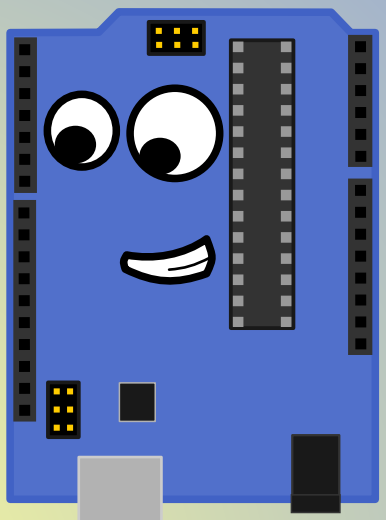
ISSUE 07 - NOV 2012

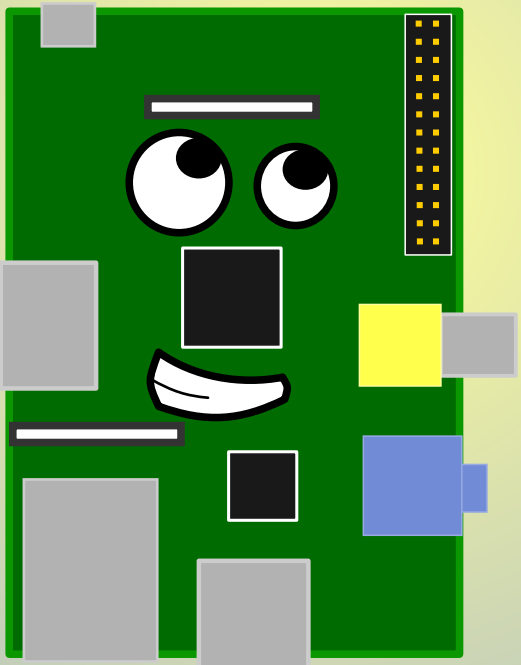
Support us even more by buying the printed copy of this mag!

The MagPi

A Magazine for Raspberry Pi Users

Arduino And RasPi Get Connected!





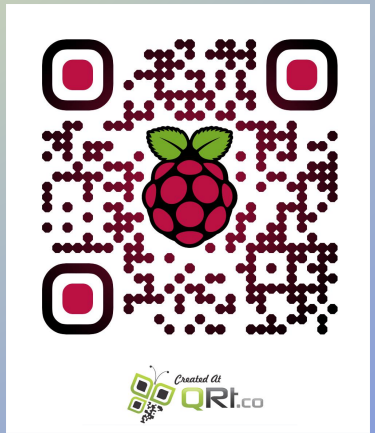
This Issue...

- Interupts
- Solar Pi
- Turbo Mode
- Pi Evolution
- C++

Plus...

- An interview with the Raspbian developers
- Make your own ladder game using PCBs
- The basics of GNU make

A chance to win a PINK RasPi Case!



http://www.themagpi.com

ISSN 2051-9990

The MagPi

The MagPi™

Welcome to Issue 7,

The Raspberry Pi and Arduino are a perfect match for real time applications where a bit more CPU power is needed for control. Coupling the two devices together opens up the possibility to use a wealth of Arduino shields too. We look forward to seeing some really interesting projects in the future.

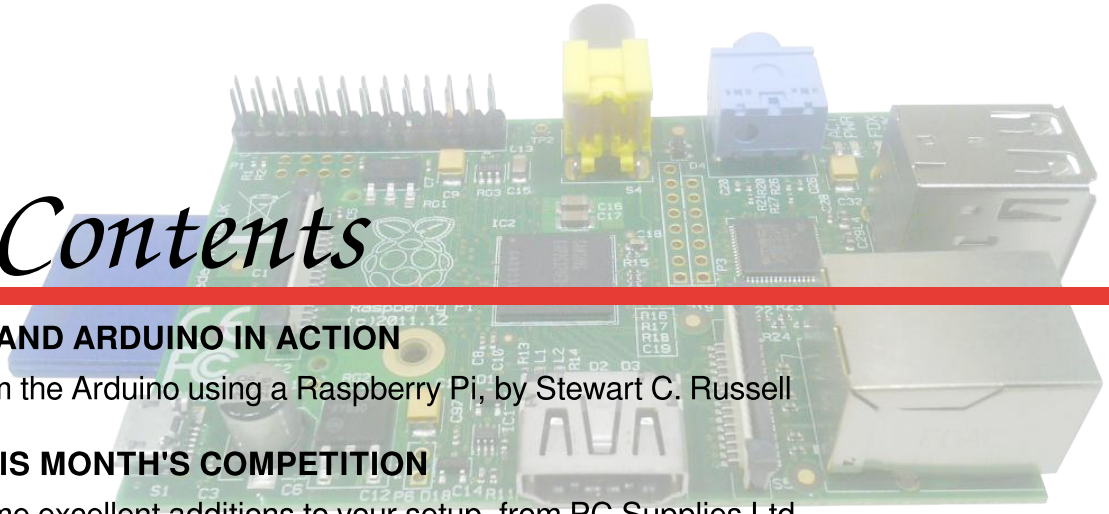
There is an interview from the lead developer of Raspbian (Debian build for the Raspberry Pi), competitions and a selection of programming articles to get your teeth into.

If you would prefer to receive your copy of The MagPi in printed form, visit http://www.modmypi.com and place your order now!

Ash Stone, Chief Editor of The MagPi

Ash Stone
Chief Editor / Administrator
Jason 'Jaseman' Davies
Writer / Website / Page Designs
Tim 'Meltwater' Cox
Writer / Photographer / Page Designs
Chris 'tzj' Stagg
Writer / Photographer / Page Designs
Ian McAlpine
Page Designs / Graphics / Writer
Joshua Marinacci
Page Designs / Graphics
Lix
Page Designs / Graphics
PaisleyBoy
Page Designs / Graphics
Sam Marshall
Page Designs / Graphics
Aaron Shaw
Page Designs / Graphics
Nick
Page Designs / Graphics

Matt '0the0judge0'
Administrator / Website
Matthew Timmons-Brown
Writer
Gordon Henderson
Writer
Colin Deady
Writer / Page Designs
Stewart C. Russell
Writer
W.H.Bell
Writer
Colin Norris
Editor / Graphics
Antiloquax
Writer
Richard Ryniker
Writer
Alex Kerr
Writer



Contents

Raspberry Pi & Arduino

While there are many I/O boards under development for the Raspberry Pi, the Arduino is well established. This article shows you how talk to an Arduino through Python and the Firmata protocol.

DIFFICULTY: INTERMEDIATE

This example combines output (setting the brightness of an LED with a graphical slider) with input (reading the temperature from an LM58).

Required Materials
Raspberry Pi
Arduino
Internet Connection
Small solder-less breadboard
LM35 temperature sensor (http://www.ti.com/product/lm35)
5 mm red LED
120Ω resistor
4× male-male jumper wires (here coloured red, yellow, blue and black)
Short-ish breadboard jumper (17.8 mm or 0.7", here coloured black)

Firmata (http://firmata.org) is a simple serial protocol that allows you to read and write I/O ports on the Arduino from a host computer. It's most often used with the graphical programming language "Processing" (http://processing.org) but there is support for other languages.

Installing the Arduino IDE and Firmata

The Arduino IDE is already included in the Raspbian repositories, so you can install it all with:

```
$ sudo apt-get install arduino
```

If it's the first time you've run it, the IDE may ask you to create a folder for its programs (called “sketches”).

Next, you'll have to choose what Arduino board you're using from the Tools/Board menu (I'm using an Uno, but I have also tested this on an older Duemilanove board).

You'll also need to choose which serial port to use from Tools/Serial Port — for an Uno that would be /dev/ttyACM0, and older boards tend to use /dev/ttyUSB0.

To install the Firmata sketch onto your Arduino, select File / Examples / Firmata / StandardFirmata and click the Upload button.

The IDE goes off and compiles your sketch, and uploads it. If all you get is blinking lights and a ‘Done uploading’ message, success! If you get any kind of red error messages, then there's likely something up with the connection or power to the Arduino.

I'd strongly recommend connecting your Arduino either through a powered hub or applying external power, as the Raspberry Pi is a bit limited in what it can power over USB.

