu_sched
  9 root       20   0     0     0   0 S  0.0  0.0  0:00.00 rcu_bh
 10 root      rt  0     0     0   0 S  0.0  0.0  0:00.01 migration/0
 11 root      rt  0     0     0   0 S  0.0  0.0  0:00.01 migration/1
 12 root      20   0     0     0   0 S  0.0  0.0  0:00.00 ksoftirqd/1
 14 root      0 -20     0     0   0 S  0.0  0.0  0:00.00 kworker/1:0H
 15 root      rt  0     0     0   0 S  0.0  0.0  0:00.01 migration/2
 16 root      20   0     0     0   0 S  0.0  0.0  0:00.01 ksoftirqd/2
 18 root      0 -20     0     0   0 S  0.0  0.0  0:00.00 kworker/2:0H
 19 root      rt  0     0     0   0 S  0.0  0.0  0:00.01 migration/3
 20 root      20   0     0     0   0 S  0.0  0.0  0:00.01 ksoftirqd/3
 22 root      0 -20     0     0   0 S  0.0  0.0  0:00.00 kworker/3:0H
 23 root      0 -20     0     0   0 S  0.0  0.0  0:00.00 khelper

```

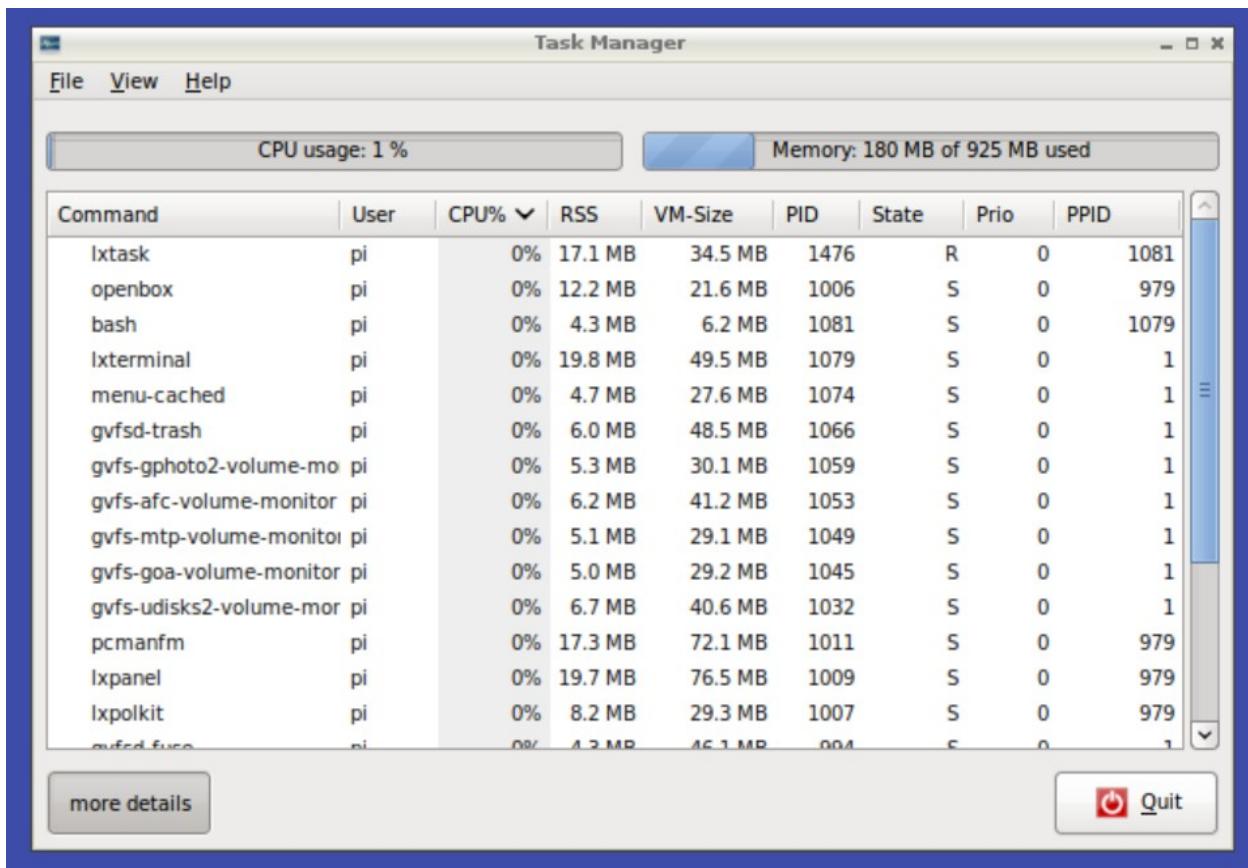
One should spend quite some time studying the manual page to become skilled in the meaning of each of the options. Without explaining where you can see them, the `top` output includes %CPU utilization, how the CPU is being used, how much memory is free, the list of top running processes, how many processes are active ... and much more.

See also:

- [top\(1\) - Linux man page](#)

## lxtask

Running `lxtask` displays an X-Windows based task list.



## vmstat

The `vmstat` command logs statistics about number of processes, virtual memory utilization, swap space, IO traffic, and CPU usage.

```
$ vmstat
procs -----memory----- ---swap--- -----io---- -system-- -----cpu-----
 r b    swpd   free   buff   cache   si   so    bi    bo   in   cs us sy id wa st
 0 0      0 836308 13416 57168    0    0     3    0 106 35  0  0 100  0  0
```

See also:

- `man(8) – vmstat`

## Performance characteristics of different programming languages

There is usually a trade-off between programming abstraction levels and performance. We know that a Pi runs a CPU and that the CPU executes CPU instructions. If we were to write our applications directly in assembly language then we would likely achieve optimal performance. However, in reality, we find that is rarely done. The reason for that is that we are willing to sacrifice that level of performance for the time, ease and maintainability of the source code itself. This is a story that repeats for higher and higher

levels of abstraction. For example, simple C programming translates pretty efficiently to assembly language but yet C is both portable across CPUs and gives us higher levels at which to work. Up from C is C++ adding an Object Oriented flavor. Again, C++ translates to assembly language and hence native executable code. Further up the scale we move away from natively executable code to interpreted code such as JavaScript. Here we have new paradigms that don't have equivalents in the instruction sets of processors such as lambda functions and closures ... features which our human minds can use to simplify our algorithmic tasks when writing programs but which take more execution resources to achieve.

Typically these interpreted programs are parsed at run-time. With some advanced interpreters, compilation is done on the fly or just in time and with other interpreters, no compilation is performed and the instructions are simply re-interpreted each time they are executed. Between interpreted and native executable code there are languages such as Java which compile their source code to a fictitious assembly language for a CPU architecture which does not exist in hardware. Instead a software application called a Virtual Machine "executes" these compiled instructions in software. The advantage here is that a program compiled once to produce these "bytecode" applications is portable to all platforms which have implemented the virtual machine ... irrespective of the underlying hardware CPU that is actually executing the virtual machine.

The choice of programming language you use will most commonly be made based upon your own skills and experience with one or the other. In addition, if you are using specialized and pre-existing libraries, they may only be available for a subset of languages (or even only one). However, when one writes a program ... no matter how elegant that program may be, if it doesn't "work" then it is useless. If we write an application and it takes too long that users aren't prepared to use it ... or worse, it is driving hardware components so slowly that they themselves simply don't operate ... then we may be forced to change from our higher level language of choice down to lower level but more efficient languages.

There are also times where we can design around performance problems while keeping high level concepts. The trick here is to determine why an application is not performing as we desire and see if we can't alleviate those problems with better coding design. Examples might include the use of multi-threading to perform multiple tasks in parallel or we might employ mixed languages. For example a program written in Java can invoke subroutines written in C. These subroutines could encapsulate the tasks that are expensive in Java.

## Security

Since the Pi is in reality a small computer, it is very possible that it could be attacked if on the network. This could mean remote logins, file copies, replacements of your applications or obtaining data contained within. This may seem like overkill and in most cases, you will have no concerns. However consider that you create a project that allows you to unlock your door remotely or switches on your oven. There may be bad consequences if someone could access your programs.

Can we do anything about these potential problems? The answer is of course yes and it is relatively easy to harden a Pi such that it becomes impenetrable to the bad guys.

### Firewalls ... on and off

There are many options for firewalls at different places. Sometimes these can be great, other times they can get in your way. For example, I have a desktop PC running Linux that I wanted to use to export an

NFS file system to a Pi. When I logged into the Pi and tried to mount the file system, it failed with no explanation. Only later did I find that my PC running Linux was running it in Virtual Box on Windows 10 ... and on Windows 10 I had Symantec Endpoint Protection (SEP) installed. To reach the Linux running as a guest in Windows, the network traffic first had to arrive at the Windows network stack and SEP has a default rule to block these kinds of traffic. My solution was to add a new rule that explicitly allowed traffic from my Pi.

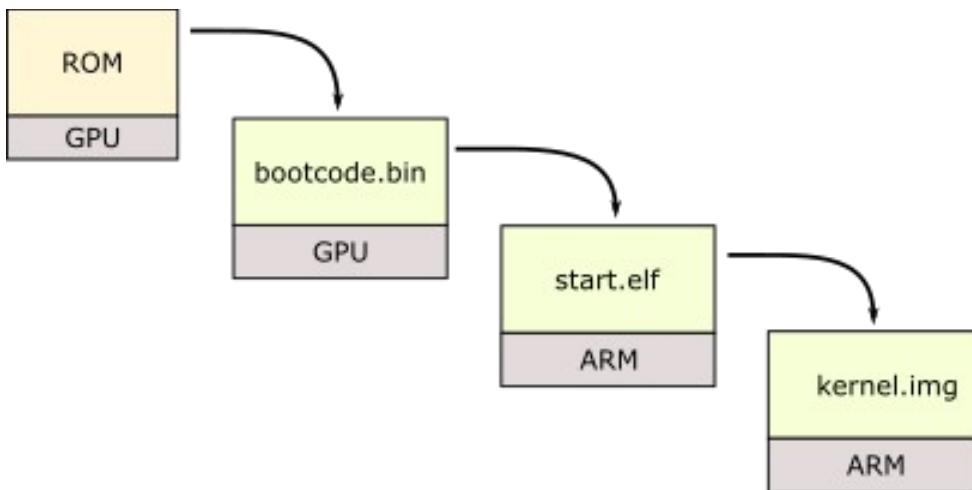
## Bare Metal Programming

So far we have discussed Pi programming from the perspective of a Linux operating system ... however, we don't *have* to run Linux. We can write applications that are loaded from SD data that "just" run on the Pi without any operating system involvement at all. Why might we do that? Perhaps we want optimum speed and control. The writing and deployment of such applications is known as bare metal programming as we are down to the bare metal of the device without any intervening layers in the way. Naturally, bare metal programming is much harder than programming at a higher level.

For bare metal programming, the high level concept is that we will create an executable program which can be placed on an SD card. If we then boot the Pi with that SD card, the program we have written will be loaded and executed. If we compare this against Linux, we find that the program that is normally loaded is Linux itself ... in our story, there simply is no Linux and the program that is loaded is, by itself, just the program.

The program that will be loaded must be called `kernel.img`. For the Pi 2 architecture, the image file can also be a `kernel7.img`. If that file isn't present, then `kernel.img` will be tried. The thinking behind this is probably that an SD card could hold two image files ... one for a Pi 1 and one for Pi 2. When a Pi boots, it can then use the correct image based on its own knowledge of the architecture in play.

Prior to loading the kernel image file, on power up, the Pi goes through a boot sequence. Initially, it is the GPU on the device that gets control which initializes access to the SD card. This initial code is held in ROM. Once the card is accessed, the GPU loads a program called `bootcode.bin` which then loads a program called `start.elf`. This program is the first one that is actually executed on the ARM processor as opposed to the GPU. Finally, the `start.elf` program loads the `kernel.img` file and passes control to it and we are now running. This is illustrated in the following diagram:



See also:

- [Baking Pi – Operating Systems Development](#)
- Forum: [Bare Metal](#)
- Github: [dwelch67 / raspberrypi – Examples](#)
- StackExchange: [What is the boot sequence?](#)

## Hardware Interfacing Examples

Although the Pi family's mission is to promote education on the topic of computing by making the devices affordable to just about anyone, there is something enticing about being able to access physical hardware through its exposed pins. On our expensive PCs, we don't readily have that luxury and if/when we get it wrong, we will be unhappy that our our PC is left a smoking ruin. With a cheap Pi, we are released from the concern and fear that may inhibit us from taking such risks. As such, we want to look into interfacing our Pis with the huge number of electronic components out there to create new and interesting projects and solutions.

The following sections provide illustrations on how to connect electronics to our Pi and then show us a variety of software techniques to drive and interact with them. Since the Pi supports a rich set of programming language choices, rather than pick just one language, we will show interaction through a variety of languages. If you only care about one language, then read the corresponding language specific recipes and discussions. However, this may be an opportunity to compare the language you are familiar with against another language since each of the programs performs exactly the same function ... you may be able to use these language comparisons as a Rosetta stone to provide insight into how different language features work.

When interacting with new components, stop and Google them. It is almost a certainty that someone, somewhere has attempted this task before. In fact, it is as likely that many folks have achieved your task in the past. As such, learn from them. Don't re-invent the wheel. When a manufacturer releases a new component, they commonly make available a document called the "data sheet". Think of this as the specification, operations guide and instructions for using that component. If it is more than a trivial device, then it will expect protocols and bit sequences with associated timings. If you have a working

knowledge of electronics including protocols such as SPI, I2C and UART and are able to write applications in C, then the chances are high that you could start from the data sheet and build your own drivers to make the component work. The question I would pose to you though is why do that? If you could just lift and re-use a high level library that performs the task why would you not just do that? Fortunately, such libraries are prolific. Some may be for the Pi and some may be for other MCUs such as the popular Arduinos. For non-Pi devices, you may have to port the code to the Pi before you can use it.

This brings us to the topic of Open Source projects. These are source code projects where the source is made publicly available and any associated licenses are very permissive. You can usually find an Open Source project that comes close to meeting your needs and you can start from there. Open Source benefits from the notion of others contributing back. This may be in the form of improvements or enhancements to the code base or it may be in the form of improvements to the documentation (if any). If you start to use an Open Source project and find problems and fix them, submit them back so that the next person who walks your path may benefit from the time and investment you yourself made.

When working on a project and it starts to come together, it is natural to feel proud of one's work. However, I caution against the idea of thinking that your Open Source library is better than an existing Open Source library. Instead, think about how you can improve what is already there as opposed to duplicating an existing effort because (perhaps) you disagree with one small aspect of such an existing project. Don't publish a new wheel just because you can, it will confuse future Open Source users when they look for a functional area and find dozens of them as opposed to just one good one.

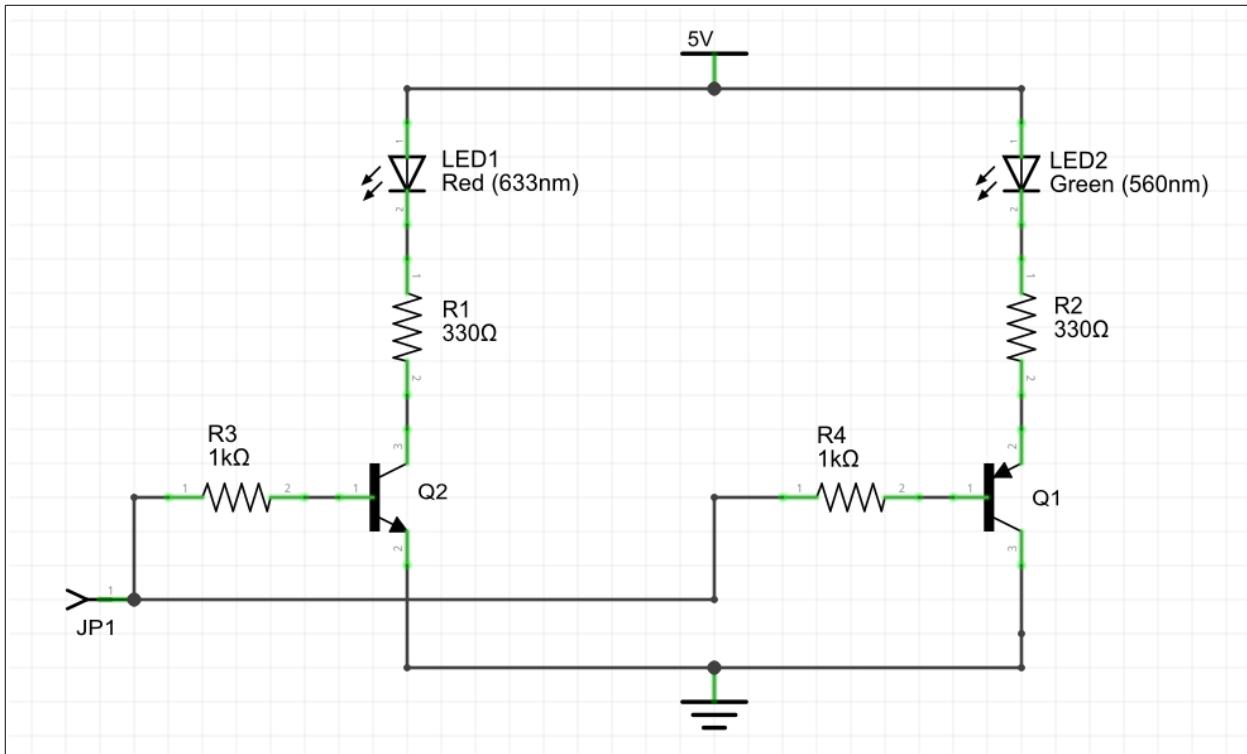
The Github repository is an excellent place to find and work with Open Source projects. It is based upon the Git source code control paradigm and Git is now available on all operating systems and built into the majority of Integrated Development Environments including Eclipse.

## **Simple GPIO Output change**

One of the simplest interfaces is the ability to change the signal level on an output GPIO from low to high or high to low. This can be used to trigger all kinds of external devices such as switching on a simple LED or energizing a relay. If performed in conjunction with toggling other pins, one can even create complex signals across multiple pins that can recreate complex protocols such as I2C or SPI.

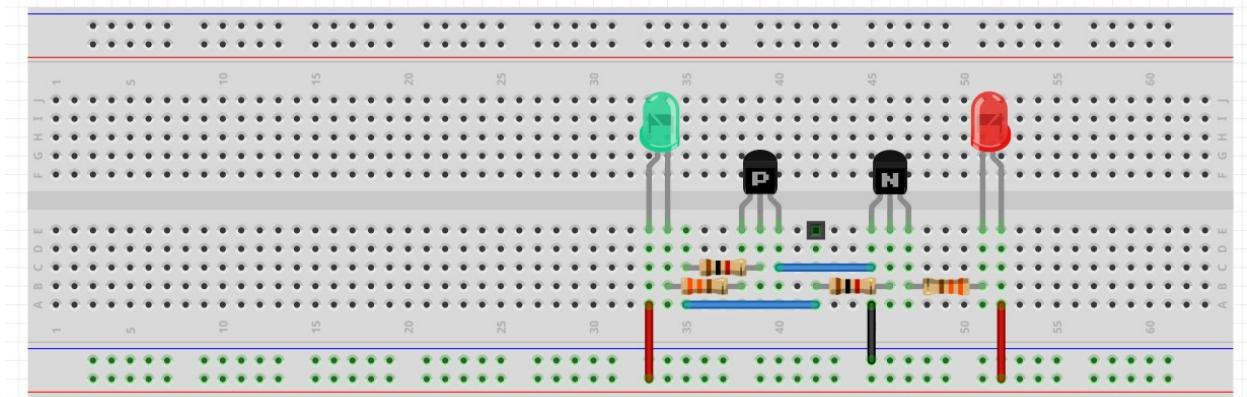
After we write our applications to test signal changes on output, we can determine if everything is working in a number of fashions. We can attach a simple meter or logic probe or get sophisticated and attach a signal analyzer. However the simplest mechanism is still the attachment of an LED with a resistor. When the signal is high, the LED lights up. When the signal is low, the LED remains dark.

We can create a bit of a more sophisticated circuit which is a simple logic probe as follows:



In this circuit, if the input is high, the Red LED (LED1) will switch on and the Green go dark, while if the input is low, the Green LED (LED2) will switch on and the Red go dark. If the input is open, then both LEDs will be on.

As a breadboard layout, here is the same circuit:



With this test circuit available to us, we are now at a place where we have an environment in which to test simple signal changes. We will pick a GPIO pin, attach our logic probe and toggle the value periodically to watch it change.

### Using the shell

We can use the WiringPi `gpio` command to toggle pin values from a script. The following illustrates this:

```

#!/bin/bash
gpio mode 0 out
value="1"
echo "Starting blinky"
while true
do
    gpio write 0 $value
    if [ $value = "1" ]
    then
        value="0";
    else
        value="1";
    fi
    sleep 0.5
done

```

First we set the mode of the pin to output to indicate that we are going to change the signal levels. Next we then write a 1 or 0 to the pin to drive it high or low.

## Using WiringPi

To change outputs using WiringPi, we setup the WiringPi environment, we set the mode of the desired pin to be output and then loop around changing the output value. The trickiest part of this code is determining the pin encoding. In this example we are using standard GPIO numbering.

```

#include <stdio.h>
#include <wiringPi.h>

#define PIN (17)

int main(int argc, char *argv[]) {
    printf("Starting blink test\n");
    int rc = wiringPiSetupGpio();
    if (rc != 0) {
        printf("Failed to wiringPiSetupGpio()\n");
        return 0;
    }

    int value = HIGH;
    pinMode(PIN, OUTPUT);

    while(1) {
        digitalWrite(PIN, value);
        delay(500);
        value= value==HIGH?LOW:HIGH;
    }
}

```

## Using Pi4J

Here is an example of using Pi4J to toggle a pin on the Pi. Be aware that the pin numbering for Pi4J is convention adopted by WiringPi.

```
package kolban;

import com.pi4j.io.gpio.GpioController;
import com.pi4j.io.gpio.GpioFactory;
import com.pi4j.io.gpio.GpioPinDigitalOutput;
import com.pi4j.io.gpio.RaspiPin;

public class Main1 {
    static final GpioController gpio = GpioFactory.getInstance();

    public static void main(String[] args) {
        System.out.println("Running Blinky");
        try {
            GpioPinDigitalOutput myLed = gpio.provisionDigitalOutputPin(RaspiPin.GPIO_00);
            boolean value = true;
            while (true) {
                myLed.setState(value);
                Thread.sleep(500);
                value = !value;
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

## Using Node.js

The following uses the wiring-pi npm library to toggle a pin every interval:

```
var wpi = require('wiring-pi');
wpi.setup('wpi');
var pin = 0;
wpi.pinMode(pin, wpi.OUTPUT);
var value = 1;
setInterval(function() {
    wpi.digitalWrite(pin, value);
    value = +!value;
}, 500);
```

## Using RasPiArduino

The following uses the RasPiArduino library:

```

#include <Arduino.h>

#define LEDPIN (17)

unsigned long startTime;
int value = HIGH;
void setup() {
    printf("Setup called!\n");
    startTime = millis();
    pinMode(LEDPIN, OUTPUT);
}

void loop() {
    if (millis() - startTime > 1000) {
        startTime = millis();
        digitalWrite(LEDPIN, value);
        printf("Value: %d\n", value);
        if (value == HIGH) {
            value = LOW;
        } else {
            value = HIGH;
        }
    }
}

```

## Using Python

Through Python we can use the `RPi.GPIO` module to interact with the GPIOs.

```

#!/usr/bin/env python
import RPi.GPIO as GPIO
import time
GPIO.setmode(GPIO.BCM)
GPIO.setup(19, GPIO.OUT)
while(True):
    GPIO.output(19, GPIO.LOW)
    time.sleep(0.5)
    GPIO.output(19, GPIO.HIGH)
    time.sleep(0.5)

```

See also:

- Python `RPi.GPIO` reference
- [Getting Started with Raspberry Pi GPIO and Python](#)

## Simple GPIO Input detection

Moving on from changing the output of a GPIO pin, we can now start to look at reading the input from a pin. If we think about the electrical characteristics of a pin, we will realize that it can have three possible

digital states. It can be connected to ground and represent a low signal. It can be connected to +ve and represent a high signal. The third state comes into play when the pin is simply not connected. In this case, the resulting signal could be considered floating and not represent anything meaningful as it may change at will between high and low. Alternatively, a resistor could be brought into play such that when the pin is not connected, the resistor could be connected to ground to pull the pin low or could be connected to +ve to pull the pin high.

To read from a pin, we must set the pin mode to be input. Once done, we can then read the signal value. The Pi also has built in resistors so we don't have to physically wire them ourselves. We can ask the Pi to introduce a pull-up resistor, a pull-down resistor or no resistor.

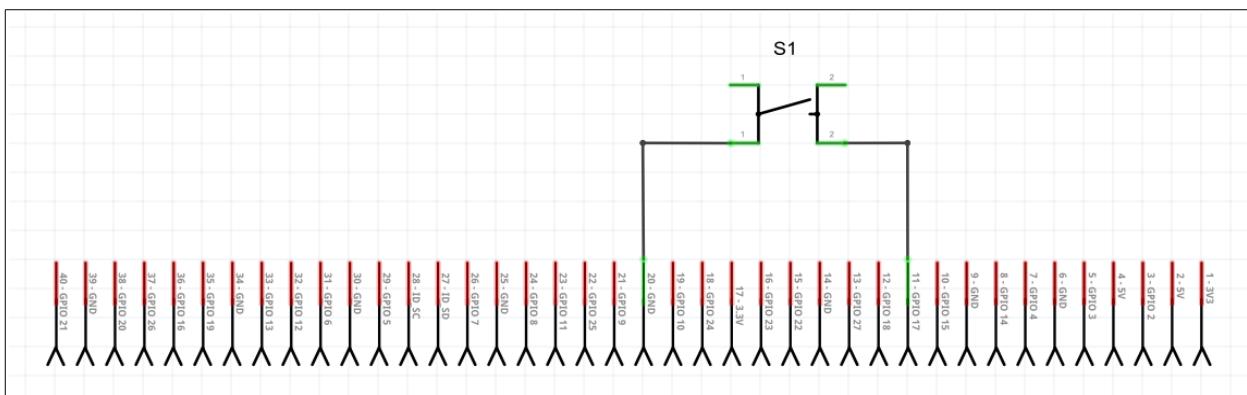
Our simplest test of input data processing will be the humble push-button. This is about the simplest electronic component there is. It consists of a plastic tab on the top which, when pressed, causes a piece of internal metal to bridge the gap between the button's pins. When the tab is released, the metal springs back and the electrical connection is broken.

If we connect one end of the button to a ground or +ve and the other end of the button to a GPIO pin, then when we press the button, the signal arriving at the GPIO pin will change and we can detect that occurrence.

Common momentary connection push buttons have 4 pins. The pins on the same side are electrically connected to each other.



Here is a simple circuit diagram illustrating a button wired to GPIO 17.



## Using the shell

We can use the `gpio read` command to read the value from a pin. Here is a simple shell example that polls a pin and reports its value.

```

#!/bin/bash
gpio mode 0 up
echo "Starting input"
while true
do
    value=$(gpio read 0)
    echo "Value is: $value"
    sleep 1
done

```

Note that we set the mode of the pin to be input with a pull-up. Since a button press will bring the signal to 0, we want to ensure that when not pressed, the signal is registered as high. If we want to switch things around, we can connect the button to 3.3V and set the mode to `gpio mode 0 down`.

## Using WiringPi

From the WiringPi library, the core function is `digitalRead()` which reads the value from a specified pin. We also use `pullUpDnControl()` to define a pull-up resistor to prevent the pin from floating when no explicit signal is provided to it.

```

#include <stdio.h>
#include <wiringPi.h>

#define PIN (17)

int main(int argc, char *argv[]) {
    printf("Starting input test\n");
    int rc = wiringPiSetupGpio();
    if (rc != 0) {
        printf("Failed to wiringPiSetupGpio()\n");
        return 0;
    }

    pinMode(PIN, INPUT);
    pullUpDnControl(PIN, PUD_UP);

    while(1) {
        int value = digitalRead(PIN);
        printf("Value is %d\n", value);
        delay(500);
    }
}

```

## Using Pi4J

```
package kolban;

import com.pi4j.io.gpio.GpioController;
import com.pi4j.io.gpio.GpioFactory;
import com.pi4j.io.gpio.GpioPinDigitalInput;
import com.pi4j.io.gpio.PinPullResistance;
import com.pi4j.io.gpio.RaspiPin;

public class Input {

    static final GpioController gpio = GpioFactory.getInstance();

    public static void main(String[] args) {
        System.out.println("Running Input");
        try {
            GpioPinDigitalInput myInput = gpio.provisionDigitalInputPin(RaspiPin.GPIO_00);
            myInput.setPullResistance(PinPullResistance.PULL_UP);
            while (true) {
                System.out.println("Pin state is: " + myInput.getState());
                Thread.sleep(500);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

## Using Node.js

For Node.js we setup the interface to WiringPi and then wake up each 500 msecs and read the value of the pin and log it to the console.

```
var wpi = require('wiring-pi');

wpi.setup('wpi');

var pin = 0;

wpi.pinMode(pin, wpi.INPUT);

var value = 1;

setInterval(function() {
    var value = wpi.digitalRead(pin);
    console.log("value is " + value);
}, 500);
```

## Using Python

```
#!/usr/bin/env python
import RPi.GPIO as GPIO
import time
GPIO.setmode(GPIO.BCM)
GPIO.setup(19, GPIO.IN)
while(True):
    value = GPIO.input(19)
    print("value is %d" %value)
    time.sleep(0.5)
```

See also:

- [Python RPi.GPIO reference](#)

## Interrupt Driven Input detection

If we continually read the value from a GPIO pin looking at the value, we are effectively polling the pin. In a loop flow based program this is an easy concept to consider and may be sufficient for many projects. However there are alternative techniques available to us. Imagine the notion of someone arriving at your front door. If you want to know when someone arrives, you could continually get up of your couch and open the door and see if anyone is there. You usually don't do that because you have other things to do and the likelihood of someone new arriving is so relatively low that the vast majority of times you open the door, no-one is there. In addition, there is also an ugly possibility that someone could arrive at the door just after you checked and leave again before you come back to check again. From your perspective, not only did you miss them, you didn't even know they ever visited in the first place.

In the real world, it is most likely that you have a door bell attached to your door. When a guest arrives they ring the bell and that acts as a notification that you should answer the door and attend to the visitor. Since the bell can ring at most any time and you will interrupt what you are doing to attend to the door, we call the door bell an instance of an interrupt.

In the Pi, we also have the notion of interrupts associated with the GPIO pins. These interrupts get registered with pins and when the signal level on the pin changes, it is the Pi itself that becomes aware of the change deep within its hardware. This interrupt detection can then be used to trigger logic in your own program to make you aware that the signal changed.

## Using the shell

We can block waiting for the signal on a pin going from low to high with:

```
gpio wfi 0 rising
```

or waiting for the signal on a pin going from high to low with:

```
gpio wfi 0 falling
```

or either direction (high to low or low to high) with:

```
gpio wfi 0 both
```

## Using WiringPi

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

#define PIN (17)

static void callback() {
    printf("The interrupt was triggered");
}

int main(int argc, char *argv[]) {
    printf("Starting input test\n");
    int rc = wiringPiSetupGpio();
    if (rc != 0) {
        printf("Failed to wiringPiSetupGpio()\n");
        return 0;
    }

    pinMode(PIN, INPUT);
    pullUpDnControl(PIN, PUD_UP);
    wiringPiISR(PIN, INT_EDGE_RISING, callback);

    while(1) {
        delay(50000);
    }
}
```

## Using Pi4J

```
package kolban;

import com.pi4j.io.gpio.GpioController;
import com.pi4j.io.gpio.GpioFactory;
import com.pi4j.io.gpio.GpioPinDigitalInput;
import com.pi4j.io.gpio.PinPullResistance;
import com.pi4j.io.gpio.RaspiPin;
import com.pi4j.io.gpio.event.GpioPinDigitalStateChangeEvent;
import com.pi4j.io.gpio.event.GpioPinListenerDigital;

public class Interrupt {

    static final GpioController gpio = GpioFactory.getInstance();

    public static void main(String[] args) {
        System.out.println("Running Interrupt");
        try {
            GpioPinDigitalInput myInput = gpio.provisionDigitalInputPin(RaspiPin.GPIO_00);
            myInput.setPullResistance(PinPullResistance.PULL_UP);
            myInput.addListener(new GpioPinListenerDigital() {

                @Override
                public void
                handleGpioPinDigitalStateChangeEvent(GpioPinDigitalStateChangeEvent changeEvent) {
                    System.out.println("The pin changed state to: " + changeEvent.getState());
                }
            });
            while (true) {
                Thread.sleep(50000);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

## Using Node.js

Similar to wiringPi in C, there is a method called `wiringPiISR()` in the Node.js `wiringPi` library. The third parameter to this function is itself a function which will be invoked when an interrupt is detected.

See also:

- [wiringPiISR](#)

## GPIO extenders

Although the Pi has a large number of exposed GPIO pins, there are always times when you find yourself wanting more. We have the capability to expand the number of GPIOs available to us through some relatively inexpensive integrated circuits.

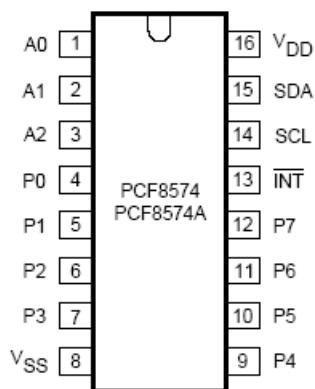
### PCF8574

An example of an extender is called the PCF8574. The PFC8574A is the same but has a different set of addresses. The PCF8574 can be found on eBay for about 40 cents each.

This is an I2C device and hence works over only two wires. Using this IC we supply a 3 bit address (000-111) that is used to select the slave address of the device. Since each address has 8 IOs and we can have up to 8 devices, this means a total of 64 additional pins.

It appears that the device will use a pull-up resistor for a high and bring the pin to ground for low. This means that if we want to use any of the pins for input, we should set their write mode to high first. This will allow either a high or low input signal to be detected. It would appear that if we set the output signal to be low and then fed a raw high signal into the device, we would have a short.

Here is the pin diagram for the device:



Here is a description of the pins:

Symbol	Pin	Description
A0-A2	1, 2, 3	Addressing
P0-P7	4, 5, 6, 7, 9, 10, 11, 12	Bi directional I/O
INT	13	Interrupt output
SCL	14	Serial Clock Line
SDA	15	Serial Data Line
V <sub>DD</sub>	16	Supply Voltage (2.5V – 6V)
Vss (Ground)	8	Ground

The address that the slave device can be found upon is configurable via the A0-A2 pins. It appears at the following address:

## PCF8574

<b>6</b>	<b>5</b>	<b>4</b>	<b>3</b>	<b>2</b>	<b>1</b>	<b>0</b>
0	1	0	0	A2	A1	A0

## PCF8574A

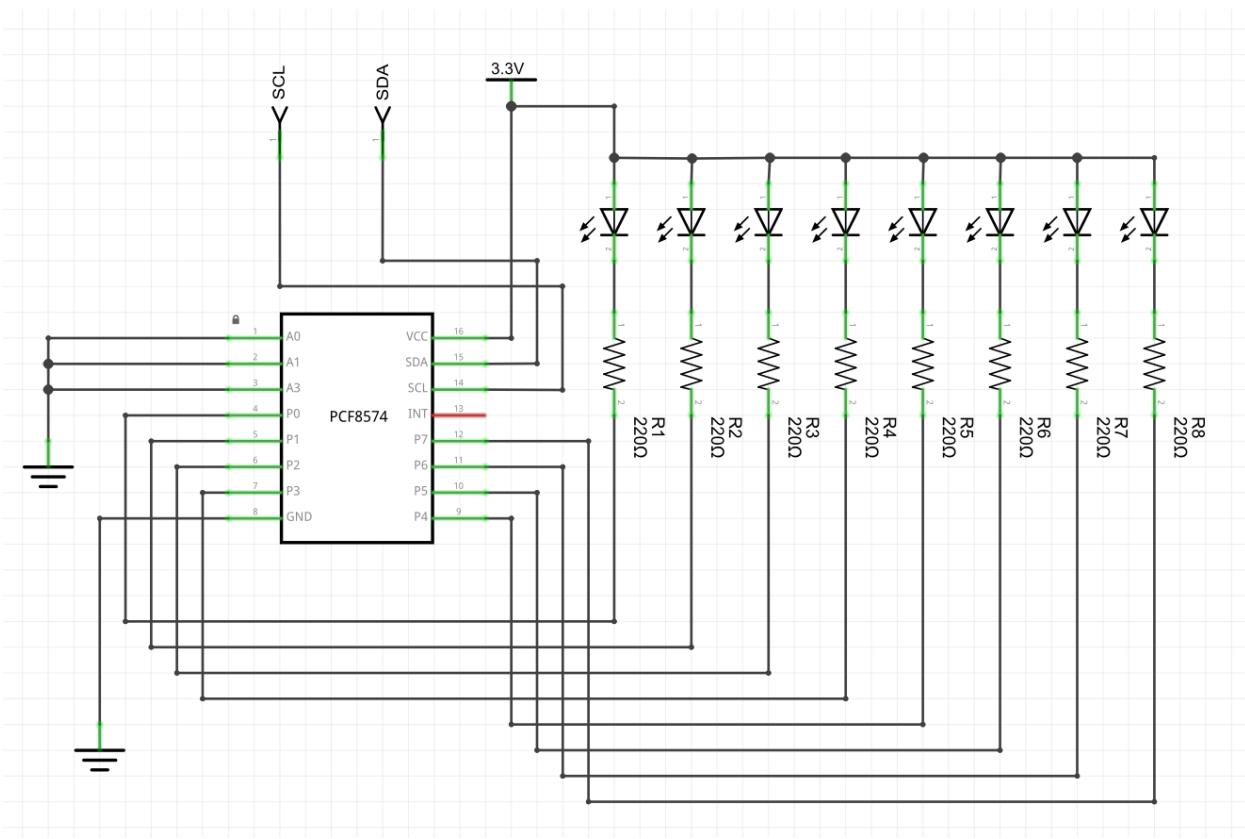
<b>6</b>	<b>5</b>	<b>4</b>	<b>3</b>	<b>2</b>	<b>1</b>	<b>0</b>
0	1	1	1	A2	A1	A0

The pins A0–A2 must not float.

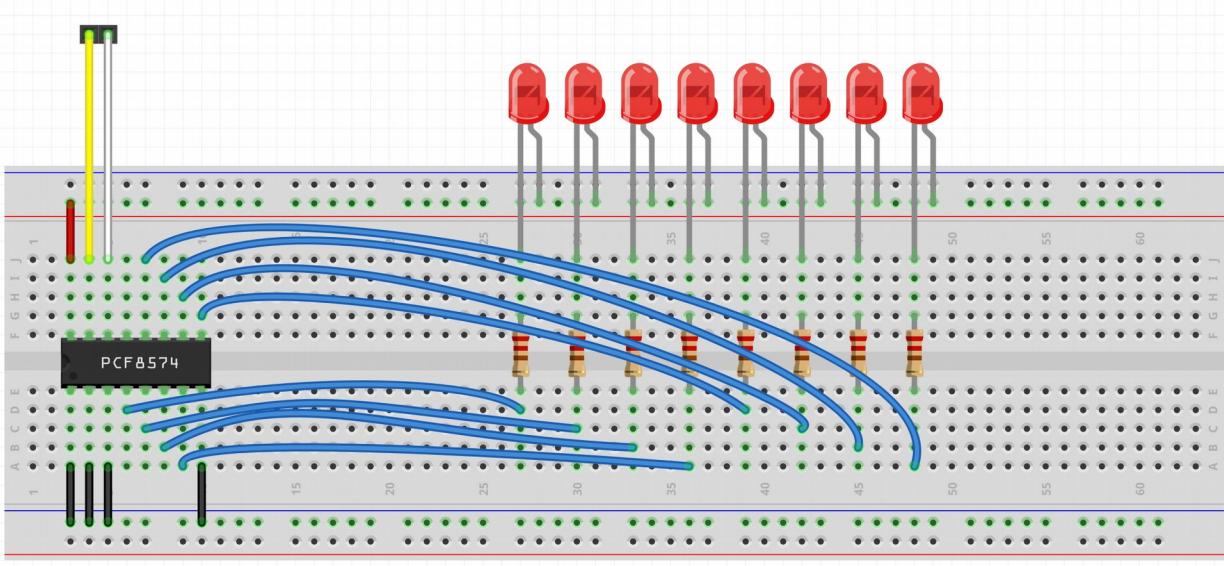
This results in the following table:

A2	A1	A0	Address PCF8574	Address PCF8574A
0	0	0	0x20	0x38
0	0	1	0x21	0x39
0	1	0	0x22	0x3a
0	1	1	0x23	0x3b
1	0	0	0x24	0x3c
1	0	1	0x25	0x3d
1	1	0	0x26	0x3e
1	1	1	0x27	0x3f

We can build a sample circuit that shows driving LEDs:



And on a breadboard:



See also:

- [Datasheet](#)

## **MCP23017**

The MCP23017 from Microchip is a 16 bit input/output port expander that uses the I2C interface. The going price for an instance of one of these on eBay is about \$1. The device has 16 GPIO pins that can be set as input or output controlled in two banks (ports). We can read or write the values of one bank at a time meaning that if we want to write all 16 bits, this would be two I2C operations and the same for reading. The device is also capable of generating interrupts for input signal detection. If we imagine a 100Khz clock rate, then to switch a bit on or off (and we can do these in groups of 8) then that would be 3 bytes of data plus acknowledgements ... ~30 bits ... which would imply a maximum switching rate of about 0.3ms.

The pin interface is:

<b>Pin</b>	<b>Label</b>	<b>Description</b>
1	GPB0	Bi-directional I/O
2	GBP1	Bi-directional I/O
3	GBP2	Bi-directional I/O
4	GBP3	Bi-directional I/O
5	GBP4	Bi-directional I/O
6	GBP5	Bi-directional I/O
7	GBP6	Bi-directional I/O
8	GBP7	Bi-directional I/O
9	VDD	Power (3.3V - 5V)
10	VSS	Ground
11	NC	Not connected
12	SCL	Serial clock input
13	SDA	Serial data input/output
14	NC	Not connected
15	A0	Address pin
16	A1	Address pin
17	A2	Address pin
18	RESET	Hardware reset
19	INTB	Interrupt output for port B
20	INTA	Interrupt output for port A
21	GPA0	Bi-directional I/O
22	GPA1	Bi-directional I/O
23	GPA2	Bi-directional I/O
24	GPA3	Bi-directional I/O
25	GPA4	Bi-directional I/O
26	GPA5	Bi-directional I/O
27	GPA6	Bi-directional I/O
28	GPA7	Bi-directional I/O

The I2C address of the device is 7 bits given by

<b>6</b>	<b>5</b>	<b>4</b>	<b>3</b>	<b>2</b>	<b>1</b>	<b>0</b>
0	1	0	0	A2	A1	A0

And hence has addresses in the range 0x20 – 0x27.

The register addresses are:

<b>Register address</b>	<b>Name</b>	<b>Description</b>
0x00	IODIRA	Direction control for port A: <ul style="list-style-type: none"> <li>• 1 – input</li> <li>• 0 – output</li> </ul>
0x01	IODIRB	Direction control for port B: <ul style="list-style-type: none"> <li>• 1 – input</li> <li>• 0 – output</li> </ul>
0x02	IPOLA	Polarity inversion for input for port A: <ul style="list-style-type: none"> <li>• 1 – inverted</li> <li>• 0 – as-is</li> </ul>
0x03	IPOLB	Polarity inversion for input for port B: <ul style="list-style-type: none"> <li>• 1 – inverted</li> <li>• 0 – as-is</li> </ul>
0x04	GPINTENA	Interrupt enable for a change for port A: <ul style="list-style-type: none"> <li>• 1 – Enable input pin for interrupt on change event</li> <li>• 0 – Disable input pin for interrupt on change event</li> </ul>
0x05	GPINTENB	Interrupt enable for a change for port B: <ul style="list-style-type: none"> <li>• 1 – Enable input pin for interrupt on change event</li> <li>• 0 – Disable input pin for interrupt on change event</li> </ul>
0x06	DEFVALA	Default values for interrupt change comparison for port A.
0x07	DEFVALB	Default values for interrupt change comparison for port B.
0x08	INTCONA	Interrupt control for port A: <ul style="list-style-type: none"> <li>• 1 – Compare the value on the pin to the value in DEFVALA</li> <li>• 0 – Compare the value on the pin to its previous value</li> </ul>
0x09	INTCONB	Interrupt control for port B: <ul style="list-style-type: none"> <li>• 1 – Compare the value on the pin to the value in DEFVALB</li> <li>• 0 – Compare the value on the pin to its previous value</li> </ul>
0x0a	IOCON	Same register as 0x0b

		<b>Bit</b>	<b>Label</b>	<b>Description</b>
		7	BANK	Register mapping: <ul style="list-style-type: none"><li>• Registers in banks</li><li>• Registers sequential</li></ul>
		6	MIRROR	Interrupt pin mapping: <ul style="list-style-type: none"><li>• 1 – INT pins connected</li><li>• 0 – INT pins separate. INTA is associated with port A and INTB is associated with port B</li></ul>
		5	SEQOP	Sequential operation mode: <ul style="list-style-type: none"><li>• 1 – Sequential operation disabled</li><li>• 0 – Sequential operation enabled</li></ul>
		4	DISSLW	Slew rate control
		3	HAEN	Hardware address enable
		2	ODR	Open drain output for INT
		1	INTPOL	Polarity of INT output: <ul style="list-style-type: none"><li>• 1 – Active high</li><li>• 0 – Active low</li></ul>
		0	N/A	Not used. Set as 0.
0x0b	IOCON	Same register as 0x0a		
0x0c	GPPUA	Pull up control for Port A: <ul style="list-style-type: none"><li>• 1 – Pull-up via a 100K resistor</li><li>• 0 – Pull-up disabled</li></ul>		
0x0d	GPPUB	Pull up control for Port B: <ul style="list-style-type: none"><li>• 1 – Pull-up via a 100K resistor</li><li>• 0 – Pull-up disabled</li></ul>		
0x0e	INTFA	Interrupt flags for Port A: <ul style="list-style-type: none"><li>• 1 – Pin caused an interrupt</li><li>• 0 – No interrupt detected</li></ul>		
0x0f	INTFB	Interrupt flags for Port B: <ul style="list-style-type: none"><li>• 1 – Pin caused an interrupt</li><li>• 0 – No interrupt detected</li></ul>		
0x10	INTCAPA	Values of GPIO when interrupt occurred on port A: <ul style="list-style-type: none"><li>• 1 – Pin was high</li><li>• 0 – Pin was low</li></ul>		
0x11	INTCAPB	Values of GPIO when interrupt occurred on port B: <ul style="list-style-type: none"><li>• 1 – Pin was high</li><li>• 0 – Pin was low</li></ul>		
0x12	GPIOA	Read – Reads the values on port A		
0x13	GPIOB	Read – Reads the values on port B		
0x14	OLATA			
0x15	OLATB			

See also:

- [Data sheet](#)

## NeoPixels

### NeoPixel theory

NeoPixels are LEDs that are driven by a single data line of high speed signaling. Most NeoPixels have a +ve and ground voltage source as well as a data line for input and a data line for output. The output of one NeoPixel can be fed into the input of the next one to produce a string of such LEDs. The input data to the LED is a stream of 24 bits of encoded data which should be interpreted as 8 bits for the red channel, 8 bits for the green channel and 8 bits for the blue channel. Each channel can thus have a luminance value of between 0 and 255. By mixing the values for each of the channels together, you can color an LED to any color you may choose. After sending in a stream of 24 bits, if we send in a second stream of 24 bits quickly after the first stream, the second stream is "pushed" through to the next LED in the chain. This can be repeated as far as desired. If we pause sending in data, the current values are "latched" into place and each LED remembers its own value.

The timings of the data signals for these LEDs can be quite tricky but fortunately great minds have already built fantastic libraries for driving them correctly so we need not concern ourselves with these low level timings and can instead concentrate on devising interesting projects and purposes to which the LEDs can be placed. There are a number of different types of these LEDs with the most common ones being known as WS2811, WS2812 or PL9823.

### NeoPixels on the Pi

The timings required to operate NeoPixels natively on the Pi are either not precise enough or because of the pre-emptive nature of Linux, not consistent enough. As such, the recommended solution is to drive them through an intermediate such as an Atmel processor.

There are reports of potential timing solutions for the Pi that use combinations of PWM and DMA.

See also:

- Adafruit – [NeoPixels on Raspberry Pi](#)

## DC Electric Motors

One should never power an electric motor directly from a Pi. It draws too much current and could damage the Pi. In addition, one can generate a "spike" of current in the opposite direction when a motor stops which can again be too much. There are a number of solutions to these problems and one of the more common is to use a driver IC such as the ULN2003A. This device can sit between the I/O pins of the Pi and the motor and drive current to the motor while insulating the Pi from possible harm.

See also:

- [ULN2003A](#)

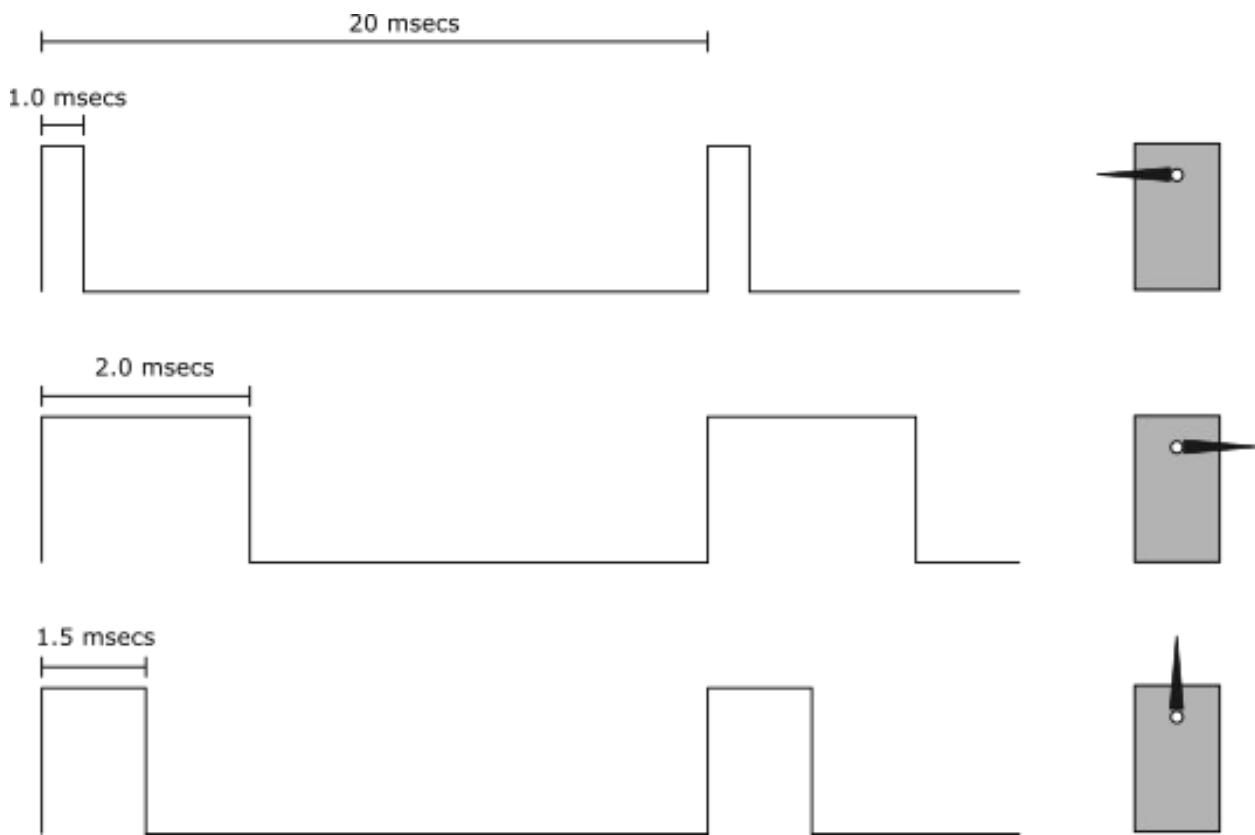
## Servos

A servo is a physical movement device similar to an electric motor. Typically, a servo has a rotation shaft that can rotate from 0 degrees to 180 degrees as a function of the signal applied to it. If one sets the signal to a specific value, the shaft will rotate to a specific position as a function of that signal. The actual minimum and maximum angles are specified by the manufacturer of any particular servo model.

The signal expected by a servo is encoded as Pulse Width Modulation (PWM). One should always consult the manufacturers data sheet to find the specific values. However, in general, the story works as follows.

The period of the PWM is 20 milliseconds. At the start of the period, the duration that the signal is high determines the rotation of the shaft. For example, it may be defined that there is a minimum value of 1 millisecond and a maximum value of 2 milliseconds. This means that if the signal is high for 1 millisecond, the shaft rotates to its minimum value. If the signal is high for 2 milliseconds, the shaft rotates to its maximum value. For high durations between 1 and 2 milliseconds, the shaft rotates proportionally to that duration.

Here is a diagram (not drawn to scale) to illustrate the principle:



Again, it is important to read the data sheet to find the correct values for minimum and maximum duty cycle widths.

The PWM signal that is used to control a servo should be as "clean" as possible. What I mean by this is that if we want to position the servo at a specific rotation, we must keep sending a PWM signal with the same duty cycle value. If the duty cycle value changes, then so will the rotation of the servo shaft. If we wish to keep the shaft fixed at some rotational value, then the duty cycle must remain at that duration. When it comes to implementations of PWM, some variants are implemented in software using high resolution timers. For example, we could set a pin high and then set a high resolution timer to inform us when it is time to set the signal low to indicate the end of the duty cycle. Caution needs to be had in multi-tasking systems or in systems with interrupt services. These are not necessarily real-time systems and the expiry of a duration timer may not be handled at precisely the time we might expect. As you might tell from the diagram, a rotational difference of 180 degrees occurs with a mere 1 millisecond (1/1000th of a second) distinction. If we are off by even a little bit from when a software timer expires, the end result can be dramatic "jitter" of the servo shaft. Not only can this give a poor result but can cause excessive wear on the servo and reduce its life.

From a Pi perspective, we have three variables to play with when calculating and setting PWM control. The first is the clock divisor, the second is the range and the third is the value. See the PWM discussion for details on those.

Here is an example set of choices for working with PWM:

Function	Value
Clock divisor	3840
Range	100
Values	2 = 1.0 msecs, 5 = 2.0 msecs

Some of the more popular servos for playing with are:

- Hossen Mini SG90 – [Data sheet](#)

We also need to remember that the voltage output from a Pi GPIO pin is 3.3v. Most of the servos require a signal at the 5v level. This means that we can't directly drive a servo's data signal input from a Pi pin. We will need to employ a logic level shifter to shift the output PWM signal from 3.3V up to 5V.

See also:

- PWM
- YouTube: [Servo Control with the Raspberry Pi](#)
- Logic Level Shifting

## Using the gpio command

When using the gpio command, we perform the task in two steps. First we define the pin that we wish to use for servo control using the `gpio mode pwm` command. Then we can set the value of the PWM signal using the `gpio pwm` command.

The pins allowable for PWM are:

Physical	Wiring Pi GPIO	Broadcom GPIO	Function
12	GPIO 01	GPIO 18	PWM0
32	GPIO 26	GPIO 12	PWM0
33	GPIO 23	GPIO 13	PWM1
35	GPIO 24	GPIO 19	PWM1

The following will setup Pin 18/01 as a PWM:

```
$ sudo gpio pwm 1
$ sudo gpio pwmc 3840
$ sudo gpio pwmr 100
```

we can now set values of the PWM output to be between ~2 and ~12 to change the position of the servo.

## Using WiringPi

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

#define PIN (18)

int main(int argc, char *argv[]) {
    printf("Starting servo test\n");
    int rc = wiringPiSetupGpio();
    if (rc != 0) {
        printf("Failed to wiringPiSetupGpio()\n");
        return 0;
    }

    pinMode(PIN, PWM_OUTPUT);
    pwmSetMode(PWM_MODE_MS);
    pwmSetClock(3840);
    pwmSetRange(100);

    while(1) {
        int i;
        for (i=3; i<12; i++) {
            pwmWrite(PIN, i);
            delay(1000);
        }
        for (i=11; i>2; i--) {
            pwmWrite(PIN, i);
            delay(1000);
        }
    }
}
```

See also:

- [pinMode](#)
- [pwmSetMode](#)

- pwmSetRange
- pwmSetClock

## GPS

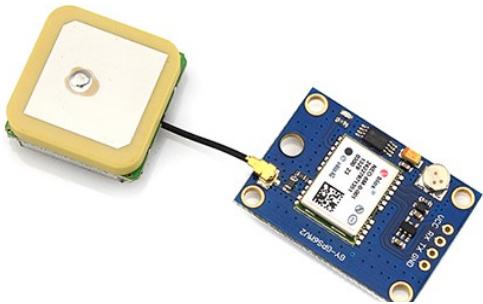
The Global Positioning System (GPS) is a set of satellites in orbit. They are continually transmitting a very precise time signal. If we have a suitable receiver, we can receive the time signals from some number of those satellites that are over head at any given time. Since the speed of radio transmission is constant and not instantaneous and each satellite is producing an extremely accurate and synchronized time signal, then a signal clock pulse emitted by all satellites at exactly the same time will be received at very slightly staggered times by the receiver as a function of the distance that the signal had to travel through space. This is the same as the distance that the satellite was from the receiver. By receiving enough signals from a sufficient number of distinct satellites and by using complex mathematics, the receiver can then triangulate its own position and thus know where it is on the surface of the Earth.