If you get stuck try the website http://www.ladyada.net/learn/arduino/

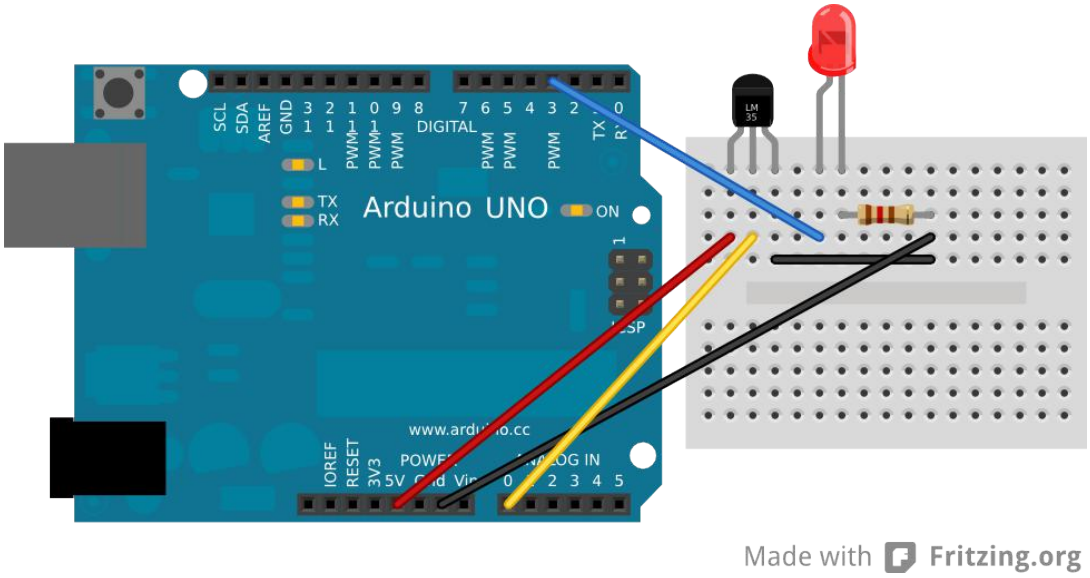
Installing pyFirmata

pyFirmata is the magic that allows your Arduino running Firmata to talk to Python. It takes a few more commands to install it:

```
$ sudo apt-get install python-serial mercurial
$ hg clone https://bitbucket.org/tino/pyfirmata
$ cd pyfirmata
$ sudo python setup.py install
```

If this succeeds, you can remove the pyfirmata folder:

```
$ cd .. ; sudo rm -r pyfirmata
```



Building the circuit

Place the parts as shown in the diagram using the following tables to assist:

ARDUINO	LM35
+5V	+Vs
A0	Vout
GND	GND

NOTE: If you accidentally connect the LM35 the wrong way round, it goes from being a temperature sensor to being a remarkable temperature generator — the D-shaped burn scar on my left index finger reminds me of this every day...

ARDUINO	LED
D3	Anode (Long Wire)
GND	Cathode (Short Wire) (Going Through the 120Ω resistor)

Running the code

Enter the progam on the next page and save it as `arduino_firmata_ty.py`. With your Arduino plugged in, and the circuit wired up, you run the program like this:

```
$ python arduino_firmata_tk.py
```

It will take a few seconds for the window to come up and, once it does, you can control the LED's brightness with the slider on screen.

The LED doesn't seem to get much brighter above 33%, you'll notice; there isn't a straight-line link between brightness and PWM input voltage.

If you touch the LM35, you'll notice the temperature reading changing. LM35 sensors are a bit noisy, so you might notice the numbers jumping about.

Once you're done, hit the window close widget, and everything should exit cleanly.

The Python code does three main tasks:

1. Set up serial communications and configure the Arduino's I/O pins.

Two important details we have to do here is set up an iterator thread to stop the Raspberry Pi's serial buffer overflowing while it receives data from the Arduino, and also to wait for the Arduino to start sending real data before we move on to the main program.

2. Define three functions to be called by the GUI:

get_temp() — this reads the value from Arduino pin A0, converts it to °C, and updates the label at the bottom of the window.

If you're familiar with Arduino, pyFirmata returns analogue values between 0.0 and 1.0 instead of the integer range 0-1023.

This routine ends by asking the main Tk process to reschedule itself after half a second (500 ms), and that way we keep updating the temperature display separate from the main loop.

Continued over page...

set_brightness() — this converts the 0-100 value from the GUI's Scale slider widget to a 0.0-1.0 floating point range, and writes it to pin D3.

cleanup() — all this routine does is turn the LED off, and tries to shut down the program neatly. It doesn't always quite manage to do this, however; sometimes you have to hit Ctrl-C in the terminal window too.

3. Set up the Tkinter GUI. Tk (and its Python version, Tkinter) is quite an old GUI system, but is also quite simple to use.

It relies on each widget (or graphical control) to run a callback routine or set a variable's value as it is clicked or changed.

So here, I'm setting up a 400 pixel wide Scale slider that calls the routine set_brightness() with the current value of the slider as it changes.

Moved fully to the right, the slide would call set_brightness(100), turning the LED fully on.

Since the window is so simple – just one Scale widget and a label — I'm using Tk's crude pack() method to arrange items in the window.

It first draws the items, then packs them together, Tetris-like, into the window frame.

Once it's done that, it schedules the first temperature reading (which schedules the next, and so on), then finally sits in the Tk mainloop() for the rest of the program, responding to your input.

Further directions

This is a very simple example of controlling an Arduino from Python.

While Firmata can control more complex outputs such as servos, it does take over the whole logic processing of the Arduino board.

Any sensor that requires complex setup or real-time control isn't going to work so well.

That aside, you've now got all the power of the Raspberry Pi hooked up to the simple robustness of the Arduino; the sky's not even the limit!

Article by Stewart C. Russell

Stewart C. Russell lives in Toronto, where he engineers wind farms and solar power plants. When he's not lurking in the world's sunny/windy places, he's at home on amateur radio (call sign VA3PID), playing banjo, fiddling with computers, or avoiding gardening. His website is http://scruss.com/blog

```
import pyfirmata
from Tkinter import *

# Create a new board object,
# specifying serial port;
# could be /dev/ttyUSB0 for older
# Arduinos
board = pyfirmata.Arduino('/dev/ttyACM0')

# start an iterator thread so
# serial buffer doesn't overflow
iter8 = pyfirmata.util.Iterator(board)
iter8.start()

# set up pins
# A0 Input      (LM35)
pin0 = board.get_pin('a:0:i')
# D3 PWM Output (LED)
pin3 = board.get_pin('d:3:p')

# IMPORTANT! discard first reads
# until A0 gets something valid
while pin0.read() is None:
    pass

def get_temp():
    # LM35 reading in deg C to label
    label_text = "Temp: %6.1f C" % (
        pin0.read() * 5 * 100)
    label.config(text = label_text)
    # reschedule after half second
    root.after(500, get_temp)
```

```
def set_brightness(x):
    # set LED
    # Scale widget returns 0 .. 100
    # pyfirmata expects 0 .. 1.0
    pin3.write(float(x) / 100.0)

def cleanup():
    # clean up on exit
    # and turn LED back off
    pin3.write(0)
    board.exit()

# set up GUI
root = Tk()
# ensure cleanup() is called on exit
root.wm_protocol("WM_DELETE_WINDOW",cleanup)

# draw a big slider for LED brightness
scale = Scale(root,
              command = set_brightness,
              orient = HORIZONTAL,
              length = 400,
              label = 'Brightness')
scale.pack(anchor = CENTER)

# place label up against scale widget
label = Label(root)
label.pack(anchor = 'nw')

# start temperature read loop
root.after(500, get_temp)
# run Tk event loop
root.mainloop()
```

NOVEMBER COMPETITION

Once again The MagPi and PC Supplies Limited are proud to announce yet another chance to win some fantastic R-Pi goodies**!**

This month there will be FIVE prizes!

Win 1 of 5 Cases

Each winner will receive a Raspberry Colour Case by PCSL. Suitable for both Model A and Model B with GPIO cable access and LED light pipes.

Made by: CYNTECH COMPONENTS from PCSLSHOP.COM

For a chance to take part in this month's competition visit:
http://www.pcslshop.com/info/magpi

Closing date is 20th November 2012. Winners will be notified in next month's magazine and by email. Good luck!

To see the large range of PCSL brand Raspberry Pi accessories visit
http://www.pcslshop.com

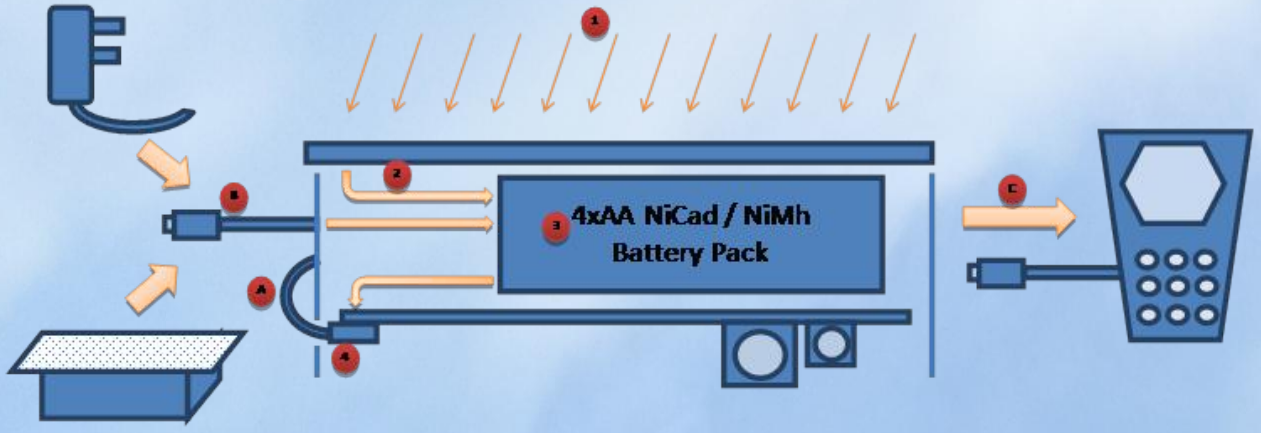
Last Month's Winners!

The 5 winners of the PCSL Limited Edition LCD mount are **Mike Bradbury (Manchester, UK)**, **David Corne (Birmingham, UK)**, **Brian Bowman (Chelmsford, UK)**, **Bart Sekulski (Bristol, UK)** and **William Green (Doncaster, UK).**

Congratulations. We will be emailing you soon with details of how to claim all of those fantastic goodies!

A Little Ray Of Sunshine...

While browsing for interesting Raspberry Pi devices, I came across the following item from CottonPickersPlace.



Unit Operation:

This solar device essentially acts as a solar charger for the 4xAA batteries placed inside, which in turn are able to supply power to the Raspberry Pi itself. The device also serves as a neat holder for the raspberry pi and as a bonus it can also be used to recharge most USB powered devices.

1. The sun shines on the solar panel.
2. Available power is used to charge the battery pack (just insert your own standard 4xAAs NiCad/NiMh of your required capacity).
3. The battery pack stores the energy and seamlessly provides power even when there is little or no sun.
4. The Raspberry Pi is powered!

A. The micro USB power lead plugs directly into the Raspberry Pi (neatly fitted upside-down into the base of the unit).

B. The full size USB plug can be used to directly power the Raspberry Pi, and if the batteries are flat add some extra charge (charging via the sun is required to fully charge the cells). This plug can also be used to daisy chain additional solar units to providing additional solar power and longer battery life (by plugging into the side USB socket (C)).

C. An additional USB socket on the side of the unit provides an auxiliary charging point for mobile phones and other USB devices. Apple iDevices such as iPods / iPhones / iPads are supported with an upgraded version if required.

Specifications	
Solar Panel	12 cell 330mA 2 Watt Polycrystalline 110x135mm
Case	Printed by BRB-3000 3D Printer with PLA (PolyLactic Acid) material – bio-degradeable. Typically takes 2h15min to print. The raspberry pi, sits neatly inside. 60x90x45mm

Typical Outputs	
Directly facing full sun	approx. 300mA+
Laying flat	200-250mA
Cloudy	30mA

Estimated Charge Times			
	Full	Flat	Cloudy
3000mAh Cells	10h	13h20m	100h
2100mAh Cells	7h	9h20m	70h
The unit will not overcharge the cells, and there is no leakage discharge when it is not charging.			

Estimated Raspberry Pi Runtime				
Model B	Full	Flat	Cloudy	Night
3000mAh Cells	23h	14h30m	7h30m	7h
2100mAh Cells	16h	10h15m	5h15m	4h50m
Model A				
3000mAh Cells	Charge	40h	11h	10h
2100mAh Cells	Charge	28h	7h45m	7h

Model B – typical estimated power usage 430mA depending on what is connected (ranges from 322mA idle to 480mA+ peaks).

Model A – Eben stated this is around 200mA, so 300mA should be a good conservative estimate.

Usage Considerations:

My own tests with some old 2500mAh Cells, provided around 4 hours of night-time usage, reasonable considering the batteries are several years old.

By using minimal peripherals and using the more efficient Model A Raspberry Pi (when released) should greatly extend the runtime. The efficiency of the Raspberry Pi can also be improved by replacing the built-in 3.3V linear regulator with a more efficient switched-mode one – an extreme measure, but reported to save about 25% of power (see http://goo.gl/dqUZL for details).

Although the Raspberry Pi can be used in many situations remotely, often you will want to use it with a screen. Currently most screens will require just as much if not more power than the Raspberry Pi itself. This may improve when the foundation can supply LCD screens directly driven from the Raspberry Pi, ultimately requiring less power than your typical LCD monitor or TV. To this end, a super-efficient e-ink based display would be an excellent option, fingers crossed that becomes a reality.

For 24 hour remote operation (something several Raspberry Pi users desire), it is likely that 2 or more solar units and plentiful sun would be required. Additionally, by designing some methods to remotely switch off the Raspberry Pi power supply and switch it back on when required would mean that an unattended remote setup would be viable. Such a system could even monitor the available battery levels and power the Raspberry Pi accordingly or set specific time intervals for operations. We would love to hear of some suitable designs!

Conclusions:

The compact unit offers a number of flexible features, and for running the Raspberry Pi from batteries it presents a good solution.

The addition of the solar panel nicely extends the runtime of the system (particularly if you live in a sunnier climate), or allows charging during the day and use at night for hours at a time. When mains powered, it offers excellent protection against power-cuts, effectively functioning as a UPS (Uninterruptible Power Supply). The unit also provides a means to quickly transfer the unit to various locations without powering off the unit (ideal for quickly moving to the big screen to play some movies, without the normal hassle of finding an available power socket, booting up etc).

CottonPickersPlace is working on a few larger panel models which support larger batteries too, which should be able to manage 24/7 operation and/or power 3rd party screens etc at the same time.

Overall, the unit offers great flexibility at excellent value for money (around £25 delivered). It is very clear that a lot of time and effort is put into each hand crafted unit. CottonPicker has clearly taken care with the design to keep the overall cost as low as possible without compromising on quality or features, in my opinon hitting a great balance.

This may just be the missing piece you've been looking for to complete your project, or just a handy case which allows you to cut the power cord whenever you feel like it!

Article by Meltwater

Available from:
www.cottonpickersplace.com
Direct link: goo.gl/w9Rs3

The Raspberry Ladder Board

The Raspberry Ladder board is a kit of parts intended to be used as an introduction to soldering and GPIO programming on the Raspberry Pi.