Putting that in perspective, an electronic module can determine physically where it is. Such modules are now common within your cell phone and within many car dashboards. They are often used in conjunction with mapping software to provide a real-time map showing your location.

Devices can be picked up for about the \$12 mark. The unit I worked with is called the GY-GPS6MV2 which is based on the u-blox NEO-6m. The pins on this breakout board are vcc, gnd, RX and TX. This is 5V device. As such, you **must** use a level shifter or voltage divider between the TX pin of the GPS and the RX pin of the pi as the Pi can't accept a 5V signal input.

Since it is a UART device, we can test this on a regular PC. If we connect a serial port terminal, we can watch the data be received. The data that comes across is in NMEA format (National Marine Electronics Association). There are various NMEA reader applications freely available which can format the data.

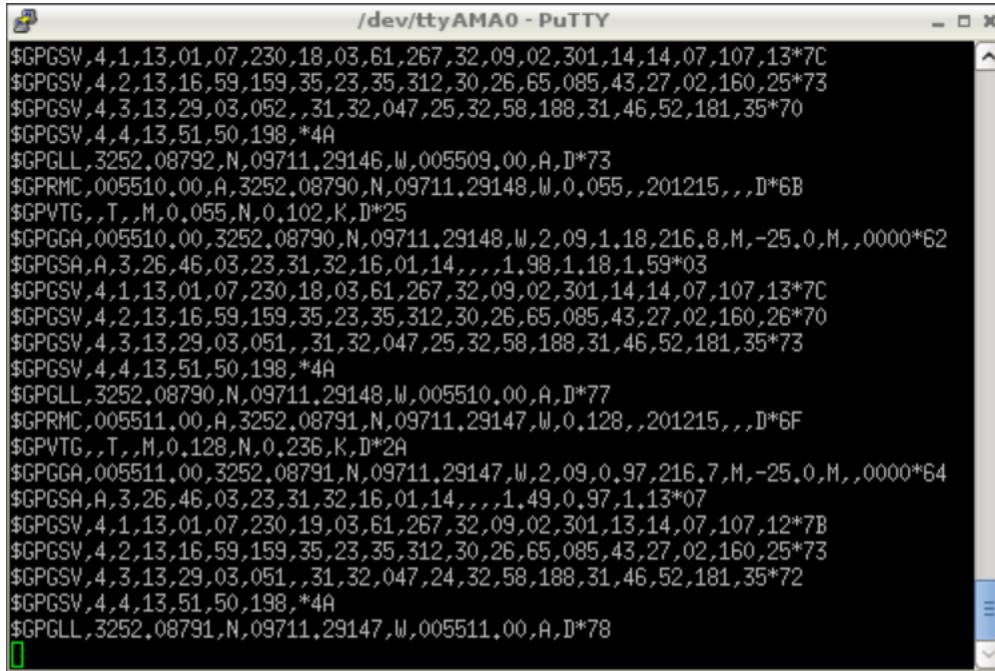
To test the data, you should really be out-doors or otherwise have an un-obstructed view of the sky. Testing in the interior of a building is basically fruitless as the GPS signals will not penetrate and you will have learned nothing.



The device has an LED on the circuit that illuminates (flashes) when a lock has been made.

On a Pi, we can connect the device directly to the UART. Connect the devices TX pin to the Pi RX through a level shifter or voltage divider, the devices RX pin to the Pi TX and the device 5V to the Pi 5V line and the devices GND to GND on the Pi. To see it in operation, stop any `getty` program listening on

/dev/ttyAMA0 and run a terminal emulator such as PuTTY to connect to the serial port. You will immediately start to see the protocol data being sent by the GPS device:



```
$GPGSV,4,1,13,01,07,230,18,03,61,267,32,09,02,301,14,14,07,107,13*7C
$GPGSV,4,2,13,16,59,159,35,23,35,312,30,26,65,085,43,27,02,160,25*73
$GPGSV,4,3,13,29,03,052,,31,32,047,25,32,58,188,31,46,52,181,35*70
$GPGSV,4,4,13,51,50,198,*4A
$GPGLL,3252.08792,N,09711.29146,W,005509.00,A,D*73
$GPRMC,005510.00,A,3252.08790,N,09711.29148,W,0.055,,201215,,,D*6B
$GPVTG,,T,,M,0.055,N,0.102,K,D*25
$GPGGA,005510.00,3252.08790,N,09711.29148,W,2,09,1,18,216,8,M,-25.0,M,,0000*62
$GPGSA,A,3,26,46,03,23,31,32,16,01,14,,,1.98,1.18,1.59*03
$GPGSV,4,1,13,01,07,230,18,03,61,267,32,09,02,301,14,14,07,107,13*7C
$GPGSV,4,2,13,16,59,159,35,23,35,312,30,26,65,085,43,27,02,160,26*70
$GPGSV,4,3,13,29,03,051,,31,32,047,25,32,58,188,31,46,52,181,35*73
$GPGSV,4,4,13,51,50,198,*4A
$GPGLL,3252.08790,N,09711.29148,W,005510.00,A,D*77
$GPRMC,005511.00,A,3252.08791,N,09711.29147,W,0.128,,201215,,,D*6F
$GPVTG,,T,,M,0.128,N,0.236,K,D*2A
$GPGGA,005511.00,3252.08791,N,09711.29147,W,2,09,0,97,216,7,M,-25.0,M,,0000*64
$GPGSA,A,3,26,46,03,23,31,32,16,01,14,,,1.49,0.97,1.13*07
$GPGSV,4,1,13,01,07,230,19,03,61,267,32,09,02,301,13,14,07,107,12*7B
$GPGSV,4,2,13,16,59,159,35,23,35,312,30,26,65,085,43,27,02,160,25*73
$GPGSV,4,3,13,29,03,051,,31,32,047,24,32,58,188,31,46,52,181,35*72
$GPGSV,4,4,13,51,50,198,*4A
$GPGLL,3252.08791,N,09711.29147,W,005511.00,A,D*78
```

See also:

- Interfacing Arduino UNO with GPS module(GY-GPS/NEO6MV2)
- Using and programming a NEO-6 GPS receiver module
- You tube: [Tutorial 15 for Arduino: GPS Tracking](#)
- TinyGPS
- u-center software
- Actisense: NMEA Reader
- gpsd

## The gpsd package

With a GPS device connected to your Pi, it is now time to see what it is saying. There is an excellent set of packages for working with GPS. The packages you want to install are "gpsd" and "gpsd-clients".

Once installed, we can run the most basic of the tools called `gpsmon`:

```
$ sudo gpsmon /dev/ttyAMA0
```

The result of this is a curses based text application that shows a wealth of data related to the GPS information being decoded from the device:

```

pi@raspberrypi: /var/run
File Edit Tabs Help
/dev/ttyAMA0 9600 8N1      NMEA0183>
Time: 2015-12-20T01:15:43.000Z Lat: 32 52' 05.134" N Lon: 97 11' 17.344" W
Cooked PVT
GPRMC GPVTG GPGGA GPGSA GPGSV GPGLL
Sentences
Ch PRN Az El S/N
0 1 224 2 0
1 3 247 59 21
2 9 306 8 19
3 14 113 1 0
4 16 146 69 33
5 23 317 42 34
6 26 65 61 40
7 27 154 7 18
8 29 43 2 0
9 31 53 25 26
10 32 185 47 36
11 133 181 53 34
GSV
(52) $GPGLL,3252.08558,N,09711.28907,W,011543.00,A,D*70\x0d\x0a

```

Ch	PRN	Az	El	S/N
0	1	224	2	0
1	3	247	59	21
2	9	306	8	19
3	14	113	1	0
4	16	146	69	33
5	23	317	42	34
6	26	65	61	40
7	27	154	7	18
8	29	43	2	0
9	31	53	25	26
10	32	185	47	36
11	133	181	53	34

Time: 011543.00  
Latitude: 3252.08558 N  
Longitude: 09711.28907 W  
Speed: 0.022  
Course:  
Status: A FAA: D  
MagVar:  
Mode: A 3  
Sats: 26 46 3 23 31 32 9 16  
DOP: H=1.36 V=2.95 P=3.25  
PPS offset:  
GSA + PPS

Time: 011543.00  
Latitude: 3252.08558  
Longitude: 09711.28907  
Altitude: 186.3  
Quality: 2 Sats: 09  
HDOP: 1.36  
Geoid: -25.0  
GGA

UTC: RMS:  
MAJ: MIN:  
ORI: LAT:  
LON: ALT:  
GST

Core amongst this data are:

- Time – The time value received from the satellites
- Lat – The latitude of the device
- Lon – The longitude of the device

The core though of the `gpsd` package is the `gpsd` demon. This demon program runs on a machine which knows how to interact with a GPS device, reads and parses the data and then makes it available as a higher level protocol.

When we run the `gpsd` demon, we can then connect to the device using `gpsmon` over the network.

Using the `gpspipe` command the protocol from `gpsd` is captured and written to `stdout`.

An alternative to `gpsmon` is `cgps` which displays character data as follows:

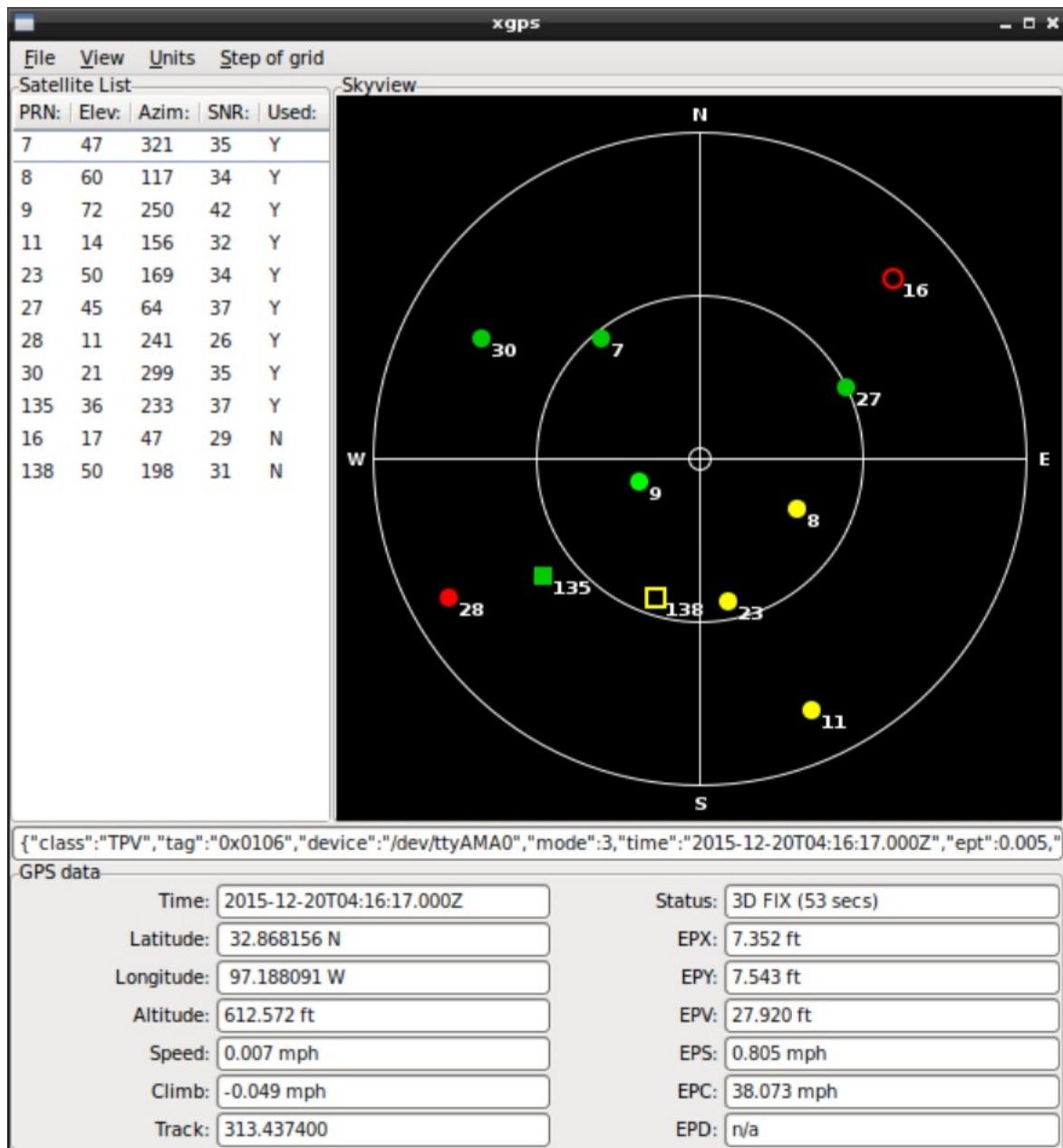
The terminal window displays two tables of data. The left table shows the user's location and status:

Time:	2015-12-20T04:14:43.000Z
Latitude:	32.868167 N
Longitude:	97.188118 W
Altitude:	568.1 ft
Speed:	0.0 mph
Heading:	290.2 deg (true)
Climb:	-12.8 ft/min
Status:	3D FIX (36 secs)
Longitude Err:	+/- 7 ft
Latitude Err:	+/- 7 ft
Altitude Err:	+/- 30 ft
Course Err:	n/a
Speed Err:	+/- 0 mph
Time offset:	0.143
Grid Square:	EM12ju

The right table lists the satellites being tracked:

PRN:	Elev:	Azim:	SNR:	Used:
7	46	320	38	Y
8	68	119	40	Y
9	72	254	41	Y
11	13	156	32	Y
23	51	169	35	Y
27	45	064	38	Y
28	11	241	24	Y
30	20	299	27	Y
135	36	233	38	Y
138	50	198	32	N

One final tool of interest to us is `xgps` which requires an X-Windows server and shows more details on the satellites:



From a programming perspective, there are libraries available for both C and C++ (as well as other languages). These libraries simplify the work we need to perform to get the information we desire. The single most important data type available with this library is called `gps_data_t`. This contains the state and data retrieved from the GPS device. Some of its fields are:

Field	Description
gps_mask_t set	
timestamp_t online	
gps_fix_t fix	
int satellites_used	
int satellites_visible	
int status	

Among these, the `gps_fix_t fix` member contains the data we most commonly want:

Field	Description
timestamp_t time	Time stamp of this fix record
int mode	Mode of the fix: <ul style="list-style-type: none"> <li>• 0 – not acquired</li> <li>• 1 – 2D acquired</li> <li>• 2 – 3D acquired</li> </ul>
double latitude	Fix latitude value
double longitude	Fix longitude value
double altitude	Altitude above sea level
double track	
double speed	Speed
double climb	
double epc	
double epd	
double eps	
double ept	
double epy	
double epv	
double epx	
double epy	

Here is an example of a C application which allows us to retrieve GPS data:

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

int main(int argc, char *argv[]) {
    printf("Starting gps client test\n");
    struct gps_data_t gps_data;
    int ret = gps_open("localhost", "2947", &gps_data);
    printf("ret from gps_open=%d\n", ret);
    if (ret != 0) {
```

```

        return 0;
    }
    gps_stream(&gps_data, WATCH_ENABLE | WATCH_JSON, NULL);
    while(1) {
        if (gps_waiting(&gps_data, 500) != 0) {
            if (gps_read(&gps_data) != -1) {
                printf("fix: %f, %f\n", \
                    gps_data.fix.latitude,
                    gps_data.fix.longitude);
            } else {
                printf("Error\n");
            }
        }
    }
    gps_close(&gps_data);
}

```

See also:

- [gpsd](#)
- main(3) – [libgps](#)
- The architecture of Open Source Applications – [gpsd](#)

## GPS from your cell phone

You are probably aware that your cell phone also has a GPS receiver built into it. This allows your phone to know where it is and do interesting things such as show your location on a map (see Google Maps for example). Can we use your phone's GPS in conjunction with a Pi?

The answer is (of course) yes. We have a number of choices as to how we retrieve GPS data from the phone. The first we will look at is using the browser. All modern browsers are compliant with the HTML5 standard and that supports access to sensor information such as GPS. If we load a web page, that page can contain JavaScript which requests the current GPS location from the browser. This is considered a privileged operation. We don't necessarily want any web app we visit to know our position. As such, the browser opens a pop-up dialog and requires us to explicitly confirm that this is an allowed operation. If we approve, then the data is made available to the page ... and hence to the embedded JavaScript ... and hence onwards to where ever that script may desire to send it. For our purposes, this could be our Raspberry Pi.

Putting this all together, we now have the ability to capture our location and get it to the Pi without needing a specific GPS receiver.

Here is an example HTML page which, when run on a browser on the phone and served up from the Pi, will send the coordinates of the phone back to the Pi.

```

<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<script>

```

```

src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.4/jquery.min.js"></script>
<script>
    navigator.geolocation.getCurrentPosition(function(position) {
        jQuery.ajax({
            url: "http://192.168.1.101:8080/gps", // IP Address of PI REST Server
            data: {
                lat: position.coords.latitude,
                lng: position.coords.longitude
            }
        });
    }, function(error) {
        console.log("Error: %O", error);
    });
    console.log("Waiting for position!\n");
</script>
</head>
<body>
Hello world!!
</body>
</html>

```

It works by using the `navigator.geolocation` object which has a series of functions to allow one to obtain GPS sensor information from within the web page. When a sensor reading has been obtained, the values of the latitude and longitude are sent back to the Pi via a REST request. The JavaScript uses the services of jQuery to make the REST request.

See also:

- MDN – [Using geolocation](#)
- REST requests
- Making REST requests using jQuery

## Video / Webcams

How can we possibly resist attaching a video input source to our Pi?

Many webcams are available very cheaply and are USB attachable. Once attached to your Pi, you should check that it has been recognized. A quick way to do this is to run `lsusb` before plugging in the camera and then re-running it after the camera has been attached. The newly added device will be your camera. For example, on my system:

```

Bus 001 Device 006: ID 18ec:3399 Arkmicro Technologies Inc.
Bus 001 Device 004: ID 0bda:8176 Realtek Semiconductor Corp. RTL8188CUS 802.11n WLAN Adapter
Bus 001 Device 003: ID 0424:ec00 Standard Microsystems Corp. SMSC9512/9514 Fast Ethernet Adapter
Bus 001 Device 002: ID 0424:9514 Standard Microsystems Corp.
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub

```

The first entry (Arkmicro) was the one that was added after attaching the camera.

Once you have located your USB device entry, you can ask for details that the device has made known to Raspbian by running the command:

```
lsusb -v -s <device number>
```

This returns a lot of detailed information and we won't be going into the details of it. However, one can just about see the list of different resolutions and other attributes supported by the camera.

Since we are running on Raspbian and Raspbian is a version of Linux, we can leverage all the work that has been on video processing for Linux. This is known as the "video4linux" interface and manifests itself as devices of the form /dev/video0. An interesting thing to note is that video is considered a securable resource. What we mean by that is that access to your video camera is restricted to only certain users. This makes sense as if your Pi were to be somehow compromised or you gave others access to your Pi then, in principle, they could switch on your camera and watch you. If you were to watch me, it would be deathly boring but you can obviously see that it could be abused. When a video camera is attached, the device has restrictive file permissions:

```
$ ls -l /dev/video0
crw-rw---- 1 root video 81, 0 Jan  9 16:37 /dev/video0
```

The most powerful tool available for working with Webcams is called `fswebcam`. The `fswebcam` package is not part of the standard Raspbian distribution and needs to be installed using:

```
$ sudo apt-get install fswebcam
```

I suggest reading the manual page once installed before using it as it has a ton of options. However, the most basic is:

```
$ fswebcam <fileName>
```

for example:

```
$ fswebcam image1.jpg
```

This will grab an image from the default webcam and write into the file. If this works, you should find a file on your file system. To view it, I recommend installing the powerful package called "imagemagick". This can be installed with:

```
$ sudo apt-get install imagemagick
```

There are a lot of tools associated with the package so read the manual pages as needed.

One of the useful tools is called "display" which displays an image in an X-Windows environment. For example:

```
$ display image1.jpg
```

**Note:** When I tried this, I found that my image was pure black and it took me a long time to figure out the problem. The Webcam I was using has auto-brightness enabled. The way this works is to start sampling frames when the camera starts up and examining the light levels from those. After a few frames have been received, the on-board electronics determines the correct brightness level. However, immediately after start-up (i.e. each time I request an image), the camera hadn't yet seen enough frames to determine the level and hence the black image. The `fswebcam` tool has an option called "`--skip <frame count>`" which ignores the first number of frames before taking the desired snap. For

example, I have to run:

```
$ fswebcam --skip 10 image1.jpg
```

to correctly grab an image.

Once you have validated that the you can grab an image, it is time to move on to some of the other options of `fswebcam`. Specifying the `--resolution <width_x_height>` allows us to declare what resolution we wish to capture. Since not all devices support arbitrary resolutions, determine the allowable sizes. For example, by closely examining the output of `lsusb -v -s <device>` I found that my camera supports:

- 640x480
- 352x288
- 320x240
- 176x144
- 160x120

Some of these pairings seem "odd" to my eyes but I'm sure there are good reasons why they exist.

The image generated by `fswebcam` has a banner by default. This is a text bar at the bottom which includes the date/time the picture was captured. There are a variety of output options we can supply to control the appearance and these can be read about in more detail in the man page. The one that I do want to cover though is `--no-banner` which disables the banner.

Another tool for our consideration is called `motion`. This is quite a catch-all tool for working with video being captured from the device. Its primary ability is to continually watch the video input source and when the image contained in the video changes, perform an action such as capture the changed scene. In effect, this provides a motion capture capability. One can imagine running this application against a webcam near a doorway and only folks arriving or leaving are recorded and not the empty spaces in between. Alternatively, one could record some wildlife prone area and only when an animal enters the scene, will the recording start.

In addition to detecting motion in a scene, the tool allows us to stream video over a network. The frame rate is poor ... maybe only a few frames/second (if that) but it does work and is simple to use.

The tool is configured by a rich and detailed configuration file called `motion.conf`. The order is to look for `motion.conf` in the current directory and then in a directory called `.motion` in the home directory and then at `/etc/motion/motion.conf`. It is likely that this file will have to be edited before you use `motion`. The tool can be started as a demon which puts itself into the background. However it can also be forced to run in the foreground when testing. To run in the foreground, use:

```
$ sudo motion -n
```

Again, this is not supplied with a default Raspbian distribution so we must install it with:

```
$ sudo apt-get install motion
```

Motion can be started as either a demon or in non demon mode. When setting it up, I recommend running in the non-demon mode. This can be controlled by passing in the "-n" flag.

To say that there are a lot of options associated with motion is an understatement. We will see some of these in the following sections.

See also:

- [Using a standard USB webcam](#)
- man(1) – [fswebcam](#)
- ImageMagick – [home page](#)
- Motion – [home page](#)
- man(1) – [motion](#)
- [avconv documentation](#)

## Streaming a webcam video

Almost without exception we will want some technique for streaming webcam video. This means taking the source of the video and sending it live over the network to be shown elsewhere. The motion package has this capability baked into it. The first parameter we will look at is called `stream_port`. This is the TCP/IP port number that motion will listen upon for incoming client requests. A client is a browser that wishes to view the video stream. A value of 0 switches it off.

The next setting is called `stream_maxrate` which is the number of frames/second that should be attempted to be delivered. The default here is 2 ... which will not give a steady stream but will be very jerky. This parameter combines with `framerate` which is the maximum capture rate from the camera. It is also set very low. The stream rate that you actually see will be no better than the smaller of these two parameters. Since the video stream consumes bandwidth, there is usually no purpose in streaming images if nothing is changing. Since motion is configured to be able to detect changes, another parameter called `stream_motion` comes into play. The values of this parameter can be `on` or `off`. When `on`, the video is NOT streamed at more than 1 frame/second when there is no detected motion in the scene. However when motion is detected, video stream starts.

Here are some suggested settings for maximum frame rates:

```
stream_port 8081
stream_maxrate 100
stream_motion off
framerate 100
```

## Recording a webcam image for a period of time

A not uncommon request is to record a webcam image for a period of time. For example, we may have a sensor that acts as a trigger ... for example a proximity detector. Our puzzle then becomes that once we detect a signal, how can we record a video for some duration? The motion tool is all about sensing motion and, as such, it doesn't normally trigger recording until it does sense that motion. However, we

can instruct the tool that even if it doesn't detect motion, it should behave as though it is present. We can use the `detection` setting to achieve this. Setting it to `on` causes motion to believe that there is always motion and it will start recording. If we set it `off`, then it will no longer be emulating motion but real motion will trigger the recording. If we only wish to record when we want to record, then we can disable detection of motion ... this says "Don't watch the video output and look for motion" ... however, if we switch on `emulate_motion` while detection is disabled, then video will be recorded. Is this a hack? Possibly, but it works.

We can use REST requests to control this. Sending:

```
http://192.168.1.101:8080/0/detection/pause
```

pauses detection while:

```
http://192.168.1.101:8080/0/config/set?emulate_motion=on
```

emulates motion and:

```
http://192.168.1.101:8080/0/config/set?emulate_motion=off
```

disables emulation.

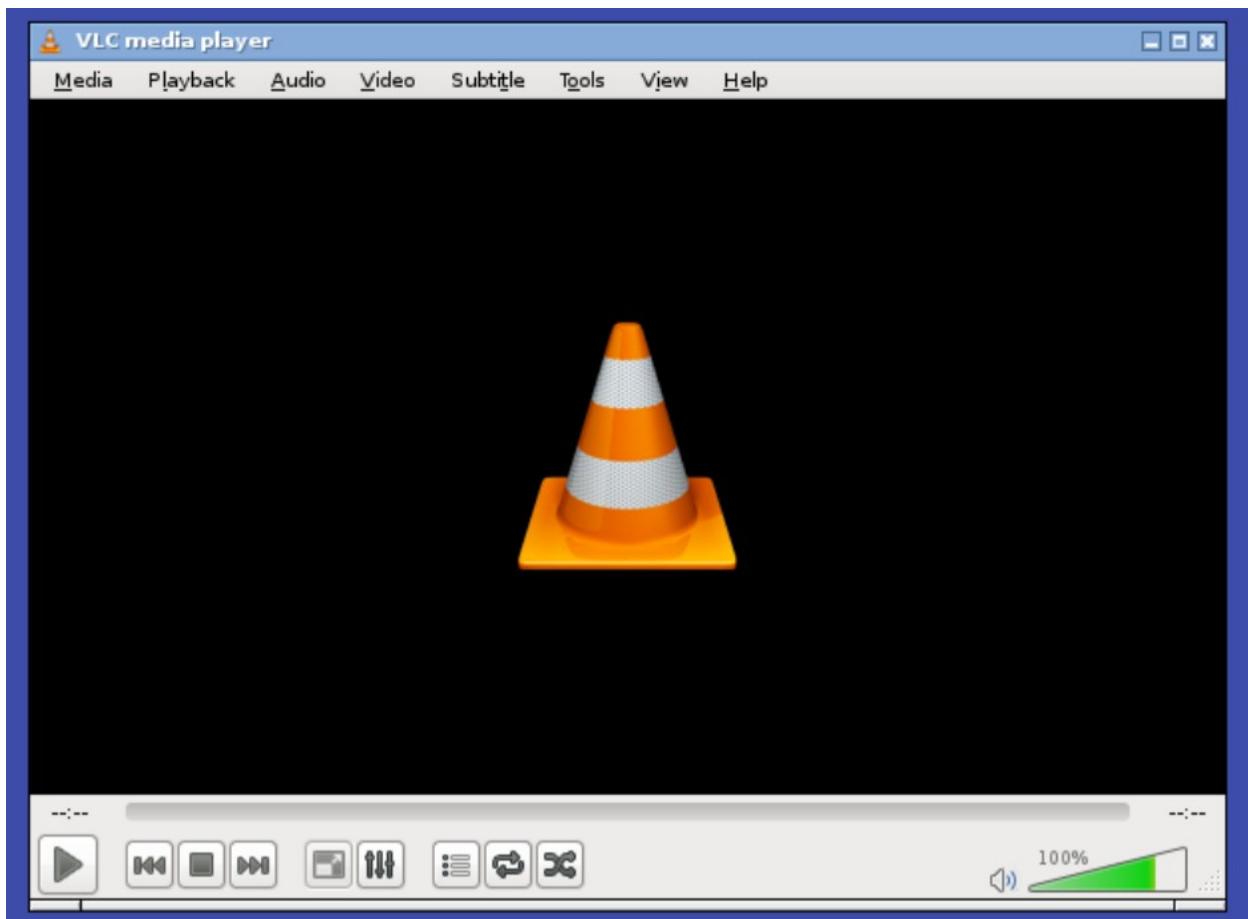
Since REST requests can be sent either based on the CRON scheduling demon or by application logic when a sensor is triggered, we now have the ability to capture video under application control.

## Playing video

Video can be played on a Pi using `omxplayer` ... however this plays directly through the GPU built into the system and hence will drive only the HDMI output and will not work in an X-Windows environment. The "`vlc`" package provides video over X-Windows.

```
$ sudo apt-get install vlc
```

Be aware that this is a **big** package. When launched it provides the ability to view video. The video files can be opened as desired.



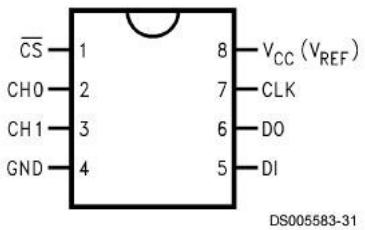
## Analog to Digital conversion - ADC0832

In our digital world, we like to think of signals as either on or off (1 or 0). However, in the real world, a lot of our data is in an analog form. For example, a light dependent resistor changes its conductive properties as the light falling upon it, an audio signal input which might vary its voltage as a function of frequency or the position of a joystick. In order to make use of these analog devices in our digital environment, we need to employ a device called an analog to digital converter. These take as input a voltage signal value and convert it into a digital representation. Some of the devices expose an address bus and some encode the resulting data as a series of digital signals along a single line.

Here we will look at just one of these devices, the ADC0832 IC from Texas Instruments. This is an 8 pin IC that produces an 8 bit output from the analog input. This means that a zero voltage will produce a value of 0 while maximum voltage will produce a value of 255 with graduations of values between these minimum and maximums.

Note that this is a 5V device but appears to be able to operate at 3.3v.

**ADC0832 2-Channel MUX  
Dual-In-Line Package (N)**



**TABLE 7. MUX Addressing: ADC0832  
Differential MUX Mode**

MUX Address		Channel #	
SGL/ DIF	ODD/ SIGN	0	1
0	0	+	-
0	1	-	+

COM internally connected to GND.

V<sub>REF</sub> internally connected to V<sub>CC</sub>.

Top View

A conversion is requested when CS (pin 1) goes from high to low. It should stay low until the reading of the analog signal and its conversion is complete.

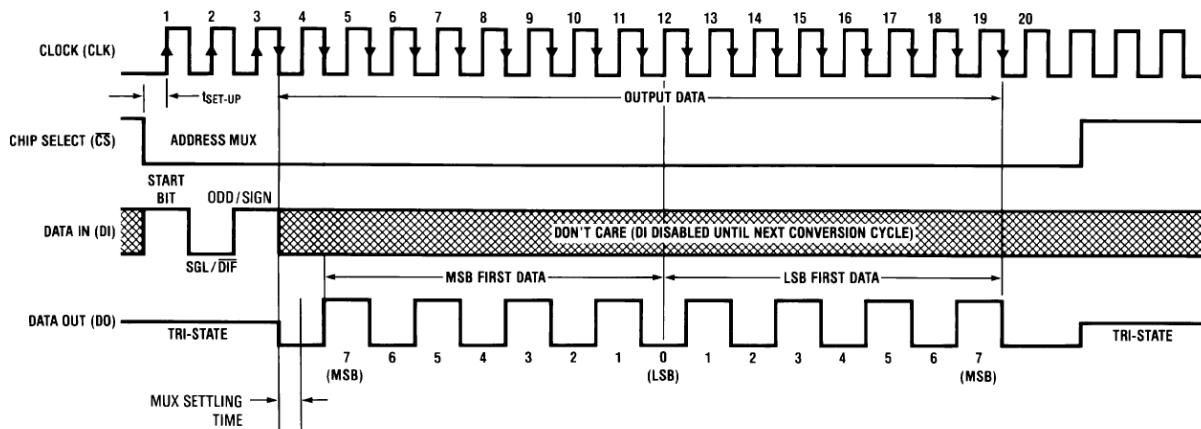
A clock signal starts low, goes high and then back to low. The clock signal should be sent to the CLK (pin 7).

The data in (DI) is used to provide instructions to the IC. Its value is clocked in on the rising edge of the clock signal. Following the start of the conversion, the DI is used to send in three bits.

- First – a Start Bit used as a flag. This should be a 1 value.
- Second – a SGL/DIF selection bit – The IC can work in an absolute or relative mode in terms of analyzing the incoming analog voltage value. The SGL (on) (Single Ended) mode means that the absolute value on the channel will be used while the DIF (off) (Differential) mode will compare the difference between the values on CH0 (pin 2) and CH1 (pin 3).
- Third – an ODD/SIGN selection bit. If we are in SGL mode, the value of this selection chooses between reading from channel 0 (CH0) or channel 1 (CH1). If we are in DIF mode, this selection choose which of the channels should be subtracted from the other (we can't have negative values).

Following this we get 8 bits of analog converted data **twice**. The first 8 bits are MSB first while the second 8 bits are LSB first. The signal data can be found on D0 (pin 5).

The following is reproduce from the data sheet and pictorially illustrates the timing diagrams:



The high level bit banging algorithm is as follows:

```

set CLK mode = Output
set CS mode = Output

// Restart here for future reads
set DIO mode = Output
set CLK = 0
set CS = 0

set DIO = 1 // Start bit
delay
set CLK = 1
delay
set CLK = 0
delay

set DIO = 1 // SGL
set CLK = 1
delay
set CLK = 0
delay

set DIO = 0 // Choose channel 0
set CLK = 1
delay
set DIO mode = Input
set CLK = 0
delay

set CLK = 1
delay
set CLK = 0
delay

loop for 8
  set CLK = 1
  get DIO /// Get a data bit

```

```

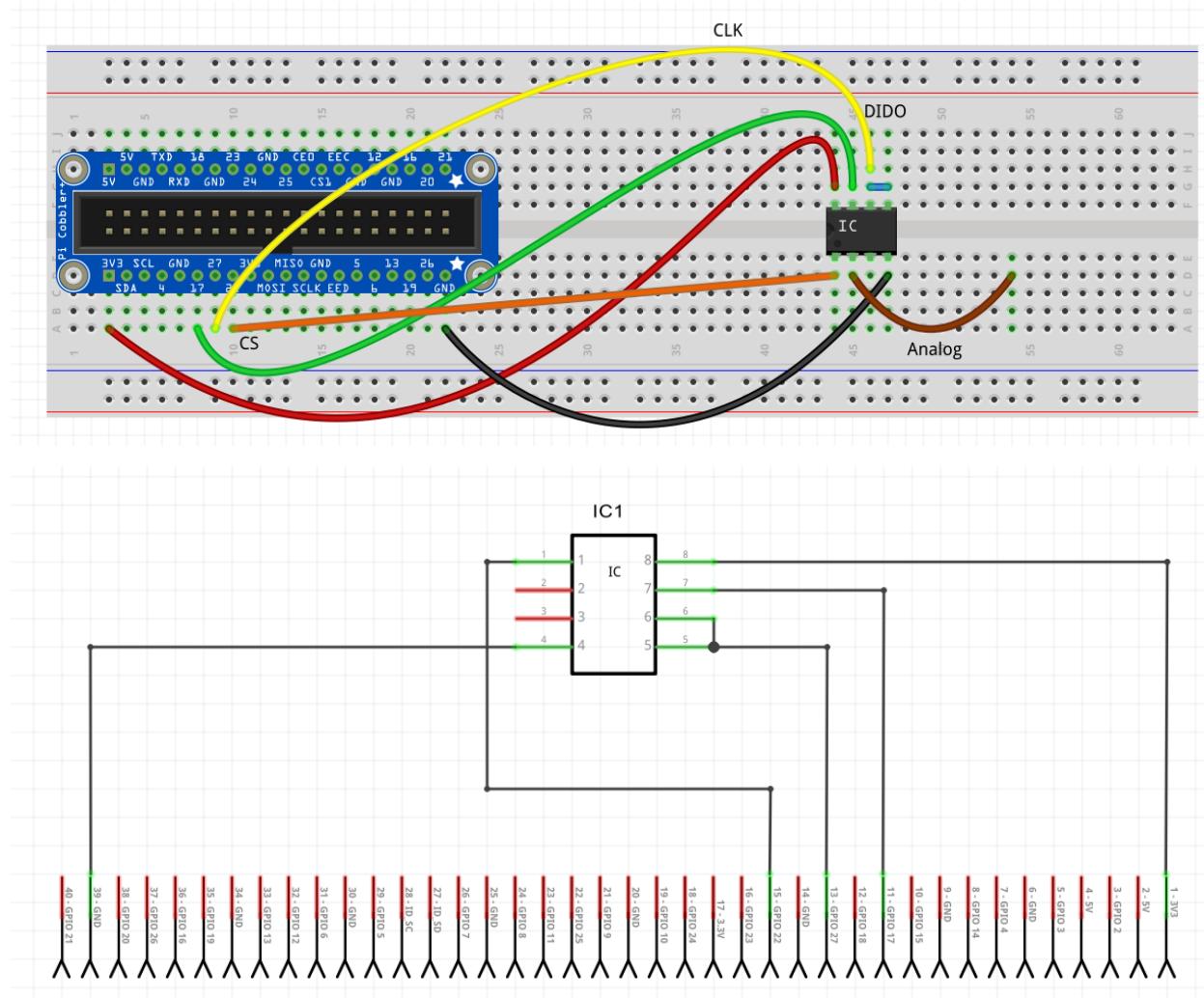
delay
set CLK = 0
delay
end of loop

set CS = 1

```

Alternatives to the ADC0832 may be the MCP3002, ADS1013/1014/1015.

The following shows an illustrative circuit connected to a Pi. The analog voltage could be tested by connecting a 10K potentiometer between high and low and tapping the center for the analog signal. Changing the setting of the potentiometer will change the value read by the Pi.



See also:

- [ADC0832 Home Page](#)
- [Data sheet](#)
- [Sensor monitoring with Raspi](#)
- [Experiment with 8 bit serial AD Convertor ADC0832](#)

## Using the shell

Since this device is not using a well known protocol, we would have to bit-bang it into operation which isn't practical from a shell script. We will ignore using the shell to interact with the device.

## Using WiringPi

The following WiringPi C application illustrates a reader using the ADC. It uses a bit-banging technique to send and receive the correct signals. If we compare the code to the high level algorithm discussed earlier, we see that it matches.

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

#define CLK 17
#define DIO 27
#define CS 22

int getValue() {
    pinMode(CLK, OUTPUT);
    pinMode(CS, OUTPUT);
    pinMode(DIO, OUTPUT);

    digitalWrite(CLK, LOW);
    digitalWrite(CS, LOW);

    digitalWrite(DIO, HIGH); // Start bit
    delay(10);
    digitalWrite(CLK, HIGH);
    delay(10);
    digitalWrite(CLK, LOW);
    delay(10);

    digitalWrite(DIO, HIGH); // SGL
    digitalWrite(CLK, HIGH);
    delay(10);
    digitalWrite(CLK, LOW);
    delay(10);

    digitalWrite(DIO, LOW); // Choose channel 0
    digitalWrite(CLK, HIGH);
    delay(10);
    pinMode(DIO, INPUT);
    digitalWrite(CLK, LOW);
    delay(10);

    digitalWrite(CLK, HIGH); // Skip a clock
    delay(10);
    digitalWrite(CLK, LOW);
    delay(10);
```

```

// Read the value
int value = 0;
int i;
for (i=0; i<8; i++) {
    digitalWrite(CLK, HIGH);
    int bit = digitalRead(DIO);
    value = value << 1 | bit;
    delay(10);
    digitalWrite(CLK, LOW);
    delay(10);
}
digitalWrite(CS, HIGH);
return value;
}

int main(int argc, char *argv[]) {
    printf("Starting ADC test\n");

    int rc = wiringPiSetupGpio();
    if (rc != 0) {
        printf("Failed to wiringPiSetupGpio()\n");
        return 0;
    }
    while(1) {
        int value = getValue();
        printf("value=%d\n", value);
        sleep(2);
    }
}

```

## Using Pi4J

The following Pi4J Java program illustrates working against the ADC from a Java environment. It uses bit banging to achieve the algorithm described previously.

```

package kolban;

import com.pi4j.io.gpio.GpioController;
import com.pi4j.io.gpio.GpioFactory;
import com.pi4j.io.gpio.GpioPinDigitalMultipurpose;
import com.pi4j.io.gpio.GpioPinDigitalOutput;
import com.pi4j.io.gpio.Pin;
import com.pi4j.io.gpio.PinMode;
import com.pi4j.io.gpio.PinState;
import com.pi4j.io.gpio.RaspiPin;

public class Main1 {
    private final Pin CLK_NUM = RaspiPin.GPIO_00;
    private final Pin DIO_NUM = RaspiPin.GPIO_02;
    private final Pin CS_NUM = RaspiPin.GPIO_03;

    private GpioPinDigitalOutput CLK;

```

```

private GpioPinDigitalOutput CS;
GpioPinDigitalMultipurpose DIO;

final GpioController gpio = GpioFactory.getInstance();

public int getValue() {
    try {
        DIO.setMode(PinMode.DIGITAL_OUTPUT);
        CLK.setState(PinState.LOW);
        CS.setState(PinState.LOW);

        DIO.setState(PinState.HIGH); // Start Bit
        Thread.sleep(10);
        CLK.setState(PinState.HIGH);
        Thread.sleep(10);
        CLK.setState(PinState.LOW);
        Thread.sleep(10);

        DIO.setState(PinState.HIGH); // SGL
        CLK.setState(PinState.HIGH);
        Thread.sleep(10);
        CLK.setState(PinState.LOW);
        Thread.sleep(10);

        DIO.setState(PinState.LOW); // Choose channel 0
        CLK.setState(PinState.HIGH);
        Thread.sleep(10);
        DIO.setMode(PinMode.DIGITAL_INPUT);
        CLK.setState(PinState.LOW);
        Thread.sleep(10);

        CLK.setState(PinState.HIGH); // Skip a clock
        Thread.sleep(10);
        CLK.setState(PinState.LOW);
        Thread.sleep(10);

        int value = 0;
        for (int i=0; i<8; i++) {
            CLK.setState(PinState.HIGH);
            int bit = DIO.getState().getValue();
            value = value << 1 | bit;
            Thread.sleep(10);
            CLK.setState(PinState.LOW);
            Thread.sleep(10);
        }
        CS.setState(PinState.HIGH);

        return value;
    } catch (Exception e) {
        e.printStackTrace();
    };
}

```

```

        return 0;
    }

    public void run() {
        CLK = gpio.provisionDigitalOutputPin(CLK_NUM);
        CS = gpio.provisionDigitalOutputPin(CS_NUM);
        DIO = gpio.provisionDigitalMultipurposePin(DIO_NUM, PinMode.DIGITAL_OUTPUT);
        while(true) {
            int value = getValue();
            System.out.println("Value = " + value);
            try {
                Thread.sleep(1000);
            } catch(Exception e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        System.out.println("Running ADC0832");
        new Main1().run();
    }
}

```

## Using Node.js

The following JavaScript implements the ADC algorithm shown earlier.

```

var wpi = require('wiring-pi');

wpi.setup('wpi');

var CLK = 0;
var DIO = 2;
var CS = 3;

function getValue() {
    wpi.pinMode(CLK, wpi.OUTPUT);
    wpi.pinMode(DIO, wpi.OUTPUT);
    wpi.pinMode(CS, wpi.OUTPUT);

    wpi.digitalWrite(CLK, wpi.LOW);
    wpi.digitalWrite(CS, wpi.LOW);

    wpi.digitalWrite(DIO, wpi.HIGH); // Start bit
    wpi.delay(10);
    wpi.digitalWrite(CLK, wpi.HIGH);
    wpi.delay(10);
    wpi.digitalWrite(CLK, wpi.LOW);
    wpi.delay(10);

    wpi.digitalWrite(DIO, wpi.HIGH); // SGL
    wpi.digitalWrite(CLK, wpi.HIGH);
    wpi.delay(10);
    wpi.digitalWrite(CLK, wpi.LOW);
    wpi.delay(10);
}

```

```

wpi.digitalWrite(DIO, wpi.LOW); // Choose channel 0
wpi.digitalWrite(CLK, wpi.HIGH);
wpi.delay(10);
wpi.pinMode(DIO, wpi.INPUT);
wpi.digitalWrite(CLK, wpi.LOW);
wpi.delay(10);

wpi.digitalWrite(CLK, wpi.HIGH); // Skip a clock
wpi.delay(10);
wpi.digitalWrite(CLK, wpi.LOW);
wpi.delay(10);

var value = 0;
for (var i=0; i<8; i++) {
    wpi.digitalWrite(CLK, wpi.HIGH);
    var bit = wpi.digitalRead(DIO);
    value = value << 1 | bit;
    wpi.delay(10);
    wpi.digitalWrite(CLK, wpi.LOW);
    wpi.delay(10);
}

wpi.digitalWrite(CS, wpi.HIGH);
return value;
}

setInterval(function() {
    var value = getValue();
    console.log("Value = " + value);
}, 1000);

```

## Analog to Digital conversion - MCP3208

Another IC that provides analog to digital conversion is MCP3208. This device goes for about \$6 on eBay as compared to about \$2 for the ADC0832. The primary distinction is that it has 12 bits of sampling as compared to 8 bits for the ADC0832. It also has 8 distinct analog input channels as compared to 4.

The device supports SPI communication. Samples can be read at 100ksps (100,000 samples/second). The device has an operating voltage of either 3.3V or 5V.

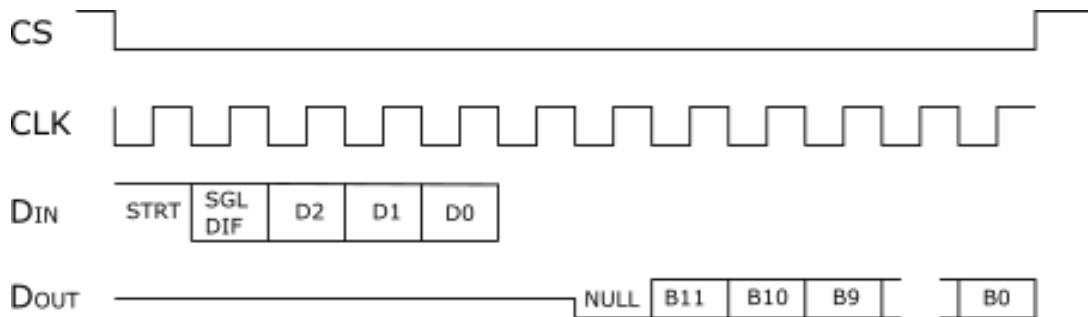
The pin out of the device is:

<b>Pin</b>	<b>Label</b>	<b>Description</b>
1-8	Channels 0-7	Input channels
9	DGND	Digital ground
10	CS/SHDN	Chip select/shutdown input – Must go high between conversations
11	D <sub>IN</sub>	Serial data in
12	D <sub>OUT</sub>	Serial data out
13	CLK	Serial clock
14	AGND	Analog ground
15	V <sub>REF</sub>	Reference voltage input – The reference voltage used by the 12 bit comparator. For example, if set to 3.3V, then the digital output will be between 0 and 4095 for a voltage between 0 and 3.3V on a channel input.
16	V <sub>DD</sub>	3.3V or 5V power

The DIN signal instructs the device which mode to use and which channels to use. The truth table for the operation is as follows:

<b>Single/Diff</b>	<b>D2</b>	<b>D1</b>	<b>D0</b>	<b>Input Configuration</b>	<b>Channel Selection</b>
1	0	0	0	Single	Chan 0
1	0	0	1	Single	Chan 1
1	0	1	0	Single	Chan 2
1	0	1	1	Single	Chan 3
1	1	0	0	Single	Chan 4
1	1	0	1	Single	Chan 5
1	1	1	0	Single	Chan 6
1	1	1	1	Single	Chan 7
0	0	0	0	Differential	Chan 0 = +, Chan 1 = -
0	0	0	1	Differential	Chan 0 = -, Chan 1 = +
0	0	1	0	Differential	Chan 2 = +, Chan 3 = -
0	0	1	1	Differential	Chan 2 = +, Chan 3 = +
0	1	0	0	Differential	Chan 4 = +, Chan 5 = -
0	1	0	1	Differential	Chan 4 = -, Chan 5 = +
0	1	1	0	Differential	Chan 6 = +, Chan 7 = -
0	1	1	1	Differential	Chan 6 = -, Chan 6 = +

When CS is brought low, we must wait 1 clock cycle before sending in the 4 bits of control information. After the 4 bits of control, there are two clock cycles that must be skipped before we read in 12 bits of data. When we see this on a logic diagram, it will make more sense:



Since we are using SPI we can use either a bit banging technique or hardware SPI. Here we will illustrate a bit banging technique. The logic will be as follows:

Start with CS High  
Start with CLK High

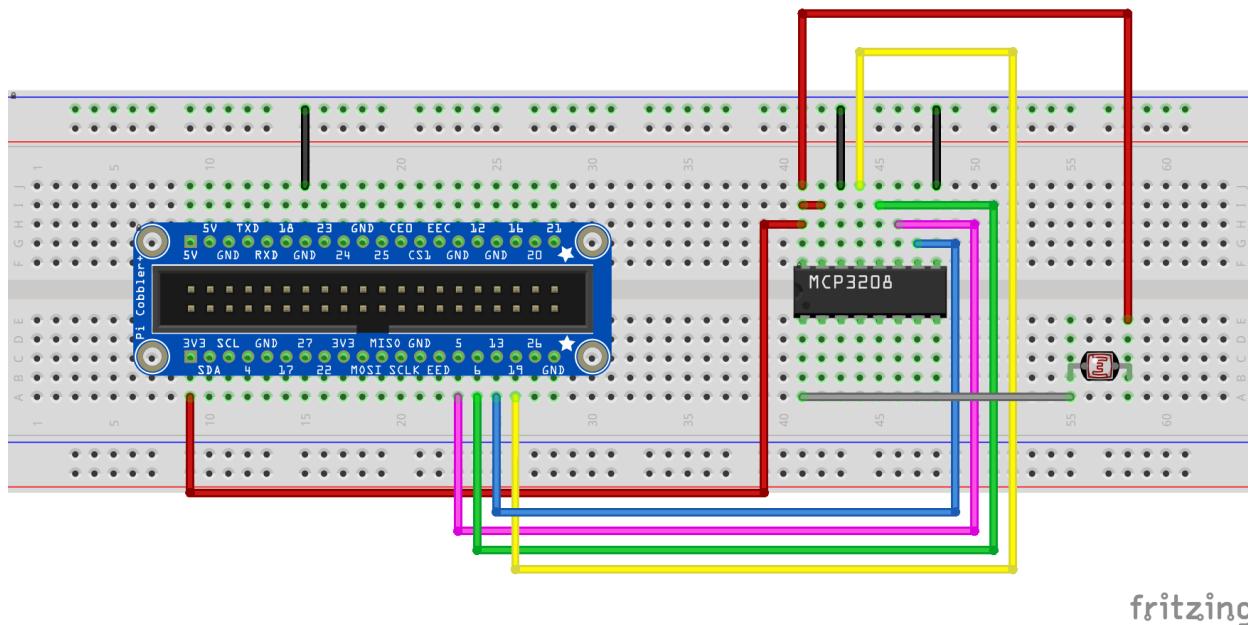
```
// Start
Set CS Low
Set Din High
Set CLK Low
Wait ½ clock
Set CLK High
Wait ½ clock

// Set Single/Differential
Set Din High (Single)
Set CLK Low
Wait ½ clock
Set CLK High
Wait ½ clock
// Set channels
Set Din Low // D2
Set CLK Low
Wait ½ clock
Set CLK High
Wait ½ clock
Set Din Low // D1
Set CLK Low
Wait ½ clock
Set CLK High
Wait ½ clock
Set Din Low // D0
Set CLK Low
Wait ½ clock
Set CLK High
Wait ½ clock
// Skip 2 clock cycles
Set CLK Low
Wait ½ clock
Set CLK High
Wait ½ clock
Set CLK Low
```

```

Wait ½ clock
Set CLK High
Wait ½ clock
// Read 12 bits
for 12 bits
    Set CLK Low
    Wait ½ clock
    Read Dout
    Set CLK High
    Wait ½ clock
// End ...
Set CS High

```



A related IC is the MCP3204 which is a 4 channel version of the same device.