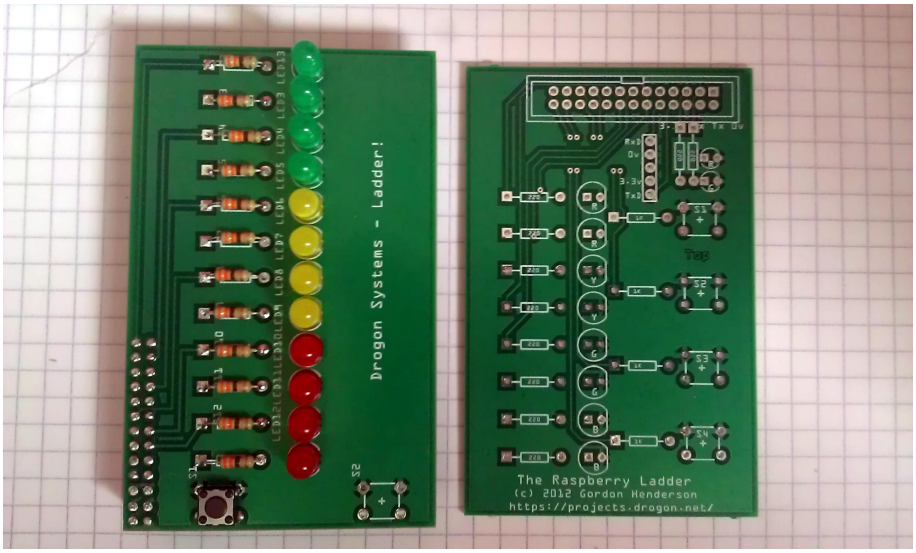
The ladder board is based on my original ladder game which I made earlier this year on a breadboard, details of which can be found here:
https://projects.drogon.net/raspberry-pi/gpio-examples/ladder-game/

This version has been modified to make it more useful for other projects and hopefully will encourage you to write your own little games and simulations. The software that I'll be providing will be a version of my original ladder game, my Tuxx crossing simulator and a new "Simon Says" game.

The kit includes a professionally made PCB, 8 standard LEDs (2 each of blue, green, yellow and red), 2 smaller LEDs (green and red), 4 push button switches, 14 resistors for the LEDs and switches and a short (ready-made) ribbon cable and IDC connectors to connect the ladder board to your Raspberry Pi.

You will need basic soldering equipment, (soldering iron, some solder, a pair of wire cutters) and some time to assemble the board. Additionally some PCB cleaner spray may be useful once it's all finished, but it's not essential. If you are comfortable with soldering then it should not take you more than 10-15 minutes to fully assemble the board, but if not, then it can be done in stages and you can even test each stage as you go.

You can buy the complete kit including PCB from Tandy for £7.00 including postage.



Prototype Raspberry Ladder Board and PCB

http://www.tandyonline.co.uk/electronics/kits/raspberry-ladder-kit.html

Soldering the PCB version:

Soldering is not difficult, but requires practice. Before you start, please read this comic strip:
http://mightyohm.com/soldercomic

Once done reading, have a look at the PCB and the components - don't remove the components from their packs at this stage - the Tandy kit will be packed in the bags in the order that you need them, but see if you can identify everything first. Study the photo of the fully assembled board to see what to expect.

Check the PCB - Look for any obvious signs of damage and identify where the components are fitted. The white printed symbols will help. The resistors have little rectangles, the switches bigger rectangles (almost square), and the LEDs circles with a flat side to them. There is a short row of 5 holes which are not used in this project and 2 longer rows of holes which are used for the ribbon cable connector.

First we need to identify which resistors are which. In the kit there are two types, 220Ω and 1000Ω. The 220Ω ones are identified by their colour banding - Red, Red, Brown and the 1000Ω ones are Brown, Black, Red. However if you are confused, then there are 10 x 220Ω resistors and 4 x 1000Ω resistors - just count them to see which is which.

Start with the 4 x 1000Ω resistors. Bend the legs at the end of the resistor and insert them into the PCB in the four locations. Resistors can go either way, but it looks better if you make them all line up the same way. Push the leads of the resistors through their holes and bend the leads outwards as shown on page 4 of the comic above.

I like to put all four in at once then use something like blu-tak to hold them in-place when I turn the board over to solder, but you may wish to do them one at a time to start with.

Assembly:

You need two hands, so make sure the board is secure. It's also a good idea to be in a well-lit location so you can see what you're doing! See the video for more ideas however, in-general touch the soldering iron to both the

lead of the component and the pad at the same time, wait 1 or 2 seconds, touch the solder to the pad or the very end of the soldering iron - it should flow immediately and fill all the gaps. Remove the solder and then (quite important this bit!) keep the soldering iron there for another second or two.

Most components will be damaged by excess heat, but do not be afraid to keep the heat there for up to 10 seconds if required. With practice you should be able to do a solder joint in about 5 seconds. Iron to pad and component lead, pause, solder until it flows, pause, remove iron. If you feel it's not a good join, then wait a moment for it to cool down and try again.

Make sure your soldering iron is clean each time you pick it up - use a damp sponge or the newer "dry wipe" systems that look like a tub of brassy springs.

Once you have soldered in your first components (or first four!) then it's time to clip the leads short. Again, this is a two-hand job and you must hold the end of the lead when you cut it - if you don't, then it will go flying off and when it hits you it will hurt. (Additionally your partner, mother, etc. will not be happy cleaning up the mess of tiny little spikes of metal!) Hold the end, and cut right next to the solder join and put in the bin.

Once you have the first four resistors soldered in, you can progress to the 220Ω resistors. Start with the two near the top of the board, then the other eight down the side.

Next is the switches. These should hold themselves in the board while soldering, but make sure you put them in the right way round - they are slightly rectangular, so if they don't seem to fit, then rotate them a quarter of a turn and try again.

Now the LEDs. Hopefully by now you should be getting the hang of soldering. I left the LEDs until now for two reasons - firstly it's generally better to solder the low components first, then the taller ones, and also to give you lots of practice soldering resistors and switches which are more resistant to overheating than LEDs are. You should still be OK for up to 10 seconds with the LEDs, but hopefully by now you should be a little quicker and more confident.

The LEDs are sensitive to which way they go in, so do look at them carefully. They have a flat on one side and this corresponds to the flat on the image on the PCB. The flat side always goes to the negative side of the circuit, and the other side (which has a longer leg) always goes to the positive side of the circuit.

Take your time when soldering these in - try to make sure they all sit flat on the PCB and that they line-up in a neat line.

Finally the GPIO connector. Fit it into the board, secure it, solder one pin then check it before soldering the rest. You may wish to go down one long line, then turn the board and go down the other line.

We're done! Hopefully your finished board will look something like the one on the facing page.

Now it's time to connect it up to a Raspberry Pi and run the test software.

Note: when you first turn on your Raspberry Pi, or reboot it with the ladder board connected, the two smaller LEDs may be glowing dimly. This is to be expected as they're being supplied with current from the Pi's on-board I^2C pull-up resistors that form part of the I^2C bus.

Testing:

The test software uses the wiringPi gpio command, so you need wiringPi installed first.

For wiringPi (if you don't already have it):

```
$ cd
$ git clone
git://git.drogon.net/wiringPi
$ cd WiringPi
$ ./bulid
```

For the raspberry ladder software:

```
$ cd
$ git clone
git://git.drogon.net/ladder
$ cd ladder
```

To run the test program:

```
$ ./ladderTest.sh
```

It should take you through a few simple steps to check that your board is working properly.

A slightly modified version of the Tux Crossing program is also there - run it with:

```
$ ./tuxx.sh
```

When it's initialised, push the bottom button to start the sequence. More software and details next month!

Full documentation is supplied in the README file about how the LEDs are connected up, and the ladderTest program is a bash script which you may copy and edit as required. You may also look at some of the GPIO example programs supplied with the wiringPi package, but the real fun starts next month when we write some more programs for it.

Article by Gordon Henderson

Interrupts and Other Activities with GPIO Pins

How to share GPIO resources among multiple applications, and use of interrupts to replace wasteful status check loops.

After some initial experiments where a Raspberry Pi operates LEDs and reads switches, when the "It works!" euphoria fades, astute users may understand there will be problems when they undertake to extend those simple programs to more complex environments.

I discuss two such issues here: how to share GPIO resources among multiple applications, and use of interrupts to replace wasteful status check loops.

There has been a frightful incidence of "run this program as root" instructions published for the Raspberry Pi user. This sounds to an experienced user rather like "Here, children; these are razor blades. Take them outside, and see what you can cut with them."

Root privilege should be viewed as a last resort. Its proper use is system creation and configuration - the establishment of a protected environment where faults in one program will not affect other applications, and cannot cause failure of the operating system. At worst, a user program that errs should compromise only the resources allocated to that program.

Linux has a large number of device drivers, programs typically part of the kernel that interface between hardware resources and application programs. Examples are file systems, which expose user-friendly functions like open, read, and write, while they manage hardware access and maintain the necessary data structures to allocate and free disk space, share access in appropriate ways between multiple programs, and handle recovery after events such as power failures.