See also:

- [Data sheet](#)

## Using WiringPi

Here is an implementation of using the MCP3208 with WiringPi. It builds on the pseudo algorithm described earlier.

```

#include <stdio.h>
#include <wiringPi.h>

#define DIN  (5) // MCP3208 Input
#define DOUT (6) // MCP3208 Output
#define CS   (13) // MCP3208 CS
#define CLK  (19) // MCP3208 Clock

#define CLK_MICROS (100)

int clockCycle = 0;

```

```

void clock() {
    if (clockCycle == 0) {
        digitalWrite(CLK, LOW);
    } else {
        digitalWrite(CLK, HIGH);
    }
    delayMicroseconds(CLK_MICROS/2);
    clockCycle = !clockCycle;
}

int getValue() {
    // Start
    digitalWrite(CS, LOW);
    digitalWrite(DIN, HIGH);
    clock(); clock();

    digitalWrite(DIN, HIGH); // Single/Differential
    clock(); clock();

    digitalWrite(DIN, LOW); // D2
    clock(); clock();
    digitalWrite(DIN, LOW); // D1
    clock(); clock();
    digitalWrite(DIN, LOW); // D0
    clock(); clock();

    // Skip 2 clock cycles
    clock(); clock();
    clock(); clock();

    int i;
    int value = 0;
    for (i=0; i<12; i++) {
        clock();
        value = (value << 1) | digitalRead(DOUT);
        clock();
    }
    digitalWrite(CS, HIGH);
    return value;
}

int main(int argc, char *argv[]) {
    printf("Starting MCP3208 test\n");
    int rc = wiringPiSetupGpio();
    if (rc != 0) {
        printf("Failed to wiringPiSetupGpio()\n");
        return 0;
    }

    pinMode(DOUT, INPUT);
    pinMode(DIN, OUTPUT);
    pinMode(CS, OUTPUT);
}

```

```

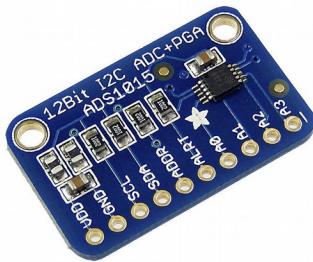
pinMode(CLK, OUTPUT);
digitalWrite(CS, HIGH);
digitalWrite(CLK, HIGH);

while(1) {
    int value = getValue();
    printf("Value = %d\n", value);
    sleep(1);
}
}

```

## Analog to Digital conversion - ADS1015

This is a pricier IC at about \$10 on eBay. It provides a 12 bit resolution for A/D conversion and is controlled by an I2C bus. It supports either 3.3V or 5V supplies. It can support either 4 single-ended channels or 2 differential channels.



The device is extremely power efficient and can operate in a number of power modes. In single-shot mode, it samples the value and makes it available and then goes into a lower power mode. In continuous operation mode, it starts calculating the next sample as soon as the previous has been calculated.

The pin out on the device is:

Pin	Label	Description
1	VDD	3.3V, 5V
2	GND	Ground
3	SCL	I2C Clock
4	SDA	I2C Data
5	ADDR	Address selection
6	ALRT	Digital comparator / Conversion ready
7	A0	Channel 0
8	A1	Channel 1
9	A2	Channel 2
10	A3	Channel 3

The device has four registers at the following addresses:

Address	Register

0	Conversion register [15:4] – Data D11:D0 – 12 bits of data [3:0] – 0 – not used																												
1	Config register:																												
	Bit	Register	Description																										
	15	OS	Operational status: W – 1 – Begin a conversion R – 0 – Currently converting R – 1 – Not currently converting																										
	14:12	MUX	Input multiplexer. Default 000 <table border="1" data-bbox="665 656 1019 1072"> <thead> <tr> <th>Bits</th><th>Positive</th><th>Negative</th></tr> </thead> <tbody> <tr><td>000</td><td>AIN0</td><td>AIN1</td></tr> <tr><td>001</td><td>AIN0</td><td>AIN3</td></tr> <tr><td>010</td><td>AIN1</td><td>AIN3</td></tr> <tr><td>011</td><td>AIN2</td><td>AIN3</td></tr> <tr><td>100</td><td>AIN0</td><td>GND</td></tr> <tr><td>101</td><td>AIN1</td><td>GND</td></tr> <tr><td>110</td><td>AIN2</td><td>GND</td></tr> <tr><td>111</td><td>AIN3</td><td>GND</td></tr> </tbody> </table>	Bits	Positive	Negative	000	AIN0	AIN1	001	AIN0	AIN3	010	AIN1	AIN3	011	AIN2	AIN3	100	AIN0	GND	101	AIN1	GND	110	AIN2	GND	111	AIN3
Bits	Positive	Negative																											
000	AIN0	AIN1																											
001	AIN0	AIN3																											
010	AIN1	AIN3																											
011	AIN2	AIN3																											
100	AIN0	GND																											
101	AIN1	GND																											
110	AIN2	GND																											
111	AIN3	GND																											
11:9	PGA	Programmable gain amplifier. Default 010																											
8	MODE	Mode: 0 – Continuous 1 – Power-down single shot (default)																											
7:5	DR	Data rate. Default 100 (1600sps)																											
4	COMP_MODE	Comparator mode. Default 0																											
3	COMP_POL	Comparitor polarity. Default 0																											
2	COMP_LAT	Latching comparitor. Default 0.																											
1:0	COMP_QUE	Comparitor queue. Default 11																											
2	Lo_threshold register																												
3	Hi_threshold register																												

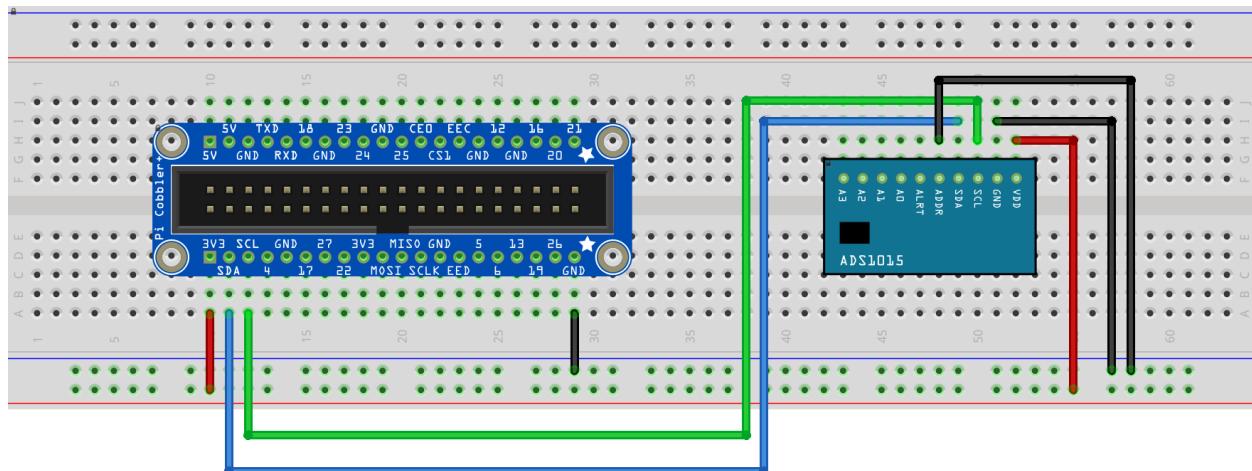
When we send a write command we send 32 bits composed of:

Address	Register	Data high	Data low
---------	----------	-----------	----------

The default address of the device is 0x48 – 0b100 1000. This can be altered by wiring the ADDR pin to different values. What this allows us to do is to have multiple instances of the devices attached to our I2C bus. If we needed more than 4 analog inputs, this would be essential.

Addr Pin	I2C Address
GND	100 1000 (0x48)
VDD	100 1001 (0x49)
SDA	100 1010 (0x4a)
SCL	100 1011 (0x4b)

Here is a typical wiring diagram:



fritzing

See also:

- [ADS1015 home page](#)
- [Datasheet](#)

### Using JavaScript

An existing library for JavaScript is available. See alphacharlie/node-ads1x15 on Github. This has the following methods exposed:

- `readADCSingleEnded`
- `readADCDifferential`
- `readADCDifferential01`
- `readADCDifferential03`
- `readADCDifferential13`
- `readADCDifferential23`
- `startContinuousConversion`
- `stopContinuousConversion`
- `getLastConversionResults`

- startSingleEndedComparator
- startDifferentialComparator
- 

See also:

- Github: [alphacharlie/node-ads1x15](https://github.com/alphacharlie/node-ads1x15)

## Analog to Digital conversion - Arduino

There are many cheap MCUs available with analog to digital conversion already built into them. The Arduino family is one such device. The Arduino devices have 6 or 8 channels (different sources of input) into which a voltage between 0 to 5V can be applied. On reading the value, this is translated into integer values between 0 and 1023. This is 10bits of resolution. Compare that to the ADC0832, which has 8 bits of resolution. The Arduino API to perform the read is called `analogRead()` and takes the pin number to read from as a parameter and returns the integer value. If we couple the `analogRead()` function with one of the techniques to integrate with an Arduino from a Pi (eg. I2C) then we can see that we can obtain a digital value from analog input from an Arduino.

See also:

- Using an Arduino as a peripheral controller

## Joysticks

A joystick is an input device that commonly has two directional input. The joystick can be moved with the fingers or thumb in a forward/back direction and a left/right direction. In addition, many joysticks also have a button press capability that is activated when one pushes vertically down on the joystick.

Mechanically, a joystick is composed of two potentiometers. When pushed forward/back one potentiometer increases or decreases its resistance while when the joystick is pushed left/right a second potentiometer increases/decreases its resistance. At any given time, if we read the values of the two potentiometers, we then know where the user has moved the joystick.

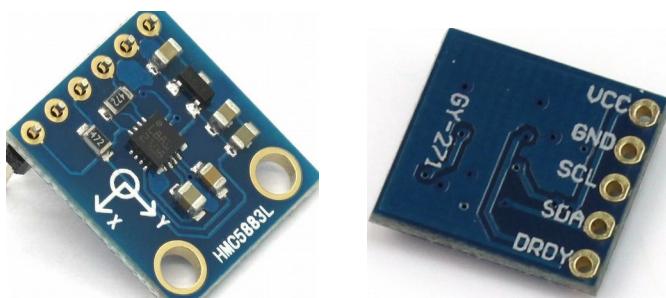


Typically, we combine a joystick with an analog to digital converter with at least two input channels. The ADC0832 is an excellent candidate for that function. One channel will correspond to the X direction and a second channel the Y direction. If we assume that the ADC can return a value from 0-255 then the rest position of the joystick will be approximately 128 in both axis.

## Compass - HMC5883L (aka GY-271)

In the real world, a compass is a device which can be used to determine the direction of magnetic north. There are also electronic components that can perform the same task. One such component is the HMC5883L. Strictly speaking, the device measures the intensity of any magnetic field around it. Since field strength is a vector quantity (has both magnitude and direction), we will be measuring the field strength in the X, Y and Z axis.

This is a 3.3V device.



Its pin configuration is (Left-Right as seen from chip side with connector at top):

Pin	Function
DRDY	Data ready/Interrupt
SDA	I2C Data line
SCL	I2C Clock line
GND	Ground
VCC	+ve voltage – 3.3V

The I2C address of the device is 0x1E. If we run

```
$ i2cdetect -y 1
```

we will see the device as being present:

```
0 1 2 3 4 5 6 7 8 9 a b c d e f
00: -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- 1e --
20: -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- --
```

60: -----  
70: -----

From I<sup>2</sup>C there is a set of registers that can be accessed:

Address	Description	Access
00	Configuration register A	Read / Write
01	Configuration register B	Read / Write
02	Mode register	Read / Write
03	Data output X MSB register	Read
04	Data output X LSB register	Read
05	Data Output Z MSB register	Read
06	Data Output Z LSB register	Read
07	Data output Y MSB register	Read
08	Data output Y LSB register	Read
09	Status register	Read
10	Identification register A	Read
11	Identification register B	Read
12	Identification register C	Read

Think of the device as having a register cursor. We can move the cursor simply by sending the id of the register to move to. For example, sending the device 0x03 will move the cursor to register 3.

Subsequent reads will read from the cursor location **and** auto-increment the cursor. So moving to register 3 and reading 6 bytes will read the next 6 register values [0x03-0x08].

Configuration Register A (register 00) is as follows:

Name	Bits	Description
	7	Reserved. Must be 0.
MA	6:5	Averaged samples per measurement: <ul style="list-style-type: none"> <li>• 00 = 1 (default)</li> <li>• 01 = 2</li> <li>• 10 = 4</li> <li>• 11 = 8</li> </ul>
DO	4:2	Data output rate in measurements/second: <ul style="list-style-type: none"> <li>• 000 = 0.75</li> <li>• 001 = 1.5</li> <li>• 010 = 3</li> <li>• 011 = 7.5</li> <li>• 100 = 15 (default)</li> <li>• 101 = 30</li> <li>• 110 = 75</li> <li>• 111 = Reserved</li> </ul>
MS	1:0	Measurement configuration <ul style="list-style-type: none"> <li>• 00 – Normal (default)</li> <li>• 01 – Positive bias</li> <li>• 10 – Negative bias</li> <li>• 11 – Reserved</li> </ul>

Configuration Register B (register 01) is as follows:

Name	Bits	Description
GN	7:5	Gain values: <ul style="list-style-type: none"> <li>• 000 = 1370</li> <li>• 001 = 1090 (default)</li> <li>• 010 = 820</li> <li>• 011 = 660</li> <li>• 100 = 440</li> <li>• 101 = 390</li> <li>• 110 = 330</li> <li>• 111 = 230</li> </ul>
	4:0	Reserved. Must be 0.

Mode Register (02)

Name	Bits	Description
HS	7	High speed I2C control
	6:2	Reserved. Must be 0.
MD	1:0	Operating mode: <ul style="list-style-type: none"> <li>• 00 = Continuous</li> <li>• 01 = Single measurement (Default)</li> <li>• 10 = Idle mode</li> <li>• 11 = Idle mode</li> </ul>

The three identification registers return the ASCII values 'H', '4' and 'C'.

To create an angle in degrees, the following formula can be used:

```
angle = atan2((double)y, (double)x) * (180 / 3.14159265) + 180; // angle in degrees
```

The driving logic at a high level is:

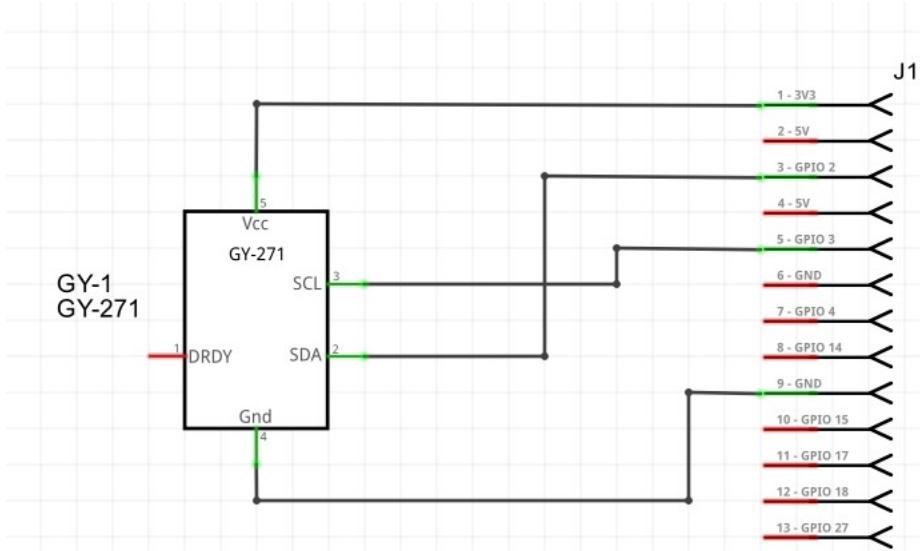
Function	Description
Write 0x02	Select mode register
Write 0x00	Continuous measurement mode
Write 0x03	Select register 3
Request 6 bytes	Read the next 6 bytes
Read 6 bytes	X, Z, Y value pairs MSB/LSB

we combine MSB and LSB of data with:

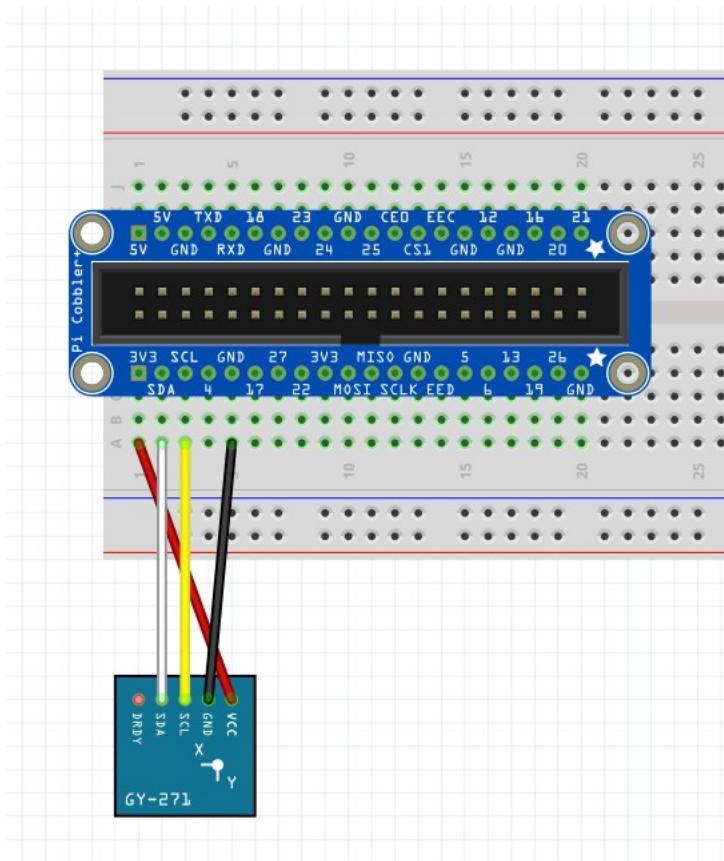
```
data[0] << 8 | data[1];
```

Attachment of the device to a Pi is particularly easy. As you can see it only needs 4 wires to connect.

Here is a schematic of how an instance of the board may be wired to the Pi.



Here is a possible layout on a bread board.



When we use a compass, the result we usually want is "wish way is North". On a real physical compass, we have a needle that points in that direction and we know our answer. Through a digital device, what we should expect to get back is an angle that the device would have to be oriented for it to point North. This assumes that there is a reference mark on the device from which the angle has meaning.

When experimenting with the device, don't bring strong magnets too close to the device as it may become magnetized or otherwise damaged.

See also:

- [Data sheet](#)
- [Sparkfun Tutorial](#)
- Instructables – [Interfacing Digital Compass \(HMC5883L\) with Raspberry Pi 2 using Python3](#)
- [Arduino Nano + GY-271 \(Digital Compass module\) + OLED](#)
- [Electrodragon - HMC5883L - Three-Axis Compass](#)
- YouTube – [Arduino How To: HMC5883L Compass Magnetometer Tutorial](#)
- YouTube – [Use the HMC5883L 3-axis sensor with an Arduino – Tutorial](#)
- YouTube – [Let's build an Arduino electronic Compass using the HMC5883L and a Ring of LEDs - Tutorial](#)
- Applications of Magnetic Sensors for Low Cost Compass Systems
- Application Note: Compass Heading Using Magnetometers
- Wikipedia – [atan2](#)
- Github – [Arduino Library](#)

## Using the shell

Although the raw data can be retrieved using the `i2cget` command it is unlikely that you will be performing any calculations in shell script due to the shell's poor ability to perform math. One might try using the Unix command `bc` (not standard on Raspbian).

## Using WiringPi

We can easily use the WiringPi I2C functions to interact with the device. Access is relatively simple with no complex operations. Once we have retrieved the values, we can perform the math to calculate the angle to magnetic North.

```
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <wiringPi.h>
#include <wiringPiI2C.h>
#include "hmc5883l.h"

#define PI 3.14159265

void checkRC(int rc, char *text) {
    if (rc < 0) {
        printf("Error: %s - %d\n", text, rc);
        exit(-1);
    }
}
```

```

int main(int argc, char *argv[]) {
    printf("HMC5883L starting\n");
    // Setup Wiring Pi
    wiringPiSetup();

    // Open an I2C connection
    int fd = wiringPiI2CSetup(HMC5883L_ADDRESS);
    checkRC(fd, "wiringPiI2CSetup");

    // Perform I2C work
    wiringPiI2CWriteReg8(fd, HMC5883L_REG_MODE, HMC5883L_MODE_CONTINUOUS);

    uint8_t msb = wiringPiI2CReadReg8(fd, HMC5883L_REG_MSB_X);
    uint8_t lsb = wiringPiI2CReadReg8(fd, HMC5883L_REG_LSB_X);
    short x = msb << 8 | lsb;

    msb = wiringPiI2CReadReg8(fd, HMC5883L_REG_MSB_Y);
    lsb = wiringPiI2CReadReg8(fd, HMC5883L_REG_LSB_Y);
    short y = msb << 8 | lsb;

    msb = wiringPiI2CReadReg8(fd, HMC5883L_REG_MSB_Z);
    lsb = wiringPiI2CReadReg8(fd, HMC5883L_REG_LSB_Z);
    short z = msb << 8 | lsb;

    double angle = atan2((double) y, (double)x) * (180 / PI) + 180;

    printf("x=%d, y=%d, z=%d - angle=%f\n", x, y, z, angle);

    return 0;
}

```

See also:

- I2C

## Using Pi4J

The Pi4J project provides a pre-built class called

`com.pi4j.component.gyroscope.honeywell.HMC5883L` that provides an encapsulation to the device. The class has a constructor of:

```
new HMC5883L(I2CBus)
```

calling the object's `readGyro()` reads the immediate values of the sensor which can then be accessed through the properties called X, Y and Z which are instances of `AxisGyroscope` which has a method called `getRawValue()` to retrieve the value.

```

package kolban;

import com.pi4j.component.gyroscope.honeywell.HMC5883L;
import com.pi4j.io.i2c.I2CBus;
import com.pi4j.io.i2c.I2CFactory;

public class Main1 {

```

```

public static void main(String[] args) {
    System.out.println("Running HMC5883L");
    try {
        I2CBus bus = I2CFactory.getInstance(I2CBus.BUS_1);
        HMC5883L hmc5883l = new HMC5883L(bus);
        hmc5883l.enable();
        while(true) {
            hmc5883l.readGyro();
            System.out.println("X=" + (short)hmc5883l.X.getRawValue() + " ,Y=" +
(short)hmc5883l.Y.getRawValue() + ", Z=" + (short)hmc5883l.Z.getRawValue());
            Thread.sleep(1000);
        }
    } catch (Exception e) {
        e.printStackTrace();
    };
}
}

```

See also:

- Github – [source file](#)

## Using Node.js

Similar to the WiringPi example for C, the execution within Node.js is also straight forward. The most interesting parts are the logic to convert two consecutively read bytes into signed 16 bit values corresponding to their desired intent. This can be seen in the function called `toShort()`.

```

var wpi = require('wiring-pi');

wpi.setup('wpi');
var HMC5883L_ADDRESS = 0x1e;

var HCM5883L_REG_CONFIG_A = 0x00;
var HCM5883L_REG_CONFIG_B = 0x01;

var HMC5883L_REG_MODE = 0x02;
var HMC5883L_REG_MSB_X = 0x03;
var HMC5883L_REG_LSB_X = 0x04;
var HMC5883L_REG_MSB_Z = 0x05;
var HMC5883L_REG_LSB_Z = 0x06;
var HMC5883L_REG_MSB_Y = 0x07;
var HMC5883L_REG_LSB_Y = 0x08;
var HMC5883L_REG_STATUS = 0x09;
var HMC5883L_REG_ID_A = 0x0a;
var HMC5883L_REG_ID_B = 0x0b;
var HMC5883L_REG_ID_C = 0x0c;

var HMC5883L_MODE_CONTINUOUS = 0x00;
var HMC5883L_MODE_SINGLE = 0x01;

```

```

var fd = wpi.wiringPiI2CSetup(HMC5883L_ADDRESS);
wpi.wiringPiI2CWriteReg8(fd, HMC5883L_REG_MODE, HMC5883L_MODE_CONTINUOUS);

function toShort(value) {
    if ((value & (1<<15)) == 0) {
        return value;
    }
    return (value & ~ (1<<15)) - (1<<15);
}

setInterval(function() {
    var msb = wpi.wiringPiI2CReadReg8(fd, HMC5883L_REG_MSB_X);
    var lsb = wpi.wiringPiI2CReadReg8(fd, HMC5883L_REG_LSB_X);
    var x = toShort(msb << 8 | lsb);

    msb = wpi.wiringPiI2CReadReg8(fd, HMC5883L_REG_MSB_Y);
    lsb = wpi.wiringPiI2CReadReg8(fd, HMC5883L_REG_LSB_Y);
    var y = toShort(msb << 8 | lsb);

    msb = wpi.wiringPiI2CReadReg8(fd, HMC5883L_REG_MSB_Z);
    lsb = wpi.wiringPiI2CReadReg8(fd, HMC5883L_REG_LSB_Z);
    var z = toShort(msb << 8 | lsb);
    console.log("x=" + x + ", y=" + y + ", z=" + z);
}, 1000);

```

## Accelerometer and Gyroscope - MPU-6050 (aka GY-521)

An accelerometer measures the force of linear acceleration. Simply put, that when an object is at rest (i.e. sitting peacefully on your desk), it is not being accelerated (well ... it actually is but we'll come back to that later). When you start to move the object, it undergoes acceleration. Note that this is not the same as speed. Acceleration is the rate of change in speed over time. When you are sitting as a passenger in your car and the car is moving at a constant speed then if the ride were very smooth and you closed your eyes, you wouldn't know you were moving. However, if the gas or brake are pressed, your car would change speed and you would feel acceleration while the speed changed. You would feel yourself pushed back in your seat when the gas is pressed and you would feel yourself wanting to move forward if the brake is pressed. Similarly, if you stand in an elevator and press a button, you feel yourself being pushed into the floor when it starts to rise and you would find yourself wanting to rise when it comes to a stop.

Acceleration is a vector quantity meaning that it has a directional quality associated with it. In your car (assuming you are driving forward and we call the forward direction  $x$ ) then stepping on the gas produces an acceleration in the  $x$  direction while stepping on the brake produces an acceleration in the  $-x$  direction. Similarly with the elevator, when you rise there will be an acceleration in the  $z$  direction and when you stop, there will be an acceleration in the  $-z$  direction. There is one more twist to the story ... gravity. Although you may not have thought about it, as you sit in your chair there is a force acting on you that wishes to accelerate you. That force is called gravity and the direction of the acceleration is  $-z$ . If it weren't for the chair, your body would be accelerated towards the floor. Even though your body isn't moving, we can consider this an acceleration force being applied. What this means for us is that when a device such as the MPU-6050 is sitting on your desk, it will be reporting that it is accelerating downwards.

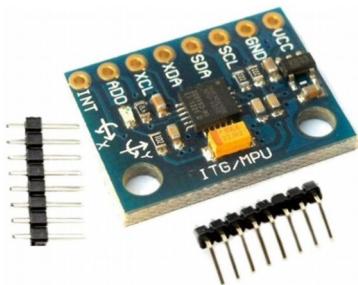
The practical implication of this is very interesting. If the MPU-6050 is flat on the desk, it will report an acceleration in its  $-z$  direction. If we now were to tilt the device, the acceleration due to gravity is still present but now the direction of that force will have changed. The magnitude of the acceleration will remain the same but the measurements will now show it spread over the  $x$ ,  $y$  and  $z$  axes. Working backwards, by examining the values of the acceleration on the  $x$ ,  $y$  and  $z$  axes, we can calculate the orientation of the device relative to the direction of gravity (i.e. relative to up and down).

Having just discussed the concept of measurement of acceleration through the notion of an accelerometer, we can now look at how a gyroscope comes into play. A gyroscope measures the change in velocity (if at all) of a device rotating around its own axis. A gyroscope measures changes in angular velocity.

The MPU-6050 combines these to measure both linear acceleration and angular velocity. The device measures both qualities in the  $x$ ,  $y$  and  $z$  axis and hence is considered a measurement in 6 degrees of freedom (linear acceleration in  $x$ ,  $y$  and  $z$  and angular velocity in  $x$ ,  $y$  and  $z$ ). The measurement values have a 16 bit resolution.

If our goal is to measure the orientation of the device, we may be able to see that the orientation can be found by examining the accelerometer values however these values only result in accurate answers if the device is at rest. If it is moved, then the acceleration due to movement can produce jittery results. If we look at the gyroscope, if we start from a known orientation (eg. flat on the desk), then in principle we should also be able to determine the device's current orientation by adding together all the gyroscopic changes that have happened to it since its original known (calibrated) position. Unfortunately, both of these techniques introduce errors. Using the accelerometer, we can determine our orientation by averaging values over time to remove jitter. As such, it gives good values over time but poor for short term measurement. The gyroscope gives us good angular momentum values in the short term but the errors accumulate over time when we try and calculate the orientation of the device from its base state by successive addition of delta values. Fortunately, we can combine these two techniques through a variety of algorithms to give us a good value that is built from the combination of the two measurements.

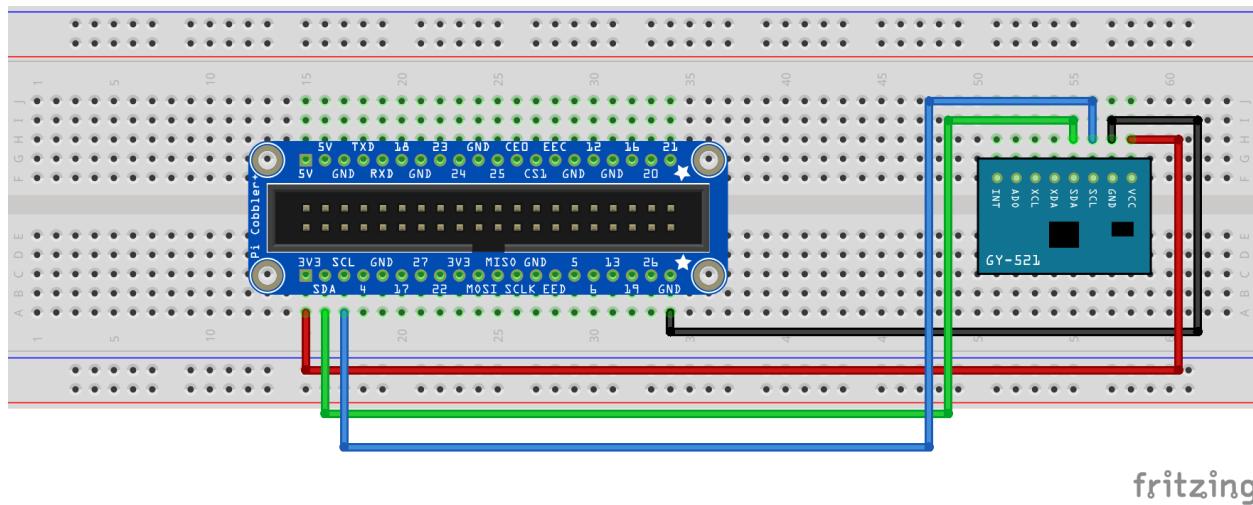
The MPU-6050 device connects via an I<sup>2</sup>C bus (default address is 0x68 but can be changed to 0x69). This is 3.3V device and hence is safe to connect to a Pi's pins directly.



The device has the following pins exposed from the breakout boards:

Pin	Description
VCC	3.3V
GND	Ground
SCL	I2C Clock
SDA	I2C Data
XDA	Auxiliary I2C Data
XCL	Auxiliary I2C Clock
ADO	I2C address selection, either 0x68 or 0x69. Low = 0x68, high = 0x69.
INT	

A sample wiring of the device to a Pi looks as follows:



fritzing

Register	Offset	Name	Description
0x3B	0	ACCEL_XOUT_H	AccelX High
0x3C	1	ACCEL_XOUT_L	AccelX Low
0x3D	2	ACCEL_YOUT_H	AccelY High
0x3E	3	ACCEL_YOUT_L	AccelY Low
0x3F	4	ACCEL_ZOUT_H	AccelZ High
0x40	5	ACCEL_ZOUT_L	AccelZ Low
0x41	6	TEMP_OUT_H	Temp High
0x42	7	TEMP_OUT_L	Temp Low
0x43	8	GYRO_XOUT_H	GyroX High
0x44	9	GYRO_XOUT_L	GyroX Low
0x45	10	GYRO_YOUT_H	GyroY High
0x46	11	GYRO_YOUT_L	GyroY Low
0x47	12	GYRO_ZOUT_H	GyroZ High
0x48	13	GYRO_ZOUT_L	GyroZ Low

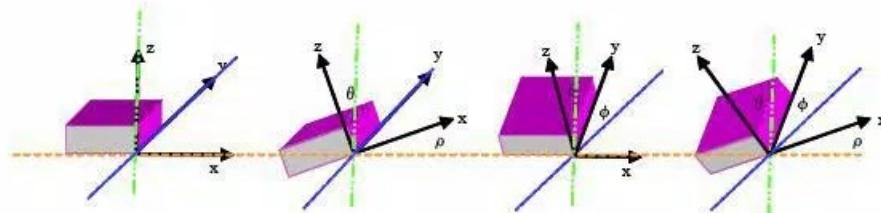
You will sometimes see these devices referred to as Inertial Measurement Units (IMUs).

The value returned from the accelerometer is a raw value. The actual result we want is measured in gravities (g) which is about 9.81 m/s/s. The device has various sensitivities built in. To get to the real value, we need to divide by a scaling factor as show below:

Sensitivity	Factor
2g	16384
4g	8192
8g	4096
16g	2048

Similarly, there is an angular velocity scaling:

Sensitivity	Factor
250 o/s	131
500 o/s	65.5
1000 o/s	32.8
2000 o/s	16.4



**Figure 8. Three Axis for Measuring Tilt**

$$\rho = \arctan\left(\frac{A_x}{\sqrt{A_y^2 + A_z^2}}\right)$$

$$\phi = \arctan\left(\frac{A_y}{\sqrt{A_x^2 + A_z^2}}\right)$$

$$\theta = \arctan\left(\frac{\sqrt{A_x^2 + A_y^2}}{A_z}\right)$$

Technical task ... send in accel values X, Y and Z

See also:

- [MPU-6050 Data Sheet](#)
- [MPU6050 Register Map and Descriptions](#)
- [MPU-6050 Accelerometer + Gyro](#)
- [I2Cdevlib – MPU-6050](#)
- [Interfacing Raspberry Pi and MPU-6050](#)
- [Filters](#)
- YouTube: [MPU6050+Raspberry Pi 2](#)
- [InvenSense – MPU-6050 Home Page](#)
- [Gyroscopes and Accelerometers on a Chip](#)
- [Using an accelerometer for inclination sensing](#)
- YouTube: [MPU-6050 Data with a Complementary Filter](#)

## Using the shell

Although the raw data can be retrieved using the `i2cget` command it is unlikely that you will be performing any calculations in shell script due to the shell's poor ability to perform math. One might try using the Unix command `bc` (not standard on Raspbian).

## Using WiringPi

Using the WiringPi library we can leverage the I2C interface to access the device.

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

#define MPU6050_ADDRESS (0x68)
#define MPU6050_REG_PWR_MGMT_1 (0x6b)
#define MPU6050_REG_DATA_START (0x3b)

void checkRC(int rc, char *text) {
    if (rc < 0) {
        printf("Error: %s - %d\n");
        exit(-1);
    }
}

int main(int argc, char *argv[]) {
    printf("MPU6050 starting\n");
    // Setup Wiring Pi
    wiringPiSetup();

    // Open an I2C connection
    int fd = wiringPiI2CSetup(MPU6050_ADDRESS);
    checkRC(fd, "wiringPiI2CSetup");

    // Perform I2C work
    wiringPiI2CWriteReg8(fd, MPU6050_REG_PWR_MGMT_1, 0);
    while(1) {
        uint8_t msb = wiringPiI2CReadReg8(fd, MPU6050_REG_DATA_START);
        uint8_t lsb = wiringPiI2CReadReg8(fd, MPU6050_REG_DATA_START+1);
        short accelX = msb << 8 | lsb;

        msb = wiringPiI2CReadReg8(fd, MPU6050_REG_DATA_START+2);
        lsb = wiringPiI2CReadReg8(fd, MPU6050_REG_DATA_START+3);
        short accelY = msb << 8 | lsb;

        msb = wiringPiI2CReadReg8(fd, MPU6050_REG_DATA_START+4);
        lsb = wiringPiI2CReadReg8(fd, MPU6050_REG_DATA_START+5);
        short accelZ = msb << 8 | lsb;
    }
}
```

```

msb = wiringPiI2CReadReg8(fd, MPU6050_REG_DATA_START+6);
lsb = wiringPiI2CReadReg8(fd, MPU6050_REG_DATA_START+7);
short temp = msb << 8 | lsb;

msb = wiringPiI2CReadReg8(fd, MPU6050_REG_DATA_START+8);
lsb = wiringPiI2CReadReg8(fd, MPU6050_REG_DATA_START+9);
short gyroX = msb << 8 | lsb;

msb = wiringPiI2CReadReg8(fd, MPU6050_REG_DATA_START+10);
lsb = wiringPiI2CReadReg8(fd, MPU6050_REG_DATA_START+11);
short gyroY = msb << 8 | lsb;

msb = wiringPiI2CReadReg8(fd, MPU6050_REG_DATA_START+12);
lsb = wiringPiI2CReadReg8(fd, MPU6050_REG_DATA_START+13);
short gyroZ = msb << 8 | lsb;

printf("accelX=%d, accelY=%d, accelZ=%d, gyroX=%d, gyroY=%d, gyroZ=%d temp=%d\n",
      accelX, accelY, accelZ,
      gyroX, gyroY, gyroZ,
      temp/340.0 + 36.53);

sleep(1);
} // End of loop
} // End of main

```

## Using JavaScript

Again, using JavaScript is simply a mapping from the C WiringPi functions to the JavaScript library. Note the bit twiddling because we don't have a "signed short" data type natively in JavaScript.

```

var wpi = require('wiring-pi');
wpi.setup('wpi');

var MPU6050_ADDRESS = (0x68)
var MPU6050_REG_PWR_MGMT_1 = (0x6b)
var MPU6050_REG_DATA_START = (0x3b)
var fd = wpi.wiringPiI2CSetup(MPU6050_ADDRESS);
wpi.wiringPiI2CWriteReg8(fd, MPU6050_REG_PWR_MGMT_1, 0);

while(true) {
  var msb = wpi.wiringPiI2CReadReg8(fd, MPU6050_REG_DATA_START);
  var lsb = wpi.wiringPiI2CReadReg8(fd, MPU6050_REG_DATA_START+1);
  var accelX = msb << 8 | lsb;
  if (accelX > 0x8000) {
    accelX = accelX - 0x10000;
  }

  var msb = wpi.wiringPiI2CReadReg8(fd, MPU6050_REG_DATA_START+2);
  var lsb = wpi.wiringPiI2CReadReg8(fd, MPU6050_REG_DATA_START+3);
  var accelY = msb << 8 | lsb;
  if (accelY > 0x8000) {
    accelY = accelY - 0x10000;
  }
}

```

```

var msb = wpi.wiringPiI2CReadReg8(fd, MPU6050_REG_DATA_START+4);
var lsb = wpi.wiringPiI2CReadReg8(fd, MPU6050_REG_DATA_START+5);
var accelZ = msb << 8 | lsb;
if (accelZ > 0x8000) {
    accelZ = accelZ - 0x10000;
}

var msb = wpi.wiringPiI2CReadReg8(fd, MPU6050_REG_DATA_START+8);
var lsb = wpi.wiringPiI2CReadReg8(fd, MPU6050_REG_DATA_START+9);
var angX = msb << 8 | lsb;
if (angX > 0x8000) {
    angX = angX - 0x10000;
}

var msb = wpi.wiringPiI2CReadReg8(fd, MPU6050_REG_DATA_START+10);
var lsb = wpi.wiringPiI2CReadReg8(fd, MPU6050_REG_DATA_START+11);
var angY = msb << 8 | lsb;
if (angY > 0x8000) {
    angY = angY - 0x10000;
}

var msb = wpi.wiringPiI2CReadReg8(fd, MPU6050_REG_DATA_START+12);
var lsb = wpi.wiringPiI2CReadReg8(fd, MPU6050_REG_DATA_START+13);
var angZ = msb << 8 | lsb;
if (angZ > 0x8000) {
    angZ = angZ - 0x10000;
}

console.log("AccelX=" + accelX + ", AccelY=" + accelY + ", AccelZ=" + accelZ);
console.log("AngX=" + angX + " ,AngY=" + angY + " , AngZ=" + angZ);

wpi.delay(1000);
}

```

## Using Arduino libraries

The Arduino libraries can also be used to take an existing Arduino sketch and run it on the Pi.

```

// MPU-6050 Short Example Sketch
// By Arduino User JohnChi
// August 17, 2014
// Public Domain
#include<Wire.h>
const int MPU_addr=0x68; // I2C address of the MPU-6050
int16_t AcX,AcY,AcZ,Tmp,GyX,GyY,GyZ;
void setup(){
    Wire.begin();
    Wire.beginTransmission(MPU_addr);
    Wire.write(0x6B); // PWR_MGMT_1 register
    Wire.write(0); // set to zero (wakes up the MPU-6050)

```

```

    Wire.endTransmission(true);
}
void loop() {
    Wire.beginTransmission(MPU_addr);
    Wire.write(0x3B); // starting with register 0x3B (ACCEL_XOUT_H)
    Wire.endTransmission(false);
    Wire.requestFrom(MPU_addr,14,true); // request a total of 14 registers
    AcX=Wire.read()<<8|Wire.read(); // 0x3B (ACCEL_XOUT_H) & 0x3C (ACCEL_XOUT_L)
    AcY=Wire.read()<<8|Wire.read(); // 0x3D (ACCEL_YOUT_H) & 0x3E (ACCEL_YOUT_L)
    AcZ=Wire.read()<<8|Wire.read(); // 0x3F (ACCEL_ZOUT_H) & 0x40 (ACCEL_ZOUT_L)
    Tmp=Wire.read()<<8|Wire.read(); // 0x41 (TEMP_OUT_H) & 0x42 (TEMP_OUT_L)
    GyX=Wire.read()<<8|Wire.read(); // 0x43 (GYRO_XOUT_H) & 0x44 (GYRO_XOUT_L)
    GyY=Wire.read()<<8|Wire.read(); // 0x45 (GYRO_YOUT_H) & 0x46 (GYRO_YOUT_L)
    GyZ=Wire.read()<<8|Wire.read(); // 0x47 (GYRO_ZOUT_H) & 0x48 (GYRO_ZOUT_L)
    Console.print("AcX = "); Console.print(AcX);
    Console.print(" | AcY = "); Console.print(AcY);
    Console.print(" | AcZ = "); Console.print(AcZ);
    Console.print(" | Tmp = "); Console.print(Tmp/340.00+36.53); //equation for
temperature in degrees C from datasheet
    Console.print(" | GyX = "); Console.print(GyX);
    Console.print(" | GyY = "); Console.print(GyY);
    Console.print(" | GyZ = "); Console.println(GyZ);
    delay(333);
}

```

See also:

- [MPU-6050 Accelerometer + Gyro](#)

## Motion detectors - Passive Infrared Sensor

When you walk towards the door of your local large grocery store, you probably find that the doors slide open by themselves. There has to be some smarts somewhere that detects your presence and knows to operate the motors that slide the doors open. This device is called a Passive Infrared Sensor and is one of the simplest devices to use. They can be found on eBay for about \$2.00 or less.



Their wiring could not be simpler. They have three pins labeled +5v, GND and Out. When motion is detected within the sensor range of the device, the output pin transitions from low to high and stays there until a configurable delay time says that is desirable to go low again.

There isn't much more to say about them than this. Because they have no input requirements other than power, and the output can be voltage divided from 5V to 3.3V for the Pi, all we need do is choose a GPIO

pin for input and wire it up. We can then poll or trigger on a value change on the pin and we will have learned that there is movement within the range of the device.

See also:

- Wikipedia – [Passive infrared sensor](#)

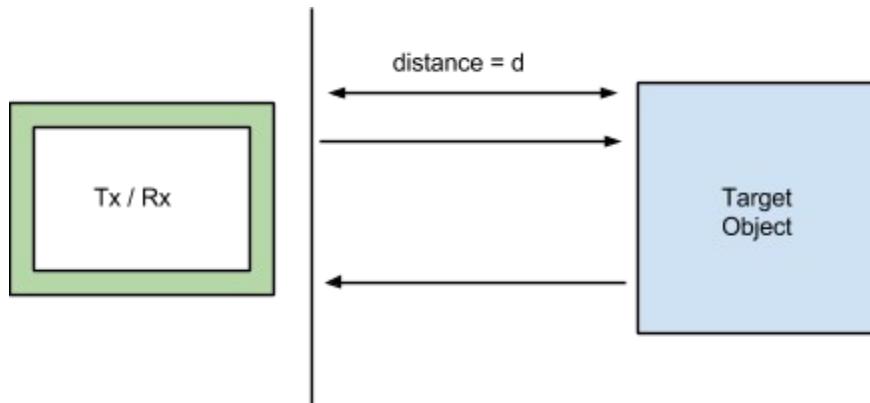
## Ultrasonics - HC-SR04

Imagine that we wish the Pi to discover the distance to an object, how might we go about achieving that? One way is through the use of an ultrasonic transmitter / receiver. These devices are very cheap and also very easy to use. The principle of operation is very simple. Even though when someone speaks to you, it feels like their voice reaches you instantly, this isn't actually the case. Sound moves through the air at a relatively slow rate (relative to light for example). The approximate speed of sound is 340 meters/second. Thus if you were 340 meters away, a loud sound would take 1 second to reach you. If you were 680 meters away, the sound would take two seconds to reach you. You probably recognize that this is what happens in a thunder storm. If you see a lightning flash, it can be many seconds before you hear the thunder. This is because light travels **very** fast but sound travels much slower.

The HC-SR04 device emits a pulse of sound but at such a high frequency, our ears can't hear it. When the sound is emitted it travels outwards away from the device. If there is an object in its path, some of that sound is reflected back the way it came ... back towards the HC-SR04. If the HC-SR04 receives an echo from a previous pulse, it indicates that on an output pin.

The HC-SR04 has a high resolution timer within it that starts when the output is transmitted and stops when an echo is received. The duration of the time between transmission and echo response is output to us. We now have learned the duration of time between when a sound was emitted and the echo received. Since we know the speed of sound and we know a time period, we now know how far the sound wave traveled. Since the distance it traveled is the distance to a remote object and back again, if we halve the distance ... we have learned the distance to that object.

Speed of sound is 340.29 m/s ( $340.29 * 39.3701$  inches/sec). Call this  $V_{\text{sound}}$ .



If  $T_{\text{echo}}$  is the time for echo response then:

$$d = \frac{(T_{echo} \times V_{sound})}{2}$$

Also the equation for expected  $T_{echo}$  lengths is given by:

$$T_{echo} = \frac{2 \times d}{V_{sound}}$$

For example:

Distance	Time
1cm	$2 * 0.01 / 340 = 0.058 \text{ msec} = 59 \text{ usecs}$
10cm	$2 * 0.1 / 340 = 0.59 \text{ msec} = 590 \text{ usecs}$
1m	$2 * 1 / 340 = 5.9 \text{ msec} = 5900 \text{ usecs (5.9 msec)}$

There are some important limitations in using an ultrasonic distance sensor. The distance is calculated by reflecting a sound wave off a remote object and receiving the response. This works great if the target object is perpendicular to the source of the sound. However, if the target object is at an angle to the source then the results can be either off or the target may simply not be detected. To see why this is, consider a wall at an angle greater than  $45^\circ$  to the source. When the sound wave strikes the wall, rather than be reflected back to the source, it will be reflected off somewhere else. If we remember back to a bouncing object from high-school, the angle of incidence equals the angle of reflection. So a perfect response will be found when the sound wave strikes the object at exactly  $90^\circ$ . Anything else and we have introduced either error or a missed response.



The HC-SR04 is a 4 pin device

- $V_{CC}$  – The input voltage is 5V.
- Trig – Pulse to trigger a transmission ... minimum of 10 microseconds.
- Echo – Pulses low to high to low when an echo is received. Warning, this is a 5V output.
- Gnd – Ground.

Send a minimum of a 10 microseconds pulse to the Trig pin (low to high to low). Later, the Echo will go low/high/low. The time that Echo is high is the time it takes the sonic pulse to reach a back-end and bounce back.

Because the Echo response is a 5V signal, it is vital to reduce this to 3.3V for the Pi. A voltage divider will work.

See also:

- MagPi #31 – [Parking Sensors](#)
- [Datasheet](#)
- YouTube – [Ultrasonic Sensor with the Raspberry Pi](#)
- YouTube: [Arduino Tutorial: Ultrasonic Sensor HC SR04 distance meter with a Nokia 5110 LCD display](#)

## Using the shell

Because we are measuring how long a pin is high for and the chances are that we will be measuring in durations less than 1 millisecond, shell commands are just too slow to perform this accurately.

## Using WiringPi

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

#define TRIG (17)
#define ECHO (27)

int main(int argc, char *argv[]) {
    printf("Starting HC-SR04 test\n");
    int rc = wiringPiSetupGpio();
    if (rc != 0) {
        printf("Failed to wiringPiSetupGpio()\n");
        return 0;
    }

    int value = HIGH;

    pinMode(TRIG, OUTPUT);
    pinMode(ECHO, INPUT);
    digitalWrite(TRIG, LOW);

    delay(50);

    while(1) {
        digitalWrite(TRIG, HIGH);
        delayMicroseconds(10);
        digitalWrite(TRIG, LOW);
        unsigned int echoStart = millis();
        while(digitalRead(ECHO) == LOW && millis()-echoStart < 1000) {
            // do nothing
        }
        if (millis()-echoStart < 1000) {
            // Mark start
            unsigned int start = micros();
            while(digitalRead(ECHO) == HIGH) {
                // do nothing
            }
            // Mark end
            unsigned int end = micros();
            unsigned int delta = end-start;

            double distance = 34029 * delta / 2000000.0;
            printf("Distance: %f\n", distance);
        }
        sleep(1000);
    } // End while(1)
}
```

## Using Pi4J

Although Java is a great language, it is much slower than compiled C. Unfortunately, plain Java doesn't have the resolution to measure time ranges sub millisecond.

## Using Node.js

Similar to Java, it is not expected that JavaScript, which is an interpreted language, will be fast enough to natively measure time intervals that are sub-microsecond.

## Audio using the ICSH030A

The ICSH030A is a module from ICStation that incorporates storage for audio and the ability to play it back. There can be four tracks of audio installed in the memory in either 8bit or 16bit PCM format. The audio can be installed via a UART adapter from a supplied application called `ICWaveDownload`. The module has 12 pins:

Pin	Label	Description
1	VCC	5v
2	D4 / TXD	Channel 4 / TXD
3	D3 / RXD	Channel 3 / RXD
4	D2	Channel 2
5	D1	Channel 1
6	GND	Ground
7	VO-	Volume down
8	VO+	Volume up
9	OUT	Audio
10	NC	Not used
11	BUSY	High when playing
12	MODE	Mode selection <ul style="list-style-type: none"><li>• High – play mode</li><li>• Low – record mode</li></ul>

The MODE pin should be High for normal play back and Low for recording. Bringing one of D1-D4 Low will play the track.

The logic to use this device is trivial ... once it is loaded with audio data, simply pulse the appropriate data line (D1, D2, D3 or D4) and the audio will play.

See also:

- [Datasheet and ICWaveDownload](#)

## Buzzers and Piezos

A buzzer can be acquired very easily. This is commonly a piezo crystal that oscillates when a voltage is applied. We can combine this with a transistor as a switch because it is unlikely that we can drive the buzzer directly from the output of a GPIO. The buzzer can be activate or deactivated by sending a GPIO

high or low. Here is an example of a JavaScript program that listens for an incoming web request and, should it arrive, causes a buzzer to beep 3 times.

```
var AUDIO_PIN = 21;

var http = require('http');
var wpi = require('wiring-pi');
wpi.setup('gpio');
wpi.pinMode(AUDIO_PIN, wpi.OUTPUT);
wpi.digitalWrite(AUDIO_PIN, wpi.LOW);

const PORT=8080;

function handleRequest(request, response){
  response.end('Client request arrived: ' + request.url);
  audioSos();
}

var server = http.createServer(handleRequest);

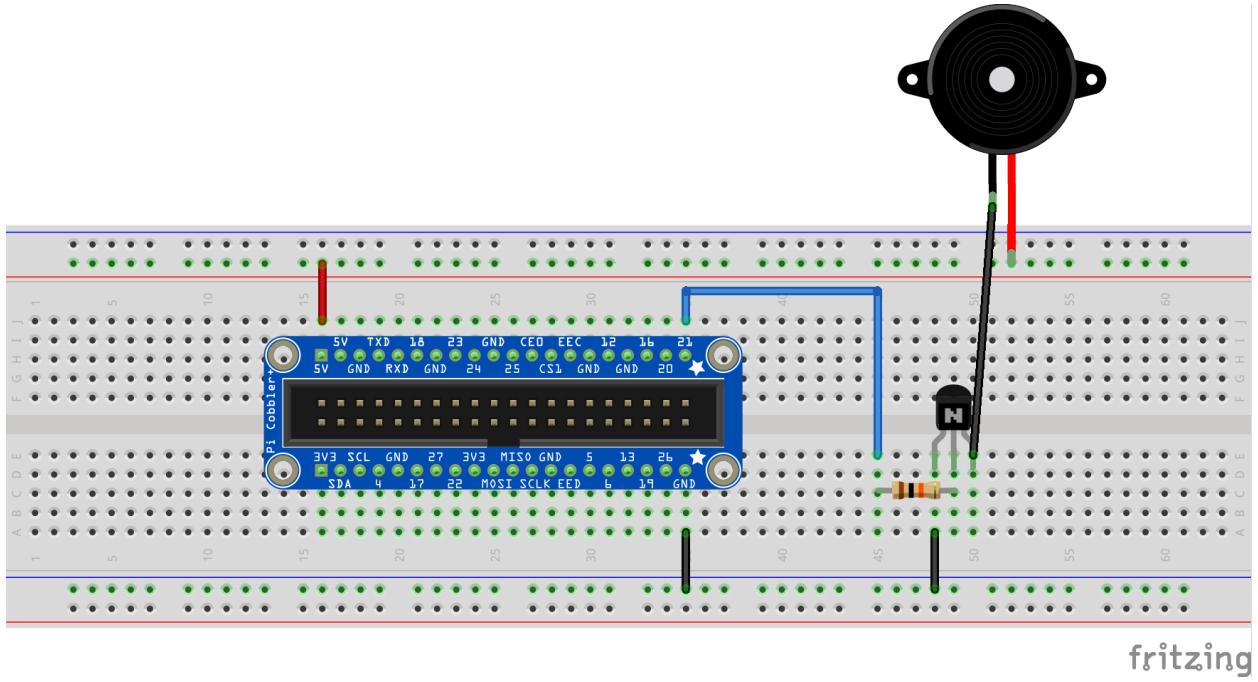
server.listen(PORT, function(){
  console.log("Server listening on: http://localhost:%s", PORT);
});

function audioOn() {
  wpi.digitalWrite(AUDIO_PIN, wpi.HIGH);
}

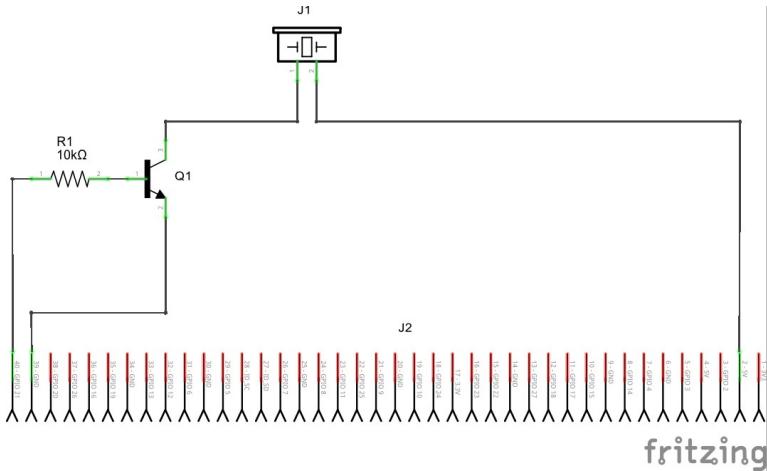
function audioOff() {
  wpi.digitalWrite(AUDIO_PIN, wpi.LOW);
}

function audioSos() {
  var count = 0;
  var state = 1;
  var id;
  audioOn();
  id = setInterval(function() {
    if (state == 1) {
      audioOff();
      state = 0;
      count++;
      if (count == 3) {
        clearInterval(id);
      }
    } else {
      audioOn();
      state = 1;
    }
  }, 1000);
}
```

Here is the corresponding breadboard:



And the same circuit as a schematic:



## FM Radio

The IC called TEA5767 provides FM frequency radio receiver which is digitally controlled via the I2C bus. The address of the device is 0x60. Modules for this IC can be found on eBay for about \$4. The device can be powered from either 3.3V or 5V. The reception frequency is from 76MHz to 108MHz.

The pins on it are:

<b>Pin</b>	<b>Description</b>
1	SDA
2	SCL
3	GND – should be grounded
4	NC
5	VCC – 3.3V – 5V
6	GND – should be grounded
7	Left Out
8	Right Out
9	GND
10	Antenna

Control of the device is by writing 40 bits (5 bytes) of information:

Byte 1		
7	MUTE	<p>Mute control:</p> <ul style="list-style-type: none"> <li>• 1 – Audio is muted</li> <li>• 0 – Audio is not muted</li> </ul> Default: 0
6	SM	<p>Search mode:</p> <ul style="list-style-type: none"> <li>• 1 – Search mode on</li> <li>• 0 – Search mode off</li> </ul> Default: 0
5:0	PLL[13:8]	
Byte 2		
7:0	PLL[7:0]	
Byte 3		
7	SUD	<p>Search up/down:</p> <ul style="list-style-type: none"> <li>• 1 – Search up</li> <li>• 0 – Search down</li> </ul> Default: 1
6:5	SSL[1:0]	Default: 0b01
4	HLSI	Default: 0
3	MS	<p>Mono or stereo:</p> <ul style="list-style-type: none"> <li>• 1 – Mono</li> <li>• 0 – Stereo</li> </ul> Default: 0
2	MR	<p>Mute right:</p> <ul style="list-style-type: none"> <li>• 1 – Right channel muted</li> <li>• 0 – Right channel not muted</li> </ul> Default: 0

1	ML	Mute left: <ul style="list-style-type: none"> <li>• 1 – Left channel muted</li> <li>• 0 – Left channel not muted</li> </ul> Default: 0
0	SWP1	Default: 0
Byte 4		
7	SWP2	Default: 0
6	STBY	Default: 0
5	BL	Band limits: <ul style="list-style-type: none"> <li>• 1 – Japanese</li> <li>• 0 – US/Europe</li> </ul> Default: 0
4	XTAL	Default: 1
3	SMUTE	Soft mute: <ul style="list-style-type: none"> <li>• 1 – Soft mute is on</li> <li>• 0 – Soft mute is off</li> </ul> Default: 0
2	HCC	Default: 0
1	SNC	Stereo noise canceling: <ul style="list-style-type: none"> <li>• 1 – Stereo noise canceling is on</li> <li>• 0 – Stereo noise canceling is off</li> </ul> Default: 0
0	SI	Default: 0
Byte 5		
7	PLLREF	Default: 0
6	DTC	Default: 0
5:0	N/A	Default: 0

See also:

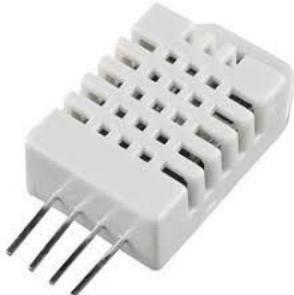
- [Datasheet](#)
- Arduino – [TEA5767 Radio](#)
- Github: [andykarpov/TEA5767](#)
- [TEA5767 FM Radio Digital Tuner with Raspberry Pi 2](#)

## Temperature and Humidity - DHT22

One of the most popular temperature and humidity sensors available is called the DHT22. This is a simple device which only uses a single wire for data communications. The timing on the protocol is at

the microsecond level so Java and JavaScript direct GPIO access is going to be too slow. However for C/C++ access, we are just fine.

Physically, the device has 4 pins 1 of which is not connected. The others are Vcc (3V-5V), GND and Data. A 4.7K resistor should be attached between Vcc and Data to pull data high.



This device provides an interesting case study on protocol descriptions from data sheets and the mapping to bit-banging I/O.

## Using WiringPi

An existing Open Source sample can be found at [https://github.com/technion/lol\\_dht22](https://github.com/technion/lol_dht22).

To build, run the following:

```
$ git clone https://github.com/technion/lol_dht22.git  
$ cd lol_dht22  
$ ./configure  
$ make
```

You will now have an executable called `lol_dht22`. This takes as a parameter the WiringPi pin number to which the data pin of the DHT22 is connected. When run, it will return data such as:

```
$ sudo ./loldht 1  
Raspberry Pi wiringPi DHT22 reader  
www.lolware.net  
Humidity = 47.60 % Temperature = 24.50 *C
```

Within the project is a source file called `dht22.c` which contains the logic to access the device. This can be used as the basis for your own projects.

## Using the Arduino APIs

The Arduino APIs provide a pre-supplied class called `DHT22`. This class has methods which include the following:

- `readData()` - Read and store the latest values from the device
- `float getHumidity()` - Return the latest retrieved humidity value
- `float getTemperatureC()` - Return the latest retrieved temperature value in degrees centigrade

Samples are provided as part of the package.

## Temperature and pressure - BMP180

The BMP180 module can be found on eBay for less than \$2.00. This device provides both temperature and barometric pressure readings via an I2C bus. It is less expensive than the DHT22 but replaces humidity for air pressure measurement. Arguably, the BMP180 is also easier to use as it does not require as finicky data stream timings.

This is a 3V device. The I2C address of the device is 0x77.

The I2C commands to read the device are described in detail in the data sheet. At a high level, we read out a series of EEPROM registers and then ask for both the temperature and pressure. The values returned for these later values need to them be arithmetically combined with the EEPROM registers using some equations and the result will be two outputs. A temperature in oC and an air pressure in Pa.

The pin out of the device is:

Pin	Label	Description
1	3.3	Not used
2	SDA	I2C SDA
3	SCL	I2C SCLK
4	GND	GND
5	VCC	3.3V

Once connected, we can run i2cdetect to list the devices on the I2C bus and we should find the BPM180 at 0x77.

```
$ i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          --- --- --- --- --- --- --- --- ---
10:          --- --- --- --- --- --- --- --- ---
20:          --- --- --- --- --- --- --- --- ---
30:          --- --- --- --- --- --- --- --- ---
40:          --- --- --- --- --- --- --- --- ---
50:          --- --- --- --- --- --- --- --- ---
60:          --- --- --- --- --- --- --- --- ---
70:          --- --- --- --- --- 77
```

See also:

- [Data sheet](#)
- I2C
- Adafruit – [Using the BMP180 with Raspberry Pi](#)
- Sparkfun – [BMP180 Barometric Pressure Sensor Hookup](#)

- [BMP180 I2C Digital Barometric Pressure Sensor](#)

## Using the Arduino APIs

Adafruit provide an Arduino library for interacting with the device. This is part of their unified sensor series. It has been tested with the Arduino environment running on the Pi and found to work without issue. Stay away from the similarly named project called Adafruit-BMP085-Library as it appears to be buggy.

See also:

- [Adafruit Unified Sensor Driver](#)
- [Adafruit Unified BMP085/BMP180 Driver](#)

## Pressure strips - FSR 402

Force Sensitive Resistors are devices that can measure the force being applied to them. They change their resistance properties as a function of the force. Physical force is measured in Newtons and is given by the equation:

$$F = m \cdot a$$

Where F = force in Newtons, m = mass in kilograms and a = acceleration in meters/second/second. If this seems a little too much, consider that when we place a 1Kg object on a scale, gravity is attempting to accelerate that object at 1g (1 gravity) which is equal to 9.80665 m/s<sup>2</sup>. We'll keep it simple and call it 9.8.

So ... the force of an object due to gravity is:

Mass	Newtons
1g	0.0098N
10g	0.098N
100g	0.98N
1Kg	9.8N

With these understandings of the mass of an object and the direct relationship to force and the fact that a force sensitive resistor can measure force ... then we have the notion that a force sensitive resistor can measure the weight of something. The device reduces its resistance as force is applied.



When we consult the data sheet, the first thing to note is that the change in resistance is non-linear. What we mean by that is that if we double the pressure, we do not halve the resistance. Rather there is a logarithmic curve. Documentation also suggests that these aren't accurate devices for measuring absolute pressure but rather relative pressure and that instances of the device vary in their results. What this means is that if we have two instances of the device and we apply the same pressure to both, they may very well result in two different resistance values. The usefulness of the devices as absolute weighing devices is thus suspect ... however, if we want to measure relative force ... for example to detect when an object place on top of one is full vs empty, then they can come into play.

See also:

- [FSR 402 Data sheet](#)
- Wikipedia: [Newton](#)
- Wikipedia: [Gravity](#)

## Ambient light sensor - BH1750FVI

<Parts on order>

## Infrared receivers

Infrared is the frequency range of light that is not visible to our eyes. If we look at a source of infrared light, we simply won't see any emissions even though they are being emitted. There are LEDs available that can generate infra-red frequency light and there are corresponding detectors that can measure the incoming amount of infrared light. Since these light sources don't distract us, we can use them for signaling by having an infra-red transmitter send pulses of light to an infra-red receiver. This is precisely how many home remote control hand-sets work for devices such as TVs and music systems.

What we might want to do is to control the Pi remotely using an infra-red hand-set.

There are many infra red hand-sets that we can use that can be purchase on eBay for a few dollars. Alternative we can use any hand-set from an old TV.



We will also need to wire in an infra-red receiver circuit to our Pi.

With the transmitter (hand-set) and the receiver (electronics connected to Pi) we are about ready to go. The next thing we have to consider is how to receive the data transmitted and how to decode it.

We need to install the LIRC package:

```
$ sudo apt-get install lirc
```

Next we need to modify the `/etc/modules` file to include the instructions to load the LIRC support.

```
lirc_dev
lirc_rpi
```

The Directory Tree Overlay for the LIRC driver has parameters available for our use. We can configure these by editing `/boot/config.txt` and editing/adding the line which reads:

```
dtoverlay=lirc-rpi,<param>=<value>
```

The following parameters are defined:

Parameter	Default	Description
gpio_out_pin	17	GPIO pin used for output
gpio_in_pin	18	GPIO pin used for input
gpio_in_pull	down	Pull up, down or off the input pin
sense	-1	Use auto detection logic. The values can be: <ul style="list-style-type: none"> <li>• 0 – Force active-high</li> <li>• 1 – Force active-low</li> <li>• -1 – Use auto detection</li> </ul>
softcarrier	on	Switch on or off the software carrier
invert	off	Invert the signal on the output pin
debug	off	Produce additional debug messages

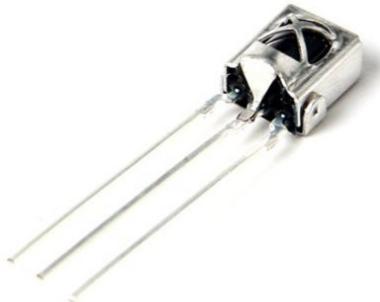
After configuring the Pi to use LIRC, we can then reboot. We can check `dmesg` and we may see messages similar to:

```
lirc_dev: IR Remote Control driver registered, major 244
lirc_rpi: module is from the staging directory, the quality is unknown, you have been
warned.
```

```
lirc_rpi: to_irq 499
lirc_rpi: auto-detected active high receiver on GPIO pin 19
lirc_rpi lirc_rpi: lirc_dev: driver lirc_rpi registered at minor = 0
lirc_rpi: driver registered!
```

We will also find the device driver interface at /dev/lirc0.

One of the more popular infrared receivers is the VS1838B.



The pin out of this device (from left to right facing the front):

Pin	Function
1	Data
2	GND
3	Vcc

With the circuit wired up, we can test the operation by running the command:

```
$ sudo mode2 -d /dev/lirc0
```

What this command does is watch the input device and retrieve the pulses that arrive encoded. The encoding is a period of time the IR is active and a period of time that the IR is inactive. The duration of these pulses provides the encoding.

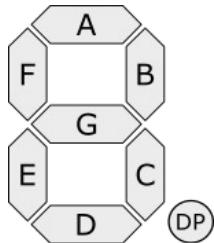
Now we start to move up the chain to the higher level functions. We need to map the pulses sent by the IR transmitter to their meanings. A file called `lirc.conf` contains these codes. We can try and find an existing file for the remote we are using or else we can build our own file using the `irrecord` command. When finally set up, we can use the `irw` command to listen at a higher level for the incoming transmissions and validate that the keys entered on the transmitter are the correct keys detected by the Pi.

See also:

- main(1) – [irrecord](#)
- [LIRC – Linux Infrared Remote Control](#)
- [Setting Up LIRC on the RaspberryPi](#)
- [How to Control the GPIO on a Raspberry Pi with an IR Remote](#)

## LED 7-Segment displays

The 7-Segment display is an LED device that is composed of 7 line segments that can be arranged to display numbers. There is also an eighth LED that is used to display a decimal point. The makeup of the segments are shown in the following diagram:



Although a seven segment display looks "quite dated" they should not be discounted. If what one wants to do is show a numeric value that can be read from a distance and which is very cheap to use, then this device may be just perfect.

It is common to see 7-segment displays used in conjunction with the MAX7219 or MAX7221 ICs. These ICs know how to drive up to eight multiplexed 7-segment displays with data received over an SPI bus. As such, the Pi can send in an SPI signal and these devices can easily display the results.

See also:

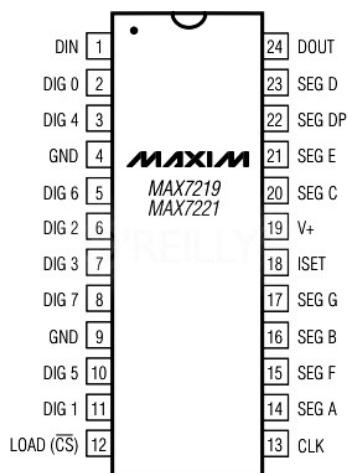
- Wikipedia – [Seven-segment display](#)

## MAX7219/MAX7221 - Serial interface, 8-digit, led display drivers

The MAX7219 and MAX7221 are ICs that can drive up to eight digits of 7 segment displays or drive a single matrix of 8x8 LEDs. The LEDs should be common cathode. The device works using the SPI protocol.

This is a 5V device. A 3.3V device known as the MAX6951 is also available.

The physical layout of the IC looks as follows:



Place a 10uF electrolytic and 0.1uF ceramic as close to the device as possible.

For 7-Segment displays, these should be common cathode.

The pin out of the device is shown in the next table.

Name	Pin	Description
DIN	1	Serial data input (MOSI). Data loaded on clock rising edge.
DIG0-DIG7	2, 3, 5-8, 10, 11	Connections to each of the eight digits.
GND	4, 9	<b>Both</b> ground pins must be connected.
CS	12	Chip select. Data is loaded into serial register while CS is low and latched on CS rising edge.
CLK	13	The clock for the serial data,
SEGA-SEG <sub>G</sub> , DP	14-17, 20-23	Connection to each of the eight segments in a digit.
ISET	18	Connect to V <sub>DD</sub> through a resistor to set peak segment current. The resistor values are shown in a following table. The resistor is referred to as R <sub>SET</sub> . The current sent to a segment is 100 times the current entering ISET.
V <sub>DD</sub>	19	5V
DOUT	24	Serial data output for daisy chaining.

Data format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	X	X	X	Register Address				Data							

Registers

Name	Bits	Description
Shutdown	0b1100 (0xc)	Should the device shutdown the display <ul style="list-style-type: none"> <li>0 – Shutdown. In shutdown mode the display is blanked.</li> <li>1 – Normal</li> </ul>
Decoding	0b1001 (0x9)	How is decoding performed? <ul style="list-style-type: none"> <li>0x00 – No decoding for digits 7-0</li> <li>Corresponding bit on – Code B decode for digit (0-9, E, H, L, P and '-')</li> </ul>
Intensity	0b1010 (0xa)	The brightness of the display 0000 – 1111 with 0000 being minimum and 1111 being maximum.
Scan Limit	0b1011 (0xb)	Controls how many digits are to be included in the scan multiplexing. <ul style="list-style-type: none"> <li>0b0000 – Digit 0</li> <li>0b0001 – Digits 0, 1</li> <li>0b0010 – Digits 0, 1, 2</li> <li>0b0011 – Digits 0, 1, 2, 3</li> <li>0b0100 – Digits 0, 1, 2, 3, 4</li> <li>0b0101 – Digits 0, 1, 2, 3, 4, 5</li> <li>0b0110 – Digits 0, 1, 2, 3, 4, 5, 6</li> <li>0b0111 – Digits 0, 1, 2, 3, 4, 5, 6, 7</li> </ul>
Display Test	0b1111 (0xf)	Test the display. <ul style="list-style-type: none"> <li>0 – Normal operation</li> <li>1 – Test mode</li> </ul>
No-Op	0b0000 (0x0)	Don't perform any task. Useful for passing data through the device to a daisy chained instance.
Digit 0	0b0001 (0x1)	Set the value for digit 0.
Digit 1	0b0010 (0x2)	Set the value for digit 1.
Digit 2	0b0011 (0x3)	Set the value for digit 2.
Digit 3	0b0100 (0x4)	Set the value for digit 3.
Digit 4	0b0101 (0x5)	Set the value for digit 4.
Digit 5	0b0110 (0x6)	Set the value for digit 5.
Digit 6	0b0111 (0x7)	Set the value for digit 6.
Digit 7	0b1000 (0x8)	Set the value for digit 7.

The value of the RSET resistor can be seen in the following table:

LED Forward Current	1.5V	2.0V	2.5V	3.0V	3.5V
10ma	66.7k	63.7k	59.3k	55.4k	51.2k
20ma	29.8k	28.0k	25.9k	24.5k	22.6k
30ma	17.8k	17.1k	15.8k	15.0k	14.0k
40ma	12.2k	11.8k	11.0k	10.6k	9.7k

Anodes of the LEDs must be connected to the SEG<sub>x</sub> lines while cathodes must be connected to the DIG<sub>x</sub> lines.

There are boards available which have MAX7219's already mounted and ready for work including 8x8 LED matrices. These are much easier to work with than wiring together a rats-nest of links. The boards cost less than \$2 an instance. The pin out from the board are:

Pin	Label	Description
1	Vcc	+ve
2	GND	Ground.
3	DIN	Data in.
4	CS	Chip Select.
5	CLK	Clock.

See also:

- [MAX7221 Home Page](#)
- [Drive MAX7219/MAX7221 with common anode displays](#)
- Arduino – [The MAX7219 and MAX7221 LED drivers](#)
- Github: Arduino – [wayoda/LedControl library](#)
- Instructables – [16x8 LED dot matrix with MAX7219 module](#)
- YouTube – BrainyBits - [How to use MAX7219 Dot LED matrix with Arduino](#)
- YouTube – [Scrolling text using the MAX7219 and an Arduino](#)
- YouTube – [Arduino tutorial: LED Matrix red 8x8 64 Led driven by MAX7219 \(or MAX7221\) and Arduino Uno](#)

## Using the Arduino APIs

An excellent open source library for the Arduino already exists with samples. It is called `wayoda/LedControl`. If we download this for our Arduino IDE and install it, we can then run the `LCDemoMatrix` sample which displays different patterns of output.

See also:

- Github: Arduino – [wayoda/LedControl library](#)

## Using Python

A particularly good library is the one found on Github at `rm-hull/max2719`. This uses the SPI hardware of the Pi.

First we import the module using:

```
import max7219.led as led
```

Now we can create a device object that represents the device either as a 7-segment display or as an 8x8 matrix. To create the 7-segment display device we use:

```
device = led.sevensegment()
```

while to create the matrix we use

```
device = led.matrix();
```

Methods on a generic device include:

- brightness(intensity) – a range from 0 – 15
- scroll\_left() - scroll the image to the left
- scroll\_right() - scroll the image to the right

Methods on a 7-segement device include:

- letter(deviceId, position, char, dot=False, redraw=True)
- write\_number(deviceId, value, base=10, decimalPlaces=0, zeroPad=False, leftJustify=False)

Methods on a matrix device include:

- letter(deviceId, asciiCode, font=None, redraw=True)
- scroll\_up(redraw=True)
- scroll\_down(redraw=True)
- show\_message(text, font=None, delay=0.05)
- pixel(x,y, value, redraw=True)
- invert(value, redraw=True)
- orientation(angle, redraw=True) – The angle can be one of 0, 90, 180 or 270.

See also:

- Github: [rm-hull/max7219](#)

## Using JavaScript

There is an npm package called max7219. This package is a little old and doesn't appear to have been touched in a couple of years. It has a pre-requisite of SPI used to drive the SPI bus. Unfortunately, the package wants to force us to use SPI 0.1.0 while the latest SPI is at 0.2.0. This causes SPI 0.1.0 to be downloaded but this failed to compile (Jessie). To circumvent the problem, I installed SPI 0.2.0 by hand, downloaded the tarball for max7219, extracted it, edited the package.json and changed the SPI dependency to be 0.2.0. Then I installed max7219 from this modified package and now all seems to work as desired. This is of course an ugly story and an issue has been raised against the project (see [#1](#)).

However, once working, it seems to be working as desired. For example, the following displays "3.141":

```
var MAX7219 = require("max7219");
var disp = new MAX7219("/dev/spidev0.1");
disp.setDecodeAll();
disp.startup();
disp.setDigitSymbol(0, "1");
disp.setDigitSymbol(1, "4");
disp.setDigitSymbol(2, "1");
```

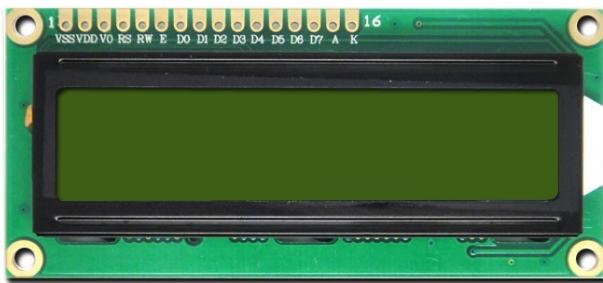
```
disp.setDigitSymbol(3, "3", true);
disp.setScanLimit(4);
```

See also:

- [npm - max7219](#)

## LCD dot matrix display - HD44780

These are extremely cheap and popular 16 column by 2 row LCD displays. Obviously in that small space you can't show much but you can show enough for many functions.



The device is a 5V device but since it is write only, we don't have to worry about level shifting. The device has a slew of wiring connections though and if one consults the data sheet, one will find a wealth of options. While it wouldn't be too difficult to write a C language interface to the device yourself, it would seem better to leverage an existing library. One is provided for us in the `wiringPiDev` library.

Pin	Name	Description
1	GND	Ground
2	VDD	Source (+ve)
3	VE (Contrast)	
4	RS (Register Select)	Register select <ul style="list-style-type: none"> <li>• 1 – data input</li> <li>• 0 – instruction input</li> </ul>
5	R/W	Read/Write <ul style="list-style-type: none"> <li>• 1 – read</li> <li>• 0 – write</li> </ul>
6	Enable	Starts data read/write. Data is clocked on the falling edge.
7	DB0	Used in 8bit mode
8	DB1	Used in 8bit mode
9	DB2	Used in 8bit mode
10	DB3	Used in 8bit mode
11	DB4	Used in 4bit or 8bit mode
12	DB5	Used in 4bit or 8bit mode
13	DB6	Used in 4bit or 8bit mode
14	DB7	Used in 4bit or 8bit mode. Also contains the BUSY flag.
15	Back Led+	Back light LED +ve
16	Back Led-	Back light LED -ve

The device has two banks of RAM. One is for display data and is called DDRAM while the other is for character generation and is called CGRAM. DDRAM is 80 bytes in size. The selection of which bank of RAM to be used is made by setting the RS (register select) pin.

The device can be used in either a 4 bit transfer mode or an 8 bit transfer mode. In 8 bit mode, all the pins labeled DB0-DB7 are used. In 4 bit mode, only the pins labeled in DB4-DB7 are used. To transfer 8 bits of data in 4 bit mode, two transfers are required. The order of data is the 4 high order bits in the first transmission and the 4 low order bits in the second transmission.

For two lines of display, take care to realize that the address of the 1st position on line two does **not** immediately follow the last position on line 1.

Address	1	2	3	...	39	40
Line 1	0x00	0x01	0x02	...	0x26	0x27
Line 2	0x40	0x41	0x42	...	0x66	0x67

The combination of RS and R/W provide control information:

RS	R/W	Operation
0	0	Perform an internal operation.
0	1	Read busy flag and address counter.
1	0	Write data.
1	1	Read data.

Instructions:

- Clear display

Clears the display and sets the DDRAM address counter to 0.

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	0	0	0	0	0	1

- Return home

Sets the DDRAM address counter to 0. Also resets any display shifting that may be present.

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	0	0	0	0	1	-

- Entry mode properties

Sets the address counter direction for auto-increment or auto-decrement. Also specifies whether or not the display should be shifted.

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	0	0	0	1	I/D	S

A constant called `LCD_ENTRYMODESET` with a value of `0x04` may be used as a base. Other related constants are:

- `LCD_ENTRYRIGHT` = `0x00`
  - `LCD_ENTRYLEFT` = `0x02`
  - `LCD_ENTRYSHIFTINCREMENT` = `0x01`
  - `LCD_ENTRYSHIFTDECREMENT` = `0x00`
- 
- Display mode properties

Sets display controls.

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	0	0	1	D	C	B

- Controls display visibility as a whole (D)
  - 0 – off – `LCD_DISPLAYOFF`
  - 1 – on – `LCD_DISPLAYON`
- Controls visibility of cursor (C):
  - 0 – off – `LCD_CURSOROFF`
  - 1 – on – `LCD_CURSORON`
- Controls blinking state of cursor (B)
  - 0 – off – `LCD_BLINKOFF`
  - 1 – on – `LCD_BLINKON`

A constant called `LCD_DISPLAYCONTROL` with a value of `0x08` may be used as a base. Other related constants are:

- `LCD_DISPLAYON` = `0x04`
- `LCD_DISPLAYOFF` = `0x00`
- `LCD_CURSORON` = `0x02`

- LCD\_CURSOROFF = 0x00
- LCD\_BLINKON = 0x01
- LCD\_BLINKOFF = 0x00
- Cursor or display shift

<b>RS</b>	<b>R/W</b>	<b>DB7</b>	<b>DB6</b>	<b>DB5</b>	<b>DB4</b>	<b>DB3</b>	<b>DB2</b>	<b>DB1</b>	<b>DB0</b>
0	0	0	0	0	1	S/C	R/L	x	x

- Function Mode properties

<b>RS</b>	<b>R/W</b>	<b>DB7</b>	<b>DB6</b>	<b>DB5</b>	<b>DB4</b>	<b>DB3</b>	<b>DB2</b>	<b>DB1</b>	<b>DB0</b>
0	0	0	0	1	DL	N	F	x	x

- Data length (4 or 8 bit) (DL)
  - 0 – 4bits – LCD\_4BITMODE
  - 1 – 8bits – LCD\_8BITMODE
- Number of lines to display (N)
  - 0 – 1 line – LCD\_1LINE
  - 1 – 2 lines – LCD\_2LINE
- Character font (F)
  - 0 – 5x8 dots – LCD\_5x8DOTS
  - 1 – 5x10 dots – LCD\_5x10DOTS

A constant called `LCD_FUNCTIONSET` with a value of 0x20 may be used as a base. Other related constants are:

- LCD\_8BITMODE = 0x10
- LCD\_4BITMODE = 0x00
- LCD\_2LINE = 0x08
- LCD\_1LINE = 0x00

- LCD\_5x10DOTS = 0x04
- LCD\_5x8DOTS = 0x00

- Set CGRAM address

Set the address counter (AC) for CGRAM

<b>RS</b>	<b>R/W</b>	<b>DB7</b>	<b>DB6</b>	<b>DB5</b>	<b>DB4</b>	<b>DB3</b>	<b>DB2</b>	<b>DB1</b>	<b>DB0</b>
0	0	0	1	AC5	AC4	AC3	AC2	AC1	AC0

- Set DDRAM address

Set the address counter (AC) for DDRAM

<b>RS</b>	<b>R/W</b>	<b>DB7</b>	<b>DB6</b>	<b>DB5</b>	<b>DB4</b>	<b>DB3</b>	<b>DB2</b>	<b>DB1</b>	<b>DB0</b>
0	0	1	AC6	AC5	AC4	AC3	AC2	AC1	AC0

- Read busy flag and address

<b>RS</b>	<b>R/W</b>	<b>DB7</b>	<b>DB6</b>	<b>DB5</b>	<b>DB4</b>	<b>DB3</b>	<b>DB2</b>	<b>DB1</b>	<b>DB0</b>
0	1	BF	AC6	AC5	AC4	AC3	AC2	AC1	AC0

Always check the busy flag (BF) to determine if the device is busy before performing an operation.

- Write data to RAM

Writes data to either CGRAM or DDRAM. The choice of whether CGRAM or DDRAM is written is based on the last previous address change of the address counter. For example, if the address counter for DDRAM was last set, then the next write will be to DDRAM.

<b>RS</b>	<b>R/W</b>	<b>DB7</b>	<b>DB6</b>	<b>DB5</b>	<b>DB4</b>	<b>DB3</b>	<b>DB2</b>	<b>DB1</b>	<b>DB0</b>
1	0	D7	D6	D5	D4	D3	D2	D1	D0

After a write to RAM, the address is automatically incremented in preparation for the next write operation.

- Read data from RAM

Reads data from either CGRAM or DDRAM. The choice of whether CGRAM or DDRAM is read is based on the previous address change of the address counter. For example, if the address counter for DDRAM was last set, then the next read will be from DDRAM.

<b>RS</b>	<b>R/W</b>	<b>DB7</b>	<b>DB6</b>	<b>DB5</b>	<b>DB4</b>	<b>DB3</b>	<b>DB2</b>	<b>DB1</b>	<b>DB0</b>
1	1	D7	D6	D5	D4	D3	D2	D1	D0

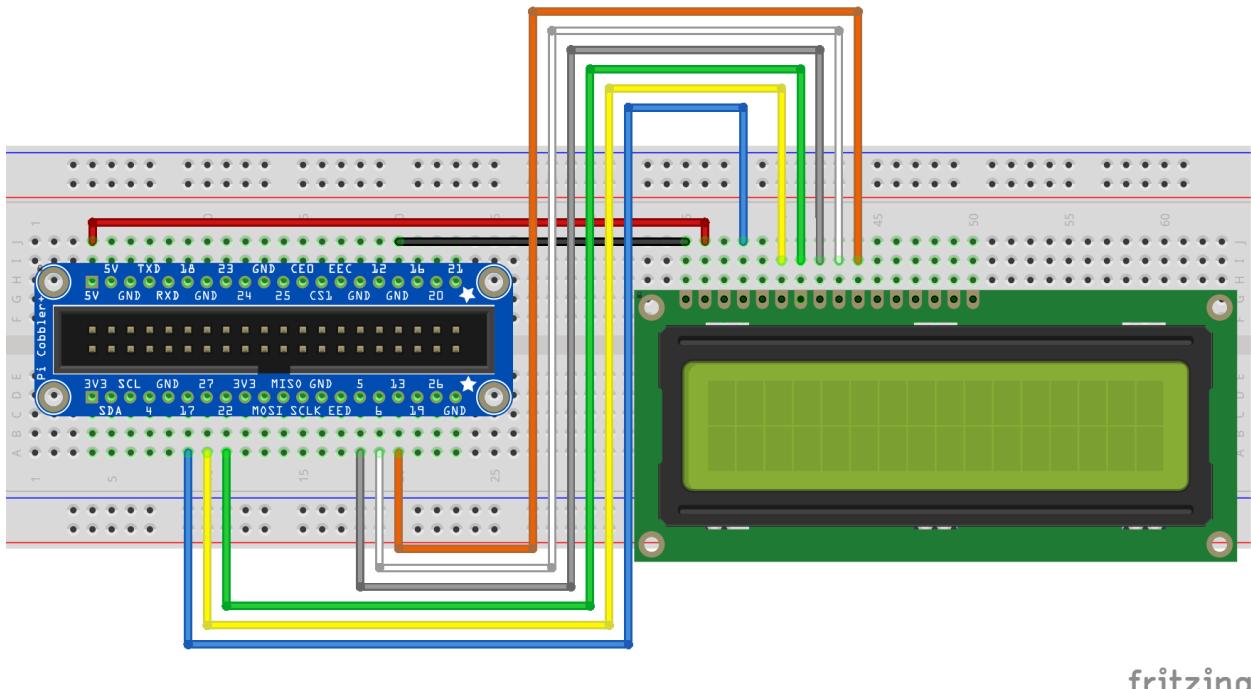
After a read from RAM, the address is automatically incremented in preparation for the next write operation.

Character table

	0000	0001	0010	0011	0100	0101	0111	1000	1001	1010	1011	1101	1110	1111
0000				0	@	P	`	p						
0001			!	1	A	Q	a	q						
0010			"	2	B	R	b	r						
0011			#	3	C	S	c	s						
0100			\$	4	D	T	d	t						
0101			%	5	E	U	e	u						
0110			&	6	F	V	f	v						
0111			'	7	G	W	g	e						
1000			(	8	H	X	h	x						
1001			)	9	I	Y	i	y						
1010			*	:	J	Z	j	z						
1011			+	;	K	[	k	{						
1100			,	<	L		l							
1101			-	=	M	]	m	}						
1110			.	>	N	^	n	→						
1111			/	?	O	—	o	←						

In the following diagram, we have wired the HD4478 pins as follows:

GPIO	HD4478 function
GPIO17	RS
GPIO27	Enable
GPIO22	D0
GPIO5	D1
GPIO6	D2
GPIO13	D3



See also:

- WiringPi – [LCD Library \(HD44780U\)](#)
- wiringPiDev – LCD

## Using WiringPi

Here is a sample C program that uses `wiringPiDev` library to drive the LCD.

```

#include <stdio.h>
#include <wiringPi.h>
#include <lcd.h>

int main(int argc, char *argv[]) {
    printf("Starting LCD test\n");

    int rc = wiringPiSetupGpio();
    int handle = lcdInit(2, // rows
        16, // columns
        4, // bits
        17, // rs
        27, // enable
        22, // d0
        5, // d1
        6, // d2
        13, // d3
        0,0,0,0); // d4-d7
    lcdClear(handle);
    lcdPuts(handle, "Hello World!");
    int i=0;
    while(1) {
        lcdPosition(handle, 0, 1);
        lcdPrintf(handle, "%d", i);
        i++;
    }
}

```

## Using Java

The Java class supplied with Pi4J at `com.pi4j.component.lcd.impl.GpioLcdDisplay` provides a wrapped for the HD44780.

## LCD display - Nokia 5110 - PCD8544

This little LCD screen has a resolution of 84x48 pixels and can be picked up on eBay for about \$3. The underlying IC is the PCD8544 but it is also known as the Nokia 5110. The device is driven by an SPI interface.



The pins on the board are:

Pin	Description
1	Vcc
2	GND
3	SCE / CS – Chip enable. A low edge indicates start of data transmission.
4	RST
5	D/C
6	DN <MOSI>
7	SCLK
8	Backlight LED

The data sheet explains well the sequence of commands that need to be sent to drive the display however, as always, look to leveraging what already exists. The fantastic folks at Adafruit with their Adafruit GFX graphics library have also provided an Arduino library for working with the device. This library has been ported to work with the Pi. As such, all you need are copies of those libraries and you are ready to go.

Frame Buffer:

DC – 24, RST – 25

```
sudo modprobe fbtft_device name=nokia5110 gpios=dc:24,reset:25,led:23
```

See also:

- [Data Sheet](#)

## Using WiringPi

For C and WiringPi, there is an existing native Open Source project called [binerry/RaspberryPi](#) which provides C code for driving the display directly through WiringPi.

The port of the Adafruit library also uses WiringPi as the GPIO subsystem.

See also:

- Github – [binerry/RaspberryPi](#)

## Using Pi4J

For Java, there is an existing Open Source project called [CleversonSA/JPCD8544](#) which provides a Java class for driving the display.

See also:

- Github – [CleversonSA/JPCD8544](#)

## OLED 128x64 - SSD1306

Another screen device that is readily available are the small OLED displays. These are based on an IC called the SSD1306 and can be found on eBay using that phrase as a search. The typical resolution is 128x64. The price for an instance seems to range between \$7 and \$12.

The device can operate at either 3.3V or 5V.



The pin out on this device is

Pin	Description
1	GND – Ground
2	VCC – 3.3v or 5.0v
3	D0 – Clock
4	D1 – MOSI
5	RES – Reset
6	DC – Data / Command
7	CS – Chip Select

The device can support a variety of protocols including 3 or 4 wire SPI and I2C. Depending on the model / variety of the board it you obtain, it is likely that it will be physically configured in one particular mode. Examine the board and the associated documentation.

An open source Github project called [hallard/ArduPi\\_OLED](#) is available which will drive the device.

See also:

- [SSD1306 Data sheet](#)
- [New Adafruit generic OLED display driver for Raspberry Pi](#)
- Adafruit – [OLED displays for the Raspberry Pi](#)
- [Controlling an Adafruit SSD1306 SPI OLED With a Raspberry Pi](#)
- Adafruit – [Monochrome OLED breakouts](#)
- YouTube – [Raspberry Pi - OLED Displays!](#)

## Using the Android API

The Adafruit Arduino library for the SSD1306 was tested and worked just fine. This allows us to drive the SSD1306 device through the Arduino IDE. One must modify a library supplied header file to specify the correct height of the display.

See also:

- Adafruit – [Monochrome OLED breakouts](#)
- GitHub – [adafruit/Adafruit\\_SSD1306](#)

## TFT 1.44" Color - ILI9163C

This little color display provides a color TFT with a resolution of 128x128 pixels. On eBay, it can be found for about \$4 a unit. The driver chip is an ILI9163.

## TFT 2.4" 320x240 - TJCTM24024-SPI

A TFT (thin-film-transistor) display is a larger display capable of color output. These displays come in a variety of sizes. Although there are many displays for many manufacturers we should look for commonality between them. It appears that the driver for many of these devices is a chip set called ILI9341.

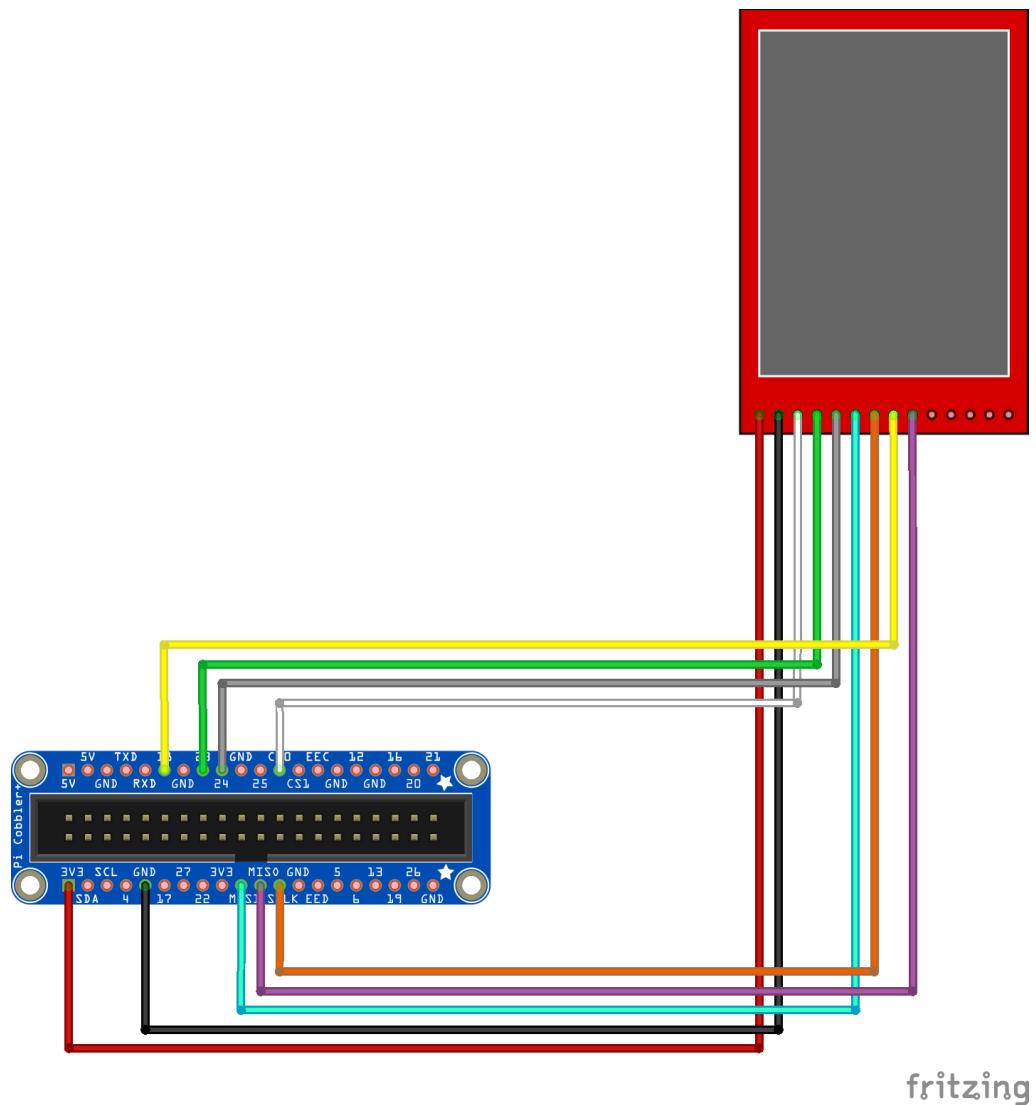
The pin out is as follows:

Pin	Label	Description	fbtft (rpi-display)
1	VCC	3.3V	VCC
2	GND	Ground.	GND
3	CS	Chip select.	GND
4	Reset	Reset.	GPIO23
5	D/C	Data or Command	GPIO24
6	MOSI	Master Out / Slave In.	MOSI
7	SCK	Clock.	SCK
8	LED		GPIO18
9	MISO	Master In / Slave out.	MISO

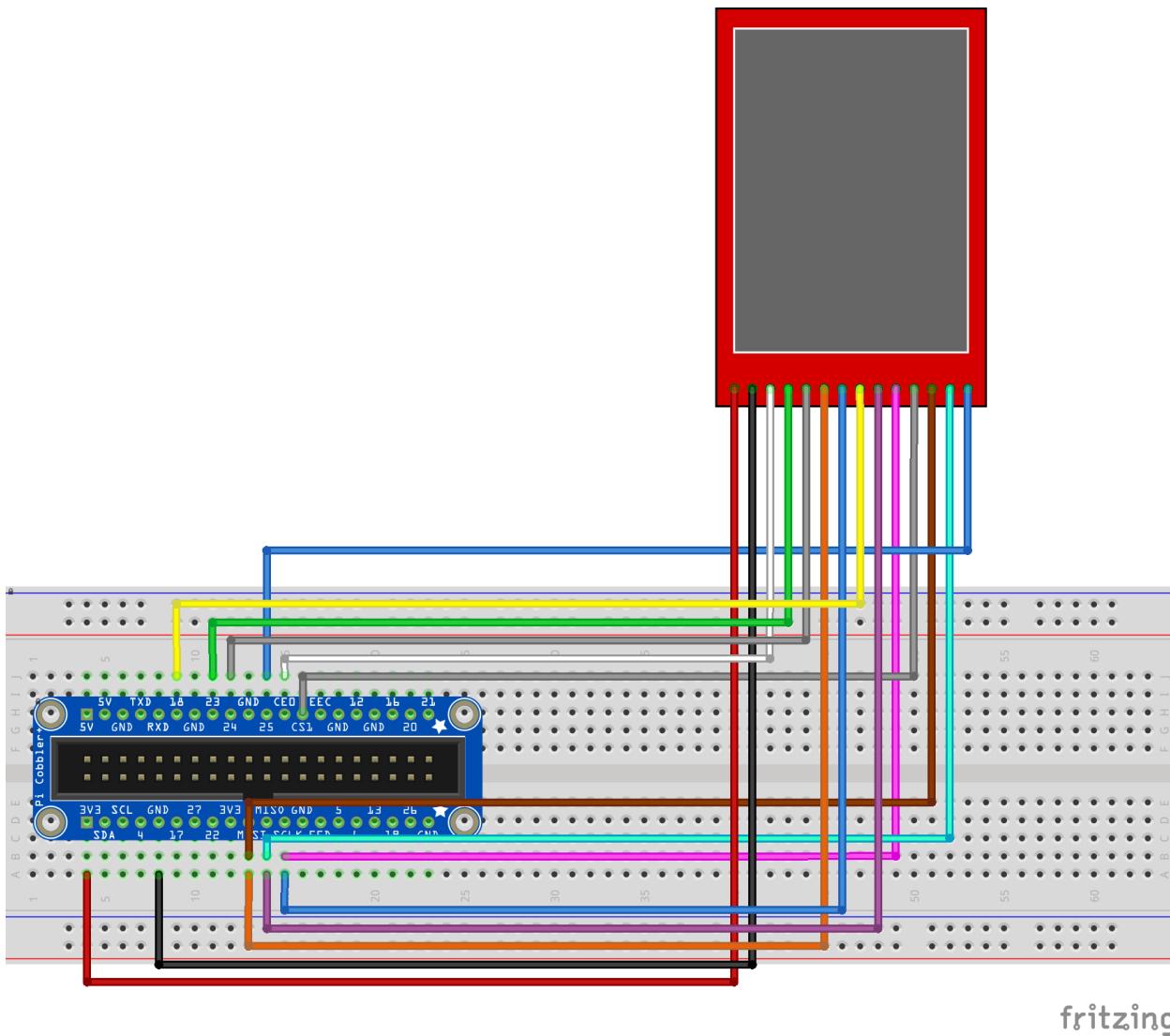
To use with the Frame Buffer TFT Driver, I recommend the driver named "rpi-display". The default configuration for that driver is:

GPIO Pin	Label
18	led
23	reset
24	dc
MOSI	MOSI
MISO	MISO
CS0	CS
SCK	SCK

Here is an example visual of how it can be wired:



If we also want to include the touch screen wiring, the following can be used:



See also:

- [Datasheet](#)
- [Github – Adafruit\\_ILI9341](#)
- [Github – Adafruit-GFX-Library](#)
- [Instructables – TFT2.2 Inch display on Arduino](#)
- Forum – [TJCTM24024-SPI – ILI9341 – Arduino UNO R3](#)

## Frame Buffer TFT driver

When we look at a graphics display such as you will find on your monitor or phone, you need to pause for a moment and think about how far computer technology has come and so quickly. There was a time when a computer's output (if it had any) were streams of characters (7 bit ASCII) that were sent to a terminal and echoed to the user. You can still see this reflected prevalently in Unix environments and Raspbian is

no exception. Today we have graphics displays with mouse cursor control and color and high resolution ... but have you considered how that all works?

Let us take a step back and look at a monitor. A monitor is a rectangular grid of pixels where each pixel can have a color. The colors are usually described as being composed of red, green and blue values where their intensity can also be controlled. A red value of 0 is none at all while a red value of 255 is brightest red possible. Same for green and blue. This means that a pixel is usually composed of 24 bits ... 8 bits for each of the colors. By mixing the intensity of each of the colors, every color we need can be produced. However, remember that this is still for just one pixel. Your screen is composed of a lot of pixels ... a width x height number of pixels to be precise. For example, if your monitor is 1920 x 1024 then you will have 19,201,024 pixels.

When we want to draw a character on the screen at a certain location, what the computer does is look up the graphical details of that character (i.e. what it will look like). That data is stored in a font file. From the data in the font file, the computer then colors the appropriate pixels to provide the representation of the character. Basically, there is a mapping from an intent (draw the character 'A') to the population of pixels values (set pixel values so that when seen on the screen, it will look like 'A').

Since there are many different types of graphical outputs (printers, screens, tablets, graphics cards ...), we need a separation between what the computer wants to do (draw the letter 'A') and how it is mechanically sent to the graphical output. That separation is called a "frame buffer". Think of a frame as a snap shot of what the computer would like to be on the screen. The frame buffer can be thought of as an area of memory that logically addresses a screen. If one were to set values into the frame buffer corresponding to the pixels then the belief is that by doing this, they would appear on the output of the screen. The computer then can separate itself from mechanical interactions with the display and consider instead just the logical desire of filling in the correct values in the frame buffer.

Now we turn our attention to the other side of the story. If the computer is writing data to the frame buffer, something must be reading that data and making it "real" on the screen. This is where a hardware device driver comes into play. The device driver knows how to take the logical frame buffer and "do what is necessary" to update the physical display.

For our TFT displays which are connected by SPI, what this will mean is that the device driver will see the change of pixels and issue the corresponding SPI data transmissions to the physical electronics IC contained in the TFT. The device driver is the component that understands the different data bus commands to send to different TFT devices. This also means that to use a TFT we have to enable SPI support.

We mentioned earlier that large amount of memory that is used to represent a frame buffer. However this does not have to be real memory. For example, the computer believes it is writing to a memory address corresponding to a pixel ... but the frame buffer implementation may choose to merely take that request and pass it on to the physical device as opposed to storing it locally in real memory.

There is a kernel module called `fbtft_device` that provides the ability to display Pi output on a TFT. The module knows about a number of pre-supplied screens and if you are using one of those screens, you can run:

```
$ sudo modprobe fbtft_device name=<Device Name>
```

To see the list of known devices, we can run the special command:

```
$ sudo modprobe fbtft_device name=list
```

You will **not** see any output on the console as what we are doing here is touching a kernel module however if we now run dmesg to look at the kernel log, we will see the output at the end.

Currently, the list that I see is:

adafruit18	ili9481	sainsmart32_fast
adafruit18_green	itdb24	sainsmart32_latched
adafruit22	itdb28	sainsmart32_spi
adafruit22a	itdb28_spi	spidev
adafruit28	mi0283qt-2	ssd1331
adafruit13m	mi0283qt-9a	tinylcd35
agm1264k-fl	mi0283qt-v2	tm022hdh26
dogs102	nokia3310	tontec35_9481
er_tftm050_2	nokia5110	tontec35_9486
er_tftm070_5	piscreen	upd161704
flexfb	pitft	waveshare32b
flexpfb	pioled	waveshare22
freetronicsoled128	rpi-display	
hx8353d	s6d02a1	
hy28a	sainsmart18	
hy28b	sainsmart32	

Parameters can also be supplied to the device driver. The defined parameters are:

Parameter	Description
busnum	SPI bus number
cs	SPI chip select (default 0)
speed	SPI speed in Hz
mode	SPI mode
rotate	0, 90, 180, 270
bgr	Set to Blue Green Red
gpios	The logical pins are reset, dc and led. The format of mapping is: gpios=<logical>:<physical>,<logical>:<physical> ...
fps	Frames per second
gamma	
txbuflen	
startbyte	
init	
verbose	0 – silent 1 – show gpios used 2 – and show devices after registration 3 – and show devices before registration

We can also create a custom device using the custom option. In addition to the previous options, this adds:

Parameter	Description
width	Display width
height	Display Height
buswidth	Display bus width

To make permanent the loading of the `fbtft_device`, we need to request that it be loaded at boot time and also that it be loaded after SPI has been initialized. To achieve that, create a new file called `/etc/modules-load.d/fbtft.conf`. Files in this folder contain the names of kernel modules to load. In the `fbtft.conf` file, add the following two lines:

```
spi-bcm2835
fbtft_device
```

This instructs the kernel to load the SPI driver and then the `fbtft_device` driver.