Root privilege makes it easy to interfere with system activities. If one is lucky, the result is immediate panic and the system crashes. In less fortunate circumstances, malicious software could be installed in a system: this software can then communicate over an Internet connection with criminals who seek personal information or might exploit your Raspberry Pi for nefarious activities.

Linux has a general facility to manage GPIO resources. It creates a convenient interface for user programs, protects GPIO resources used by device drivers such as I2C and SPI, and delivers pin-specific access so one application does not need to worry about what other programs do with other GPIO pins. This individual pin interface is important, because without it every GPIO application would have to worry about race conditions with other applications that share a bank of GPIO pins (locks, interrupt management, or other complexities would be needed).

The Linux GPIO facility uses files in the /sys/class/gpio/ directory. Yes, like many system configuration or control files, these files are owned by root. I shall ignore this for now, to make description of the interface easier, but promise to return later and present a tool to encapsulate the privileged operation in a responsible way.

Setting Up The Pins

The echo command is commonly used in shell procedures to display messages to standard output, or with output redirection to write to a file. A simple example:

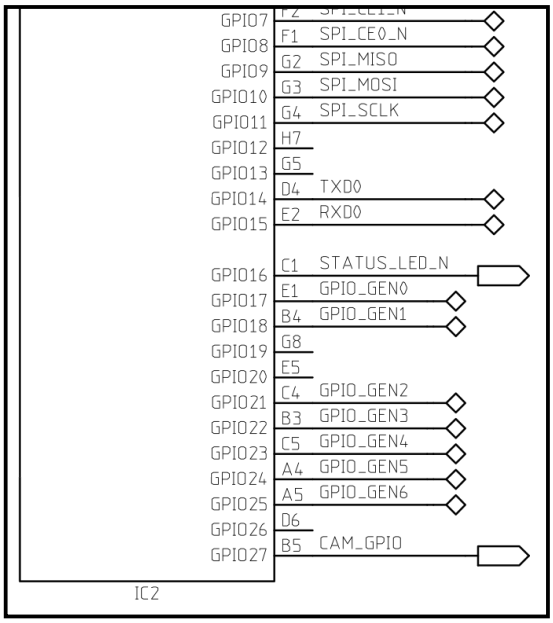
```
echo Hello there.
```

writes the output "Hello there." With output re-direction:

```
echo Hello there. >file_01
```

creates the file "file_01" that contains the same message.

The echo command will be used for some examples of GPIO use. Pin 23 is used because it is convenient and easily accessible at terminal 16 of the 26-terminal Raspberry Pi header. It is labeled GPIO_GEN4 on the Raspberry Pi schematic (http://www.raspberrypi.org/wp-content/uploads/2012/04/Raspberry-Pi-Schematics-R1.0.pdf).



To create a user interface for pin 23, use sudo or, as root, execute:

```
echo 23 >/sys/class/gpio/export
```

This causes the kernel to create a /sys/class/gpio/gpio23 directory which contains four files relevant to this discussion: active_low, direction, edge, and value. The initial values in these files (if there is no external connection to this pin) are:

```
active_low    0
direction     in
edge          none
value         0
```

To make this an output pin:

```
echo out
>/sys/class/gpio/gpio23/direction
```

If the output value should be initialized first, before the output driver is enabled, one of the following may be used to set pin direction with an initial value:

```
echo low >/sys/class/gpio/gpio23/
direction
echo high
>/sys/class/gpio/gpio23/direction
```

Please note GPIO reference changes for pins 3,5 & 13 on Revision 2.0

To set this pin output on or off:

```
echo 1 >/sys/class/gpio/gpio23/value
echo 0 >/sys/class/gpio/gpio23/value
```

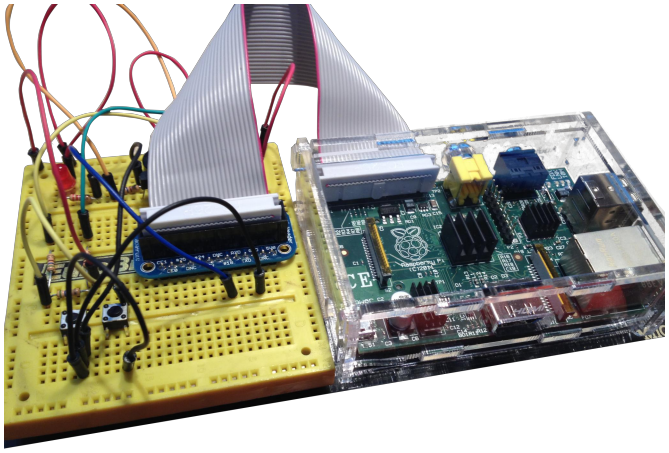
To invert the pin logic:

```
echo 1 >/sys/class/gpio/gpio23/
active_low
```

Do this before reading an input or setting an output value. When active_low is 1 (or anything other than 0) and value is set to 1, the pin is driven low, etc.

Continued over page...

How fast can this mechanism change GPIO pin values? A simple python program http://ryniker.ods.org/raspberrypi/MagPi/gpio23-max.py will generate pulses at 19 kHz. If this is written in C (see http://ryniker.ods.org/raspberrypi/MagPi/23-maxa.c) the frequency increases to roughly 120 kHz. The actual frequency varies because the Raspberry Pi does other things that temporarily suspend the loop - clock maintenance, network activity and other user and system processes.



The Program

As promised earlier, here is the program http://ryniker.ods.org/raspberrypi/MagPi/gpio_control.c that executes operations which require root privilege in order to export a GPIO pin for use by ordinary users. Comments at the beginning of the program describe how to compile and install it. Once it is installed (by root), because of its "setuid" characteristic, the program runs with an effective userid of root. Therefore, it has the privilege needed to export or unexport a GPIO pin and set appropriate permissions for the files used to control that pin.

Programs that execute with root privilege should be written by really paranoid programmers. Most of the code in gpio_control.c simply checks that the arguments are reasonable, and tries to be informative if anything unusual happens.

To use gpio_control to export pin 23 so all of the pin manipulations discussed earlier do not require root privilege, simply execute:

```
gpio_control 23 export
```

gpio_control.c may be easily configured, before it is compiled, to allow GPIO access to all users or only users in the caller's group.
Each of the 26 GPIO pins may be individually configured to permit or forbid export.

The Raspberry Pi uses GPIO pin 16 to control the "Status OK" green LED. If one tries to export GPIO pin 16, the operation fails because the kernel is using this resource:

```
ryniker@raspberrypi:~$ gpio_control
16 export
export failed: Device or resource busy
```

Other kernel programs may acquire GPIO pins, which can make them unavailable to users. This is good. Little harm could come from a user turning the status LED on and off, but what about the kernel I2C driver? It could easily suffer erratic failures if the pins it uses are changed in ways it cannot understand.

The kernel remembers the state of GPIO pins. For example, suppose a pin is exported, set by the user as an output pin, then unexported. The userspace files disappear, but the pin remains an output pin with the last value set. If this pin is again exported, the userspace files are recreated to manifest the saved state.

The echo command is convenient to use in shell scripts, occasionally on the command line, but Python is much easier for real programs. The dozen lines in gpio23-max.py should provide a simple example.

Now that the basic elements of GPIO control have been exhibited, this facility can be used to replace the "infinite loop" operation, where a program repeatedly reads the value of an input signal and performs some operation when it changes; with a vastly more efficient program that only runs when the input signal changes. With only one input, and absolutely nothing else to do until it changes, a loop may not be a problem. However, such a loop wants to consume 100 percent of the CPU resource, and therefore competes aggressively with everything else that might want some piece of the Raspberry Pi.

One can introduce a delay in the poll loop, say a "sleep 0.5" command to delay one-half second before starting the next loop iteration.

This allows other activities to run during the sleep period, but means there is an average delay of one-quarter second before any change in the input is observed. Shorter delay, faster response, more wasted CPU... ugly choice.

As the number of inputs grows, and the number of responses to those inputs becomes larger and more varied, it often is necessary to manage tasks with different priorities. Interrupts are the means to quickly connect an input such as "There is a chasm directly in front of the vehicle" to the response "Stop!".