Since we also want to pass configuration options to the `fbtft_device`, create another file called `/etc/modprobe.d/fbtft_device.conf`. The contents of files in this directory are options passed into modprobe. Within the `fbtft_device.conf` file add the following (or similar as your device needs):

```
sudo modprobe fbtft_device name=nokia5110 gpios=dc:24,reset:25,led:23
```

The frame buffer used by the console can be set with `con2fbmap`. Running `con2fbmap 1` will show where console 1 is being sent. To switch the console to use frame buffer 1, run:

```
$ con2fbmap 1 1
```

To see the identities of frame buffers, we can `cat /proc/fb`. This will list the frame buffers defined on the Pi. For example:

```
$ cat /dev/fb
0 BCM2708 FB
1 fb ili9341
```

We can also set the TFT frame buffer to be used by X-Windows. On a Pi there is a built in GPU which expects to read the frame data from an existing frame buffer. The default is called `/dev/fb0`. Writing into this frame buffer drives the HDMI output. By default, the X-Windows server that runs on the Pi generates its graphics output by writing into this frame buffer. In order to have X-Windows output shown on our SPI connected TFT, we need to instruct X-Windows to use a different frame buffer ... namely `/dev/fb1` which is the frame buffer of the TFT. Edit the file:

```
/usr/share/X11/xorg.conf.d/99-fbturbo.conf
```

Find the line which reads:

```
Option "fbdev" "/dev/fb0"
```

and change to:

```
Option "fbdev" "/dev/fb1"
```

There is a partner utility called `fbcop` which copies the frame buffer from `/dev/fb0` to `/dev/fb1`. This is supplied in source form and must be compiled.

I want to give you a warning about using this device driver and using other SPI based devices. The SPI protocol says that many devices can be connected to the MOSI and MISO lines but only one of them should be active at any one time. It is the responsibility of the environment to ensure that only one is indeed active. However, Pi is a multi process environment and, by and large, these processes don't know what each other is doing and hence there is very little in the way of cooperation between them. If the `fbft` drivers are enabled, you should be very cautious in using other SPI devices connected to the same SPI pins.

See also:

- Github: [notro/fbtst](#)
- Github: [tasanakorn/rpi-fbcop](#)
- [Module source at Linux Kernel](#)
- SPI
- [How to Setup an LCD Touchscreen on the Raspberry Pi](#)

## Touch screen input

Having looked at what it takes to drive output to a TFT screen we now look at input from the screen. Some devices have touch screen capabilities which means that pressure or touch applied to the screen can be registered as data input.

The theory behind a touch screen is to imagine that it is made up of a fine grain set of cells. When touch is made somewhere on the screen, the resistance of some cells change. This tells a processor (like the TSC2046 or the ADS7846) which cells changed. This is then transmitted via SPI as a pair of coordinates.

Looking at the rear of some TFT displays we find some touch related pins labeled:

Pin	Label	Description
1	T_IRQ	Interrupt request.
2	T_DO	MOSI
3	T_DIN	MISO
4	T_CS	Device select
5	T_CLK	Clock

The Pi has a driver for touch screens called "ads7846". This is supplied as a Device Tree driver. To enable in the Pi, we need to edit `/boot/config.txt` and add a line that reads:

```
dtoverlay=ads7846,speed=500000,penirq=<IRQ Pin Number>
```

If all has gone well, dmesg will show a line:

```
ads7846 spi0.1: touchscreen, irq 501
```

If we find that the orientation of movement is backwards, add `swapxy=1` to the `dtoverlay` line.

My final working configuration was:

`/boot/config.txt`

```
dtoverlay=ads7846,speed=50000,penirq=21,swapxy=1
```

`/etc/modules.load.d/fbtft.conf`

```
spi-bcm2835  
fbtft_device
```

`/etc/modprobe.d/fbtst_device.conf`

```
options fbtft_device name=rpi-display rotate=270
```

See also:

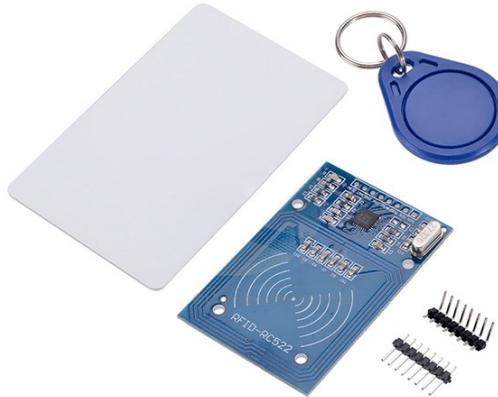
- [I'd like to have some LCD on my Pi](#)

## RFID MFRC522

Radio frequency identification (RFID) is the notion that we can have a receiver that is listening for low power radio frequency emissions. When a device that emits a signal on the correct frequency is brought in range, the receiver is triggered. Typically, the range of such a device is only a few inches and the transmitter itself is a passive device ... meaning it has no power source within it. We see these kinds of devices in hotels when they can be used to open doors. Instead of inserting a physical key or even a key card into a slot, we simply bring the card "near" or "tap it" on the lock and the door opens.

The transmitters can also transmit small amounts of information to the receiver and that information can be used allow or disallow access.

For our purposes, we can use the MFRC522 receiver and associated cards and transmitters. If you see the phrase PICC ... this stands for "proximity integrated circuit card" which is the formal name for the RFID cards and tags.



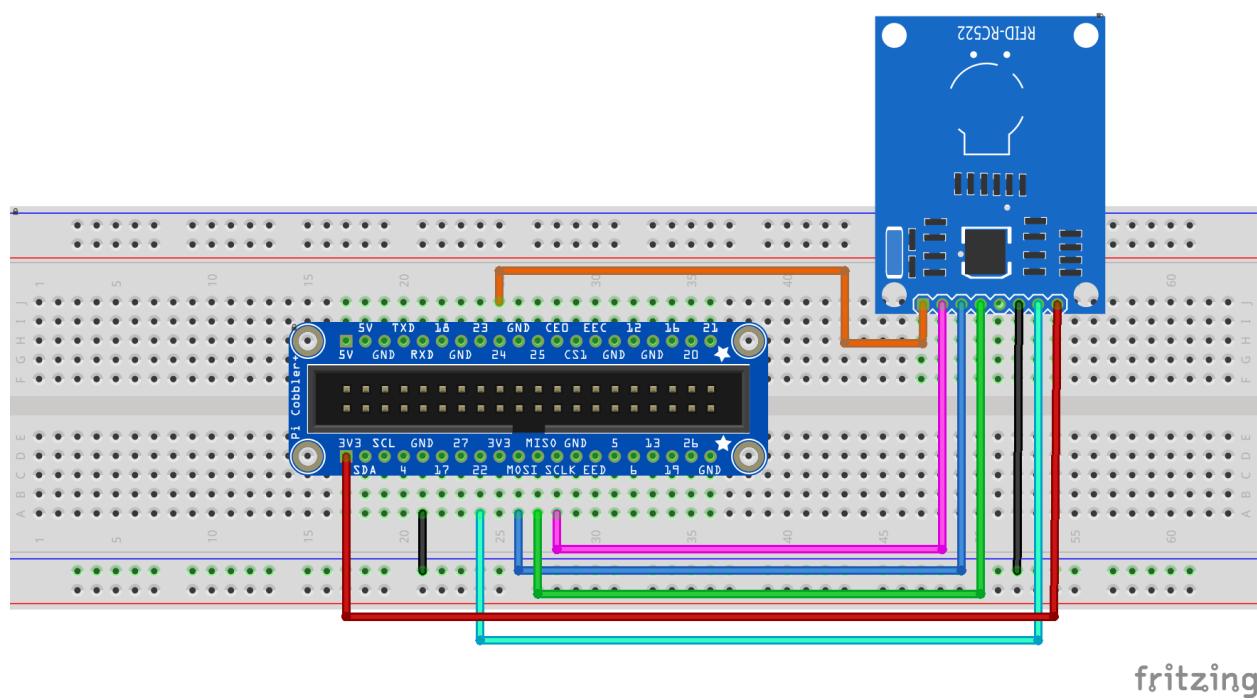
A receiver and some transmitters can be picked up on eBay for under \$3.

The MFRC522 is a 3.3V device. Do not try and power it from the 5V source. The cards can hold up to 1K bytes of data and have a range of 1cm or 2cm. The communication between the receiver and the Pi is via the SPI protocol.

The pin out on the device is:

<b>Pin</b>	<b>Label</b>	<b>Description</b>
1	3.3V	Source
2	RST	Reset
3	GND	Ground
4	IRQ	Can be left unconnected
5	MISO	Master In / Slave Out
6	MOSI	Master Out / Slave In
7	SCK	Clock
8	SDA	Slave select or SDA. Quite why this is labeled SDA is a mystery as that is an I2C term and this board only supports SPI. It does seem to serve as a slave select.

Here is an example breadboard layout:



When connected, there is an LED on the board that lights when power is applied. Personally, I like it when working with new peripherals when they show an indication of life.

The data on the card is broken out into sectors, blocks and bytes. There are 16 sectors identified as 0 through 15. Each sector contains 4 blocks. These are identified as 0 through 3. Each block contains 16 bytes. If we look at the total ... 16 sectors \* 4 blocks \* 16 bytes we end up with 1024 bytes (or 1K).

See also:

- Wikipedia – [Radio-frequency identification](#)
  - [MFRC522 Data sheet](#)

- [How to use RFID-RC522 on Raspbian](#)
- [quick and cheap RFID with RPi and rc522](#)
- [RC522 reader for raspberry pi \(C\)](#)
- YouTube: [Tutorial:Using a RFID Card Reader with the Arduino](#)
- Github: [miguelbalboa/rfid](#) – Arduino library for MFRC522
- Github: [mxgxw/MFRC522-python](#) – Python class to interface with MFRC522
- Github: [ondryaso/pi-rc522](#) – Python library for RC522 module

## MFRC522 - Low levels

This is a 3.3V device, you will ruin it if you attempt to connect it to a 5V source.

The chances are that this section will be of little value to you. Most of the components I work with have relatively simple interfaces but this one has a lot of options and unless one wants to become a deep student of the device, it is likely that one of the pre-existing libraries will be what is most beneficial. However, it is my intent that, over time, I will read the data sheet carefully and combine that with the current known open source implementations and try and provide at least a readers guide to understanding the protocols and algorithms.

The device has a lot of registers:

Address	Name	Description
0x00	Reserved	Reserved for future use.
0x01	CommandReg	Starts and stops command execution.
0x02	ComlEnReg	Control of interrupt requests.
0x03	DivlEnReg	Interrupt request bits.
0x04	ComIrqReg	
0x05	DivIrqReg	
0x06	ErrorReg	Error status of last command executed.
0x07	Status1Reg	Status register.
0x08	Status2Reg	Status register.
0x09	FIFODataReg	
0x0a	FIFOLevelReg	Number of bytes in FIFO queue.
0x0b	WaterLevelReg	Threshold for FIFO queue under and over flow warning.
0x0c	ControlReg	Misc control bits.
0x0d	BitFramingReg	Bit oriented frames.
0x0e	CollReg	Collision detection handling.
0x0f	Reserved	Reserved for future use.
0x10	Reserved	Reserved for future use.

0x11	ModeReg	General transmit and receiver controls.																		
		<table border="1"> <thead> <tr> <th>Pin</th><th>Name</th></tr> </thead> <tbody> <tr><td>7</td><td>MSBFFirst</td></tr> <tr><td>6</td><td>N/A</td></tr> <tr><td>5</td><td>TxWaitRF</td></tr> <tr><td>4</td><td>N/A</td></tr> <tr><td>3</td><td>Polarity of MFIN</td></tr> <tr><td>2</td><td>N/A</td></tr> <tr><td>1:0</td><td>CRC Preset</td></tr> </tbody> </table>	Pin	Name	7	MSBFFirst	6	N/A	5	TxWaitRF	4	N/A	3	Polarity of MFIN	2	N/A	1:0	CRC Preset		
Pin	Name																			
7	MSBFFirst																			
6	N/A																			
5	TxWaitRF																			
4	N/A																			
3	Polarity of MFIN																			
2	N/A																			
1:0	CRC Preset																			
0x12	TxModeReg	Transmit rate.																		
		<table border="1"> <thead> <tr> <th>Pin</th><th>Name</th></tr> </thead> <tbody> <tr><td>7</td><td>TAuto</td></tr> <tr><td>6:5</td><td>TGated</td></tr> <tr><td>4</td><td>TAutoRestart</td></tr> <tr><td>3:0</td><td>TPrescaler_Hi</td></tr> </tbody> </table>	Pin	Name	7	TAuto	6:5	TGated	4	TAutoRestart	3:0	TPrescaler_Hi								
Pin	Name																			
7	TAuto																			
6:5	TGated																			
4	TAutoRestart																			
3:0	TPrescaler_Hi																			
0x13	RxModeReg	Receive rate.																		
0x14	TxControlReg	Controls antenna driver pins.																		
		<table border="1"> <thead> <tr> <th>Pin</th><th>Name</th></tr> </thead> <tbody> <tr><td>7</td><td>InvTx2RFOn</td></tr> <tr><td>6</td><td>InvTx1RFOn</td></tr> <tr><td>5</td><td>InvTx2RFOff</td></tr> <tr><td>4</td><td>InvTx1RFOff</td></tr> <tr><td>3</td><td>TX2CW</td></tr> <tr><td>2</td><td>N/A</td></tr> <tr><td>1</td><td>TX2RFE</td></tr> <tr><td>0</td><td>TX1RFE</td></tr> </tbody> </table>	Pin	Name	7	InvTx2RFOn	6	InvTx1RFOn	5	InvTx2RFOff	4	InvTx1RFOff	3	TX2CW	2	N/A	1	TX2RFE	0	TX1RFE
Pin	Name																			
7	InvTx2RFOn																			
6	InvTx1RFOn																			
5	InvTx2RFOff																			
4	InvTx1RFOff																			
3	TX2CW																			
2	N/A																			
1	TX2RFE																			
0	TX1RFE																			
0x15	TxASKReg	Transmit modulation setting. Bit 6 1 Forces a 100% ASK modulation																		
0x16	TxSelReg	Analog module control.																		
0x17	RxSelReg	Receiver settings.																		
0x18	RxThresholdReg	Thresholds for receiving.																		
0x19	DemodReg	Demodulator settings.																		
0x1a	Reserved	Reserved for future use.																		
0x1b	Reserved	Reserved for future use.																		
0x1c	MfTxReg	MIFARE transmit parameters.																		
0x1d	MfRxReg	MIFARE reception parameters.																		
0x1e	Reserved	Reserved for future use.																		

0x1f	SerialSpeedReg	UART serial speed.
0x20	Reserved	Reserved for future use.
0x21	CRCResultReg	CRC calculation.
0x22	CRCResultReg	CRC calculation.
0x23	Reserved	Reserved for future use.
0x24	ModWidthReg	Modulation width.
0x25	Reserved	Reserved for future use.
0x26	RFCfgReg	Receiver gain.
0x27	GsNReg	Conductance of antenna pins.
0x28	CWGsPReg	Detailed
0x29	ModGsPReg	Detailed.
0x2a	TModeReg	Timer settings.
0x2b	TPrescalerReg	
0x2c	TReloadRegH	
0x2d	TReloadRegL	
0x2e	TCounterValReg	
0x2f	TCounterValReg	
0x30	Reserved	
0x31	TestSel1Reg	Test signal configuration.
0x32	TestSel2Reg	Test signal configuration.
0x33	TestPinEnReg	
0x34	TestPinValueReg	
0x35	TestBusReg	
0x36	AutotestReg	
0x37	VersionReg	MFRC522 software version.
0x38	AnalogTestReg	
0x39	TestDAC1Reg	
0x3a	TestDAC2Reg	
0x3b	TestADCReg	

Be careful when using registers, they are encoded as follows:

7	6	5	4	3	2	1	0
1 = read	address				0		
0 = write							

As you can see, an address contains whether we are reading or writing to it ... it is also shifted one bit to the left.

To say that there is a lot of work in building access to RFID by hand is an understatement. However, by looking closely at the work already performed by others in various open source projects, we can get a feel for the steps of an implementation:

### Initialization

```
Reset
TModeReg      ← 0x8d - TAuto=1, non-gated, TPrescalerHi=b1101
TPrescalerReg ← 0x3e - Low 8 bits of TPrescaler
TReloadRegL   ← 30
TReloadRegH   ← 0
TxASKReg     ← 0x40 - Force 100% ASK
ModeReg       ← 0x3d - Not MSBFIRST, TxWaitRF, MFin active High, CRCPreset=0x6363
AntennaOn
```

### AntennaOn

```
If TxControlReg→Tx1RFEEn is off or TxControlReg→Tx2RFEEn is off
  TxControlReg→Tx1RFEEn = 1
  TxControlReg→Tx2RFEEn = 1
```

## Using Python

Once install via the instructions in the tutorials and as per the documentation found at the Github project, we can start dumping the content of RFID cards. Examine the sample called "Dump.py" to see an example.

This package **needs** a modification made to /boot/config.txt. Specifically, the following line must be added:

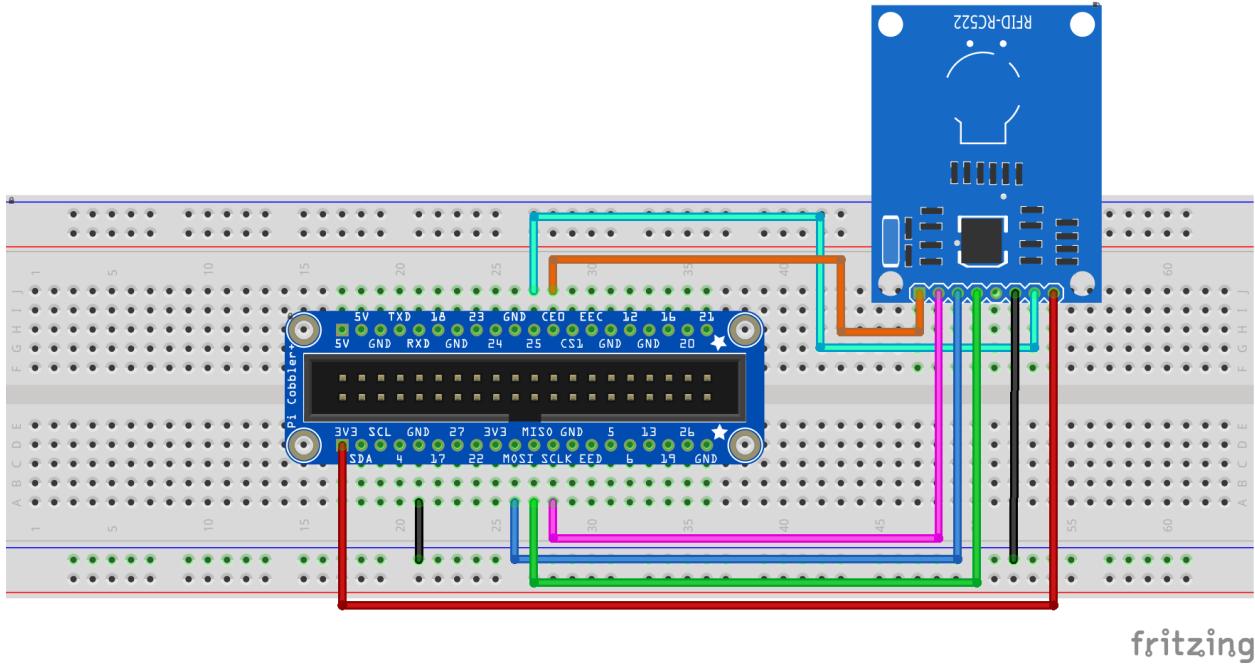
```
dtoverlay=spi-bcm2708
```

See also:

- [How to use RFIC-RC522 on Raspbian](#)
- Github: [mxgxw/MFRC522-python](#) – Python class to interface with MFRC522

## Using JavaScript

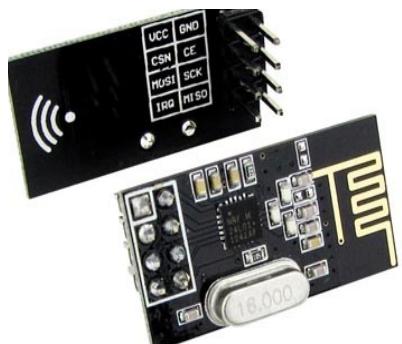
An effort was undertaken to port the Python MFRC522 library to JavaScript. This uses the following breadboard layout:



After some debugging and testing the results were successful. The project was submitted to the author of the Python package for his consideration as a parallel package for JavaScript.

## Communications - nRF24L01

The nRF24L01 is an ultra cheap communication device. It allows one to transmit and receive data over large distances (hundreds of feet). It transmits and receives packets between like devices. Do not confuse it with WiFi or Bluetooth. You can transmit from one nRF24 to another nRF24 but not to anything else. There are a number of reasons to consider this device for projects over others. First is the ultra low cost. An instance of the device can be picked up for less than a dollar. Second is the peer-to-peer nature of the devices. There does not need to be a master/slave relationship in the story. In fact, two or more devices can exchange packets with each other without any of them acting as controllers. There are occasions where this is exactly what is needed.



The device operates on 3.3V and uses SPI as a protocol. The pin-out on the nRF24L01 is:

Pin	Description
Vcc	3.3V
GND	Ground
MISO	SPI Master In/Slave Out.
MOSI	SPI Master Out/Slave In.
SCK	SPI System clock.
CSN	SPI Chip Select – Enable the device (active low).
CE	RX or TX mode. The Data sheet calls this Chip Enable which seems an awfully strange name for a communication mode selector.
IRQ	Maskable interrupt.

See also:

- Nordic Semi – [Home Page](#)
- [Data sheet](#)
- [Lamp Switch](#)
- [SPI – Serial Peripheral Interface](#)
- [Tutorial: Ultra Low Cost 2.4 GHz Wireless Transceiver with the FRDM Board](#)
- [Julian Ilett #1](#)
- [NRF24L01 - How To](#)
- [Adding a nRF24L01 to a breadboard or stripboard](#)
- [Tutorial 0: Everything You Need to Know about the nRF24L01 and MiRF-v2](#)

## Using the Arduino APIs

An Arduino library for the nRF24 is available called RF24. This library is highly active and well documented. It appears to be a fork of the RF24 library originally created by Manicbug. It has a detailed set of documentation including descriptions of all the class APIs.

A typical flow would be:

```
RF24 radio(CE_PIN, CSN_PIN);  
radio.begin();  
radio.setPALevel(RF24_PA_LOW);  
radio.openWritingPipe(addressTX);  
radio.openReadingPipe(1,addressRX);  
// To be a receiver ...  
radio.startListening();
```

To stop being a receiver, call:

```
radio.stopListening();
```

Since there are only two modes, a receiver and a transmitter, when one stops being a receiver, one immediately becomes a transmitter.

To determine if there is data available to receive, call:

```
radio.available()
```

To read data, we can call:

```
radio.read(&address, length);
```

To write, we can call:

```
radio.write(&address, length);
```

The library comes with some samples.

- `GettingStarted` – Two devices. One can act as the transmitter and one as the receiver. They can be flipped around through user input from the serial port.

Before the library can be used, an instance of it must be constructed. The constructor looks as follows:

```
RF24(cePin, csPin)
```

The pins for `ce` and `cs` need not be fixed and hence we must instruct the library as to which pins we actually used. Take care that when an instance is created you specify the pins in the correct order as there is no guard against that.

Before any other functions can be called, we must call `begin()`. This performs the initialization of the device plus sets defaults for many of the operational parameters.

Every device has an address associated with it. This allows a transmission from one source device to be addressed to a specific destination device. When a transmission occurs, devices other than the one with the matching address will ignore the transmission. The device requires the size of the address to be supplied and fixed. The allowable address sizes are 3, 4 or 5 bytes. The default is 5 bytes. The size of the address can be changed with the `setAddressWidth()` method however you are unlikely going to want to change the default value. You must remember to supply the full data for the address. For example, don't set an address of "123" when the address size is 5 bytes as your device address will become "123<?><?>" and you will likely waste time trying to find out why no data is coming to you.

At this point, we must consider what we are going to do with the device from here on. The device has two modes of operation. It can transmit data or it can receive data but it can't do both simultaneously.

Let us look first at being a transmitter. First we create a "pipe" over which the data will flow. We supply the address of the destination.

```
openWritingPipe(const unit8_t *address)
```

For example:

```
unit8_t address = "1Node";
openWritingPipe(address);
```

We are now ready to actually transmit some data.

```
bool write(const void *data, uint8_t length);
```

This method will transmit the data pointed to by the data pointer for the number of bytes supplied in length. There is a maximum size as specified by the `getPayloadSize()` method and you should not exceed that. The method will block until the data has been transmitted or the transmission request has timed out. The time out is short, about 70ms so you won't block for long. Upon return from the method call, we can determine whether the transmission was successful or if it failed. A return value of `true` means we succeeded while `false` means as failure.

The device has a maximum transmission size that is 32 bytes. You can not transmit packets larger than this. The default maximum transmission set in the library is 32 bytes which is the maximum that the device will permit. You can change this with the `setPayloadSize()` method but it is unlikely you will need that function. Simply remember that we can not ever send a payload size greater than 32 bytes.

The other mode of the device is that of a receiver where we actually listen for incoming data. As mentioned, the device can either be in transmit mode or receive mode.

To receive incoming data, we need to supply the address of the device from which we are expecting incoming data to arrive. We do this using the `openReadingPipe()` method.

```
openReadingPipe(uint8_t number, const uint8_t *address)
```

Once we have specified the address of the transmitting device we will receive data from, we can start actively listening using the `startListening()` method. Should we wish to end listening mode, we can call the `stopListening()` method which will return us to transmission mode.

Once we have entered listening mode, we can ask the library if there is data available for us to read. The method for this is `available()`. If there is data available, the return value is `true` and `false` otherwise.

To actually read the data that was received, we can use the method called `read()`.

```
read(void *buf, uint8_t len)
```

This method supplies a buffer into which the received data will be stored. The length of the buffer is also supplied.

A debug file called `printDetails()` is provided which logs the state of the nRF24 to the `stdout`. In order to use this one must:

- Include `<printf.h>`
- Call `printf_begin()`

When the `printDetails()` function is then called, a status of the nRF24 is written to the Serial port.

This can be a powerful tool for debugging problems. Here is an example of the output:

```
STATUS      = 0x0e RX_DR=0 TX_DS=0 MAX_RT=0 RX_P_NO=7 TX_FULL=0
RX_ADDR_P0-1 = 0x0045444f4e 0x0045444f4e
RX_ADDR_P2-5 = 0xc3 0xc4 0xc5 0xc6
TX_ADDR     = 0x0045444f4e
RX_PW_P0-6   = 0x20 0x20 0x00 0x00 0x00 0x00
EN_AA       = 0x3f
EN_RXADDR   = 0x02
RF_CH       = 0x4c
RF_SETUP    = 0x03
```

```
CONFIG      = 0x0f
DYNPD/FEATURE = 0x00 0x00
Data Rate    = 1MBPS
Model        = nRF24L01+
CRC Length   = 16 bits
PA Power     = PA_LOW
```

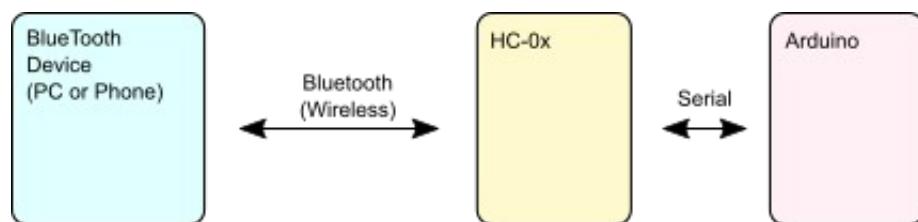
See also:

- [Github project](#)
- [Project Blog](#)
- [Forum thread](#)

## Bluetooth - HC-05/HC-06

Bluetooth is a wireless protocol used to connect devices to one another via a point-to-point link. One side acts as the master while the other side acts as the slave. Blue tooth masters are commonly cell phones or PCs or in car audio devices while Bluetooth slaves are commonly headphones, keyboards or mice. Through cheap bluetooth adapters we can connect the Pi as either a master or a slave.

In this section we discuss the devices called HC-05 and HC-06. Both of these are Bluetooth to serial converters. What that means is that data *arriving* at the HC-0x over Bluetooth will be made available via a serial stream while any serial data delivered to the HC-0x will be transmitted as Bluetooth data.

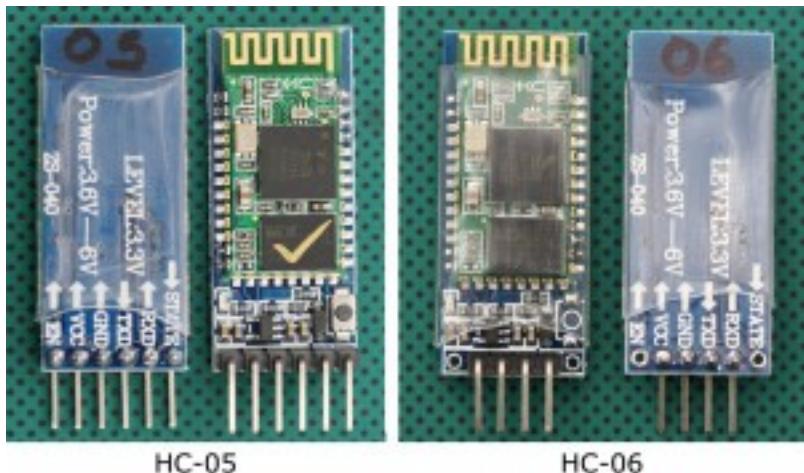
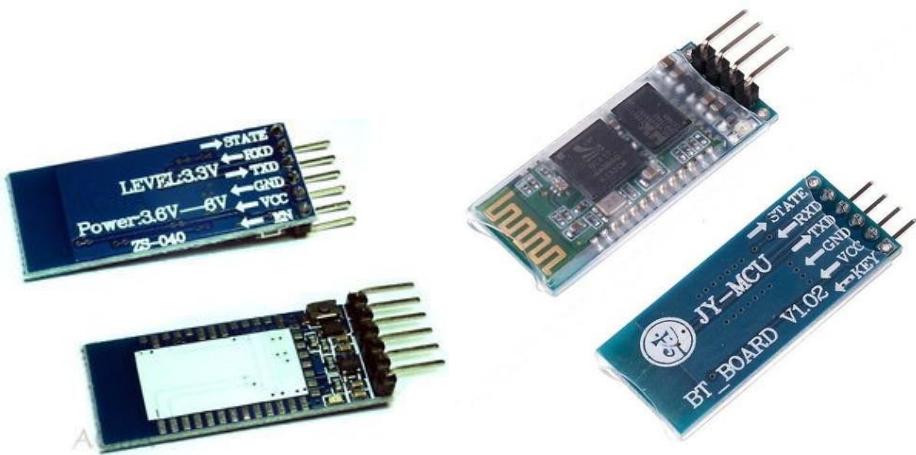


HC-05 can be configured as a Master **or** a Slave while the HC-06 is **only** a Master or a Slave. An HC-06-M is a master while an HC-06-S is a slave. These are hard-set during fabrication. In practice, the only HC-06's that are seen are slaves.

Since the HC-05 has more functions than the HC-06, the HC-05 should generically be used in preference however it may be more expensive (relatively speaking).

The mounting board for the device appears to be called a ZS-040.

These devices are 3.3V.



The module containing the device is called JY-MCU.

The default pin code is 1234.

Serial terminal parameters are:

Baud rate: 9600

Data bits: 8

Stop bit: 1

Parity: No parity

The HC-0x is controlled via AT commands. The HC-05 has more AT commands at its disposal than the HC-06.

	<b>HC-05</b>	<b>HC-06</b>
Bluetooth name	HC-05	linvor
Password	1234	1234
Default baud rate	9600	9600
Leds	Led1 <ul style="list-style-type: none"> <li>• Slow blink – AT mode 2</li> <li>• Fast blink – AT mode 1</li> </ul>	Led <ul style="list-style-type: none"> <li>• Blinking – AT mode</li> <li>• Steady – connected</li> </ul>

Notes:

- It appears that the HC-06 is in AT mode at power up until connected to a master.

A quick and easy way to test one of these devices is with nothing more than a PC and a USB-to-UART connector. We cross connect the RX/TX of the devices ensuring that there is only 3.3V arriving on the HC-0x RX pin. We can also get 5V or 3.3V from the USB device itself and hence power the HC-0x directly from the USB. The HC-0x will now start to blink signifying that it is ready for a connection. Since the HC-0x now appears as a COM port on the PC, attaching a terminal (Putty) to the COM port will allow us to send and receive data from the HC-0x. If we now run a Bluetooth terminal emulator (say on a phone or pad) and connect to the HC-0x, when we type on the Bluetooth terminal emulator, it will appear on the PC terminal. If we type on the PC terminal, it will appear on the Bluetooth terminal.

This was tested and worked first time.

AT Commands

Command	Description
AT	Test connection.
AT+VERSION	Return the version information.
AT+NAME<myName>	Set the device name.
AT+PIN<PinCode>	Set the device pin code.
AT+BAUD1	Set baud rate to 1200.
AT+BAUD2	Set baud rate to 2400.
AT+BAUD3	Set baud rate to 4800.
AT+BAUD4	Set baud rate to 9600.
AT+BAUD5	Set baud rate to 19200.
AT+BAUD6	Set baud rate to 38400.
AT+BAUD7	Set baud rate to 57600.
AT+BAUD8	Set baud rate to 115200.
AT+BAUD9	Set baud rate to 230400.
AT+BAUDA	Set baud rate to 460800.
AT+BAUDB	Set baud rate to 921600.
AT+BAUDC	Set baud rate to 1382400.

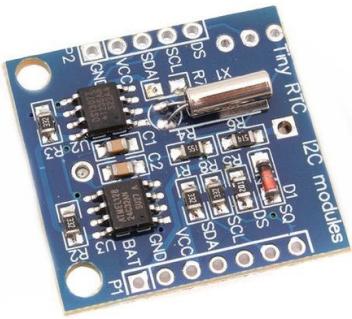
See also:

- [HC Serial Bluetooth Products – User Instructional Manual](#)

## Real time clocks

When we start a Pi, it doesn't know what the real time is. We can tell it once booted but even then, once we power it off, it will forget again. If the Pi is connected to the Internet, it can learn the real time by using a network time protocol. However, if our Pi is not connected to the Internet and is network isolated, by default, it has no mechanism to know the current time. This is what we can add a real-time-clock (RTC) module. A real-time clock module is a small piece of electronics that contains a battery and is taught the current date and time. Because it has an on-board battery, it can remember the time even when the Pi is powered off. When the Pi boots, it can be configured to ask the RTC module for the current date/time and use that from then on.

There are many RTC modules available but one of the more prevalent is the DS1307.



These can be picked up on eBay for about \$1. The RTC uses battery backup when not powered. This is a standard CR2032 type. When inserting the battery, it is +ve down. When soldering on header pins, think through how you will mount the board when complete. It may be that you want to solder the header pins such that you can easily access and replace the battery once attached to your PCB or strip board.

This device provides a real time clock capability that includes hours, minutes, seconds, month, day of month, day of week and year. Month calculations including leap year support are also accommodated. Interestingly, it also provides 56 bytes of non-volatile RAM. The device uses I2C for communication.

Note: Before plugging your board to the Pi, realize that this is a 5V module. If you connect 5V to the Pi's GPIO pins bad things can happen because they are 3.3V pins. The I2C protocol uses pull-up resistors to bring the SDA and SCL lines high by default. If we were to look at the schematic diagram of this module we would find that resistors R2 and R3 are pull-up resistors. Unfortunately, we have two problems. First is that the Pi already pulls up the pins and secondly the module pulls them high to a 5V line which is too much for the Pi. The solution is to physically remove these two resistors from your module before use. The resistors are clearly marked on the board as R2 and R3 and carry the mark 332 (3.3k).

The boards have 5 pins on one side 7 on the other.

<b>Pin</b>	<b>Label</b>	<b>Description</b>
1	DS	DS18B20 Temp.
2	SCL	I2C clock.
3	SDA	I2C data line.
4	Vcc	Vcc (5V).
5	GND	Ground.

Pin	Label	Description
1	SQ	Square wave output.
2	DS	DS18B20 Temp.
3	SCL	I2C clock.
4	SDA	I2C data line.
5	VCC	Vcc (5V).
6	GND	Ground.
7	BAT	Battery voltage

Once plugged in, we will see the RTC at I2C address 0x68:

```
$ i2cdetect -y 1
      0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- --
10:          -- -- -- -- -- -- -- -- -- -- -- --
20:          -- -- -- -- -- -- -- -- -- -- -- --
30:          -- -- -- -- -- -- -- -- -- -- -- --
40:          -- -- -- -- -- -- -- -- -- -- -- --
50:          -- -- -- -- -- -- -- -- -- -- -- --
60:          -- -- -- -- -- -- 68 -- -- -- -- --
70:          -- -- -- -- -- -- -- -- -- -- -- --
```

Once connected, it is time to start looking at the software side of the house. Jessie provides a kernel level module to integrate with the clock but it is not enabled by default. To do this we must edit /etc/modules and add a line to load the clock module. Here is an example:

```
i2c-bcm2708
i2c-dev
rtc-ds1307
```

Note that it must come after the modules relating to I2C.

Next we must tell the I2C subsystem that an RTC clock is present and tell it what I2C address it is present on. We do that by echoing "ds1307 0x68" into the file /sys/class/i2c-adapter/i2c-1/new\_device. For example:

```
$ sudo bash
# echo "ds1307 0x68" > /sys/class/i2c-adapter/i2c-1/new_device
```

Following this we will find some new device driver files up in /dev. Namely we will find /dev/rtc and /dev/rtc0.

We can now ask the hwclock what time it thinks it is with:

```
$ sudo hwclock -r
Tue 05 Jan 2016 07:24:22 PM CST -0.156382 seconds
```

If the clock has not been set, then it will be some odd time back in the past. If so, we can set the hardware clock from the current date/time with:

```
hwclock --systohc -D --noadjfile -utc
```

and if we re-run `hwclock -r` we will see the proper time. At this point, the `hwclock` is set and keeping real-time. When next the Pi reboots we need to instruct it to read the time from the `hwclock` which will have kept it for us and sync the system time from that value. This means that the Pi has to re-associate the `hwclock` with I2C and then execute:

```
hwclock -s
```

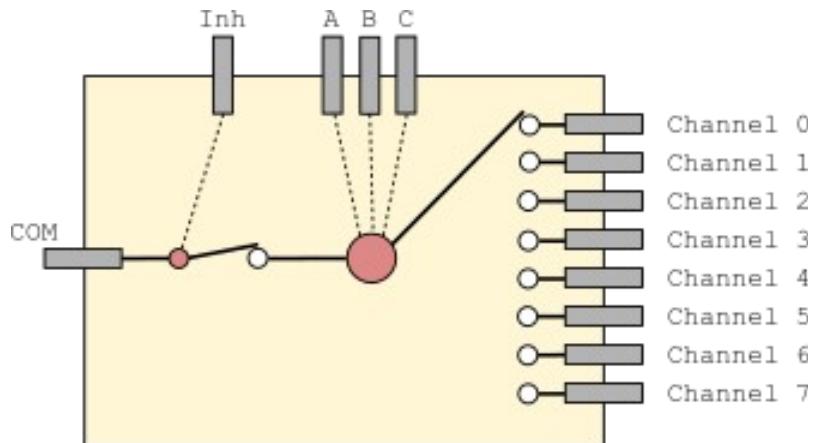
This will synchronize the `hwclock` back to the OS date/time.

See also:

- Instructables – [Set up Real Time Clock \(RTC\) on Raspberry Pi](#)
- Adafruit – [Adding a real-time-clock to a Raspberry Pi](#)
- Tutorial – [Using DS1307 and DS3231 Real-time Clock Modules with Arduino](#)
- [Maxim integrated datasheet](#)
- [Raspberry Pi – Adding a Real Time Clock \(RTC\)](#)
- man(8) – [hwclock](#)

## CD4051 - Digital switches

Many of the previous hardware integration examples have demonstrated how to configure the Pi with modules that are themselves pre-built circuits. Here we are going to discuss a single raw integrated circuit called the CD4051. Officially, it is described as an analog multiplexer/demultiplexer ... however them there are some big words. What it really is is a digital switching circuit. Logically, you can think of it in terms of the following diagram:



There is a common pin from which data can be either read or written. That pin will be connected internally to **one** of the channel pins (0 through 7). The selection of which pin is governed by the digital signals on A, B and C. The 3 bits of information there select the corresponding channel. Finally there is an inhibit pin which if set will prevent any connection between the input and the output. The connections between the common pin and the channel pin is fully analog ... it can be used to provide a digital signal or an analog signal ... in either direction.

With this background, how then might we use such an IC? Imagine we only have one Analog to digital convertor input on our Pi but we want to take readings from multiple sensors. We could switch between each of the sensors taking a reading before moving onto the next.

The CD4051 is very cheap, under \$0.40 an instance as found on eBay.

Its physical pin layout is:

Pin	Description
1	Channel #4
2	Channel #6
3	Common In or Out
4	Channel #7
5	Channel #5
6	Inhibit
7	Vee (Ground)
8	Vss (Ground)
9	Input C
10	Input B
11	Input A
12	Channel #3
13	Channel #0
14	Channel #1
15	Channel #2
16	Vdd (+ve)

See also:

- Texas Instruments – [CD4051 data sheet](#)
- YouTube: [4000 Series Logic ICs: The 4051 8-channel analog multiplexer/demultiplexer IC](#)

## Robotics

I worried about titling a section "robotics" because what we will end up with will be nothing like R2D2, C3PIO, the Terminator or any other "thing" that science fiction would have us desire or believe. However, just as the current controversy on two wheeled balanced boards being called "hover boards" ... we still know what they mean. For our discussion, robotics here will be synonymous (rightly or wrongly) with movement or control of movement. Examples would include wheeled devices (including wheels, treads, spiders etc), arm control (eg. a crane), drones (flying) and more. By itself, robotics and the Pi is as huge a topic as any possibly might be, so we will take our time and cover what we can.

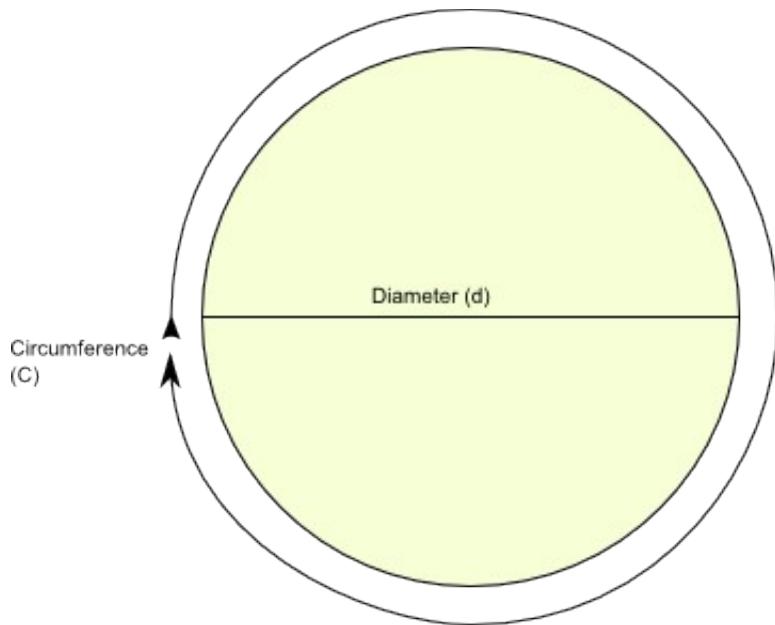
## Locomotion

Imagine we wish our Pi to move. The simplest story would be to put some wheels on it. However, a wheel by itself doesn't cause any motion. There has to be a drive that will turn the wheels. This will be a motor of some sort. Let us now peer at the qualities of a turning wheel. There are primarily two ... speed and torque. Speed can be measured as how fast it takes a wheel to rotate about its axis. Another way of saying this is how long it takes for a wheel to perform one revolution. The faster a wheel spins, the shorter the time to complete the revolution. When we have a wheeled Pi, the speed of the wheel governs how fast (or slow) the robot will move.

The second concept we need to look into is the notion of torque. This one is a bit more subtle. We can define it as the amount of force that the wheel needs to supply to move. Each one of us has a certain strength. I can lift a 1 pound weight easily. I can just about lift a sack of potatoes if it were about 50 pounds. I am not sure (if or how long for) I could lift a 100 pound sack of cement. This is our analogy of torque. Each motor is able to provide a rotational force to a wheel (its strength) but the mass of the Pi, chassis and other components may be too much and the motor may not be strong enough to even move it never mind move it at the desired speed.

There are a number of different types of motors available to us. The simplest is the DC motor. This is supplied a direct current and the motor freely spins. Typically it rotates at high speed which I define to mean 1000's of times per second. This is obviously no use to our wheels. We might not know exactly how fast we want them to spin, but 1000's of times per second is far too fast. The solution to this is to introduce the notion of a geared motor. A geared motor introduces different resolutions of gears such that 1 turn of the motor shaft might be 1/16 or 1/32 or 1/64 (or some other metric) of a wheel rotation. This divides down the speed of rotation but also acts as a torque enhancer. In loose principle, if I can "just about" lift 100 pounds of weight with a 10 foot lever then if I used a lever 10 times as long (100 feet) I could lift 1000 pounds ... it would still require the 10 times the exertion of 100 pounds but it would be spread out over a longer (and hopefully sustainable) period. DC motors are very cheap and easy to control. Their downsides are that they are not precise. If we supply the same power source to two instances of a DC motor, it is my experience that they will rotate at slightly different speeds. The reason this is a problem is that we typically have multiple motors in play that cause our Pi to move. Usually at least one on the left and one on the right. If their speeds are even just a little distinct from each other, the chassis will skew in forward movement either to the left or to the right.

The diameter of the wheel has a direct bearing on the speed of the chassis. To see this, let us look at a simple diagram of a wheel:



Some simple math gives us the circumference ( $C$ ) =  $\pi * \text{diameter } (d)$ . So if we measure the diameter of our wheel, we can determine the circumference. If we took the circumference as a strip of paper and wrapped it around the outside of the wheel, then in one revolution, the wheel would have laid down one circumference worth of distance.

Other than DC motors, we have other choices. The next we will look at is the stepper motor. The stepper motor offers ultra fine grained control over movement. Common stepper motors allow us to move the wheel in increments of 1/4096th of a degree. That is a ridiculously fine granularity and much more than I can imagine we would ever need.

### The H-Bridge

When we think about a regular DC motor, we realize that if we apply power to it, it starts to rotate. Great. However for many of our projects, we wish the motor to rotate both backwards and forwards and not just in one direction. Rotating in the forward direction will rotate the wheel one way while rotating in the backwards direction will rotate the wheel the other way. To rotate a DC motor in the opposite direction means that we have to reverse the polarity of the input. In addition, DC motors require lots of power. We simply can't drive the motor from the output of pins. It might be tempting to connect the inputs to a motor to two GPIO pins and drive one high and the other low to rotate in one direction and logically change their outputs to switch the motor to rotate in the other direction. Simply put ... don't do it. You are pretty much guaranteed to ruin your Pi. However there is a solution through what is called the H Bridge. The H bridge is called such because of the typical appearance of it when shown in a schematic in the circuit diagram. The most common IC for an H bridge is the L293D which provides an implementation of **two** H bridges in one IC. You will typically need two anyway as you are likely to have at least two motors (one for the left wheel and one for the right). If we look at just one H bridge, it takes as input two input signals called IN1 and IN2. These signals are then used to control the output signal on OUT1 and OUT2 ... however the IN1 and IN2 can be simple digital signals as can be found on the output of a Pi GPIO. The OUT1 and OUT2 can be sourced from much higher voltages and currents. This means

that there is a decoupling between the signal used to control the polarity of the output from the voltage delivered at the actual output. In addition, the L293D includes internal diodes to handle the reverse current induced when a motor stops. Finally, there is a third input which is called ENABLE1. ENABLE1 is high, the outputs are driven however if ENABLE1 is pulled low, then the outputs are disabled. The value of this is that we can then use PWM to generate pulses of output ... and this can be used to control the speed of output of the motors.

The L293D has the following connections:

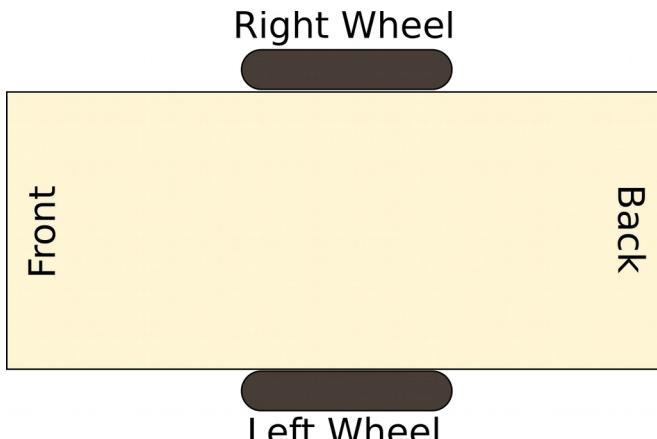
Label	Description
IN1	Input signal used to drive OUT1
IN2	Input signal used to drive OUT2
ENABLE1	Enable OUT1 and OUT2
OUT1	Output to motor drive by IN1
OUT2	Output to motor drive by IN2
IN3	Input signal used to drive OUT3
IN4	Input signal used to drive OUT4
ENABLE2	Enable OUT3 and OUT4
OUT3	Output to motor drive by IN3
OUT4	Output to motor drive by IN4
Vs	Logic source
VSS	Motor source
GND	Ground

See also:

- [Wikipedia – H bridge](#)

## The robot project

From a purely schematic perspective, our robot will have two driving wheels. This does not mean that it will balance on only two wheels, there will be other mechanisms to stop it tipping over. However, to keep the model in our heads, the following is how we should think of it:



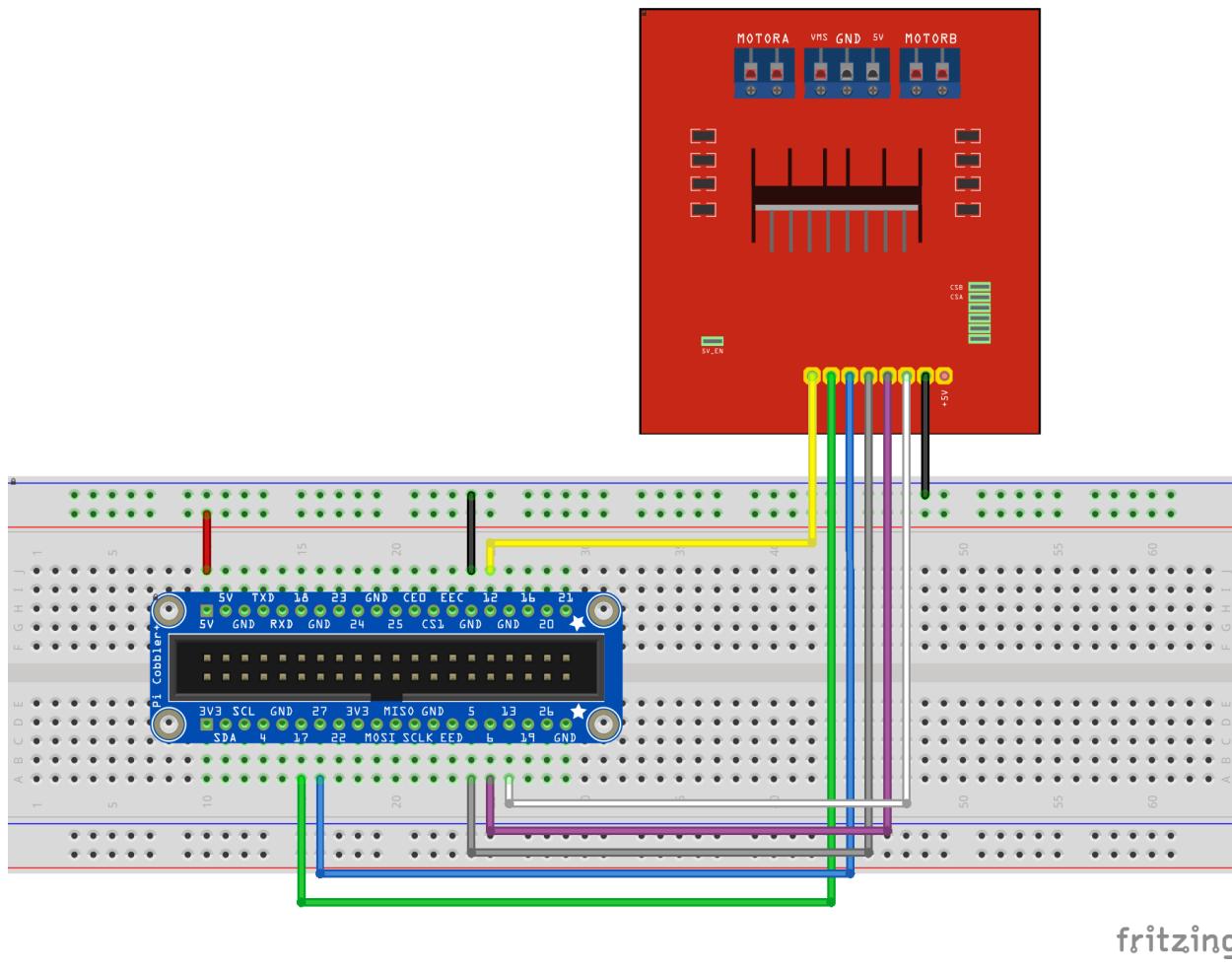
With this model in mind, we can now start to talk about the rotation of the wheels. Since a wheel can rotate in one of two directions at a time (clock-wise and counter clock-wise) we will say that a wheel spins "forwards" if doing so would move the robot in the direction we label as front and a wheel spins "backwards" if it would move the robot in the direction we label as back. Some though will show that the actual rotation of the wheels will be opposite from each other if both move forwards or backwards ... but now we have a way of thinking in terms of wheels having a "goal" rather than in simply rotating in one direction or another. We can now think of a wheel as moving the robot forwards or moving it backwards.

We need two pins per motor for setting motor inputs IN1 and IN2 (Motor 1) and IN3 and IN4 (Motor 2). We also need two PWM pins to control ENABLEA and ENABLEB. For the motor input pins, those need only be basic GPIOs.

We will choose the following:

Function	Pin	Wire color
IN1	GPIO17	Green
IN2	GPIO27	Blue
ENABLEA	GPIO12 (PWM0)	Yellow
IN3	GPIO5	Grey
IN4	GPIO6	Purple
ENABLEB	GPIO13 (PWM1)	White

here is an example breadboard using this wiring model:



The motors attached to the board were the cheap mass produced DC gear kind:



We will use JavaScript as the programming language for controlling the robot. Obviously there are many languages available to us however JavaScript offers one of the easiest.

To make a wheel spin, we need to apply opposite signals to its inputs. For example setting `IN1=1` and `IN2=0` will cause the wheel to spin in one direction while `IN1=0` and `IN2=1` will cause it to spin in the

opposite direction. Setting both values for input the same will cause it to stop. Either both 0 or both 1 will work the same so we will choose both 0 as the stop indication.