Another Python Program

http://ryniker.ods.org/raspberrypi/MagPi/interrupt_test23.py will illustrate GPIO interrupt handling. This program configures pin 23 as an input, sets the pin's edge file to "both" so interrupts will occur for "falling" and "rising" transitions, then opens the pin's value file. Invocation of select.poll() creates a polling object "po", then po.register() adds the GPIO pin's value file as one of the sources which can satisfy a subsequent po.poll() request. This program uses only the one interrupt source, but other GPIO pins, and many other sources, can be registered with the poll object. For instance, a pipe that connects to another process could be an interrupt source, or a socket that receives data over a network from a remote system.

The second operand to po.register specifies which of three conditions will be recognized as interrupts. The select.POLLPRI value specifies only "priority data to read" will cause an interrupt. The other possible conditions - "data available" and "ready for output" - are always true for a GPIO pin, therefore a poll operation when either of these is enabled would always complete immediately. If other interrupt sources are registered with po, they might use these conditions.

Sometimes, the absence of an expected signal may be important. The po.poll(60000) call will wait for an interrupt, but only for 60 seconds (60,000 milliseconds), before it returns an empty list of interrupt signals to indicate it timed out.

The kernel maintains the value file for a GPIO pin with two bytes of content: a 0 or 1 character to represent the pin's current value, and a newline character. f.seek(0) resets the current location in the file to the beginning, so the value of the first character may be read again.

Expanding The GPIO

Only a few GPIO pins are accessible on the Raspberry Pi, but several people have shown how inexpensive ICs such as MCP23017 can use the I2C interface to expand this number. A design such as http://shop.ciseco.co.uk/k002-slice-of-pi-o/ can be used up to 8 times to add 128 digital I/O pins to a Raspberry Pi. Use the MCP23017 open-drain interrupt configuration to connect interrupt signals from multiple devices to a single GPIO pin. A pull-up resistor to 3V3 keeps the input high, until a connected device drives it low. When an interrupt occurs, the interrupt handler has to read values from all the interrupt-generating devices to learn which have active interrupt signals (there may be several), launch the appropriate response programs, then clear all the interrupt requests (so the GPIO input returns to the high state) to allow the next interrupt to occur.

A Summary Of The URLs

Raspberry Pi schematic:
http://www.raspberrypi.org/wp-content/uploads/2012/04/Raspberry-Pi-Schematics-R1.0.pdf

The programs:
http://ryniker.ods.org/raspberrypi/MagPi/gpio23-max.py
http://ryniker.ods.org/raspberrypi/MagPi/23-maxa.c

IO Expander:
http://shop.ciseco.co.uk/k002-slice-of-pi-o/

Article by Richard Ryniker

The Raspbian distribution of Debian provides the operating system behind most Raspberry Pi installations. In this month's The MagPi we interview Mike Thompson the founder of Raspbian.

Q: Why did you choose to start Raspbian?

I read in January that the Foundation was planning to use Fedora and I wondered if I could do the same thing but with Debian as that is my preferred Linux distribution on ARM processors. For a long time I have been interested in learning how to build an operating system and also to contribute something back to the Debian community. Raspbian gave me this opportunity.

I realised it was going to take a certain amount of resources, time and effort to create Raspbian. I started asking questions on the forums to understand what someone would have to consider if they undertook this. I poked around to see how to make it happen and it unfolded from there.

Raspbian is a joint effort between myself and Peter Green (Plugwash) who is a Debian developer.

Q: Why the Raspberry Pi, as there are other relatively low cost boards available?

I have a personal interest in inexpensive Linux systems and am very encouraged by the Raspberry Pi Foundation demonstrating the desire and need in the market for a system such as the Raspberry Pi. Ultimately my interest is in seeing these systems at a $5-$10 price point. It may take a few years to get there but things could get a lot more interesting in this world when there is large scale access to very cheap but relatively sophisticated computers such as the Raspberry Pi.

Q: How did you go from having a working build of Raspbian to being the official OS for the Raspberry Pi Foundation?

Peter Green and I were well under way with the project when in the middle of June we got hints that the Foundation was interested in Raspbian. When I started my expectation was to create an alternative to Fedora used by maybe 10-20% of Raspberry Pi users and I did not expect Raspbian to become the "official" Linux distribution on Raspberry Pi. After releasing the first few test images of Raspbian and a significant part of the repository was built people started getting enthusiastic saying they were hoping the Foundation was going to choose a Debian based distribution. I knew the Foundation was aware of Raspbian via the forums and that if they thought it was worthwhile they would make a choice to use it.

Q: How do Raspbian and the Foundation's releases differ?

Raspbian is a recompilation of the packages that comprise Debian ARM Wheezy hardfloat with the compilation settings tuned for the ARMv6 processor in the Raspberry Pi. We are using the kernel work coming out of the Foundation unchanged because the binary interfaces into the kernel do not have any floating point components. That saved a lot of effort enabling us to concentrate on the recompilation of the packages.

Alex Bradbury, the Foundation's lead developer, worked on the Foundation's Raspbian image. As Raspbian is essentially a clone of Debian he took the same scripts he had used for the Debian based image, made minor changes and used them to build their own Raspbian based image. I think he was pleased to see that we were closely following Debian, hence it was a fairly trivial process to create a Raspberry Pi optimised version of Debian based on our Raspbian packages.

Q: How suitable did the Freescale iMX53 Quick Start Board you bought in March prove to be as a build platform?

We are still using 8 Freescale iMX53 boards to build Raspbian packages. These are fairly fast systems with a 1GHz ARMv7 processor and 1GB of RAM. When building, some of the

packages require lots of RAM to build enormous link structures in memory and we run into 1.5-2GB of swap as we are exceeding the available RAM. A typical modern PC with 4GB of memory may take an hour or so to build a large package, but on these ARM systems it can take upwards of 20-30 hours. Having 8 systems available for parallel builds was needed in May and June when we were building the bulk of the 18000 source packages which translates into just under 38000 binary packages for Raspbian. If we only had one system we would still be building packages today. We are using modified versions of Debian's own build tools to distribute the building of packages across the 8 systems.

I came into this project with very limited experience of building operating systems and had to learn everything needed. Fortunately for me, Peter Green joined and his experience with Debian and Debian build tools was essential to making Raspbian possible. I had been a software developer all my career but never attempted any build on this scale. I now fully understand why companies have build engineers that just focus on building large software projects!

Q: How dependent were you on the upstream work undertaken by the Linux community?

Extremely dependent. Raspbian would not be possible if the group in Debian who created armhf had not done their upstream work 18 months prior, albeit they went for ARMv7 and not ARMv6. Peter Green is adamant, and I think correctly so, that Raspbian be as close to an official Debian release as possible without actually being an official release. As long as we maintain that commitment with Raspbian, it will remain a firm base for the Raspberry Pi Foundation and the community.

Downstream, keeping Raspbian so close to Debian reduces the risk of just two guys working on it. Peter Green ensured everything we have done is completely open. If we were to close up shop tomorrow our work is out there mirrored in 30-40 places around the world. Anyone with knowledge of building Debian could easily pick it up and keep maintaining it. Therefore Raspbian is a low risk to the Foundation.

Q: Are there any other performance gains that can be had in Raspbian?

I think we are maxed out on the software side of things. Replacing the CPU with an ARMv7 or adding more memory [Editor's note: the Pi has just started shipping with 512MB RAM!] would be great as some people are hitting the ceiling, for example with web browsing on a GUI desktop.

I think in general software efficiency has gone by the wayside, especially with GUI applications. I always value lean and efficient use of memory for computation. Unfortunately the reality is that lots of RAM and a powerful CPU is now needed for most GUI apps. We should still encourage people to learn to program efficiently with limited resources. If systems like the Raspberry Pi had been available eight years ago we may have seen a lean branch of productivity software requiring less resources in general on all computer platforms.

Compared to Turbo Pascal that came out on CP/M in the early 1980s, and later Turbo C, both of which featured small, fast and fully integrated development environments, modern GUI based development environments take up enormous resources and do not run well, if at all, on the Raspberry Pi. It is sad that today there is no real equivalent of Turbo Pascal or Turbo C on the Raspberry Pi as these systems went away when GUIs came in. I believe there is a huge opportunity to bring these types of tools back for the comparatively low resource environment of the Raspberry Pi.

Q: What work is left to do on Raspbian?

We are now largely in maintenance mode. As Debian releases updates to packages we pull them down, build and push out to the repositories. Personally, I have achieved my goal with Raspbian of creating an ARMv6 hardfloat version of Debian.

I am happy that Raspbian has enabled so many things in the Raspberry Pi community. It is also great that I have been able to give back to the wider Linux community and I hope this will lead to thousands more Debian users in the future.

Mike Thompson is a Computer Engineer living in the San Francisco Bay Area. He has a diverse background in embedded systems design, handheld/mobile application development, PC application development and large scale Internet systems design. He is a serial entrepreneur who co-founded two previous companies and is the founder and a lead developer for Raspbian, the leading operating system for the Raspberry Pi.

TURBO PI

Fancy getting some more oomph from your Pi? Go Turbo!

The Raspberry Pi's processor has a clock speed of 700MHz. This means it performs 700,000,000 cycles every second. The clock speed of a processor is an indication of how fast it can perform operations. It is measured either in megahertz (MHz) or gigahertz (GHz) with 1000MHz equal to 1GHz. So, the higher the MHz the faster the processor will operate.

What are Overclock and Overvolt?

While 700MHz is the design speed of the Raspberry Pi processor, there is a way of getting faster performance... Overclocking... and thanks to the latest Raspbian image it is easier than ever!

Overclocking is the process of making a component or computer run faster than its designed speed, though it can involve a trade-off with increased instability and decreased processor life. For the Raspberry Pi these side effects are so minimal you would be foolish NOT to perform some overclocking!

Overclocking requires additional power. If you want to overclock your Raspberry Pi to 900MHz and higher you will need to provide extra power by 'overvolting'. How far you can overclock depends on several factors; your Raspberry Pi, the quality of your power supply and possibly also your SD card. Because of manufacturing tolerances, 700MHz is the manufacturer's guaranteed performance. But every Raspberry Pi is different and so each one has different limits.

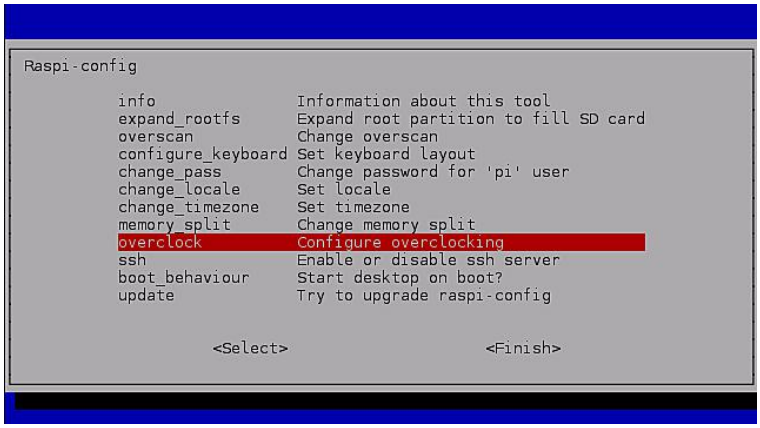
There are several overclock and overvolt settings. These are detailed at `http://elinux.org/RPi_config.txt#Overclocking_options` but the latest Raspbian image contains an easy configuration option. Importantly this allows you to overclock and overvolt your Raspberry Pi while still keeping the warranty intact.

Power supply considerations

When overclocking it is important that you use a good power supply. I use a Kindle charger that is high quality and provides a current of 850mA. Original Apple iPhone chargers are also good choices and provide 1A of current. Beware of third party iPhone chargers. The Raspberry Pi Foundation has identified that some of these do not perform as specified.

CAUTION: Make a backup

Before we begin, it is worth noting that there is a possibility that overclocking may corrupt the data on your SD card, especially if the Turbo option is chosen. Make sure you have a backup of any important documents and also of your `/boot/config.txt` file. The easiest way to do this is to copy the files to a USB thumb drive or upload them to an online storage service such as Dropbox.



Alternatively you can make a complete backup image of your SD card using the same Win32DiskImager program that you used to create your Raspbian SD card. This time, instead of writing an image to the SD card you

are going to create a backup image by reading from the SD card. Insert your SD card into an SD card reader in your Windows machine and start Win32DiskImager. Enter a name for your backup image and click on `Read`. When the backup is complete, the file size of the backup image file should be very similar to the size of your SD card.

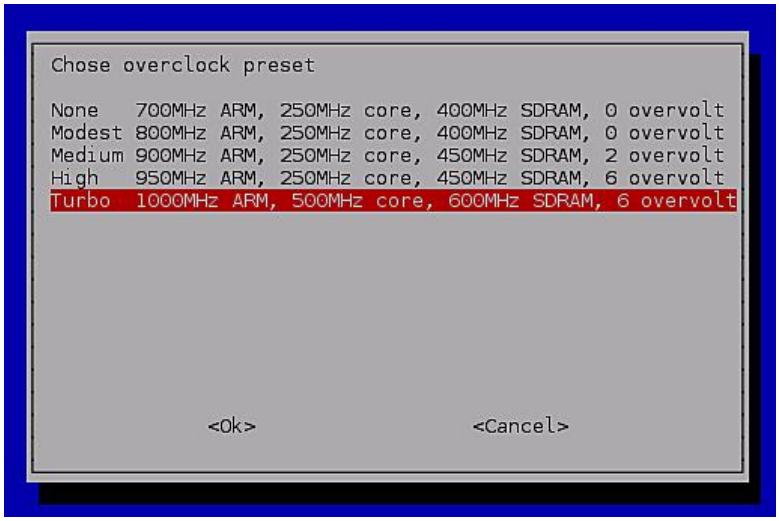
Overclocking

To change the overclock settings start your Raspberry Pi. From the terminal enter the command:

```
$ sudo raspi-config
```

A blue box should appear, as shown on the opposite page. This is the same configuration menu that appears when you start the Raspbian image for the first time. The first thing to do is to update the raspi-config tool. Use the arrow keys to scroll down and choose the “update” option. Wait for raspi-config to look for its latest version. Once that has finished we can get on with overclocking!

Now scroll down and choose the “overclock” option. You can select how far you would like to overclock. There are five preset options for you to pick from; None, Modest, Medium, High or Turbo.



Use the arrow keys and choose your preferred overclock preset. It is best to start with Turbo to see if that is stable. If not then try High, followed by Medium then Modest. After making your choice, exit from the raspi-config tool and reboot. You must reboot before the changes will be made. The changes are written to the file `/boot/config.txt`.

Did I just break my Pi?

But what if your Raspberry Pi does not boot up any more? This means that the overclock settings have exceeded the operating limits of your Raspberry Pi, but don't worry as it is easy to fix. First unplug the power to your Raspberry Pi, wait a few seconds then power up your Raspberry Pi again. Immediately press and hold down the `<SHIFT>` key on your keyboard. Watch the text on your screen. You will see the following text:

```
[ ok ] Checking if shift key is held down: Yes.
Not switching scaling governor.
```

This means that the overclock settings are ignored and will allow you to boot up as normal. If you were unlucky, it is possible your SD card file system got corrupted and you will have to restore from your backup image. If you have powered up successfully, you can open the terminal and enter the command:

```
$ sudo raspi-config
```

This time, when you choose “overclock” try a slower option.

If you operate your Raspberry Pi headless via SSH or another remote access application and cannot hold down the `<SHIFT>` key during start-up then you need to change the overclock settings manually using another computer with an SD card slot. You want to edit the file `config.txt`. For Linux and Apple Mac you can use their default text editor programs. For Windows you will find it awkward to edit the config.txt file using Notepad. For a better alternative I suggest you download the free TextPad program from `http://www.textpad.com`. The entries that you need to edit are `arm_freq`, `core_freq`, `sdram_freq` and `over_voltage`. Use the image to the left as a reference for the values you should use.

You have chosen an overclock preset and your Raspberry Pi appears to have started again without any problems. How do you know it will be reliable?

Continued over page...

Reliability tests

There are a couple of tests you can perform to determine if your Raspberry Pi will be reliable. It is no fun having a fast but unreliable system. If you have Quake 3 installed this is a great test of both the CPU and GPU overclock settings. Another very easy test that I use is to simply perform an upgrade of the Raspbian image. To do this start your Raspberry Pi, open the terminal and enter the following commands:

```
$ sudo apt-get update
$ sudo apt-get upgrade -y
```

This is a good test of the CPU and SDRAM overclock settings. Depending on the number of updates this could take 30 minutes but you will now have the latest system.