When opposite signals are applied to the inputs, the wheel will spin at maximum rate. In order to control the speed, we need to pulse the enable value from 0 to 1 and back to 0. The time spent at 1 will be the time the motor is powered while the time spent at 0 will be the time the motor is unpowered. As we vary the relative frequency of on to off we will effectively be controlling its speed. Since we are pulsing signals on and off at a given frequency, this is a perfect opportunity to utilize pulse-width modulation (PWM). The Pi has two PWM channels called `PWM0` and `PWM1`. We configure `ENABLEA` to `PWM0` and `ENABLEB` to `PWM1`.

From a function perspective, we wish the following capabilities to be exposed:

- `speed(speedValue)` – Set the speed. A value between 0 (stopped) and 1 (maximum).
- `stop(wheel)` – Stop the wheel from turning.
- `forward(wheel)` – Set the wheel turning forward.
- `backward(wheel)` – Set the wheel turning backward.

The following constants are also provided:

- `LEFT_WHEEL` – The symbolic that represents the left wheel.
- `RIGHT_WHEEL` – The symbolic that represents the right wheel.

### Robot - Network commands

It is a good assumption that we will want to both send requests to the robot to control its operation as well as receive telemetry from it. We will assume that the Pi will have a network adapter on it. How then do we wish to send and receive commands? In this example, we will use MQTT protocol and Mosquitto as the MQTT broker. We start by installing Mosquitto.

Next, we need to look at the content of command messages that can be received. These messages will be published to the MQTT topic "`robot/command`". Since we are building JavaScript applications, JSON will be a great format. We now define the format of a JSON object:

```
{  
  command: <Command>  
  speed: <speed>  
}
```

The set of commands we will support are:

- `FORWARDS` – Make the robot go forwards
- `BACKWARDS` – Make the robot go backwards
- `STOP` – Make the robot stop

Since we wish to our robot controller to behave as an MQTT subscriber, we need to install the NPM package called "`mqtt`".

See also:

- Mosquitto MQTT
- Node.js JavaScript – MQTT

### Balancing wheel power

When you buy something that is cheap relative to alternatives in the market, there is likely a good reason for that. DC motors come in all shapes, sizes and prices. Some of the robot motors with gears seem extremely cheap and there is a good reason for that ... they are likely not of as high quality as more expensive ones. Consider this to be under the heading of "you get what you pay for". For our robot story, this manifests itself in variability of speed and power between motor instances and this manifests as an ugly problem. If we supply the same voltage to two distinct motors, we would like them to spin at the same speed with the same power ... but due to the distinctions between them, they will likely differ. What that means to us is that one motor will spin faster than the other with the result that our robot will turn in circles when what we really wanted it to do was move in a nice straight line. The radius of the circle will depend on the relative speed of the two motors. The closer they are to the same speed, the wider the circle.

Fortunately there is a solution. What we can do is adjust the speed of the motors such that they get closer together. If two motors are given maximum voltage and they spin at different rates, then we have to reduce the voltage of the faster motor to match the speed of the slower. If we are using PWM, then this will be the reduction of the PWM value for the faster wheel.

By using networking and the ability to send distinct speed control commands to each wheel while the robot is moving, we can dynamically try different values until we find ones that work.

### Compass bearing

Given that we are able to attach solid state compass sensors to our Pi, we can determine which direction the robot is facing. If we know which direction magnetic north is then we can say that the front of the robot is some number of degrees off that direction. As the robot continues to move forward and we are hoping for straight line movement, we can be watching for change in the orientation of the robot. If this occurs, we can then dynamically alter our relative wheel speeds to bring ourselves back in line. If our movement is smooth enough and our sensor sensitive enough then, in principle, our robot can self correct its own course.

## **Windows IoT**

The Windows IoT allows us to run a version of Microsoft Windows called Windows 10 IoT Core on a Pi 2. Note that the Pi 2 is mandatory, Windows 10 IoT Core will not run on Pis smaller than thus. In order to use it, you will need an empty 8GByte SD card.

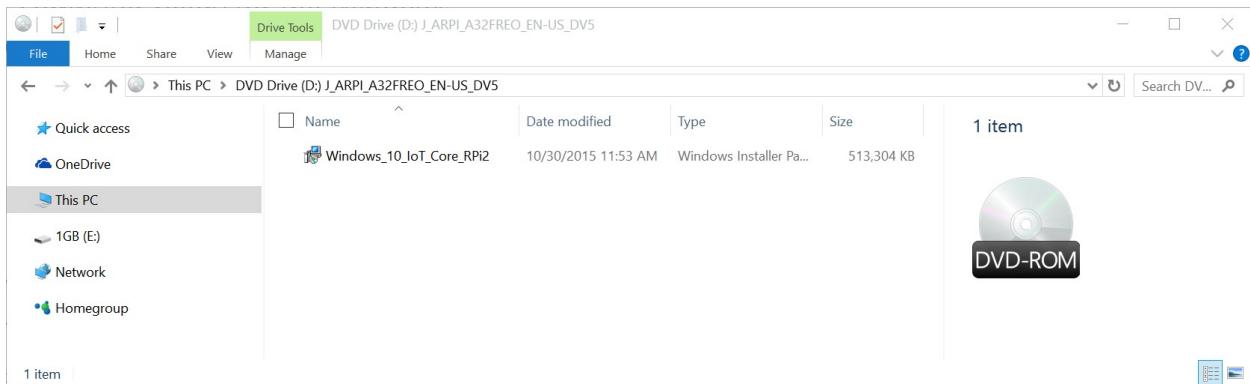
To start, we must download the Windows 10 IoT core from here:

<http://ms-iot.github.io/content/en-US/Downloads.htm>

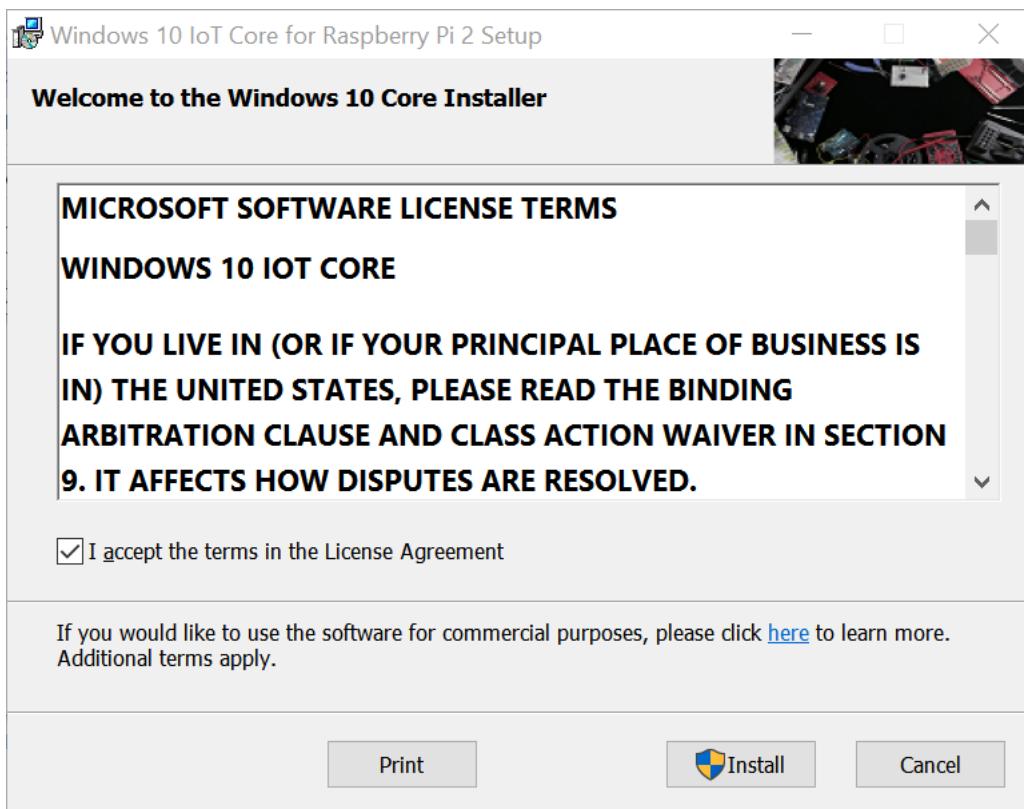
The download is 502MBytes.

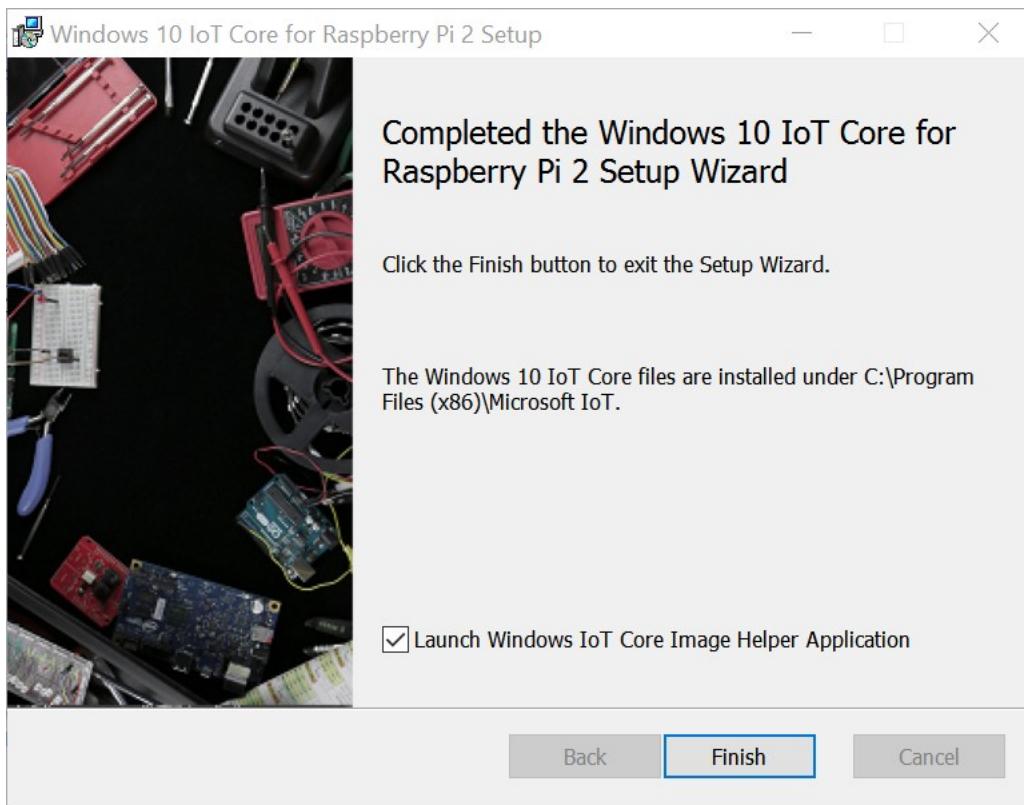
The result will be a file that is a ".iso" file that is a virtual disk that can be mounted.

Double clicking the ISO file will mount it as a virtual drive. From there we will find an installer:



Now we run the installer and we will be presented with an installer wizard:

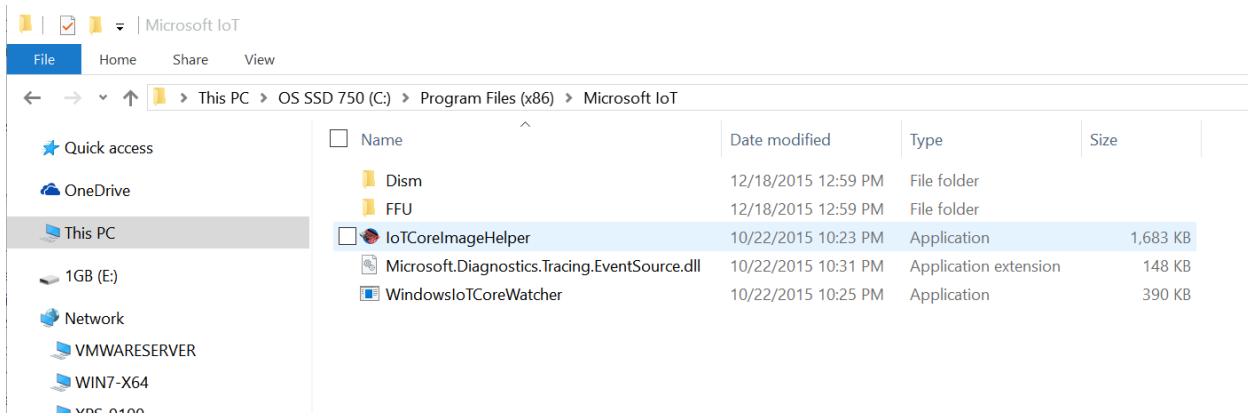




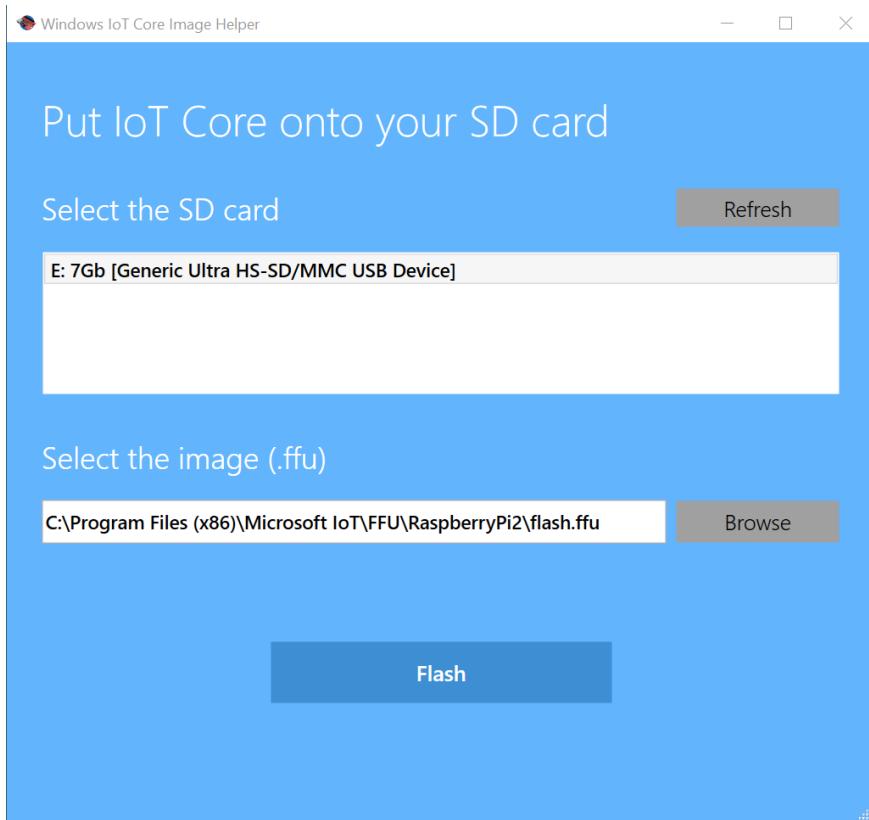
At the completion of the installation, the files will be found at:

C:\Program Files (x86)\Microsoft IoT

The program called `IoTCoreImageHelper` is the tool to build a SD image for the Pi.



After launching the Image Helper tool, we get to specify the flash file that we wish to load into the SD Micro card:



A DOS window appears showing progress:

```
C:\Program Files (x86)\Microsoft IoT\dism\dism.exe
Deployment Image Servicing and Management tool
Version: 10.0.10586.0
Applying image
[=====          15.0% ]
```

At the completion, we will have a confirmation message:

## Completed!

Windows IoT Core is now onto your SD card. Go ahead and plug it into your device.

Ok

After inserting the new card in your Pi, you can boot it up. Be aware that on the first boot, it seems to take some minutes to complete the start-up.

You will most certainly want to connect your device to the network either using Ethernet or one of the supported WiFi dongles.

Once started, you may have to configured your WiFi credentials in order to access the network.

See also:

- [Windows IoT](#)
- [Forum on Windows IoT on the Pi](#)

## Connecting the dashboard

The dashboard is a web based application that you can use to manage your device. It can be accessed by opening a browser and pointing it to:

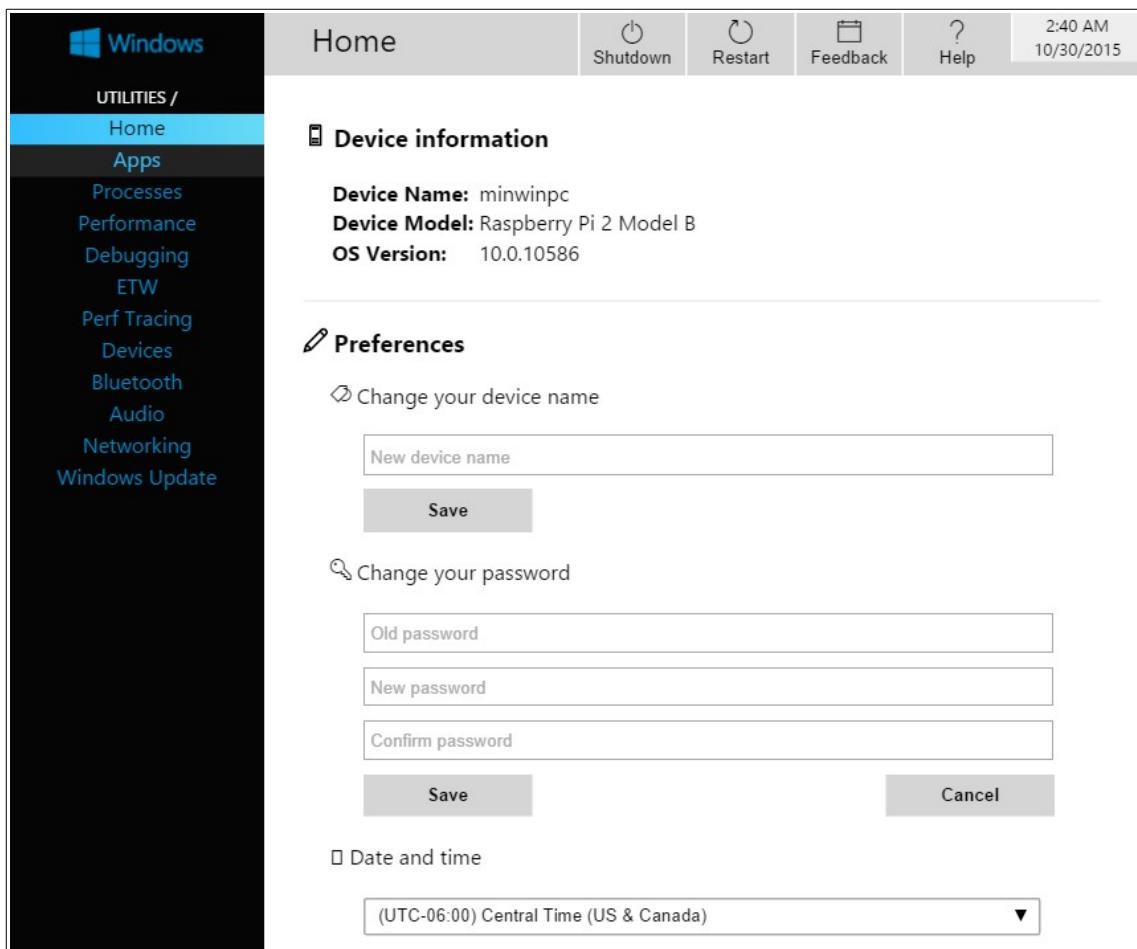
`http://<Your device IP address>:8080`

You will be prompted to login. The default userid and password are:

userid: Administrator

password: p@ssw0rd

Once logged in you will see a screen that looks as follows:



Notice the tabs on the left. Each one of these corresponds to its own configuration settings screen. We will discuss the pages as and when we come to settings we need within them.

At this point though I do recommend that you rename your device. I changed mine to "windows10iot" and that is what you will see in the remainder of the examples. You will need to restart your device after changing the name.

## Opening a shell to the Windows 10 IoT

Start a Windows PowerShell as administrator.

Start the Windows Remote Management demon with:

```
net start WinRM
```

Next, we need to set the device as a trusted partner with:

```
Set-Item WSMAN:\localhost\Client\TrustedHosts -Value windows10iot
```

Now we are ready to start a remote session to the device. The command to achieve that is:

```
Enter-PSSession -ComputerName windows10iot -Credential windows10iot\Administrator
```

You will be prompted for credentials. Now the PowerShell appears to freeze for 30 seconds or more. This is ok, connection is being made. After a period of time we will see the prompt.

```
[windows10iot]: PS C:\Data\Users\Administrator\Documents>
```

From within the PowerShell you can perform all kinds of administrative commands.

### Process management commands

- get-process
- tlist
- kill

The TList command lists the running tasks (processes). There are many options to this command to show the output in various levels of structure and detail.

### Network commands

- ping
- netstat
- netsh
- ipconfig
- nslookup
- tracert
- arp

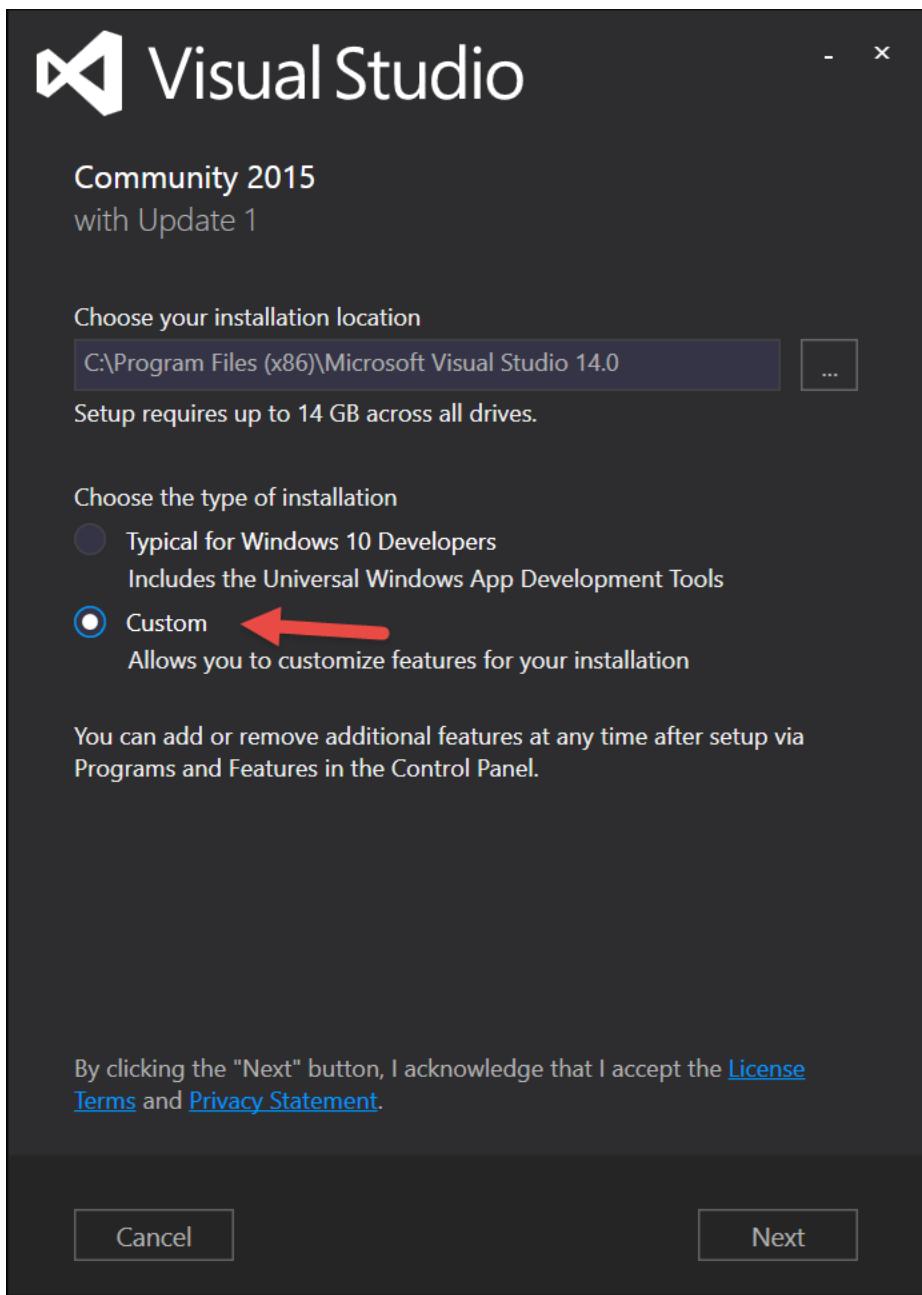
See also:

- msdn – [TList command](#)

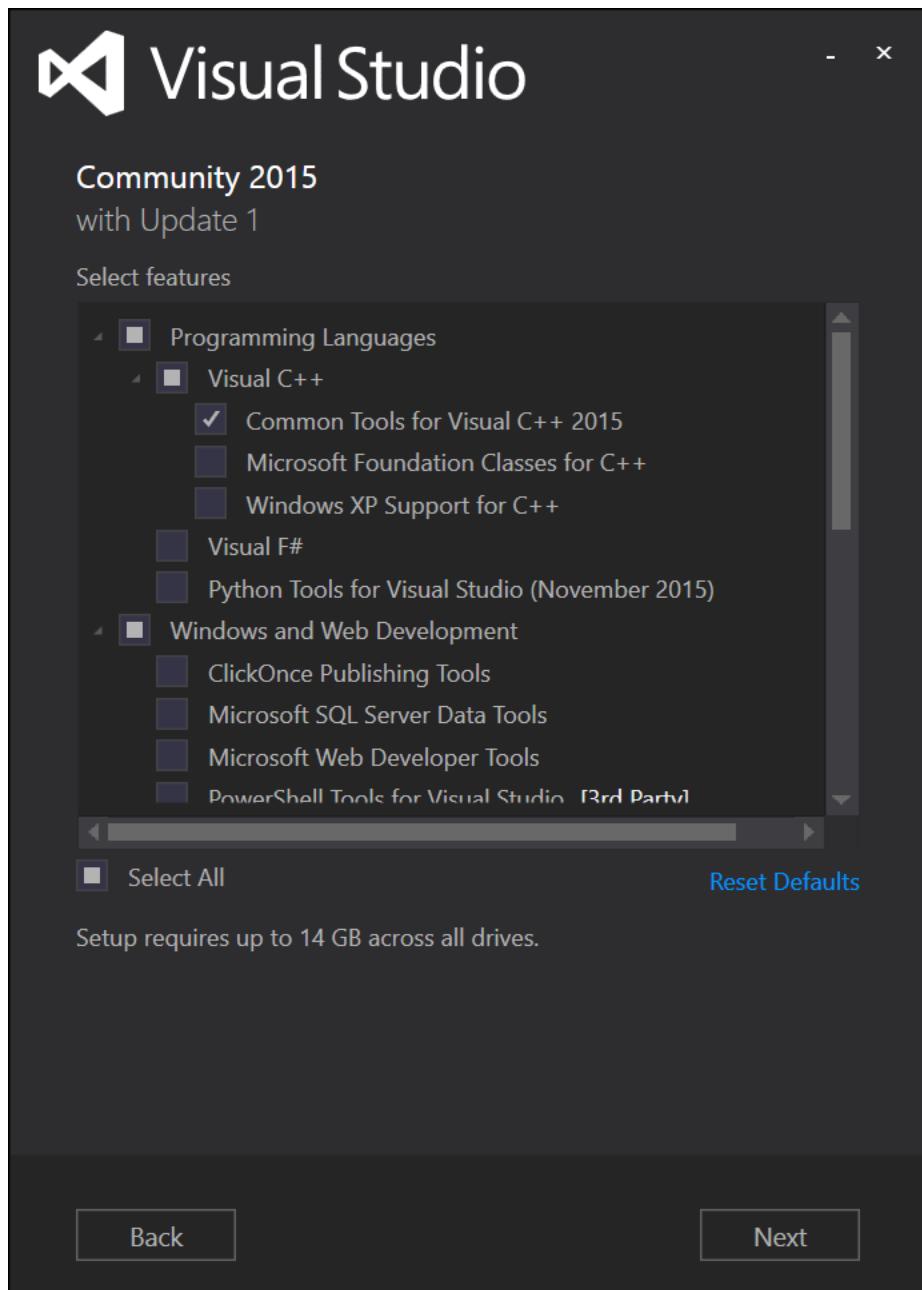
## Installing Visual Studio Community 2015

To develop applications, you will need to install the Visual Studio Community 2015 product.

Select the Custom installation.



Click Next to move onto the component selection.



In the Component selection ensure that the following are selected:

- Programming Languages → Visual C++ → Common Tools for Visual C++ 2015
- Tools for Universal Windows Apps (1.2) and Windows 10 SDK



# Visual Studio

Community 2015

with Update 1

Selected features

## MICROSOFT SOFTWARE

Windows 8.1 SDK and Universal CRT SDK

Common Tools for Visual C++ 2015

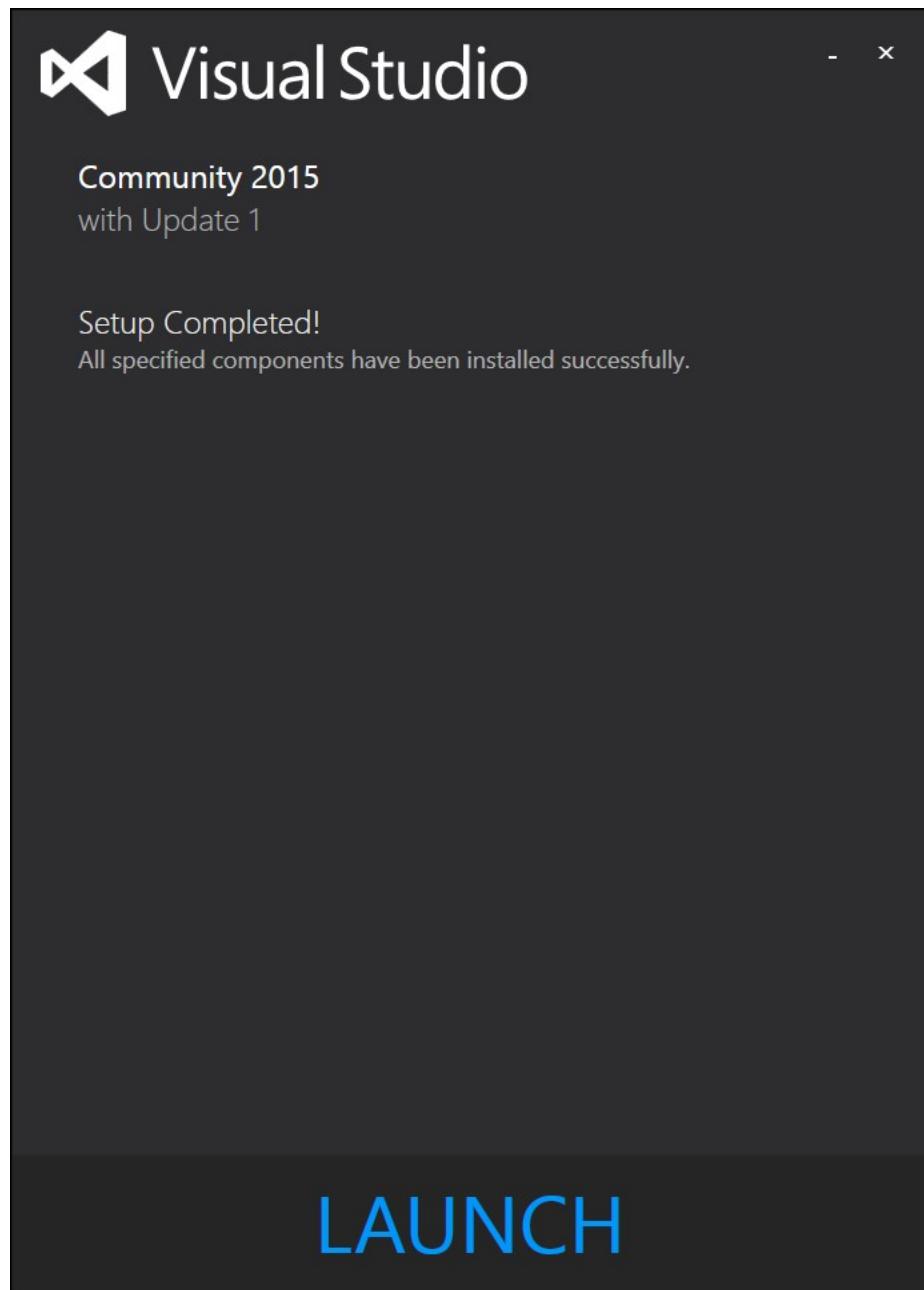
Tools for Universal Windows Apps (1.2) and Windows 10 SDK (10.0.10586)

Setup requires up to 14 GB across all drives.

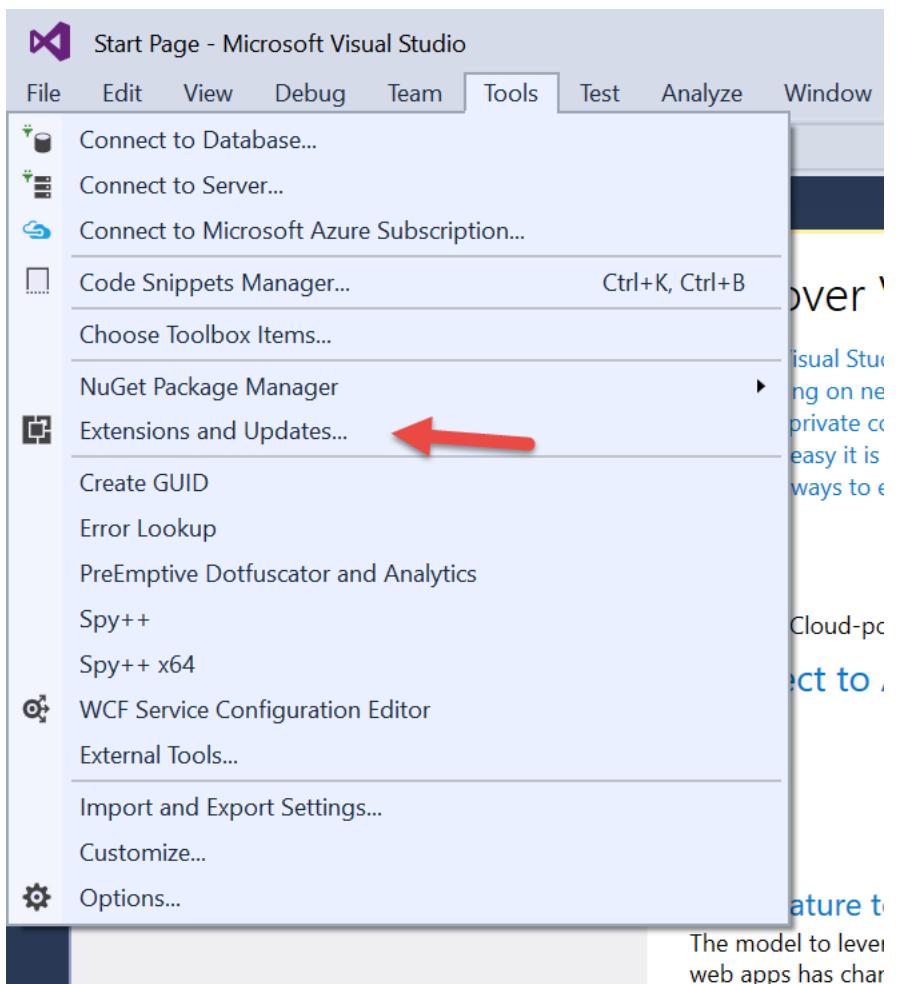
Back

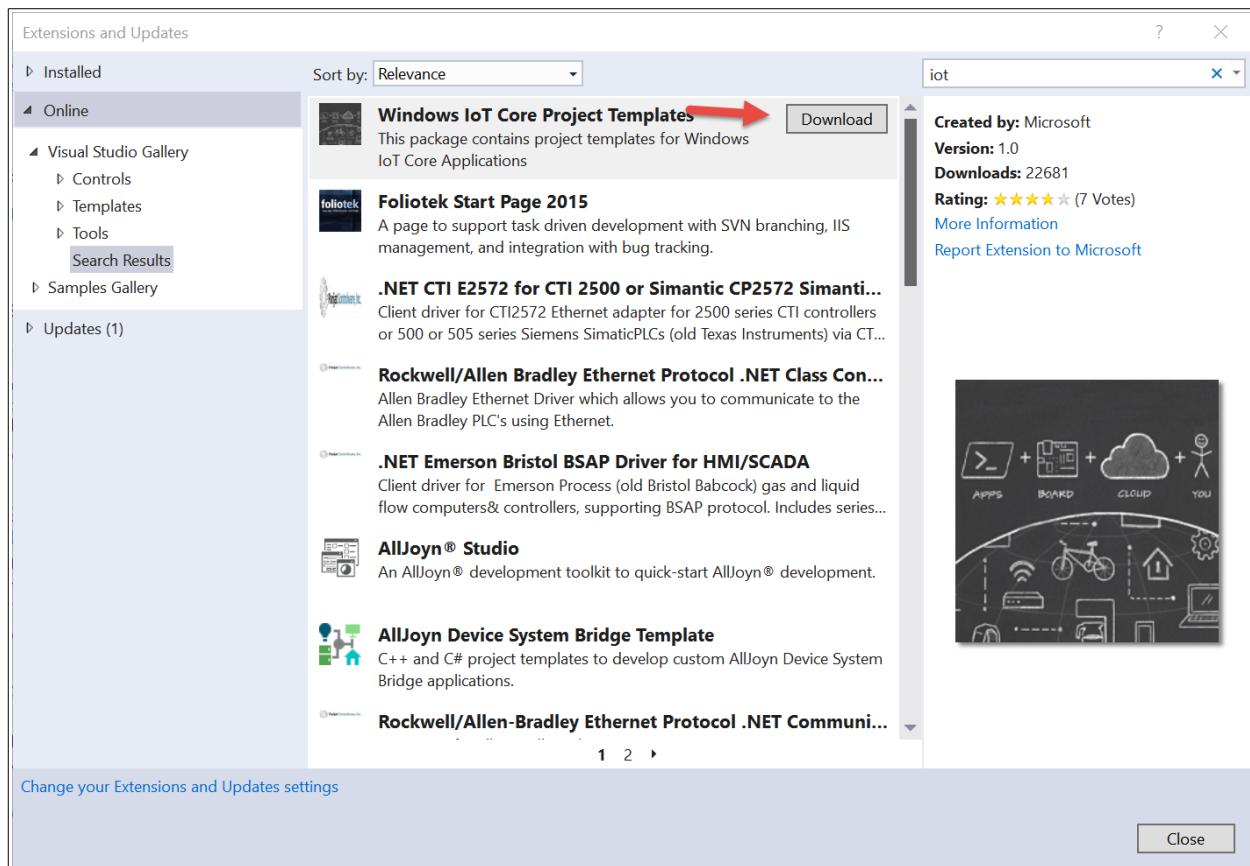
 Install

This may take some time to complete. After final installation, we will see:



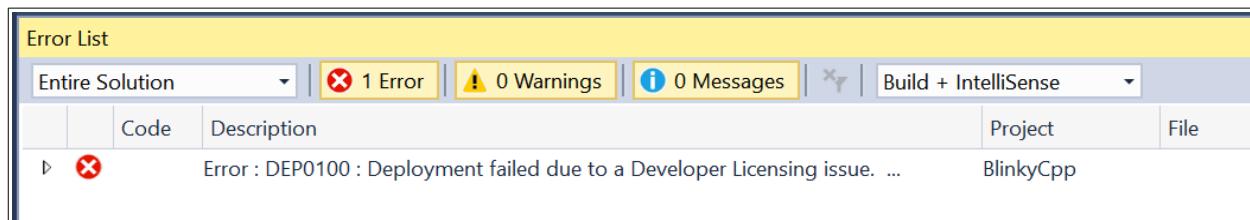
Once launched we need to install the Windows IOT templates. From the Tools menu select Extensions and Updates....





To validate that it installed, see the "Installed" category and then the "tools" sub category.

Finally, on your PC where Visual Studio is installed, you must flag it as being in "Developer mode". Should you omit that step, Visual Studio will warn with:



## Running samples

A wealth of Windows 10 IoT samples have been provided on Github here:

<https://github.com/ms-iot/samples>

These can be downloaded as a ZIP and opened in Visual Studio.

## **Processing.org**

Processing is the name given to an open source project that provides a programming environment that is primarily graphical in nature. What it does is allow the programmer to draw a rich variety of 2D and 3D graphics and take data from external sources to be used as input to what is drawn. Why is this of interest to us in the Pi land? First of all Processing can run directly on the Pi. It also has built in logic to retrieve data directly from a Pi's hardware subsystem (GPIOs, I2C, SPI, Serial etc). In addition, Processing has networking primitives built into it such that it can retrieve data over the network, for example from a Pi. Don't mistake Processing's graphics abilities for standard UI building, it isn't that though it *could* be used for that (though I would not recommend it). Rather, Processing is much more for displaying mathematically generated data or very sophisticated graphics modeling.

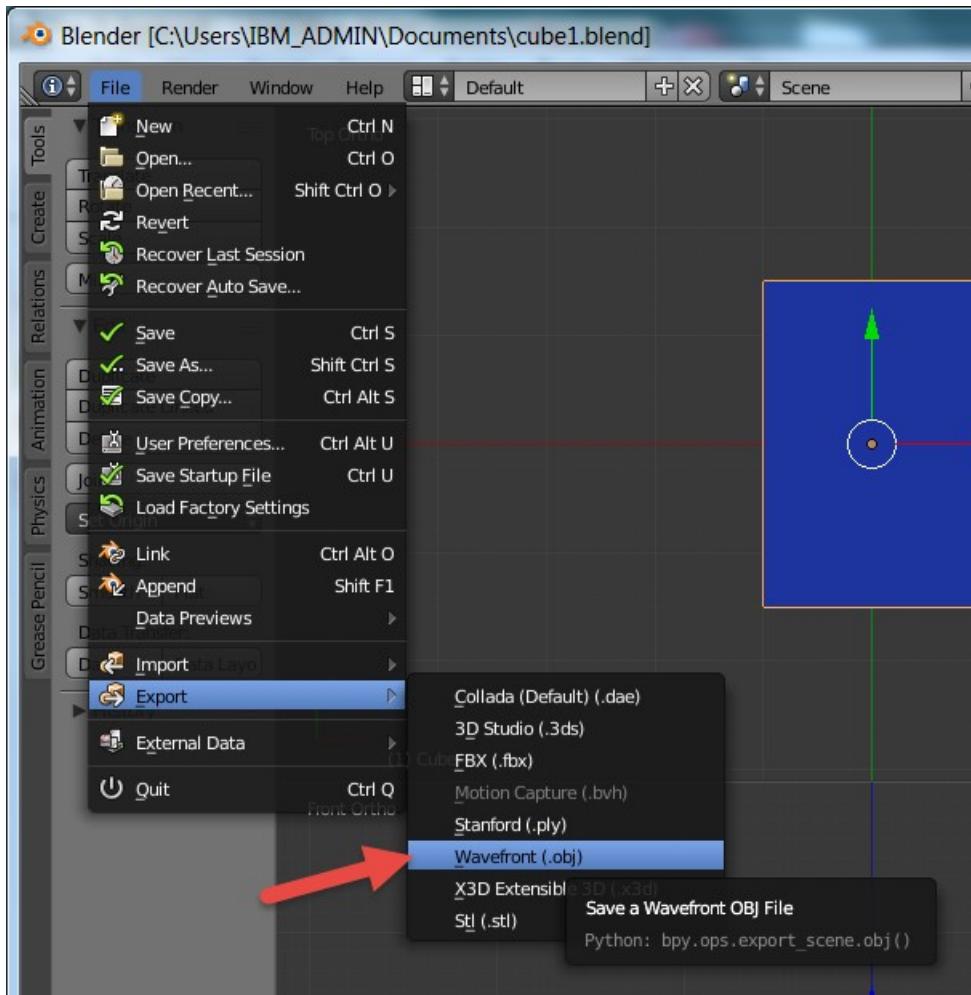
See also:

- [Processing.org](http://Processing.org)
- Github: [Processing](https://github.com/processing/processing)

## **Importing Blender graphics**

Blender is a world-renowned open source 3D graphics package. Using blender one can easily create 3D models or import a wide variety of pre-existing 3D models that can be found on the net. Processing has the ability to import Blender authored models. When we create a blender model, we can export it in OBJ format which is the format that can be imported into Processing. The API to achieve this is called `loadShape()`. This is not currently available in `p5.js`.

To export a model from Blender in OBJ format, select `File > Export > Wavefront`:



Once chosen, we have the opportunity to provide a set of options for the export, these are the ones I have been using without incident:



## Processing and network interfaces

To use the networking library, we must `import processing.net.*`. Once done, we can create either socket clients or servers. To create a client, we create an instance of the Client class. For example:

```
Client myClient;

void setup() {
    myClient = new Client(this, "1.2.3.4", port);
}
```

Now in the `set draw()` logic, we can access the Client object through its methods to send and receive data. As an example, to check if we have any data and to retrieve it we can use:

```
if (myClient.available() > 0) {
    String s = myClient.readString();
}
```

To be a server, the logic is similar.

```
Server myServer;

void setup() {
    myServer = new Server(this, port);
}
```

To see the data available, we can call the `Server.available()` method. This will return either a `Client` object corresponding to the first client with data available or null if there is no client with data having been sent.

```
void draw() {  
    Client myClient = myServer.available();  
    if (myClient != null) {  
        String s = myClient.readString();  
    }  
}
```

See also:

- [Processing – network library](#)

## Electronics

One of the key stories of the Pi is that we can use it to interact with the real-world through a variety of electronics devices connected to its headers. Thankfully, we don't have to be rocket scientist electrical engineers in order to get this going. There are many free sources of knowledge on these topics on the Internet including step by step tutorials. However, there are certain items that one would not normally hear about when working in the Pi that are related to electronics that I feel are of real value for you to hear about.

### Drawing circuits and breadboards - Fritzing

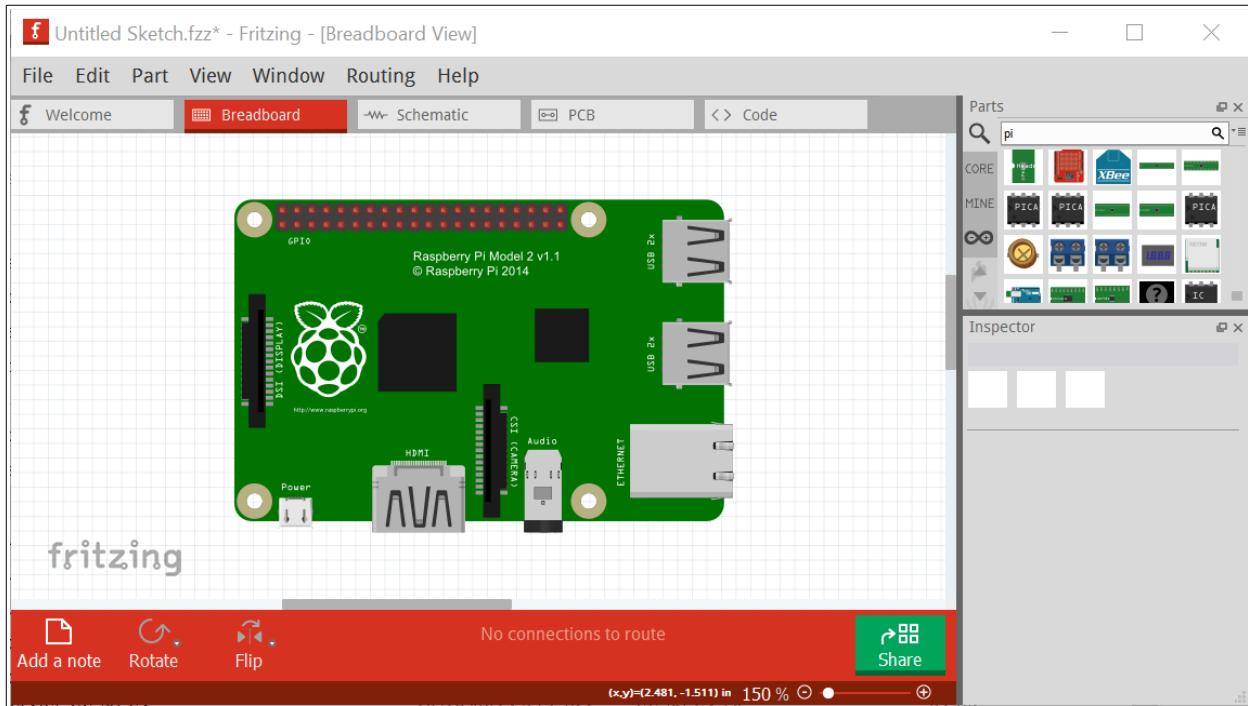
The fantastic open source package called Fritzing ([www.fritzing.org](http://www.fritzing.org)) is one of the best packages available for both drawing circuit diagrams as well as laying out breadboards. End users can create their own parts and share them via the Internet. The tool is very easy to use and provides an integrated circuit diagram editor plus a breadboard layout editor.

The software is free to download at:

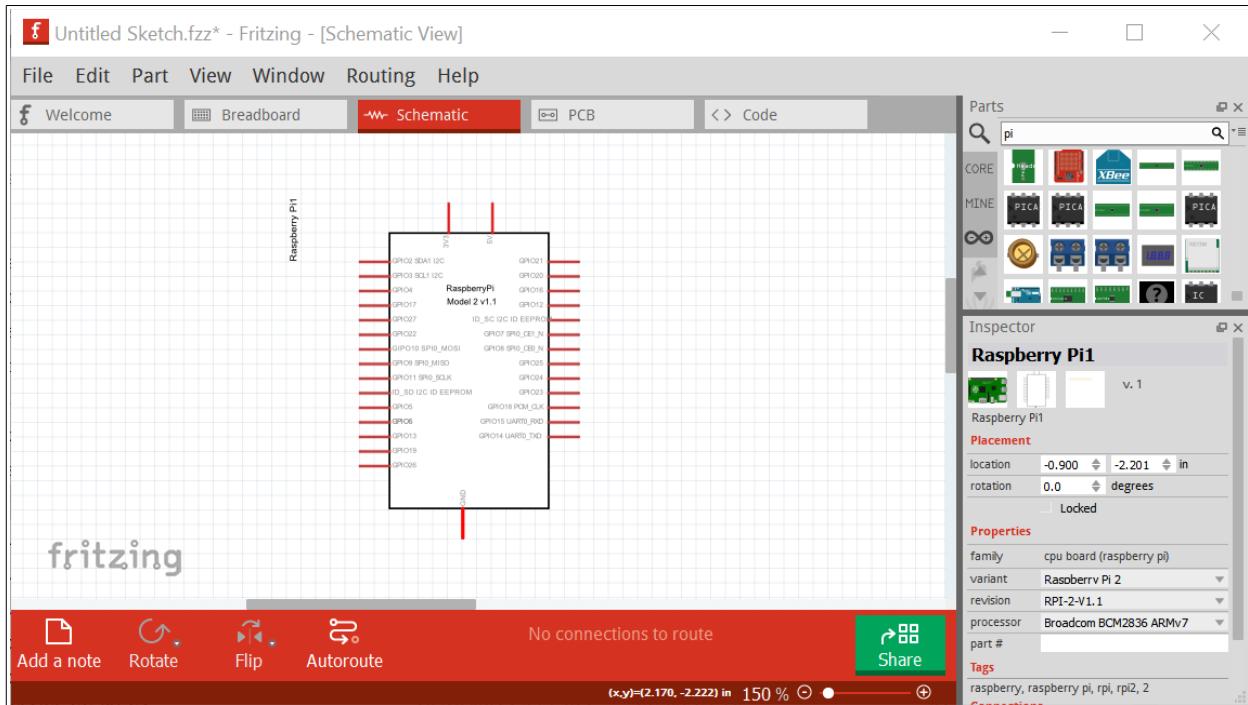
<http://fritzing.org/home/>

I recommend the use of Fritzing for all but the most trivial of projects. By using Fritzing to visualize both a schematic and physical layout when it comes time to assemble a project, you will have a blue print to work from. It is tempting to build a project going straight to the breadboard but resist that. Invariably, you will spend more time debugging than you would have spent drawing and then wiring. In addition, you have an artifact in the form of the Fritzing images that can be referred to again and again over time. Ideally you will share these images on the Internet for others to use. If nothing else, when you come back to this project a month later you will have somewhere better to start with than any written notes you may or may not have taken.

The following shows a screen shot but does not do the power of the tool justice. The images can be wired and linked together to produce visual representations of circuits. Here we show a physical Pi.



In the schematic editor, we see its logical layout:



One of the true benefits of this package is that we are not constrained by the availability of parts.

This takes a little time and practice and one should be handy with the open source SVG graphics editor called Inkscape.

See also:

- [Fritzing home page](#)
- [Inkscape](#)

## Using a logic analyzer

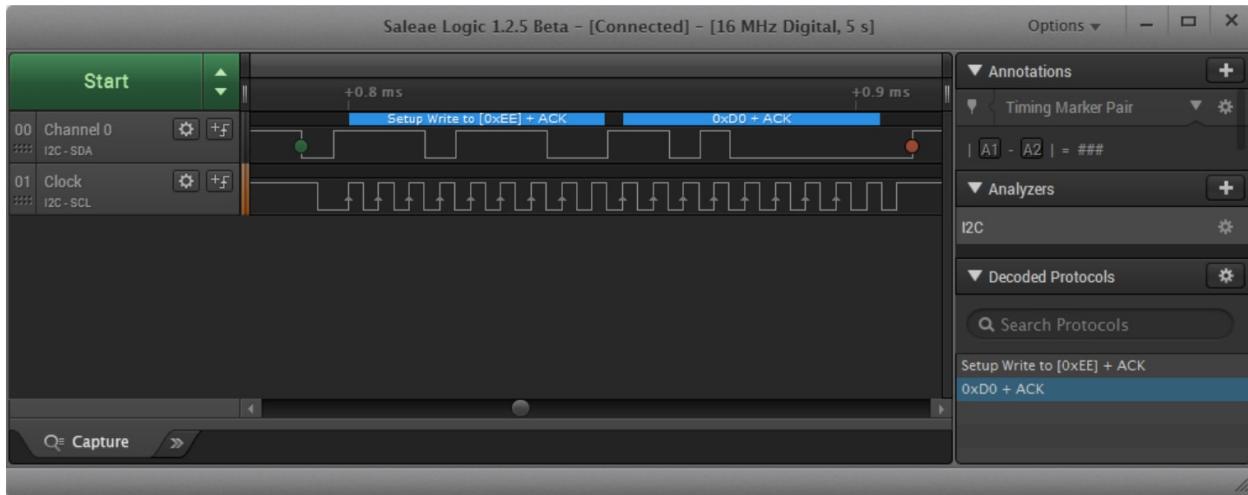
When you have built a circuit that includes your Pi and you run your application and the results are not what you are expected, you are in trouble. You need tools to diagnose your problem. No matter how carefully you examine your circuit with a magnifying glass, you will not be able to see the bits and bytes flowing around. What you need is a device called a logic analyzer. Commonly, this is a device which plugs into your USB port of your PC and allows you to attach connector wires to various places on your circuit. The number of concurrent wires you can connect is the number of channels available on the analyzer. Once you have connected the wires, you run your application/circuit on the Pi. While it is running, you start a recording on the PC. The PC and the logic analyzer now samples the data on the wires and hence the data at the locations of where those wires are connected in your circuit. The sampling rate can be exceedingly high such that changes in signal values at a 100 nanoseconds or less can be detected. The data is recorded in the memory of the PC. After a few seconds of sampling, you can stop the recording. Now that you have accumulated the data, you take your time and use the logic analyzer software to examine the results. This is usually in the form of a time-line with a row per channel recorded. The row shows a line with values of low or high corresponding to the signal. You can scroll along the line as well as zoom in or zoom out to show more or less detail for smaller or larger periods of time. If you recorded multiple channels, you can see the relationships and timing of signals as they change together or apart.

When working with protocols such as I2C or SPI, a logic analyzer can be invaluable. If you assemble a circuit and write an application and it isn't working as it should, you need some mechanism to perform diagnostics. Trying to debug SPI or I2C without an analyzer is like trying to juggle with only one hand.

There are a number of vendors of logic analyzers on the market. One of the better suppliers is called Saleae. They make a sample visualizer called Saleae Logic that is extremely elegant and performs all the activities that one could sensibly want.

If one searches on eBay, one can also find very cheap versions of analyzers at around the \$10 price point.

Here is an example screen shot captured using the excellent Saleae Logic tool:



Not only does this tool clearly illustrate the wave forms, but it is also able to decode some of the well known protocols such as I2C and SPI.

See also:

- [Saleae](#)
- [Wikipedia – Logic analyzer](#)

## Building a prototyping environment for the Pi Zero

Primarily because the Pi Zero is only \$5, many of us can take risks with those devices and buy multiple instances. When we look at how we "play" with and prototype solutions, we see that there are common patterns and mini circuit components that we continually build over and over again. As such, there is value in investing some time and money building a permanent Pi Zero circuit board that hosts common circuit functions and components that we want to use in conjunction with a Pi Zero. Some of these include:

- Logic probe – A couple of LEDs with transistor circuits that will show the signal value input upon it.
- Power supply – Component-ized power supplies are available that provide 3.3V and 5V signals.
- USB→UART adapters – A USB→UART adapter allows us to connect a serial device from our PCs to the Pi.
- Male and female headers – Connecting male and female headers to the board allows us to simply plug-in any number of components as we desire using jumper wires.

When we have designed our desired prototype board, we can then make it permanent through a piece of strip board and soldering the components upon it.

## Overcoming trepidation of using Integrated Circuits

When it comes to wiring a Pi to other modules, most folks appear to be happy to do that when the modules expose pins which are clearly labeled. Plugging a pin on a device that is clearly labeled as

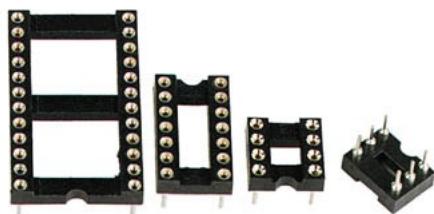
"MOSI" or "CLK" to the corresponding pin on the Pi header seems to be well within everyone's grasp. However when we introduce integrated circuits, something changes. Integrated Circuits (ICs) don't have friendly labels associated with their pins, they are smaller and seem to instill a degree of concern that isn't warranted.



Physically, the ICs that we will be working with come as small, black devices with pins. Commonly 8 or 16pin but sometimes more and sometimes less. They should be bread-board friendly meaning that you can push them into a bread-board for testing. The biggest mystery over the devices is how to connect them properly ... and this is where the data-sheet of the device becomes your greatest friend. Each manufacturer of an IC produces a document called a data-sheet that describes in a **lot** of detail how to use it. For hobbyists, that can seem like too much detail. There are charts and graphs showing all kinds of information about its characteristics. However, from our perspective, what we really are looking for is the pin configuration. Once we learn which pin performs which function, we are well on our way to success.

When buying ICs, make sure you understand the difference between DIP (Dual in-line package) which are ICs with the pins in parallel rows that can be plugged into bread-boards. These are also called "through the hole" components. The primary alternative format is called surface-mount. These can not be directly used with bread-boards and for hobbyist and ease of tinkering considerations, should be discounted in favor of DIP.

When it comes to assembling circuits and soldering them to a PCB or strip board, if the chip is expensive or you feel you don't want to commit it to just one project, you can solder in an IC socket into which the IC can be inserted and subsequently removed.



There are sockets for all the different DIP IC dimensions and they can also be found cheaply.

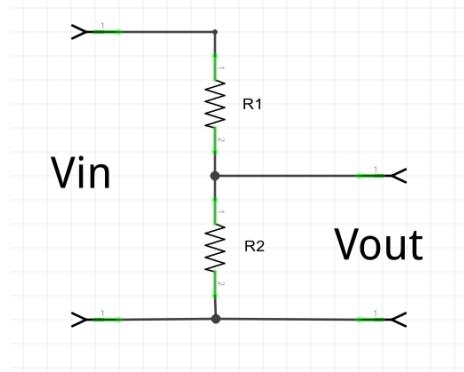
See also:

- Wikipedia: [Dual in-line package](#)
- Wikipedia: [List of integrated circuit package dimensions](#)

## Logic Level Shifting

We have already read that the Pi GPIO pins are 3.3V tolerant. This means that the maximum input voltage must be 3.3V. If you go higher than that, you are at risk for frying your Pi. But what if we have a peripheral device that outputs a 5V signal? Conversely, the maximum output voltage on a Pi GPIO is 3.3V. If we feed that as input to a peripheral that is expecting a 5V input, there is no assurance that it will detect the signal correctly. Fortunately there is a solution in a circuit called a logic level shifter. This circuit can convert a 3.3V signal to a 5V signal and can also convert a 5V signal to a 3.3V signal.

If you don't have access to such logic level circuits, an alternative solution is to create a voltage divider. This will protect 3V Pi inputs from over voltage from a 5V source. This is a simple circuit made with two resistors. A higher voltage is provided to input than output and using a correct combination of resistors, the input voltage is proportionally scaled to the output voltage.



There is an equation available to us:

$$V_{out} = \frac{R2}{R1+R2} \cdot V_{in}$$

As an example of a solution to this equation, if Vin is 5V and we want Vout to be 3.3V and choose 1K for R2 then ...

$$3.3 = \frac{1000}{R1+1000} \times 5$$

Which evaluates to R1 having a value of 515Ohms. We can use a higher value for R1 but not lower.

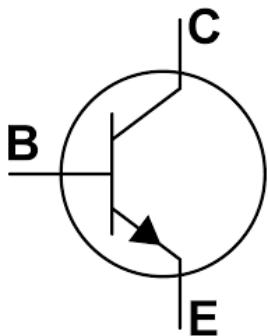
See also:

- Sparkfun – [Bi-Directional Logic Level Converter Hookup Guide](#)
- Wikipedia – [Voltage divider](#)

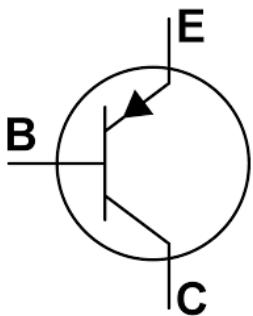
## Transistors as switches

If we consider that a GPIO pin can produce a signal of high or low we see that the signal can be used to perform work. However there is a maximum current that can be drawn from any given I/O pin. If, for example, we wished to attach a device to a pin which would draw more current than the pin is rated to supply, we may very easily damage our Pi. This is where a transistor can come into play. A transistor can be thought of as an electronic switch. A transistor has three pins labeled emitter, base and collector. If a

small current flows between base and emitter a correspondingly higher current will be allowed to flow between collector and emitter. There are two types of transistor we will come across called NPN and PNP. The difference between them is whether or not the "switch" is triggered by a high signal or a low signal. The following is the schematic symbol for an NPN transistor. Normally we connect the emitter to ground and a load between the collector and +ve. If we then connect the base to the output of an I/O pin (through a resistor), then when the pin goes high, a small current will flow between B and E and a much higher current will flow between C and E.



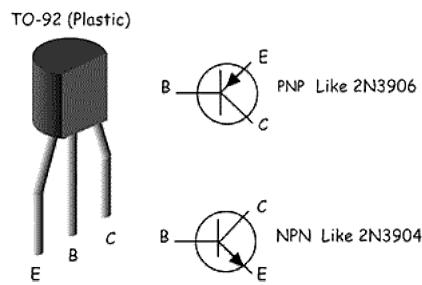
For the other type of transistor (PNP) whose symbol is shown in the following diagram, the emitter is usually connected to +ve and the collector to the load and then ground. With the base pin connected to a GPIO (through a resistor), when the base goes low, the small current flowing from the emitter through the base will be greatly amplified as the current flowing through the emitter to the collector.



By using a transistor, we are no longer directly drawing load through the GPIO pin but instead simply using the signal present on the pin to energize the transistor.

Common transistors that can easily be obtained include:

Name	Type
2N3904	NPN
2N3906	PNP
PN2222	NPN



## Migrating code from the Arduino

The Arduino devices are fantastic microprocessors that have a huge following. As such, the vast majority of peripherals and electronics already have Arduino libraries available for them. When we encounter a new device that we wish to integrate with, we can always start from scratch with its data sheet and build an interface to it from scratch. However, unless we are truly masochists, there is no real need to do that. Instead what we can do is find an already existing library from a technology such as the Arduino and port it to the Pi. Of course, if a library already exists for the Pi, one would use that instead.

If we are faced with an Arduino sketch that achieves what we want in an Arduino environment and wish to get that going on a Pi, we need to have skills not only in Pi programming but also some skills in Arduino programming. If nothing else, we need to be able to read Arduino code and map the lower level Arduino specific APIs to comparable Pi APIs. That is what this section is about. Here we will examine some of the more common Arduino functions that we are likely to come across and map those to the corresponding Pi functions.

### Arduino I2C - Wire class

The Arduino has an I2C interface called "Wire".

See also:

- [Wire Library](#)

#### Wire.begin

This API sets up the Arduino as an I2C bus master.

```
Wire.begin()
```

#### Wire.beginTransmission

Start a communication stream to the I2C slave. The syntax is:

```
Wire.beginTransmission(address)
```

## **Wire.write**

Transmit data to the I2C slave. There are a few formats of the API:

```
Wire.write(byte)
Write.write(buffer, length)
```

## **Wire.endTransmission**

End a stream to the I2C slave.

```
Wire.endTransmission()
```

## **Wire.read**

Read a byte back from the slave.

```
byte = Wire.read()
```

## **Wire.requestFrom**

Request a stream of data from the given slave.

```
Wire.requestFrom(address, quantity)
Wire.requestFrom(address, quantity, stop)
```

## **Scheduling applications**

Imagine you have a desire to record the temperature every hour. You could always write an application that puts itself to sleep until the top of the hour, records some data and then repeats the sleep. This will work but there is perhaps a better way. What you could do is write an application that records a data record every time it is launched and now schedule that application to run at periodic intervals. Raspbian provides a tool called "cron" that allows us to to configure arbitrary commands that run at arbitrary intervals.

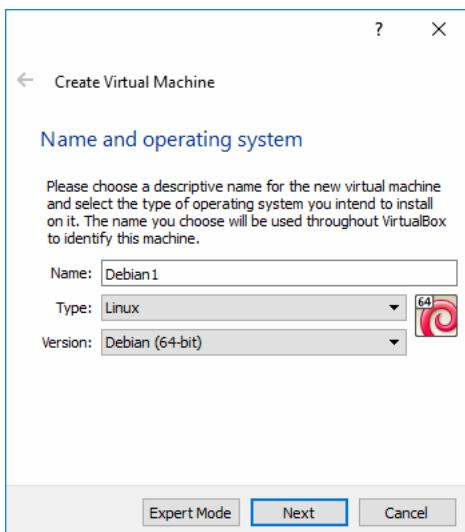
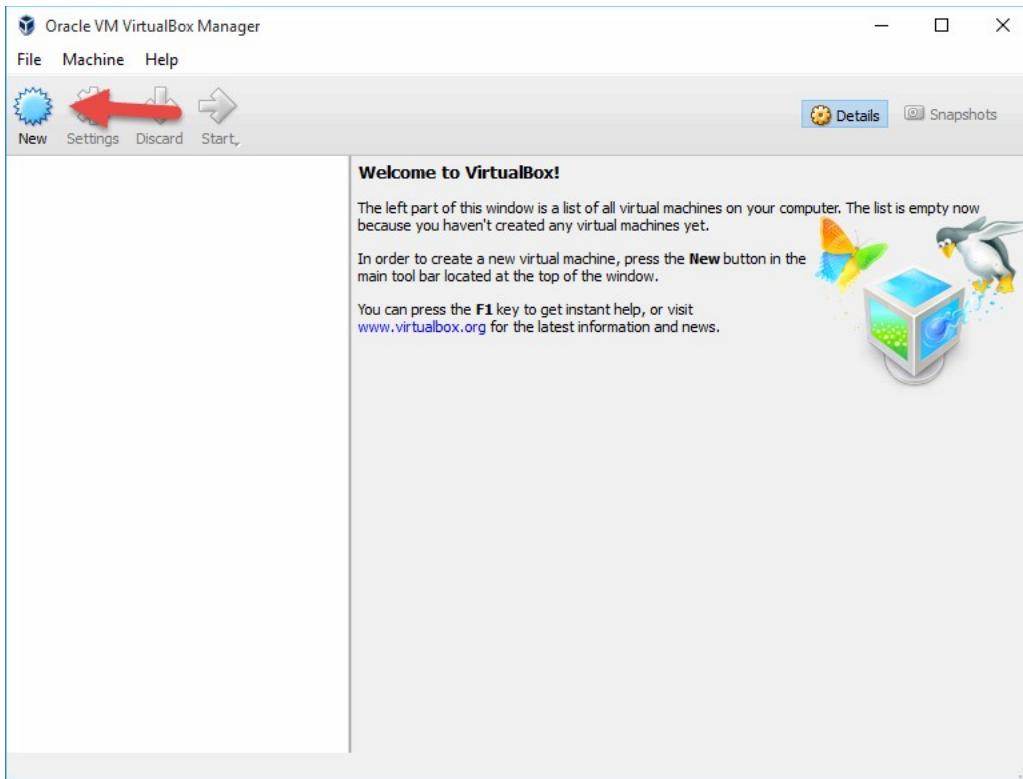
## **Setting up Linux on Virtual Box on Windows**

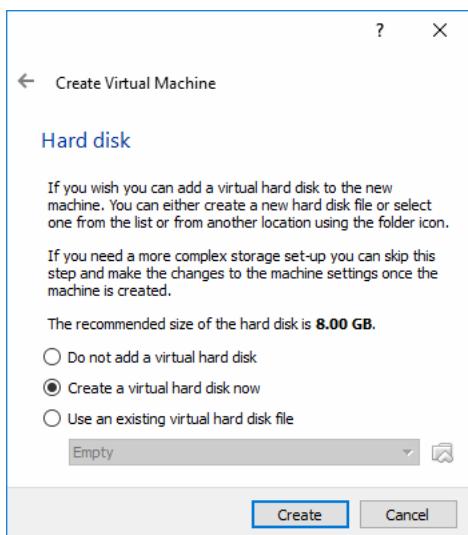
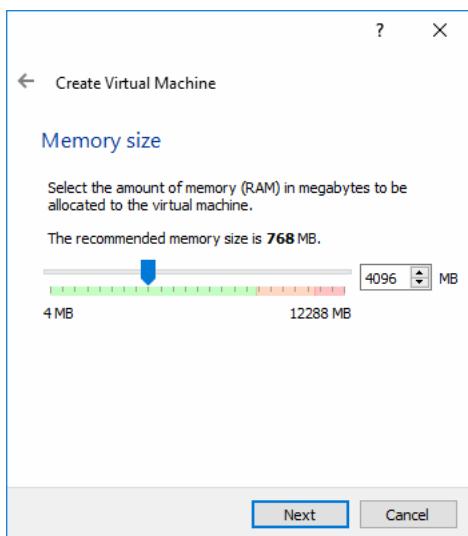
For the longest time I resisted doing development work on Linux. I am a windows user and happy to be such. However when I moved into Pi development, that forced me to work in the Linux environment and my views changed. I still don't want to run Linux for my day to day desktop work but for development, given a choice, that will be my environment. However, rather than have multiple computers, monitors and other such things around, what I wanted to do was just run one computer that does **both** Linux and Windows. Fortunately, Oracle Virtual Box comes to the rescue. Using the free Virtual Box software, I can host a guest operating system (Linux) on top of my host operating system (Windows 10). The result is the best of both worlds and I am delighted.

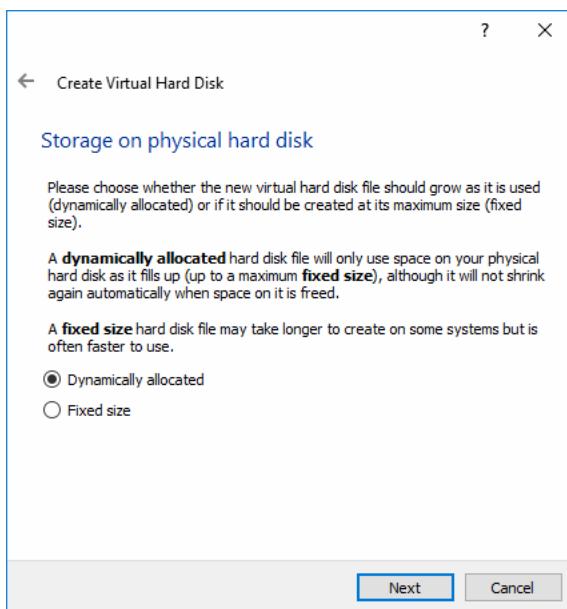
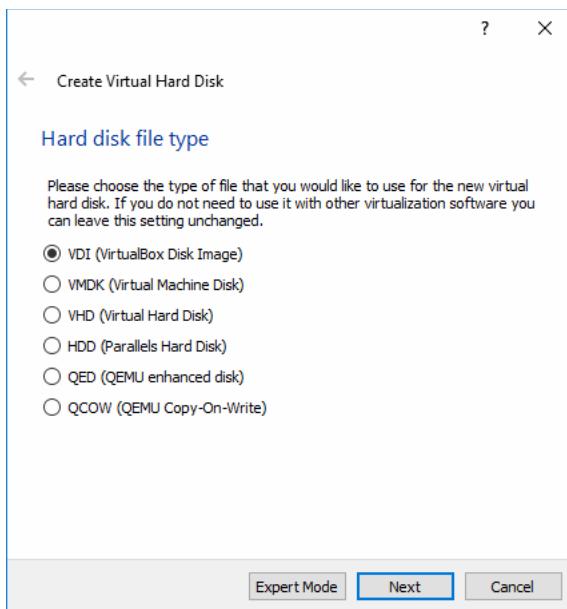
Once setup, you will have a Linux environment that is indistinguishable from a "real" Linux environment. I choose Ubuntu as my Linux distribution.

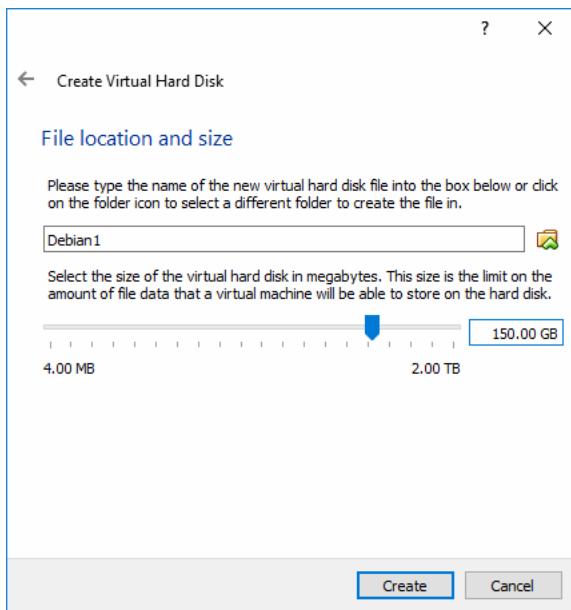
Here is a walk through of a Debian 8.3.0 install.

1. Download the `debian-8.3.0-amd64-CD-1.iso` file
2. Create a new image in Virtual Box.

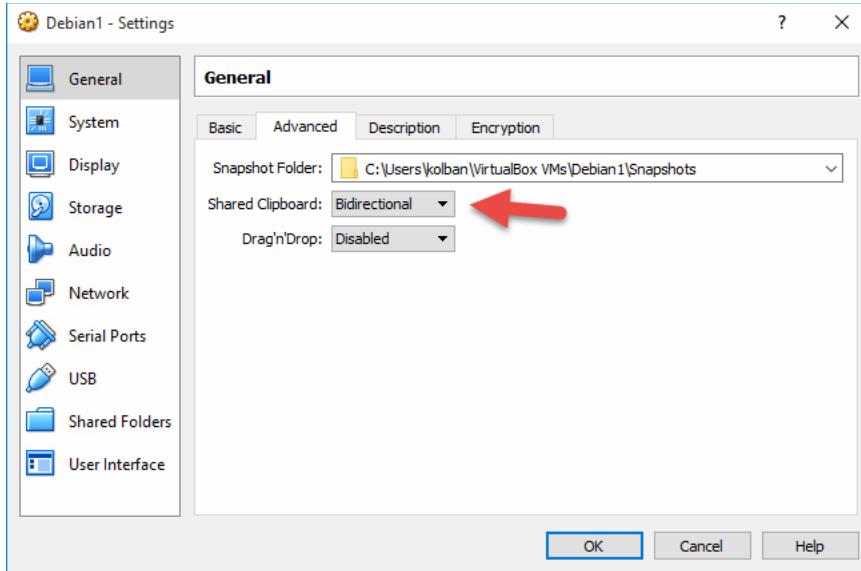


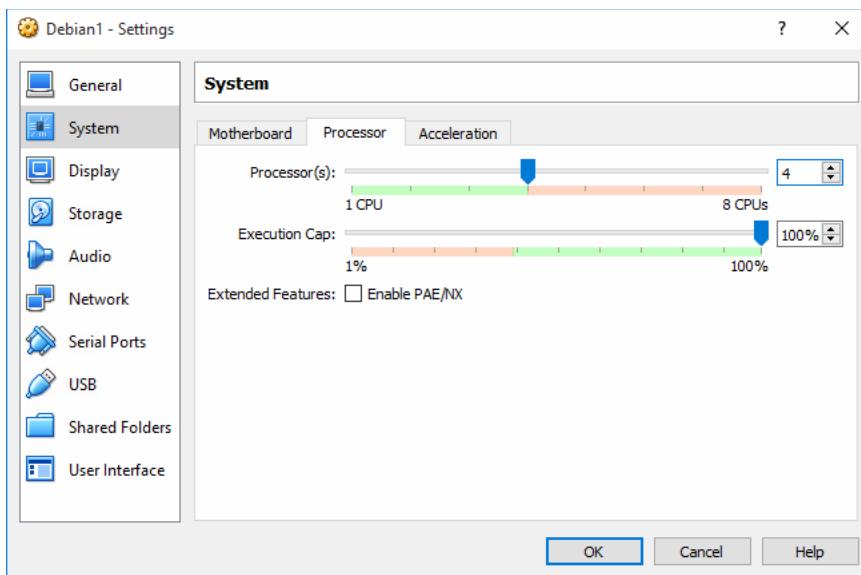




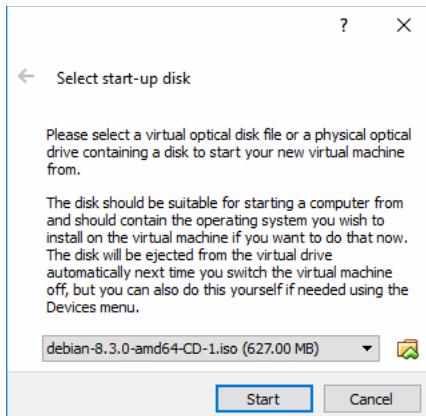


### 3. Change some of the settings on the image

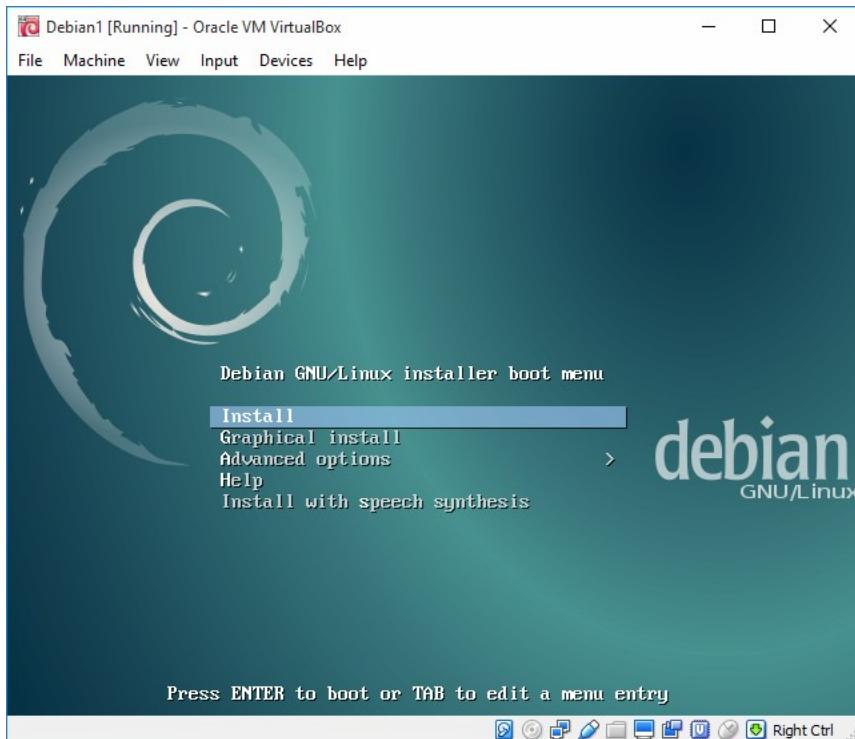




4. Start the image for the first time



5. Select how you want to install Debian

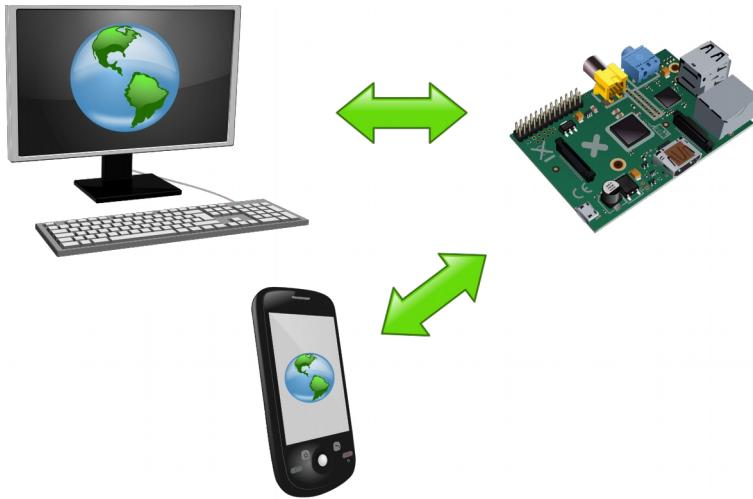


6. Follow the on-screen instructions for setting up the Debian environment. After answering a slew of questions, Debian will install. This may take a while.
7. s
8. s
9. s
10. s

## Web programming

There is a school of thought that says the browser will become the universal desktop for all applications. Instead of applications having to drive the screen locally using a variety of different UI technologies such as X-Windows, JavaFX, Swing, Windows UI and all the many, many others, applications will instead serve up HTML that runs within the browser. Any data need by the UI will be requested by the browser from the app using REST. Any actions that need to be performed such as data base updates will again be performed by the App on request of the user through the browser. Not only will the browser become the UI but it opens up the opportunity for remote interaction with the application where today a thick UI can only be used locally.

Another important consideration is the variety of devices that can host a browser. Not only does this include PCs but of course includes phones and tablets.



See also:

- REST requests

## Browser security

To allow geolocation from files loaded from the file system, we need to start Chrome with the extra flags "`--allow-file-access-from-files`".

```
$ google-chrome --allow-file-access-from-files
```

To allow yourself to make cross site calls unsecured, start with "`--disable-web-security`".

```
$ google-chrome -disable-web-security
```

## jQuery - JavaScript framework

The jQuery JavaScript framework provides a lot of the service to JavaScript in the browser that aren't otherwise available. Many of these relate to DOM programming which is the technology used to change how the browser displayed web page is shown. However there are other features of jQuery that have non-UI applicability.

### Using jQuery

The simplest way to use jQuery is to include it within the `<head>` of your page. For example:

```
<script  
src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.4/jquery.min.js"></script>
```

You can bring it in from a variety of CDN locations or serve a copy of it up from your local Webserver.

## Making REST requests using jQuery

From with any JavaScript application, sooner or later we will be making REST calls. The function called "`jQuery.ajax()`" makes the REST call for us. It takes a parameter which is the settings for the request which is a JavaScript object. That object controls how the request shall be made.

- `url` – The target of the request.
- `data` – An object to be passed as name/value pairs for the query string. If the method is `POST` then this is the data sent in the body of the message.
- `contentType` – The Mime Content-Type HTTP header value.
- `success` – A JavaScript function that is called when a successful return occurs from the REST request. The function has the signature of:
  - Anything data
  - String textStatus
  - jqXHR jqXHR
- `error` – a JavaScript function that is called when an error return occurs. The function has the signature of:
  - jqXHR
  - textStatus
  - errorThrown
- `context` – The JavaScript object (commonly passed as this) that will be the context used for the callback functions.
- `method` – The type of HTTP command. Commonly either "GET" or "POST".
- `username` – A username to be passed.
- `password` – A password to be passed.

See also:

- [jQuery.ajax\(\)](#)

## Reference Information

See also:

- [elinux.org](#)

## Recommended software add-ons

When working with new environments, it can be frustrating to find that a tool or capability you rely on was not installed simply because you forgot to perform that task. As such, it is a good idea that when you

come to a new environment you pre-install those tools. It is also good to keep a list of what you need so that you can remember what you need.

Package	Pi	Linux PC	Description
geany	•	•	Small IDE / Editor
gedit	•	•	General purpose editor
git	•	•	Source code tools
Eclipse Mars		•	Full function IDE

## Magazines

Although the notion of IT magazines may seem a bit of an anathema in our Internet/Web/Blog/PDF world, they are still alive and kicking. There is a specialist magazine for the Pi called "The MagPi" which is available both in paper and PDF versions. The PDFs can be downloaded free of charge including back-issues. The web site for this magazine is:

<https://www.raspberrypi.org/magpi/>

## Forums

Forums (aka message boards) are an invaluable source of both knowledge and assistance for your work on the Pi. A forum is a place where you can ask a question and, if someone knows an answer, can post that answer for your assistance. Since the forums are public, you can peruse and search previous questions and answers. There is a very good chance that a question you have has been previously asked and answered and may be just what you need.

It is a good deed to look at questions when you have time and see if you yourself can't supply answers. They don't need to be authoritative and, even if your answer is flat wrong, if someone posts a correction, you yourself will have grown new knowledge.

See also:

- [RaspberryPi.org](http://RaspberryPi.org)
- Stack Exchange – [Raspberry Pi](#)

## You Tube

Without question, one of the best sources of knowledge on almost any subject is You Tube. Obviously the quality of the content found there can vary dramatically ... however there are some real gems out there. Some are Pi exclusive while others cover varying electronics subjects. With experience you will find that you will be able to translate projects from one processor type (eg. Arduino) to the Pi with growing ease. Some of the channels that I recommend are:

- [Julian Ilett](#) – A very easy to listen to British individual that is a natural presenter for You Tube if ever there was one. His knowledge is excellent and his delivery very relaxed. The presentation is

of high quality with very low overhead (no fancy titles, no grating music and long un-interrupted segments). The breadth of topics covered is excellent.

- [Great Scott!](#) – Another very easy to listen to German chap who covers materials extremely well.
- [TheRaspberryPiGuy](#) – A dedicated channel to Raspberry Pi. Very easy to follow articles. High quality recordings with low overhead and focused content. Contrast this series against Julian Ilett's. While Julian's are extremely friendly and a great favorite of mine this channel gets to the meat of all content very quickly. If you want reference and shorter videos, these are likely the ones you want to examine first.

## Chat rooms

There is a live chat room for the Raspbian OS that can be found here:

<http://webchat.freenode.net/?channels=raspbian>

It uses IIR mechanism but can be viewed on a browser. The name of the chat room is #raspbian.

## WiringPi Reference information

The WiringPi library and tools have become the defacto standard on a Pi for driving the low level hardware. As such, we will likely find ourselves working with these over and over again. The following section provides summary documentation and experiences for many of the commands and function calls. One should always use this in conjunction with the formal documentation on WiringPi found at the home web site for the library.

### gpio command

When working with pin numbers, the default is to use the WiringPi numbering scheme. However, we can switch of using the Broadcom numbering scheme by supplying the `-g` flag or to physical pin numbering with the `-1` flag. As always, it is vital that you think through which pins are which otherwise you might get yourself into all kinds of trouble.

The `gpio` command has built-in support for a variety of add-on boards including PiFace and Gertboard. Neither of those boards will be discussed here nor their corresponding commands.

See also:

- [The gpio utility](#)

### gpio clock

Before setting the frequency, the pin's mode must be set to clock.

```
gpio [-g|-1] clock <pin> <frequency>
```

## gpio mode

The `gpio mode` command sets the mode of the pin. It has the following syntax structure:

```
gpio [-g|-l] mode <pin> {in|out|pwm|clock|up|down|tri}
```

The definitions are:

- `in` – Set the pin as input. Signal changes made at the pin can be read from an application.
- `out` – Set the pin as output. The application can change the state of signal at a pin.
- `pwm` – Set the pin as Pulse Width Modulation
- `clock` – Set the pin as a clock
- `up` – Set the pin as input with a pull-up resistor
- `down` – Set the pin as input with a pull-down resistor
- `tri` – Set the pin as input with tri-state (no resistor).

The pins allowable for PWM are:

Physical	Wiring Pi GPIO	Broadcom GPIO	Function
12	GPIO 01	GPIO 18	PWM0
32	GPIO 26	GPIO 12	PWM0
33	GPIO 23	GPIO 13	PWM1
35	GPIO 24	GPIO 19	PWM1

## gpio pwm

The `gpio pwm` command sets the PWM value on the given pin. Prior to setting a pin's value, the pin mode should be set to `pwm`. The command has the following syntax structure:

```
gpio [-g|-l] pwm <pin> <value>
```

The value is between 0 and 1023.

The pins allowable for PWM are:

Physical	Wiring Pi GPIO	Broadcom GPIO	Function
12	GPIO 01	GPIO 18	PWM0
32	GPIO 26	GPIO 12	PWM0
33	GPIO 23	GPIO 13	PWM1
35	GPIO 24	GPIO 19	PWM1

See also:

- `PWM`
- `pwmWrite`

## gpio pwmc

The `gpio pwmc` command sets the PWM clock value.

```
gpio pwmc <value>
```

Value must be in the range 1 and 4095. This is used to define the period. The base clock speed is 19.2MHz. The value here becomes a divisor of this clock to reduce the period. From here, we will further divide this period into a number of units defined by the range.

See also:

- `PWM`
- `pwmSetClock`

## gpio pwmr

The `gpio pwmr` command sets the PWM range.

See also:

- `PWM`
- `pwmSetRange`

## gpio pwm-bal

The `gpio pwm-bal` command sets the PWM into balanced mode. This is the default.

See also:

- `PWM`
- `pwmSetMode`

## gpio pwm-ms

The `gpio pwm-ms` command sets the PWM into mark/space mode. The default is balanced mode.

See also:

- `PWM`
- `pwmSetMode`

## gpio read

The `gpio read` command reads a value from a given pin. It has the following command syntax:

```
gpio [-g|-l] read <pin>
```

## gpio readall

The `gpio readall` command shows an attractive listing of the current settings of all the pins of the Pi including their mode, names, pins and values. This is an example of output:

Pi 2																					
	BCM	wPi		Name		Mode		V		Physical		V		Mode		Name		wPi		BCM	
				3.3v				1		2						5v					
	2	8		SDA.1		IN		1		3		4				5V					
	3	9		SCL.1		IN		1		5		6				0v					
	4	7		GPIO. 7		IN		1		7		8		1		ALTO		TxD		15	
				0v						9		10		1		ALTO		RxD		16	
	17	0		GPIO. 0		IN		0		11		12		0		IN		GPIO. 1		1	
	27	2		GPIO. 2		IN		0		13		14						0v			
	22	3		GPIO. 3		IN		0		15		16		0		IN		GPIO. 4		4	
				3.3v						17		18		0		IN		GPIO. 5		5	
	10	12		MOSI		IN		0		19		20						0v			
	9	13		MISO		IN		0		21		22		0		IN		GPIO. 6		6	
	11	14		SCLK		IN		0		23		24		1		IN		CEO		10	
				0v						25		26		1		IN		CE1		11	
	0	30		SDA.0		IN		1		27		28		1		IN		SCL.0		31	
	5	21		GPIO.21		IN		1		29		30						0v			
	6	22		GPIO.22		IN		1		31		32		0		IN		GPIO.26		26	
	13	23		GPIO.23		IN		0		33		34						0v			
	19	24		GPIO.24		IN		0		35		36		0		IN		GPIO.27		27	
	26	25		GPIO.25		IN		0		37		38		0		IN		GPIO.28		28	
				0v						39		40		0		IN		GPIO.29		29	
+-----+		BCM	wPi		Name		Mode		V		Physical		V		Mode		Name		wPi		BCM
+-----+																					

## gpio wfi

The `gpio wfi` command blocks the caller until a change of signal on the specified pin is detected. The acronym "wfi" means **Wait For Interrupt**. The waiting is interrupt driven (as opposed to polling) and hence is efficient in its usage of system resources.

The syntax of the command is:

```
gpio [-g|-l] wfi <pin> {rising|falling|both}
```

## gpio write

The `gpio write` command sets the pin's output value to low or high. The pin should previously have been defined as output.

The command's syntax is:

```
gpio [-g|-l] write <pin> {0|1}
```

## **delay**

Delay / pause the application thread for the given number of milliseconds.

```
void delay(unsigned int duration)
```

This call causes the process to delay for at least the specified number of milliseconds supplied by duration. We say at least because it could be more. If another Linux process has control when the delay period has elapsed, then our own process might not immediately get control. It will, however, be then eligible for subsequent execution. Remember that 1 millisecond is 1/1000th of a second.

## **delayMicroseconds**

Delay / pause for the given number of microseconds.

```
void delayMicroseconds(unsigned int duration)
```

This call causes the process to delay for at least the specified number of microseconds supplied by duration. We say at least because it could be more. If another Linux process has control when the delay is up, then our own process won't immediately get control. It will, however, be then eligible for subsequent execution. Remember that 1 microseconds is 1/1000000th of a second.

## **digitalRead**

Read the value from an input pin.

```
int digitalRead(int pin)
```

This call returns the value read from the pin. The result will either be HIGH or LOW which are the values 1 and 0.

The corresponding gpio command is:

```
gpio [-g|-l] read <pin>
```

## **digitalWrite**

Set the output value of a pin.

```
void digitalWrite(int pin, int value)
```

When a pin is set to a pinMode value out OUTPUT, this call defines the value of the signal found on that pin. A value of 0 means low while any other value means high. The symbolic constants of HIGH and LOW may also be used as values.

The corresponding gpio command is:

```
gpio [-g|-l] write <pin> 0/1
```

## **digitalWriteByte**

Write 8 bits of data as quickly as possible.

```
void digitalWriteByte(int value)
```

Interpret the value as 8 bits of data and write them to the first 8 pins as quickly as possible.

The corresponding gpio command is:

```
gpio wb <value>
```

## **micros**

Return the number of microseconds since the original setup call.

```
unsigned int micros(void)
```

This call returns the number of microseconds that have elapsed since the initial call to a WiringPi setup function was made.

## **millis**

Return the number of milliseconds since the original setup call.

```
unsigned int millis(void)
```

This call returns the number of milliseconds that have elapsed since the initial call to a WiringPi setup function was made.

## **physPinToGpio**

```
physPinToGpio(int physPin)
```

## **piBoardRev**

Get the Pi board revision number.

```
piBoardRev(void)
```

Each Pi type has a revision number associated with it. This call retrieves and returns that value.

## **piHiPri**

Set the priority of the application.

```
int piHiPri(int priority)
```

The priority is a number from 0 to 99 with the default being 0. The higher the number, the higher the relative priority of the application. When multiple applications are all eligible for the CPU (i.e. are ready to run), the application with the highest priority will be given the resource.

## piThreadCreate

Create a new thread.

```
int piThreadCreate(name)
```

This call starts an instance of a thread defined by the macro PITHREAD(name). For example, if we code:

```
PITHREAD(runme) {  
    printf("Hello world\n");  
}  
  
void soSomething() {  
    piThreadCreate(runme);  
}
```

This will create an instance of a thread that will run the function in parallel (multi threaded) to the creator.

## piLock

Lock a mutex.

```
piLock(int keyNum)
```

A mutex is the name given to a mutual exclusion section. It can be thought off as a boolean valued variable that is initially false. When a lock is attempted to be taken on a mutex, then if it is false, its value becomes true and the program continues. If its value is true then the call blocks until the value becomes false ... at which point it becomes true and again progress continues. The corresponding piUnlock call should be made to unlock the mutex.

In WiringPi there are 4 mutexes with key values 0 through 3.

Typically, mutexes are used in conjunction with multi-threaded applications to provide mutual exclusion while performing some changes. This prevents two parallel threads from stepping on each other.

Consider this code:

```
char *array[10];  
int index = 0;  
  
void add(char *value) {  
    array[index] = value;  
    index = index + 1;  
}
```

If we called the add() function at the same time in two parallel threads, we run the risk of getting an incorrect result. However if we code:

```
char *array[10];  
int index = 0;  
  
void add(char *value) {  
    piLock(0);  
    array[index] = value;
```

```

    index = index + 1;
    piUnlock(0);
}

```

then we have restored correctness. Only one thread will be allowed to modify the `array` and `index` variables at a time while the other has to wait for the first one to reach the code to complete.

We need to take care that we don't end up in a deadlock situation if we are working with multiple locks at the same time.

## **pinMode**

Set the mode of the pin.

```
void pinMode(int pin, int mode)
```

The mode defines the direction or type of a GPIO pin. The options available to us are:

- `INPUT` – Set the pin to be input
- `OUTPUT` – Set the pin to be output
- `PWM_OUTPUT` – Set the pin to be a PWM output
- `GPIO_CLOCK` - ???

The corresponding gpio command syntax is:

```
gpio [-g|-l] mode <pin> {in|out|pwm|up|down|tri}
```

The pins for PWM are:

Physical	Wiring Pi GPIO	Broadcom GPIO	Function
12	GPIO 01	GPIO 18	PWM0
32	GPIO 26	GPIO 12	PWM0
33	GPIO 23	GPIO 13	PWM1
35	GPIO 24	GPIO 19	PWM1

## **piUnlock**

Unlock a mutex.

```
piUnlock (int keyNum)
```

Unlock a previously locked mutex. The `keyNum` is a value between 0 and 3 and specifies which of the four pre-defined mutexes we are unlocking.

## **pullUpDnControl**

Define the pull-up or pull-down nature of the pin.

```
void pullUpDnControl(int pin, int pud)
```

The pud parameter defines the nature of the pull-up / pull-down. This should only be set on a pin defined with a `pinMode` of `INPUT`. The values that can be set are:

- `PUD_OFF` – No pull-up or pull-down
- `PUD_UP` – Define a pull-up
- `PUD_DOWN` – Define a pull-down

## **pwmSetClock**

Set the PWM divisor value.

```
pwmSetClock(int divisor)
```

The divisor must be smaller than 4096.

The corresponding `gpio` command is:

```
gpio pwmc <clock value>
```

See also:

- `PWM`
- `gpio pwmc`

## **pwmSetMode**

Set the hardware mode of operation.

```
pwmSetMode(int mode)
```

The PWM can run in a couple of modes. One is called "balanced" and the other called "mark/space".

This call sets the current mode. The values of `mode` can be one of:

- `PWM_MODE_BAL` – Run in balanced mode. This is the default.
- `PWM_MODE_MS` – Run in mark/space mode.

The corresponding `gpio` commands are:

```
gpio pwm-bal
```

or

```
gpio pwm-ms
```

See also:

- `PWM`
- `gpio pwm-bal`
- `gpio pwm-ms`

## **pwmSetRange**

Set the range of hardware PWM.

```
pwmSetRange(unsigned int range)
```

Set the range of PWM. The default is 1024.

This value can also be set through the `gpio pwmr` command.

See also:

- [PWM](#)
- [gpio pwmr](#)

## **pwmWrite**

Set the hardware PWM value.

```
void pwmWrite(int pin, int value)
```

The value has the range of 0-1023.

The pins for PWM are:

Physical	Wiring Pi GPIO	Broadcom GPIO	Function
12	GPIO 01	GPIO 18	PWM0
32	GPIO 26	GPIO 12	PWM0
33	GPIO 23	GPIO 13	PWM1
35	GPIO 24	GPIO 19	PWM1

The corresponding `gpio` command is:

```
gpio [-g] pwm <pin> <value>
```

See also:

- [PWM](#)
- [pwmSetClock](#)
- [pwmSetMode](#)
- [pwmSetRange](#)
- [pinMode](#)
- [gpio pwm](#)

## **serialClose**

Close the serial device.

```
void serialClose(int fd)
```

The device specified by `fd` is closed. No further serial access should be attempted.

### **serialDataAvail**

Get the amount of data available for reading.

```
int serialDataAvail(int fd)
```

The return from this call is the amount of data immediately available for reading.

### **serialFlush**

Discard data.

```
void serialFlush(int fd)
```

Discard the data awaiting transmission or available to be received.

### **serialGetchar**

Get the next byte of data.

```
int serialGetchar(int fd)
```

Return the next byte of data. If no data is immediately available, the call will wait up to 10 seconds for new data to become available.

### **serialOpen**

Open the serial device.

```
int serialOpen(char *device, int baud)
```

The serial device is opened.

### **serialPutchar**

Send a single byte.

```
void serialPutchar(int fd, unsigned char c)
```

Write a single byte to the serial output.

### **serialPuts**

Send a string down the serial port.

```
void serialPuts(int fd, char *s)
```

The null terminated string is sent to the serial output.

## **serialPrintf**

Send a formatted string to the serial output.

```
void serialPrintf(int fd, char *message, ...)
```

Send a formatted string to the serial output.

## **setPadDrive**

```
setPadDrive(int group, int value)
```

## **shiftIn**

Read 8 bits of data.

```
uint8_t shiftIn(uint8_t dataPin, uint8_t clockPin, uint8_t order)
```

This call reads 8 bits of data from the pin specified by `dataPin`. A corresponding `clockPin` provides the timing control which says when the pin should be sampled. The `order` parameter can be one of `LSBFIRST` or `MSBFIRST` indicating the bit order to make up the byte.

## **shiftOut**

Write 8 bits of data

```
void shiftOut(uint8_t dataPin, uint8_t clockPin, uint8_t order, uint8_t val)
```

This call writes 8 bits of data specified by `val` to the `dataPin`. The `clockPin` is changed to indicate a new bit being available. The bits are written in the order supplied by the `order` parameter which will be one of `LSBFIRST` or `MSBFIRST`.

## **softPwmCreate**

Create a PWM output in software.

```
int softPwmCreate(int pin, int initialValue, int pwmRange)
```

This call creates a PWM output that is driven by software. The `pin` parameter specifies the pin to be used for output. The `pwmRange` parameter specifies the upper bound in 100 microsecond units. For example specifying a `pwmRange` of 100 means that the value can be from 0 to 100. The `pwmRange` specifies the period so a value of 100 means 100 units of 100 microsecond which equals 10 milliseconds. The `initialValue` is in the range of 0 to `pwmRange`.

See also:

- PWM

## **softPwmWrite**

Change the value of the software PWM output.

```
void softPwmWrite(int pin, int value)
```

The output of the software PWM is changed. The `pin` parameter defines the pin to be used for PWM while `value` is the value to be written. This call should not be made before calling `softwarePwmCreate`.

See also:

- [PWM](#)

### **softToneCreate**

Create a PCM tone.

```
int softToneCreate(int pin)
```

Initialize the named pin to be used for PCM output.

### **softToneWrite**

Set the PCM tone frequency.

```
void softToneWrite(int pin, int frequency)
```

Output a PCM signal on the given pin at the given frequency. The signal will continue until canceled by setting the frequency to 0.

### **wiringPiI2CRead**

Read 8 bits of data through I2C.

```
int wiringPiI2CRead(int fd)
```

Read 8 bits of data through I2C.

See also:

- [WiringPi I2C](#)
- [wiringPiI2CSetup](#)

### **wiringPiI2CReadReg8**

Read 8 bits of data through I2C from a specific register.

```
int wiringPiI2CReadReg8(int fd, int reg)
```

Read 8 bits of data through I2C from a specific register identified by 8 bits.

See also:

- [WiringPi I2C](#)
- [wiringPiI2CSetup](#)

## wiringPiI2CReadReg16

Read 8 bits of data through I2C from a specific register.

```
int wiringPiI2CReadReg16(int fd, int reg)
```

Read 16 bits of data through I2C from a specific register identified. The order is LSB first.

See also:

- WiringPi I2C
- wiringPiI2CSetup

## wiringPiI2CSetup

Initialize I2C.

```
int wiringPiI2CSetup(int address)
```

Initialize I2C for a specific slave device given by the address. The return is the file descriptor (`fd`) that becomes the handle to the I2C device. If an error occurs, the return is -1 and `errno` should be consulted for more details.

See also:

- WiringPi I2C

## wiringPiI2CWrite

Write 8 bits of data to the I2C device.

```
int wiringPiI2CWrite(int fd, int data)
```

Write 8 bits of data to the I2C device.

See also:

- WiringPi I2C
- wiringPiI2CSetup

## wiringPiI2CWriteReg8

Write 8 bits of data to the I2C device.

```
int wiringPiI2CWriteReg8(int fd, int reg, int data)
```

Write 8 bits of data to the I2C device specifying the 8 bit register to write to.

See also:

- WiringPi I2C
- wiringPiI2CSetup

## wiringPiI2CWriteReg16

Write 16 bits of data to the I2C device.

```
int wiringPiI2CWriteReg16(int fd, int reg, int data)
```

Write 16 bits of data to the I2C device specifying register to write to. The order is LSB first.

See also:

- WiringPi I2C
- wiringPiI2CSetup

## wiringPiISR

Register an interrupt callback function.

```
int wiringPiISR(int pin, int edgeType, void (*functionName)(void))
```

This function identifies a pin and an interrupt type and when an appropriate signal occurs on the pin, the registered function will be automatically called. The `edgeType` parameter specifies the type of interrupt that is to be watched for:

- `INT_EDGE_FALLING` – Interrupt when signal goes from high to low.
- `INT_EDGE_RISING` – Interrupt when signal goes from low to high.
- `INT_EDGE_BOTH` – Interrupt on either a low to high or high to low.
- `INT_EDGE_SETUP` – Interrupt already externally defined for the pin.

The signature of the function to be called is:

```
void (*functionName)(void)
```

The function will be called in its own thread with a high priority (if the application has root authority to increase its priority). As such care should be taken that any work it does is thread safe with respect to the rest of the application.

The closest `gpio` command to this capability is the `gpio wfi` command. This command blocks waiting for a corresponding interrupt to occur at which point it will unblock and continue. The command does not poll and hence doesn't consume resources while waiting.

## wiringPiSetup

Initialize WiringPi.

```
int wiringPiSetup()
```

The pin numbering that will be used is that of WiringPi.

## wiringPiSetupGpio

Initialize WiringPi.

```
int wiringPiSetupGpio()
```

The pin numbering that will be used is based on the Broadcom numbering.

## wiringPiSetupPhys

Initialize WiringPi.

```
int wiringPiSetupPhys()
```

The pin number that will be used is physical pin numbering.

## wiringPiSetupSys

Initialize WiringPi.

```
int wiringPiSetupSys()
```

Use the `/sys/class/gpio` interface to interact with the hardware. When using this configuration, the pin numbering is Broadcom numbering (the same as `wiringPiSetupGpio`).

## wiringPiSPIDataRW

Read and Write SPI data.

```
int wiringPiSPIDataRW(int channel, unsigned char *data, int len)
```

Write and simultaneously read data from the given channel (0 or 1) through SPI. The data pointed to by the `data` parameter is written down the MOSI line and is replaced with data received through the MISO line.

See also:

- [WiringPi SPI](#)

## wiringPiSPISetup

Initialize an SPI channel.

```
int wiringPiSPISetup(int channel, int speed)
```

Initialize the SPI channel identified by the `channel` parameter (0 or 1). The `speed` is the clock speed in HZ and must be between 500,000 (500KHz) and 32,000,000 (32MHz).

See also:

- [WiringPi SPI](#)

## wpiPinToGpio

```
wpiPinToGpio(int wPiPin)
```

## wiringPiDev - LCD

The wiringPiDev – LCD functions contains code to drive the HD44780 devices. These are 16 columns by 2 row LCD displays. To use this library you will need to link with both `wiringPi` and the `wiringPiDev` libraries. You will also need to include `lcd.h`.

See also:

- WiringPi – [LCD Library](#)

### lcdInit

This is the main initialization function of the library.

```
int lcdInit (int rows, int cols, int bits, int rs, int strb, int d0, int d1, int d2,  
int d3, int d4, int d5, int d6, int d7)
```

The parameters are as follows:

- `rows` – The number of rows eg. 2
- `cols` – The number of columns eg. 16
- `bits` – The number of bits we will use in addressing. This will be either 4 or 8.
- `rs` – The RS pin (RS)
- `strb` – The Enable pin (E)
- `d0 ... d8` – The pins for the data lines. If we are using 4 bit mode, only configure d0-d3.

The return parameter is the "handle" to the current LCD display should we have multiple.

### lcdHome

Move the cursor to the home position.

```
void lcdHome(int handle)
```

### lcdClear

Clear the display.

```
void lcdClear(int handle)
```

### lcdPosition

Set the position of the next write.

```
void lcdPosition(int handle, int x, int y)
```

### lcdPutchar

Write a single character.

```
void lcdPutchar(int handle, uint8_t data)
```

### lcdPuts

Write a NULL terminated string.

```
void lcdPuts(int handle, char *string)
```

### lcdPrintf

Write a formatted string.

```
void lcdPrintf(int handle, char *message)
```

## **WiringPi Mapping from Arduino**

With the prevalence of Arduino sketches using Arduino APIs to interact with hardware, it is worth considering how these would map to the WiringPi APIs. By compiling a porting guide, we can keep a description of what is necessary to map one to the other.

Arduino	WiringPi
digitalWrite	digitalWrite
digitalRead	digitalRead
SPI.beginTransaction	N/A
SPI.setSettings	wiringPiSPISetup
SPI.transfer	wiringPiSPIDataRW
LOW	
HIGH	
Serial.println	printf
Serial.print	printf

## **Python RPi.GPIO reference**

The Python package that provides GPIO is called `python-rpi.gpio` or `python3-rpi.gpio`. It can be installed via apt-get. For example:

```
$ sudo apt-get install python-rpi.gpio
```

To use it in app, you must import the module:

```
import RPi.GPIO as GPIO
```

There are two possible pin numbering schemes for Python ... "BOARD" for Board numbering and "BCM" fro Broadcom numbering. It is mandatory to instruct the environment which one you wish to use by making a call to `setmode(GPIO.BOARD)` or `setmode(GPIO.BCM)`. We can query which one is in effect with a call to `getmode()`.

For any given pin, it can be configured as either input or output. We must call `setup()` to define the pin direction. We can either set `GPIO.OUT` for output or `GPIO.IN` for input.

The following sets a pin to be output:

```
GPIO.setup(17, GPIO.OUT)
```

while the following sets a pin to be input:

```
GPIO.setup(17, GPIO.IN)
```

Instead of a single pin, we can supply a list of pins to set them all to be input or output. When setting a pin to be output, we can also supply an initial value:

```
GPIO.setup(17, GPIO.OUT, initial=GPIO.HIGH)
```

For input pins, we can also specify any pull-up or pull-down resistor values:

```
GPIO.setup(17, GPIO.IN, pull_up_down=GPIO.PUD_UP)
```

```
GPIO.setup(17, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
```

To read a value from an input pin, we can call `input()` passing in the pin number to read:

```
myValue = GPIO.input(17)
```

The return will indicate the value read from the pin which will be either 1 (`GPIO.HIGH`) or 0 (`GPIO.LOW`).

To set the output value of an output pin, we can call `output()` to set the state. It takes the pin number we wish to use plus the desired new value.

```
GPIO.output(17, GPIO.LOW)
```

When we have finished working with GPIO pins, we can clean them up returning them to their default state with a call to `cleanup()`.

If we are watching for the value of a pin to change, we can always poll it but a better way is to block waiting for its value to change. We have a function named `wait_for_edge()` that allows us to block waiting for a change.

To wait for a pin to go from low to high we can use:

```
GPIO.wait_for_edge(17, GPIO.RISING)
```

to wait for a pin to go from high to low we can use:

```
GPIO.wait_for_edge(17, GPIO.FALLING)
```

to wait for a pin to change from either low to high or high to low we can use:

```
GPIO.wait_for_edge(17, GPIO.BOTH)
```

All of these can have an optional timeout specified in milliseconds.

Since `wait_for_edge()` is a blocking call and we have to be waiting for it before we will be informed, we have the capability to register an event using `add_event_detect()`. This will cause the value of the pin to be monitored and we can later call `event_detected()` to see if the event we were interested in has happened. If we want to be informed asynchronously, we can register a function that is called back when an event of interest to us has happened.

```
def myCallback(pinNumber):
    # Some code

GPIO.add_event_detect(17, GPIO.FALLING, myCallback)
```

If we no longer wish to be called back, we can cancel an event with `remove_event_detect()`.

See also:

- Python Programming
- [Python GPIO home page](#)

## Python SPI reference

There is a package called lthiery/SPI-Py that provides a Python library for SPI. A user of the class must import the package with:

```
import spi
```

See also:

- Github: [lthiery/SPI-Py](#)

### spi.openSPI

Open an SPI device for use

- `device=` - The device to use to interact with the SPI subsystem – default /dev/spidev0.0
- `speed=` - The speed in bits/second – default 500000
- `mode=` - default 0
- `bits=` - default 8
- `delay=` - default 0

### spi.transfer

Initiate a transfer

### spi.closeSPI

Close the use of a device.

## cURL API programming

The Curl package is a C language interface for performing a wide array of TCP/IP activities from a C language environment. It has excellent support and is well documented.

The package called `libcurl4-openssl-dev` should be installed.

To perform cURL API calls, the first activity must be a call to `curl_easy_init()`. This returns a handle that will be used in most of the subsequent calls. The handle should be released with a call to `curl_easy_cleanup()` when finished.

One of the most important CURL APIs is `curl_easy_setopt()`. This prepares the call to be made over the network.

There are many options available to us and you should study them all. However, there are ones which I consider more prevalent than others. These include:

- `CURLOPT_URL` – The network connection to the partner.
- `CURLOPT_UPLOAD` – Specify that we want to perform an upload (HTTP PUT).
- `CURLOPT_HTTPGET` – Specify that we want to perform a get request (HTTP GET).

Once the setup for a Curl call has been made, it can be performed by invoking `curl_easy_perform()`.

To recap:

1. Invoke `curl_easy_init()` to initialize
2. Invoke `curl_easy_setopt()` once per option
3. Invoke `curl_easy_perform()` to make the REST call

See also:

- [cURL home page](#)
- [libcurl man pages](#)

## Sample CURL application

Here is a sample curl application that makes a REST call to Thingspeak:

```
#include <stddef.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <curl.h>

static void checkRC(CURLcode rc, char *message) {
```

```

    if (rc != CURLE_OK) {
        printf("Error: %d %s\n", rc, message);
    }
}

int main(int argc, char *argv[]) {
    printf("About to start\n");
    char urlText[100];

    CURLcode rc;
    CURL *curl = curl_easy_init();
    if (curl == NULL) {
        printf("Failed to setup curl\n");
        exit(-1);
    }
    while(1) {
        int value = rand() % 20;
        printf("Setting value: %d\n", value);
        sprintf(urlText, "https://api.thingspeak.com/update?
api_key=D7YJKGFR1WV5D0UD&field1=%d", value);
        curl_easy_setopt(curl, CURLOPT_URL, urlText);
        curl_easy_setopt(curl, CURLOPT_HTTPGET, 1L);
        rc = curl_easy_perform(curl);
        checkRC(rc, "perform");
        sleep(20);
    }
    curl_easy_cleanup(curl);
    printf("Ending\n");
    return 0;
}

```

See also:

- REST requests
- [Thingspeak.com](#)

## Adafruit GFX library

Adafruit is a US company that specializes in electronics and computing projects and components. They contribute a massive amount to the community. One of their projects is called the Adafruit GFX library. This is a graphics library that is device independent and contains a wealth of graphical functions. This library has been written for the Arduino but has been ported to the Pi.

Included in its library are:

- void drawPixel(x, y, color)
- void drawLine(x0, y0, x1, y1, color)
- void drawFastVLine(x0, y0, length, color)
- void drawFastHLine(x0, y0, length, color)
- drawRect(x0, y0, w, h, color)

- `fillRect(x0, y0, w, h, color)`
- `drawCircle(x0, y0, r, color)`
- `fillCircle(x0, y0, r, color)`
- `drawRoundRect(x0, y0, w, h, radius, color)`
- `filledRoundRect(x0, y0, w, h, radius, color)`
- `drawTriangle(x0, y0, x1, y1, x2, y2, color)`
- `filledTriangle(x0, y0, x1, y1, x2, y2, color)`
- `drawChar(x, y, c, color, bg, size)`
- `setCursor(x, y)`
- `setTextColor(color)`
- `setTextColor(color, backgroundColor)`
- `setTextSize(size)`
- `setTextWrap(wrap)`
- `drawBitmap(x, y, bitmap, w, h, color)`
- `fillScreen(color)`
- `setRotation(rotation)`
- `width()`
- `height()`

See also:

- Adafruit – [GFX graphics library](#)
- LCD display – Nokia 5110 – PCD8544

## Interacting directly with the BCM2835 ARM Peripherals

The BCM2835 is the heart and soul of interacting with electronic components and circuits from a Pi. Generically, we will refer to all the "things" that can be connected to the Pi as peripherals. The BCM2835 implements in hardware the interaction between what an application wishes to happen and the physical interaction at the electrical level. Interacting with the BCM2835 directly is rarely needed to be done. This is because it is necessarily detailed which means that it is complex. Instead, higher level libraries such as WiringPi are available that encapsulate access to the device at much higher levels of abstraction. That said, an understanding of interfacing with the BCM2835 directly does no harm and should you decide that you need ultra low level access for a particular purpose, this is the way to go.

Interfacing with the BCM2835 is achieved by reading and writing from specific memory locations which are defined as being mapped to BCM2835 control functions. If we are working in a Raspbian

environment, we have a well understood code fragment that allows us to address real memory. It looks as follows:

```
int fd = open("/dev/mem", O_RDWR | O_SYNC);
if (fd < 0) {
    printf("Error opening /dev/mem\n");
    return;
}
void *map = mmap(
    NULL,
    4*1024,
    PROT_READ | PROT_WRITE,
    MAP_SHARED,
    fd,
    baseAddress);
if (map == MAP_FAILED) {
    close(fd);
    printf("Error mapping data\n");
    return;
}
volatile unsigned int *pData = map;
```

Lets start by looking at the identifies of the interfaces supported by the BCM2835. These are:

- GPIO
- SPI
- UART
- I2C bus
- DMA
- External Mass Media Controller
- Clocks
- Interrupts
- Timers
- PCM / I2S Audio
- PWM

The addresses for mappings are:

<b>Function</b>	<b>Pi1</b>	<b>Pi2</b>
BCM2835_BASE	0x2000 0000	0x3f00 0000
BCM2835_ST_BASE	0x2000 3000	0x3f00 3000
BCM2835_IRQ_BASE	0x2000 b000	0x3f00 b000
BCM2835_PM_BASE	0x2010 0000	0x3f10 0000
BCM2835_CM_BASE	0x2010 1000	0x3f10 1000
BCM2835_GPIO_BASE	0x2020 0000	0x3f20 0000
BCM2835_UART0_BASE	0x2020 1000	0x3f20 1000
BCM2835_PCM_BASE	0x2020 3000	0x3f20 3000
BCM2835_SPIO_BASE	0x2020 4000	0x3f20 4000
BCM2835_BSC0_BASE	0x2020 5000	0x3f20 5000
BCM2835_PWM_BASE	0x2020 c000	0x3f20 c000
BCM2835_BSCS_BASE	0x2021 4000	0x3f21 4000
BCM2835_BSC1_BASE	0x2080 4000	0x3f80 4000

Note explicitly that the base addresses for BCM2835 access differ from the Pi 1 architecture to the Pi 2.

The core clock speed of the BCM2835 on the Pi is 250MHz.

See also:

- [Low Level Programming of the Raspberry Pi in C](#)
- [BCM2835 ARM Peripherals](#)

## Auxiliary Peripherals

On the Pi the base offset is 0x2021 5000. On the Pi2 the base offset is 0x3f21 5000.

<b>Offset</b>	<b>Register Name</b>	<b>Description</b>
0x00	AUX_IRQ	Auxiliary Interrupt status
0x04	AUX_ENABLES	Auxiliary enables
0x40	AUX_MU_IO_REG	Mini Uart I/O data
0x44	AUX_MU_IER_REG	Mini Uart Interrupt Enable
0x48	AUX_MU_IIR_REG	Mini Uart Interrupt Identify
0x4c	AUX_MU_LCR_REG	Mini Uart Line Control
0x50	AUX_MU_MCR_REG	Mini Uart Modem Control
0x54	AUX_MU_LSR_REG	Mini Uart Line Status
0x58	AUX_MU_MSR_REG	Mini Uart Modem Status
0x5c	AUX_MU_SCRATCH	Mini Uart Scratch
0x60	AUX_MU_CNTL_REG	Mini Uart Extra Control
0x64	AUX_MU_STAT_REG	Mini Uart Extra Status
0x68	AUX_MU_BAUD_REG	Mini Uart Baudrate
0x80	AUX_SPI1_CNTL0_REG	SPI 1 Control Register 0
0x84	AUX_SPI1_CNTL1_REG	SPI 1 Control Register 1
0x88	AUX_SPI1_STAT_REG	SPI 1 Status
0x90	AUX_SPI1_IO_REG	SPI 1 Data
0x94	AUX_SPI1_PEEK_REG	SPI 1 Peek
0xc0	AUX_SPI2_CNTL0_REG	SPI 2 Control Register 0
0xc4	AUX_SPI2_CNTL1_REG	SPI 2 Control Register 1
0xc8	AUX_SPI2_STAT_REG	SPI 2 Status
0xd0	AUX_SPI2_IO_REG	SPI 2 Data
0xd4	AUX_SPI1_PEEK_REG	SPI 2 Peek

#### AUXIRQ - Check interrupts

<b>Bits</b>	<b>Name</b>	<b>Description</b>
2	SPI 2 IRQ	Set when SPI 2 has a pending interrupt
1	SPI 1 IRQ	Set when SPI 1 has a pending interrupt
0	Mini Uart IRQ	Set when Mini UART has a pending interrupt

#### AUX\_ENABLES - Enable the modules Mini UART, SPI1 and SPI2

<b>Bits</b>	<b>Name</b>	<b>Description</b>
2	SPI 2 enable	Set the SPI 2 module enabled / disabled
1	SPI 1 enable	Set the SPI 1 module enabled / disabled
0	Mini Uart enable	Set the Mini UART module enabled / disabled

#### AUX\_MUI\_IO\_REG - Read and write data

<b>Bits</b>	<b>Name</b>	<b>Description</b>
7:0		
7:0	Transmit	Write character to transmit FIFO
7:0	Read	Read character from receive FIFO

#### AUX MU IER REG - Enable interrupts

Bits	Name	Description
7:0	MS 8 bits of Baudrate	Used if DLAB=1
1	Enable transmit interrupt	Used if DLAB=0
0	Enable receive interrupt	Used if DLAB=0

Note: This is the correct settings as shown in the errata and differ from the spec sheet.

#### AUX MU IRR REG - Interrupt status

Bits	Name	Description
7:6	FIFO enables	
2:1	Interrupt ID / FIFO Clear	On read: <ul style="list-style-type: none"> <li>• b00 – no interrupts</li> <li>• b01 – transmit holding register empty</li> <li>• b10 – Receiver holds valid byte</li> </ul> On write: <ul style="list-style-type: none"> <li>• b01 – Clear the receive FIFO</li> <li>• b10 – Clear the transmit FIFO</li> <li>• b11 – Clear both the receive and transmit FIFOs</li> </ul>
0	Interrupt pending	Clear when an interrupt pending

#### AUX MU LCR REG

Bits	Name	Description
7	DLAB access	
6	Break	
1:0	Data size	<ul style="list-style-type: none"> <li>• b00 – 7 bit mode</li> <li>• b11 – 8 bit mode</li> </ul>

#### AUX MU MCR REG - Modem control

Bits	Name	Description
1	RTS	<ul style="list-style-type: none"> <li>• 0 – RTS line is set high</li> <li>• 1 – RTS line is set low</li> </ul>

#### AUX MU LSR REG - Data status

Bits	Name	Description
6	Transmitter idle	1 – Transmit FIFO is empty
5	Transmitter empty	1 – Transmit FIFO can accept at least one byte
1	Receiver overrun	1 – Receiver overrun
0	Data ready	1 – receive FIFO holds at least 1 character

#### AUX MU MSR REG - Modem status

Bits	Name	Description
5	CTS status	

#### AUX MU CNTL REG - Extra controls

Bits	Name	Description
7	CTS assert level	<ul style="list-style-type: none"> <li>• 0 – CTS auto flow assert level is high</li> <li>• 1 – CTS auto flow assert level is low</li> </ul>
6	RTS assert level	<ul style="list-style-type: none"> <li>• 0 – RTS auto flow assert level is high</li> <li>• 1 – RTS auto flow assert level is low</li> </ul>
5:4	RTS AUTO flow level	<ul style="list-style-type: none"> <li>• b00 – De-assert RTS when the receive FIFO has 3 empty spaces</li> <li>• b01 – De-assert RTS when the receive FIFO has 2 empty spaces</li> <li>• b10 – De-assert RTS when the receive FIFO has 1 empty space</li> <li>• b11 – De-assert RTS when the receive FIFO has 4 empty spaces</li> </ul>
3	Transmit auto flow control using CTS	<ul style="list-style-type: none"> <li>• 0 – Transmitter ignores the status line</li> <li>• 1 – Transmitter will stop if CTS line is de-asserted</li> </ul>
2	Receive auto flow control using RTS	<ul style="list-style-type: none"> <li>• 0 – RTS line is controlled by AUX_MU_MCR_REG bit 1</li> <li>• 1 – RTS line will de-assert if the FIFO reaches its "auto flow" level.</li> </ul>
1	Transmitter enable	<ul style="list-style-type: none"> <li>• 0 – UART transmitter disabled</li> <li>• 1 – UART transmitted enabled</li> </ul>
0	Receiver enable	<ul style="list-style-type: none"> <li>• 0 – UART receiver disabled</li> <li>• 1 – UART receiver enabled</li> </ul>

#### AUX MU STATE REG - Extra status bits

Bits	Name	Description
27:24	Transmit FIFO fill level	
19:16	Receive FIFO fill level	
9	Transmitter done	
8	Transmit FIFO empty	
7	CTS line	
6	RTS status	
5	Transmit FIFO full	
4	Receiver overrun	
3	Transmitter idle	
2	Receiver idle	
1	Space available	
0	Symbol available	

#### AUX MU BAUD REG - Baudrate counter

Bits	Name	Description
15:0	Baudrate	Mini UART baudrate counter

The baudrate is given by the following formula:

$$\text{baudrate} = \frac{\text{system\_clock\_freq}}{8 * (\text{baudrate\_reg} + 1)}$$

By some algebra, we also have:

$$\text{baudrate\_reg} = \frac{\text{system\_clock\_freq} - 1}{8 * \text{baud}}$$

Here are some worked values for the PI system clock frequency of 250MHz:

Baudrate	Register value
115200	270
57600	541
19200	1626
9600	3254

Sample UART algorithm:

```

AUX_ENABLES      = 0x1 // Enable the Mini UART
AUX_MU_IER_REG  = 0x0 // Doesn't appear to do anything???

```

```

AUX_MU_CNTL_REG = 0x0 // All disabled
AUX_MU_LCR_REG = 0b11 // Work in 8 bit data mode
AUX_MU_MCR_REG = 0x00 // Set RTS to low
AUX_MU_IER_REG = 0x0 // Doesn't
AUX_MU_IIR_REG = xC6 (0b1100 0110)
AUX_MU_BAUD_REG = 270 // Baud rate counter
AUX_MU_CNTL_REG = 0b10 // Transmitter enabled
// Loop until transmitter is empty
while (bit(AUX_MU_LSR_REG, transmitter empty) == 0 {
}
AUX_MU_IO_REG = <character>;

```

## BCM2835 GPIO

The BCM2835 has a whopping 54 GPIO lines at its disposal. However, these lines are also used by the other functions and hence you can't use these lines as both GPIO and their alternate functions at the same time. For example, if you wish to use I2C, then two of the GPIO lines become used for I2C clock and data. It is important to realize that there are mapping at play.

The mappings are as follows:

GPIO	Alt 0	Alt 1	Alt 2	Alt 3	Alt 4	Alt 5
GPIO0	SDA0	SA5				
GPIO1	SCL0	SA4				
GPIO2	SDA1	SA3				
GPIO3	SCL1	SA2				
GPIO4	GPCLK0	SA1				ARM_TDI
GPIO5	GPCLK1	SA0				ARM_TDO
GPIO6	GPCLK2	SOE				ARM_RTCK
GPIO7	SPI0_CE1	SWE				
GPIO8	SPI1_CE0	SD0				
GPIO9	SPI0_MISO	SD1				
GPIO10	SPI0_MOSI	SD2				
GPIO11	SPI0_SCLK	SD3				
GPIO12	PWM0	SD4				ARM_TMS
GPIO13	PWM1	SD5				ARM_TCK
GPIO14	TXD0	SD6				TXD1
GPIO15	RXD0	SD7				RXD1
GPIO16		SD8		CTS0	SPI1_CE2	CTS1
GPIO17		SD9		RTS0	SPI1_CE1	RTS1
GPIO18	PCM_CLK	SD10		BSCSL SDA / MOSI	SPI1_CE0	PWM0
GPIO19	PCM_FS	SD11		BSCSL SCL/SCLK	SPI1_MISO	PWM1
GPIO20	PCM_DIN	SD12		BSCSL / MISO	SPI1_MOSI	GPCLK0
GPIO21	PCM_DOUT	SD13		BSCSL CE	SPI1_SCLK	GPCLK1
GPIO22		SD14		SD1_CLK	ARM_TRST	
GPIO23		SD15		SD1_CMD	ARM_RTCK	
GPIO24		SD16		SD1_DAT0	ARM_TDO	
GPIO25		SD17		SD1_DAT1	ARM_TCK	

GPIO26				SD1_DAT2	ARM_TDI	
GPIO27				SD1_DAT3	ARM_TMS	
GPIO28	SDA0	SA5	PCM_CLK			
GPIO29	SCL0	SA4	PCM_FS			
GPIO30		SA3	PCM_DIN	CTS0		CTS1
GPIO31		SA2	PCM_DOUT	RTS0		RTS1
GPIO32	GPCLK0	SA1		TXD0		TXD1
GPIO33		SA0		RXD0		RXD1
GPIO34	GPCLK0	SOE				
GPIO35	SPI0_CE1	SWE				
GPIO36	SPI0_CE0	SD0	TXD0			
GPIO37	SPI0_MISO	SD1	RXD0			
GPIO38	SPI0_MOSI	SD2	RTS0			
GPIO39	SPI0_SCLK	SD3	CTS0			
GPIO40	PWM0	SD4			SPI2_MISO	TXD1
GPIO41	PWM1	SD5			SPI2_MOSI	RXD1
GPIO42	GPCLK1	SD6			SPI2_SCLK	RTS1
GPIO43	GPCLK2	SD7			SPI2_CE0	CTS1
GPIO44	GPCLK1	SDA0			SPI2_CE1	
GPIO45	PWM1	SCL0			SPI2_CE2	
GPIO46						
GPIO47						
GPIO48						
GPIO49						
GPIO50						
GPIO51						
GPIO52						
GPIO53						

### Register addresses

The physical memory register addresses for GPIO capabilities are offset from the base addresses for GPIO which we give the symbolic name `BCM2835_GPIO_BASE` which is `0x2020 0000` for most Pis and `0x3f20 0000` for the Pi 2. The offset for GPIO from the base of the BCM2835 memory start is `0x0020 0000`.

<b>Offset</b>	<b>Function</b>	<b>Description</b>	<b>Access</b>
0x00	GPFSEL0	GPIO Function Select 0	R/W
0x04	GPFSEL1	GPIO Function Select 1	R/W
0x08	GPFSEL2	GPIO Function Select 2	R/W
0x0c	GPFSEL3	GPIO Function Select 3	R/W
0x10	GPFSEL4	GPIO Function Select 4	R/W
0x14	GPFSEL5	GPIO Function Select 5	R/W
0x1c	GPSET0	GPIO Pin Output Set 0	W
0x20	GPSET1	GPIO Pin Output Set 1	W
0x28	GPCLR0	GPIO Pin Output Clear 0	W
0x2c	GPCLR1	GPIO Pin Output Clear 1	W
0x34	GPLEV0	GPIO Pin Level 0	R
0x38	GPLEV1	GPIO Pin Level 1	R
0x40	GPEDS0	GPIO Pin Event Detect Status 0	R/W
0x44	GPEDS1	GPIO Pin Event Detect Status 1	R/W
0x4c	GPREN0	GPIO Pin Rising Edge Detect Enable 0	R/W
0x50	GPREN1	GPIO Pin Rising Edge Detect Enable 1	R/W
0x58	GPFEN0	GPIO Pin Falling Edge Detect Enable 0	R/W
0x5c	GPFEN1	GPIO Pin Falling Edge Detect Enable 1	R/W
0x64	GPHEN0	GPIO Pin High Detect Enable 0	R/W
0x68	GPHEN1	GPIO Pin High Detect Enable 1	R/W
0x70	GPLEN0	GPIO Pin Low Detect Enable 0	R/W
0x74	GPLEN1	GPIO Pin Low Detect Enable 1	R/W
0x7c	GPAREN0	GPIO Pin Async. Rising Edge Detect 0	R/W
0x80	GPAREN1	GPIO Pin Async. Rising Edge Detect 1	R/W
0x88	GPAFEN0	GPIO Pin Async. Falling Edge Detect 0	R/W
0x8c	GPAFEN1	GPIO Pin Async. Falling Edge Detect 1	R/W
0x94	GPPUD	GPIO Pin Pull-up/down Enable	R/W
0x98	GPPUDCLK0	GPIO Pin Pull-up/down Enable Clock 0	R/W
0x9c	GPPUDCLK1	GPIO Pin Pull-up/down Enable Clock 1	R/W

### GPIO function select setting

Each of the 54 GPIO pins can, at any one time, have one of eight different functions assigned to it. These are input, output, function 0, function 1, function 2, function 3, function 4, function 5,

For any given pin, the function can thus be encoded in three bits:

<b>Value</b>	<b>Value Hex</b>	<b>Meaning</b>
000	0x0	Input
001	0x1	Output
100	0x4	Function 0
101	0x5	Function 1
110	0x6	Function 2
111	0x7	Function 3
011	0x3	Function 4
010	0x2	Function 5

To provide the ability to select the function for each GPIO, there are 6 BCM2835 registers called GPFSEL0 – GPFSEL5. These contain the 3 bit settings for each of the pins.

These are mapped as follows:

GPFSEL0

29:27	FSEL9
26:24	FSEL8
23:21	FSEL7
20:18	FSEL6
17:15	FSEL5
14:12	FSEL4
11:9	FSEL3
8:6	FSEL2
5:3	FSEL1
2:0	FSEL0

GPFSEL1

29:27	FSEL19
26:24	FSEL18
23:21	FSEL17
20:18	FSEL16
17:15	FSEL15
14:12	FSEL14
11:9	FSEL13
8:6	FSEL12
5:3	FSEL11
2:0	FSEL10

GPFSEL2

29:27	FSEL29
26:24	FSEL28
23:21	FSEL27
20:18	FSEL26
17:15	FSEL25
14:12	FSEL24
11:9	FSEL23
8:6	FSEL22
5:3	FSEL21
2:0	FSEL20

GPFSEL3

29:27	FSEL39
26:24	FSEL38
23:21	FSEL37
20:18	FSEL36
17:15	FSEL35
14:12	FSEL34
11:9	FSEL33
8:6	FSEL32
5:3	FSEL31
2:0	FSEL30

GPFSEL4

29:27	FSEL49
26:24	FSEL48
23:21	FSEL47
20:18	FSEL46
17:15	FSEL45
14:12	FSEL44
11:9	FSEL43
8:6	FSEL42
5:3	FSEL41
2:0	FSEL40

GPFSEL5

11:9	FSEL53
8:6	FSEL52
5:3	FSEL51
2:0	FSEL50

For example, if we wished to set Pin 17 as an output we would set the bits 23:21 of the GPFSEL1 register to 001.

The general algorithm to get the function of a pin is:

```
value = ((* ((pin / 10) * 4) + &GPFSEL0) >> (3 * (pin % 10))) & 0x7
```

#### GPIO set high level - GPSET0 and GPSET1

The registers GPSET0 and GPSET1 control setting corresponding GPIO lines high. The GPSET0 register controls GPIOs 0-31. Setting a corresponding bit to 0 has no effect but setting a corresponding bit to 1 causes the GPIO line to be driven high (assuming it is an output pin).

Similarly, the GPSET1 register controls GPIOs 32-53 using the same notion as for GPSET0.

#### GPIO set low level - GPCLR0 and GPCLR1

The registers GPCLR0 and GPCLR1 control setting the corresponding GPIO lines low. The GPCLR0 register controls GPIOs 0-31. Setting a corresponding bit to 0 does nothing but setting a corresponding bit to 1 causes the GPIO line to be driven low (assuming it is an output pin).

Similarly, the GPCLR1 register controls GPIOs 32-53 using the same notion as for GPCLR0.

#### GPIO query levels - GPLEV0 and GPLEV1

The registers GPLEV0 and GPLEV1 are used to query the values of the signals found on the GPIO lines. GPLEV0 returns the GPIO values for GPIOs 0-31 while GPLEV1 returns the GPIO values for GPIOs 32-53.

#### GPIO event detection - GPEDS0 and GPEDS1

When the BCM2835 has been asked to watch for changing levels of signals, these registers indicate which (if any) of the GPIOs have been triggered. GPEDS0 contains the status bits for GPIOs 0-31 while GPEDS1 contains the status bits for GPIOs 32-53. A 0 value for a bit means that an event was not detected while a 1 value means it was. Writing a 1 to a bit corresponding to a GPIO causes it to reset.

#### GPIO rising edge detection - GPREN0 and GPREN1

The registers GPREN0 and GPREN1 determine which GPIO lines will trigger an indication of an event when the signal changes from low to high. If enabled for a GPIO, this will set the corresponding flag in the GPEDS0 and GPEDS1 registers.

The GPREN0 enables/disables rising edge detection for GPIOs 0-31 while GPREN1 enables/disables rising edge detection for GPIOs 32-53. A 0 bit value disables while a 1 bit value enables.

#### GPIO rising edge detection - GPFEN0 and GPFEN1

The registers GPFEN0 and GPFEN1 determine which GPIO lines will trigger an indication of an event when the signal changes from high to low. If enabled for a GPIO, this will set the corresponding flag in the GPEDS0 and GPEDS1 registers.

The GPFEN0 enables/disables falling edge detection for GPIOs 0-31 while GPFEN1 enables/disables falling edge detection for GPIOs 32-53. A 0 bit value disables while a 1 bit value enables.

### GPIO high detection - GPHEN0 and GPHEN1

The registers GPHEN0 and GPHEN1 determine which GPIO lines will trigger an indication of an event when the signal is high. If enabled for a GPIO, this will set the corresponding flag in the GPEDS0 and GPEDS1 registers.

The GPHEN0 enables/disables high detection for GPIOs 0-31 while GPHEN1 enables/disables high detection for GPIOs 32-53. A 0 bit value disables while a 1 bit value enables.

### GPIO low detection - GPLENO and GPLEN1

The registers GPLENO and GPLEN1 determine which GPIO lines will trigger an indication of an event when the signal is low. If enabled for a GPIO, this will set the corresponding flag in the GPEDS0 and GPEDS1 registers.

The GPLENO enables/disables low detection for GPIOs 0-31 while GPLEN1 enables/disables low detection for GPIOs 32-53. A 0 bit value disables while a 1 bit value enables.

### GPIO Pullup/Pulldown - GPPUD

Bit	Description
1:0	00 – Off 01 – Pull down 10 – Pull up 11 – Do not use

## **BCM2835 UART**

The BCM2835 provides a couple of UARTs known as the Mini UART and a full blown UART. All UARTs on the Pi are 3.3V signal level so take care when connecting to 5V devices.

The UART pins are mapped as follows:

Pin	ALT0	ALT2	ALT3
GPIO14	TXD0		
GPIO15	RXD0		
GPIO16			CTS0
GPIO17			RTS0
GPIO30			CTS0
GPIO31			RTS0
GPIO32			TXD0
GPIO33			RXD0
GPIO36		TXD0	
GPIO37		RXD0	
GPIO38		RTS0	
GPIO39		CTS0	

As we can see, the same functions are available on different pin combinations.

There are a number of interrupts available:

## UARTRXINTR

## UARTRTINTR

### Registers

The registers for this device are based as 0x2020 1000 on Pi 1 and 0x3f20 1000 on Pi2.

Address	Name	Description
0x00	DR	Data register
0x04	RSRECR	???
0x18	FR	Flag register
0x24	IBRD	Integer baud rate divisor
0x28	FBRD	Fractional baud rate divisor
0x2c	LCRH	Line control register
0x30	CR	Control register
0x34	IFLS	Interrupt FIFO level select register
0x38	IMSC	Interrupt mask set clear register
0x40	MIS	Masked interrupt status register
0x44	ICR	Interrupt clear register
0x80	ITCR	Test control register
0x84	ITIP	Integration test input register
0x88	ITOP	Integration test output register
0x8c	TDR	Test Data register

### DR - Data Register

This is the register used to read and write data from the UART. It is broken out as follows:

Bits	Name	Description
11	OE	Overrun error
10	BE	Break error
9	PE	Parity error
8	FE	Framing error
7:0	DATA	Read received data / write transmission data

### RSRECR - Receive status / error clear Register

Bits	Name	Description
3	OE	Overrun error
2	BE	Break error
1	PE	Parity error
0	FE	Framing error

### FR - Flag Register

<b>Bits</b>	<b>Name</b>	<b>Description</b>
7	TXFE	Transmit FIFO empty
6	RXFF	Receive FIFO empty
5	TXFF	Transmit FIFO full
4	RXFE	Receive FIFO empty
3	BUSY	UART busy
0	CTS	Clear to send

### IBRD - Integer baud rate divisor

<b>Bits</b>	<b>Name</b>	<b>Description</b>
31:16	N/A	Write 0
15:0	IBRD	The integer baud rate divisor

### FBRD - Fractional baud rate divisor

<b>Bits</b>	<b>Name</b>	<b>Description</b>
31:16	N/A	Write 0
15:0	FBRD	The fractional baud rate divisor

### LCRH - Line control register

Bits	Name	Description
31:8	N/A	Write 0
7	SPS	Stick parity select <ul style="list-style-type: none"> <li>• 0 – stick parity is disabled</li> <li>• 1 - ???</li> </ul>
6:5	WLEN	Word length: <ul style="list-style-type: none"> <li>• b11 – 8 bits</li> <li>• b10 – 7 bits</li> <li>• b01 – 6 bits</li> <li>• b00 – 5 bits</li> </ul>
4	FEN	Enable FIFOs <ul style="list-style-type: none"> <li>• 0 – disabled</li> <li>• 1 – enabled</li> </ul>
3	STP2	Two stop bits selected <ul style="list-style-type: none"> <li>• 1 – Two stop bits transmitted</li> </ul>
2	EPS	Even parity select <ul style="list-style-type: none"> <li>• 0 – odd parity</li> <li>• 1 – even parity</li> </ul>
1	PEN	Parity enable <ul style="list-style-type: none"> <li>• 0 – parity disabled</li> <li>• 1 – parity enabled</li> </ul>
0	BRK	Send break <ul style="list-style-type: none"> <li>• ???</li> </ul>

### CR - Control register

Bits	Name	Description
15	CTSEN	CTS hardware flow control <ul style="list-style-type: none"> <li>• 1 – hardware flow control enabled</li> <li>• 0 – hardware flow control disabled</li> </ul>
14	RTSEN	RTS hardware flow control <ul style="list-style-type: none"> <li>• 1 – hardware flow control enabled</li> <li>• 0 – hardware flow control disabled</li> </ul>
11	RTS	
9	RXE	Receive enabled
8	TXE	Transmit enabled
7	LBE	Loopback enabled
0	UARTEN	UART enable <ul style="list-style-type: none"> <li>• 0 – UART is disabled</li> <li>• 1 – UART is enabled</li> </ul>

### IFLS - Interrupt FIFO level select register

<b>Bits</b>	<b>Name</b>	<b>Description</b>
5:3	RXIFLSEL	Interrupt control for interrupt generation <ul style="list-style-type: none"><li>• b000 – 1/8 full</li><li>• b001 – ¼ full</li><li>• b010 – ½ full</li><li>• b011 – ¾ full</li><li>• b100 – 7/8 full</li></ul>
2:0	TXIFLSEL	Interrupt control for interrupt generation <ul style="list-style-type: none"><li>• b000 – 1/8 full</li><li>• b001 – ¼ full</li><li>• b010 – ½ full</li><li>• b011 – ¾ full</li><li>• b100 – 7/8 full</li></ul>

### IMSC - Interrupt mask set / clear register

<b>Bits</b>	<b>Name</b>	<b>Description</b>
10	OEIM	Overrun error interrupt mask
9	BEIM	Break error interrupt mask
8	PEIM	Parity error interrupt mask
7	FEIM	Framing error interrupt mask
6	RTIM	Receive timeout interrupt mask
5	TXIM	Transmit interrupt mask
4	RXIM	Receive interrupt mask
1	CTSMIM	

### RIS - Raw interrupt status register

<b>Bits</b>	<b>Name</b>	<b>Description</b>
10	OERIS	Overrun error interrupt status
9	BERIS	Break error interrupt status
8	PERIS	Parity error interrupt status
7	FERIS	Framing error interrupt status
6	RTRIS	Receive timeout interrupt status
5	TXRIS	Transmit interrupt status
4	RXRIS	Receive interrupt status
1	CTSRMIS	

### MIS - Masked interrupt status register

Bits	Name	Description
10	OEMIS	Overrun error masked interrupt status
9	BEMIS	Break error masked interrupt status
8	PEMIS	Parity error masked interrupt status
7	FEMIS	Framing error masked interrupt status
6	RTMIS	Receive timeout masked interrupt status
5	TXMIS	Transmit masked interrupt status
4	RXMIS	Receive masked interrupt status
1	CTSMMIS	

### ICR - Interrupt clear register

Bits	Name	Description
10	OEIC	Overrun error interrupt clear
9	BEIC	Break error interrupt clear
8	PEIC	Parity error interrupt clear
7	FEIC	Framing error interrupt clear
6	RTIC	Receive timeout interrupt clear
5	TXIC	Transmit interrupt clear
4	RXIC	Receive masked interrupt status
1	CTSMIC	

## BCM2835 PWM

For the theory of PWM see the section on abstract PWM. Within the BCM2835, there are two distinct PWM output channels. PWM can output in two different modes. The first an easiest mode to understand is simple mark/space ratios. Given then we want to output a value between 0 and 1 then if the period width is  $P$  then the output will be high for  $v \cdot P$  and low for  $(1-v) \cdot P$ . Let us look at an example of that. If the period is 20 msecs and we wish to output a value of 0.2 then we will be high for 4 msecs and low for 16 msecs.

The other mode is a little muddier. What it wants to do is output highs and lows interspersed with each other such that the ratio of duration of High / P equals the value such that the high values over the period are as evenly spaced as possible. Control of the mode of operation of the BCM2835 is set by the MSEN bit (Mark/Space Enabled). When its value is 1, then mark/space mode is used otherwise the equal spacing mode is used.

## Registers

The registers for the PWM device are based as 0x2020 xxxx on Pi 1 and 0x3f20 xxxx on Pi2.

<b>Address</b>	<b>Name</b>	<b>Description</b>
0x00	CTL	PWM Control
0x04	STA	PWM Status
0x08	DMAC	PWM DMA Configuration
0x10	RNG1	PWM Channel 1 Range
0x14	DAT1	PWM Channel 1 Data
0x18	FIF1	PWM FIFO Input
0x20	RNG2	PWM Channel 2 Range
0x24	DAT2	PWM Channel 2 Data

### CTL - PWM Control

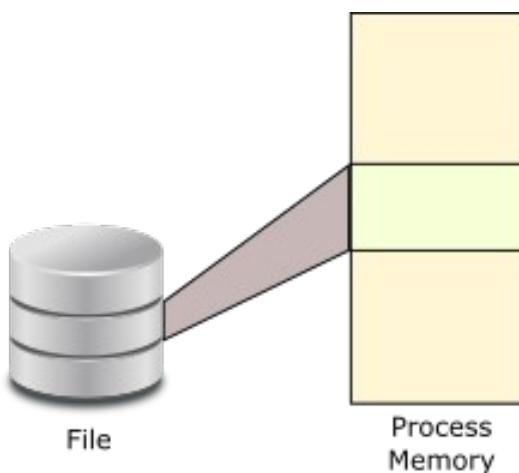
<b>Bits</b>	<b>Name</b>	<b>Description</b>
31:16	N/A	
15	MSEN2	M/S Enable for channel 2: <ul style="list-style-type: none"> <li>• 0 – PWM algorithm</li> <li>• 1 – Mark/Space algorithm</li> </ul>
14	N/A	
13	USEF2	Use FIFO for Channel 2: <ul style="list-style-type: none"> <li>• 0 – Data register is transmitted</li> <li>• 1 – FIFO used for transmission</li> </ul>
12	POLA2	Polarity for Channel 2: <ul style="list-style-type: none"> <li>• 0 – 0=Low, 1=High</li> <li>• 1 – 1=Low, 0=High</li> </ul>
11	SBIT2	Silence bit for Channel 2
10	RPTL2	Repeat data for Channel 2: <ul style="list-style-type: none"> <li>• 0 – Transmission ends when FIFO is empty</li> <li>• 1 – Last data in FIFO is retransmitted</li> </ul>
9	MODE2	Mode for Channel 2: <ul style="list-style-type: none"> <li>• 0 – PWM mode</li> <li>• 1 – Serializer mode</li> </ul>
8	PWEN2	Enabled for channel 2: <ul style="list-style-type: none"> <li>• 0 – Channel is disabled</li> <li>• 1 – Channel is enabled</li> </ul>
7	MSEN1	M/S Enable for channel 1: <ul style="list-style-type: none"> <li>• 0 – PWM algorithm</li> <li>• 1 – Mark/Space algorithm</li> </ul>
6	CLRF1	Clear FIFO: <ul style="list-style-type: none"> <li>• 1 – Immediately clears the FIFO</li> </ul>
5	USEF1	Use FIFO for Channel 1:

		<ul style="list-style-type: none"> <li>• 0 – Data register is transmitted</li> <li>• 1 – FIFO used for transmission</li> </ul>
4	POLA1	Polarity for Channel 1: <ul style="list-style-type: none"> <li>• 0 – 0=Low, 1=High</li> <li>• 1 – 1=Low, 0=High</li> </ul>
3	SBIT1	Silence bit for Channel 1
2	RPTL1	Repeat data for Channel 1: <ul style="list-style-type: none"> <li>• 0 – Transmission ends when FIFO is empty</li> <li>• 1 – Last data in FIFO is retransmitted</li> </ul>
1	MODE1	Mode for Channel 1: <ul style="list-style-type: none"> <li>• 0 – PWM mode</li> <li>• 1 – Serializer mode</li> </ul>
0	PWEN1	Enabled for channel 1: <ul style="list-style-type: none"> <li>• 0 – Channel is disabled</li> <li>• 1 – Channel is enabled</li> </ul>

### Interacting using mmap()

Imagine a file on the file system containing binary data. We can read and write from that file using system calls in order to access its data. As an alternative mechanism, imagine that we opened the file and then "mapped" it to an area of memory addressable by our process. If this were done then read and writes to and from the memory would be mapped into corresponding reads and writes from the file. In some cases this may be much more efficient. In other cases, this may provide an easier to use and more elegant programming model. It is the `mmap()` system call that allows us to map a file to an area of memory.

This is illustrated in the following diagram.



For the Pi, we have an additional usage pattern. Consider the BCM2835 I/O device. It is almost exclusively based upon reading and writing memory locations. The memory though is designed to be accessed by the kernel and as such as is not part of the virtual address space of user processes but instead

part of the global address space seen by the kernel. Again, using `mmap()`, we can map the address space of the BCM2835 device into the address space of a process and then all of a sudden we have the ability to interact with the BCM2835 through processes.

Let us start by taking a look at the specification of `mmap`. It has the following signature:

```
void *mmap(  
    void *start,  
    size_t length,  
    int prot,  
    int flags,  
    int fd,  
    off_t offset)
```

It has a specific header file that must be included that contains both its declaration and constant definitions. The header is `sys/mman.h`.

Let us now start to look at its parameters. The first is `start`. This is where in the address space of the process the memory should be placed. Most times we don't care so we can pass in `NULL`. This allows the system call to choose for us where the mapped memory can be found in our address space. The return from the call is the address actually chosen for us. The combination of `length` and `offset` define where within the source data the mapped data will be located. The `length` defines how many bytes of the source will be mapped and the `offset` specifies an offset from the start of the source data that will be mapped. To provide access control, the `prot` parameter is a combination of flags specifying what we are requesting to be able to do to the memory. The options are:

- `PROT_NONE` – Don't do anything with the memory
- `PROT_READ` – Request the ability to read the data
- `PROT_WRITE` – Request the ability to write the data
- `PROT_EXEC` – Request the ability to execute the data

The `flags` parameter provides additional control to the memory mapping. Primary among the flags are `MAP_SHARED` and `MAP_PRIVATE`. Specifying `MAP_SHARED` means that updates to the data are passed through to the underlying memory of the file that is the device. `MAP_PRIVATE` means that changes are not propagated back down to the underlying file.

A corresponding system call named `munmap()` is available to un-map a previously mapped area of storage. Its format is:

```
int munmap(void *start, size_t length)
```

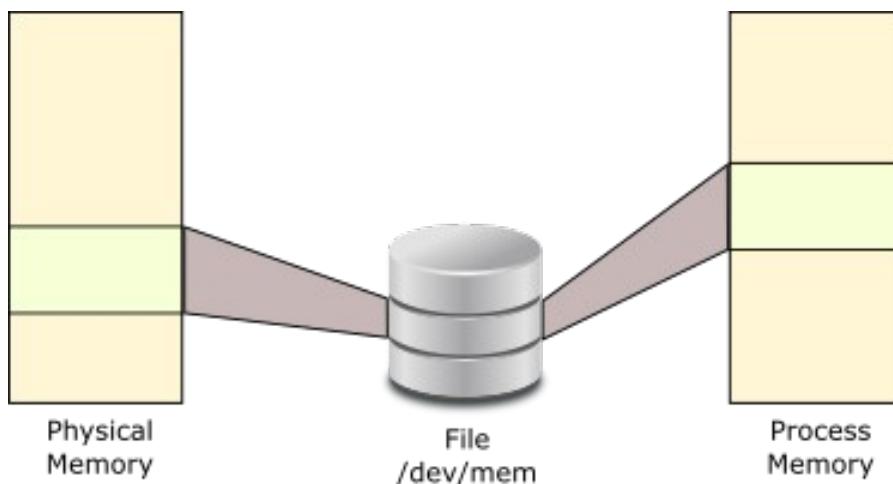
With a bit of theory of `mmap()` behind us, let us now look at how we might be able to use that with the Pi and accessing the BCM2835. The trick to our story is that there is a special device driver file called `/dev/mem`. One needs to be root in order to work with this file. What it does is provide raw access to the physical address space of the Pi and not the virtual address space seen by our process. For example, if we could open this file and read from address `0x1234` then we would be reading from the physical RAM

address of 0x1234. Contrast this with reading from address 0x1234 in our process ... which we might simply achieve with:

```
unsigned char *myPtr = (unsigned char *)0x1234;
unsigned char value = *myPtr;
```

This would read from address 0x1234 in the virtual address space of our process.

The next diagram illustrates this notion. Note that it is important to realize the /dev/mem is **not** a real file on the file system. No SD micro card I/O will be performed when one reads and writes to it. Rather it is a logical "bridge" between process memory as seen by a user level Raspbian application and the raw data of the physical RAM.



First, let us look at how we would open the /dev/mem file. We can do that with the `open()` system call.

```
int fd = open("/dev/mem", O_RDWR | O_SYNC)
```

once opened, we can create a mapping to memory with:

```
volatile unsigned char *memoryMap = map(
    NULL, // Let the OS decide where to map into our virtual address space
    4096, // Map in one block (4K) of data
    PROT_READ | PROT_WRITE, // Ask for both read and write access to the memory
    MAP_SHARED, // Ask for shared access which will cause writes to be propagated to the
    real memory
    fd, // The file descriptor to the /dev/mem device
    GPIO_MEMORY_BASE // The offset to the base of GPIO memory
)
```

Notice that we asked that the variable we are using to hold the pointer to our address map to be flagged as volatile. What this means is that the compiler makes no assumptions about its value during optimization. Imagine code such as the following:

```
unsigned char *myPtr = ...;           // Some pointer value
*myPtr = 'X';                      // Store 'X' at the pointer address
unsigned char myValue = *myPtr; // Retrieve what is at the pointer address
```

A smart compiler might believe that since it just stored 'X' at the memory location, if we read it back it will still be 'X' so why bother reading it, just assume it is 'X'. However, for our device I/O story, this would be wrong.

See also:

- man(2) – [mmap](#)
- Wikipedia – [mmap](#)

## Common C language includes

There are a plethora of include files available for the Pi and knowing which ones to include can be hard. Here are some cheat sheets.

### Missing data types

int16\_t, uint16\_t – stdint.h

boolean – stdbool.h ... however this was not found and I opted for:

```
typedef bool boolean;
```

### String functions

The majority of string functions can be found in string.h including:

strncpy

### Math functions

abs – stdlib.h

## Cheat Sheets

The next few pages are cheat sheets for various Pi items. Cheat sheets can be very useful when we know what we want and we can't commit all the details to memory. In every area of study there is a wealth of detail and the Pi is no exception. My recommendation is to learn the concepts and don't try and memorize the details nor get frustrated if you try and can't. As long as you know something exists or the principles of how to achieve a goal, the details can always be looked up later ... and hence the cheat sheet ... which in reality ... isn't any kind of "cheat" at all.

# Wiring Pi Cheat Sheet

## Timing

```
void delay(duration)
void delayMicroseconds(duration)
unsigned int micros()
unsigned int millis()
```

## GPIO

```
int digitalRead(pin)
int digitalWrite(pin, value)
int digitalWriteByte(value)
void pinMode(pin, mode)
void pullUpDnControl(pin, pud)
int wiringPiISR(pin, edgeType, functionName)
```

## Pin modes

- INPUT
- OUTPUT
- PWM\_CLOCK

## Pull-up/Pull-down

- PUD\_OFF
- PUD\_UP
- PUD\_DOWN

## Edge Type

- INT\_EDGE\_FALLING
- INT\_EDGE\_RISING
- INT\_EDGE\_BOTH
- INT\_EDGE\_SETUP

## PWM

```
pwmSetClock(divisor)
pwmSetMode(mode)
pwmSetRange(range)
pwmWrite(pin, value)
```

## PWM Modes

- PWM\_MODE\_BAL
- PWM\_MODE\_MS

## Setup

wiringPiSetup()	WiringPi
wiringPiSetupGpio()	Broadcom
wiringPiSetupPhys()	Physical

## I2C

```
int wiringPiI2CSetup(address)
int wiringPiI2CRead(fd)
int wiringPiI2CReadReg8(fd, reg)
int wiringPiI2CReadReg16(fd, reg)
int wiringPiI2CWrite(fd, reg, data)
int wiringPiI2CWriteReg8(fd, data)
int wiringPiI2CWriteReg16(fd, reg, data)
```

## Serial

```
void serialClose(fd)
int serialDataAvail(fd)
void serialFlush(fd)
int serialGetChar(fd)
int serialOpen(device, baud)
void serialPutchar(fd, char)
void serialPuts(fd, char *)
void serialPrintf(fd, char *)
```

## Pi4J Cheat Sheet

### GpioController

```
GpioPinDigitalInput provisionDigitalInputPin(Pin)  
GpioPinDigitalOutput provisionDigitalOutputPin(Pin)  
void setMode(PinMode, GpioPin)  
GpioPinDigitalMultipurpose provisionDigitalMultipurposePin(Pin, PinMode)
```

### PinMode

- DIGITAL\_INPUT
- DIGITAL\_OUTPUT
- PWM\_OUTPUT

### GpioPinDigitalOutput

```
setState(PinState)
```

### GpioPinDigitalInput

```
boolean isHigh()
```

```
boolean isLow()
```

```
PinState getState()
```

### Pin types

```
RaspiPin - GPIO_XX
```

### PinState

- HIGH
- LOW

## Linux cheat commands

### Compressed/archived files

Compressed and archived files come in a variety of formats. Some of the more common file types are tar, gz, bz2

- tar – tar -xvf <filename>
- bz2 – bzip2 -d <filename>

### Reference documents

- [BCM2835 ARM Peripherals](#)

## Credits

An effort such as this book can't be achieved without liberally accumulating knowledge and artifacts from Internet based sources. Here are just a few of the credits that come to mind. If you feel that others should be acknowledged that I may have missed, please drop me a line and I'll most certainly add:

- [www.pighixxx.com](#) – Fantastic pin-out diagrams
- [Wikipedia](#) – Creative commons images

## Research Areas

- When a BCM2835 starts, it seems to boot from start\*.elf. It seems that there are some variants of this available. Can these be used to create custom OS's for the Pis?
- What to setup to get "better" login from an X Server on Windows?
- We need to find out what the I2C protocol looks like from WiringPi. It appears it may be sending stop bits after interactions.
- Provide Node.js sample for thingspeak.
- Provide JavaScript sample for thingspeak.
- Performance measurements and tuning for C, Java and JavaScript
- Test Windows 10 IOT programming – waiting for network device
- Wire up ICStation sound IC
- Write a blinky program using bare metal
- Write a UART output program using bare metal
- Document Tasker and Pi together
- How to make a Pi an WiFi Access Point – waiting for more network devices

- Using the nRF24 as a network device
- Investigate qemu running a Pi Emulator
- Test FSR402 resistive pressure strips
- Test OV7670 CMOS camera
- Test BH1750FVI light intensity sensor
- Test the TTP223B digital touch sensor
- Using a database for storing and retrieving data
- Microphone input
- MSG07 audio analyzer
- Study the use of a Bluetooth attachment
- Use Motion to learn how to stop and start webcams
- Show processing orienting and accelerometer
- MCP23017 expander docs – show how to create 10 keys/buttons
- Investigate running arduino IDE on the PI ... see  
<https://www.raspberrypi.org/forums/viewtopic.php?f=28&t=132747>
- Add LedControl showNumber to Open source after porting to Arduino
- Show LCD1602 with I2C – waiting on parts
- FM radio IC - TEA5767
- Test 2.4GHz keyboard
- Configure a hostapd using DHCP server
- Build the Arduino IDE on PI <https://www.raspberrypi.org/forums/viewtopic.php?f=31&t=133432>
- Examine the PWM outputs on a logic analyzer
- Test the RT3070 as a Hostapd device
- Build a native Arduino from the ATMEGA 328P on a breadboard
- Study Firmata as a control mechanism for Arduino control from the Pi

#### Video ideas

- Using systemd to start programs at boot time.
- Getting Mosquitto operational on the Pi.
- Wiring a mosquitto client in a browser.