Once complete, reboot your Raspberry Pi. This time you want to pay close attention to the messages that appear during start-up. Look out for “mmc” messages or any messages related to file system errors. Additionally look out for [warn] and [fail] messages. If you see these messages this suggests a potential weakness and you should try the next lowest overclock preset.

If you have different SD cards then it is worth testing each of these. I tested three Raspberry Pis with nine different SD cards ranging in speed from class 2 to class 10. Each Raspberry Pi was a Revision 1 device with the USB poly fuses replaced with wire links. They were powered from the same USB hub which had a 2A power supply. One of the Raspberry Pis was successful up to the Medium overclock preset; the other two were successful up to the High overclock preset. None of the Raspberry Pis worked reliably with the Turbo preset.

Interestingly, the two Raspberry Pis that worked with the High overclock preset only did so with seven of the nine SD cards. They failed with the other two cards; a Transcend 4 GB class 6 card and a Lexar 16 GB class 6 card. However, your results may be different.

Monitoring

When overclocking it is very useful to know the current CPU frequency and CPU temperature. You can do this very easily in LXDE. Right-click on the Task Bar along the bottom of the screen and choose `Add / Remove Panel Items`. The Panel Preferences dialog will appear and the Panel Applets tab should be selected. Click on the `Add` button. Choose `CPUFreq frontend` and click the `Add` button. Repeat this process but this time choose `Temperature Monitor`. You may find it useful to add other applets such as `Volume Control` and `Network Status Monitor`.

Another simple test is to start Midori and visit `http://www.raspberrypi.org`. While this is loading, keep your mouse hovered over the `CPUFreq frontend` applet. You will see this change between 700MHz and the CPU frequency defined in your current overclock preset.

To watch a video about overclocking plus other Raspberry Pi topics, please visit my YouTube channel: `http://www.youtube.com/user/TheRaspberryPiGuy`.

Article by Matthew Timmons-Brown & Ian McAlpine

DID YOU KNOW?

While ordering a copy of The MagPi from `http://www.modmypi.com`, I noticed they sell a cooling kit. "The Raspberry Pi Heat Sink Kit" comprises of three small heat sinks plus thermal tape and could help improve device reliability. There is a heat sink for the SoC, GPU and power regulator.

In my own tests the CPU temperature dropped from a maximum of 61°C without the heat sink to a maximum of only 52°C with the heat sink.

The MagPi What's On Guide

Want to keep up to date with all things Raspberry Pi in your area?

Then this new section of The MagPi is for you! We aim to list Raspberry Jam events in your area, providing you with a Raspberry Pi calendar for the month ahead.

Are you in charge of running a Raspberry Pi event? Want to publicise it?
Email us at: editor@themagpi.com

Bloominglabs Raspberry Pi Meetup

When: First Tuesday of every month @ 7:00pm
Where: Bloomington, Indiana, USA

Meetings are the first Tuesday of every month starting at 7:00pm until 9:00pm, everyone is welcome. Further information is available at http://bloominglabs.org

Durham Raspberry Jam

When: Wednesday 14th November 2012 @ 5:30pm
Where: Durham Johnston School, Durham, UK

The meeting will run from 5:30pm until 7:00pm and there are a limited number of places. Tickets and further information are available at http://durhamjam-eorg.eventbrite.com

Sheffield Raspberry Jam

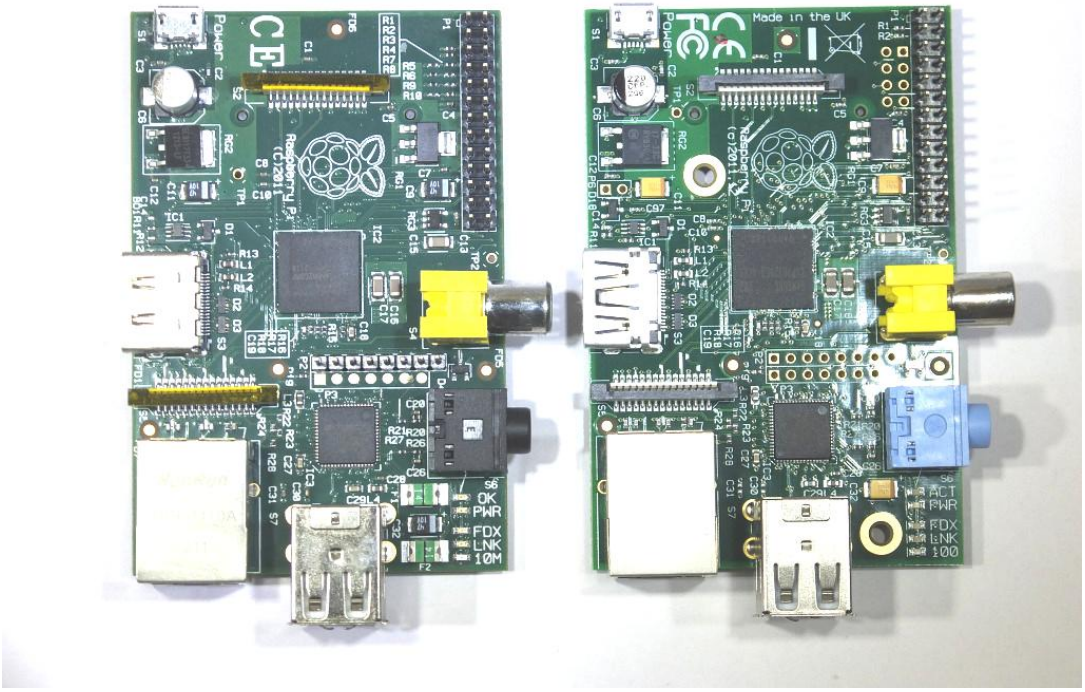
When: Last Wednesday of the month @ 6:30pm
Where: 1st Floor, The Workstation, Grinders Hill / Brown St., Sheffield, S1 2BX, UK

The meetings are hosted by GISThub. Doors open at 6:20pm and the meeting runs from 6:30pm until 8:30pm. Further information is available at http://sheffieldraspi1210.eventbrite.com

Publicise your Raspberry Jam here

PI-EVOLUTION

The pace of development surrounding the Raspberry Pi is hard to keep up with...



If you were an early adopter of the Raspberry Pi, you most likely started off running 'Debian Squeeze' - a Linux Operating System. You spent time learning how to download the image, and write it to an SD Card. Then you were probably introduced to LXDE - the lightweight x-window graphical user interface. It looked similar to Windows but things were just slightly different. You then had to work out how to use a package manager - the 'apt-get' command, 'aptitude' or perhaps 'synaptic', so that you could download and install new application packages.

You just got a decent collection of useful applications installed, when you discover that a newer version of Squeeze is available, which is better than the previous one, so you start over again. Then not much later there is yet another updated operating system - Debian Wheezy 'Beta'. But you heard that there were some problems with that version, and didn't know whether to stick with Squeeze or move to Wheezy. It was all so confusing.

Meanwhile, many other people were still waiting patiently for their Raspberry Pi order to be processed. Finally the little bundle arrived and you were so happy. Then you learned that a new and improved version of the Pi was now available, which had mount holes and various other tweaks, and you wondered if yours would now be considered 'old-hat'. Then to make things even worse, it is announced that the Raspberry Pi will now come with double the amount of memory, and for the same price!

Finally a better version of Wheezy comes along, but now it's become something called 'Raspbian Wheezy', which is much faster, and many of the bugs have been fixed. Oh, wait a second, here comes another release of it, which is even better, and I hear there is another version already planned to be released in the next couple of weeks.

Now, if you are someone that hates change, you are probably banging your head against the wall by this point, and wondering if this computer stuff is really for you after all.

In computing, things often happen very quickly, and dealing with a steady of flow of updates is something that you have to get used to. If you can't adapt to change, then you will get left behind.

Most likely the pace of development will start to slow down to a degree once the foundations are put in place - but we are not quite there yet. You can expect with the educational release of the Pi that the boards will come with a case included and probably some form of instruction manual. Also likely is a much improved image, bundled with many goodies. Despite this, we

must not expect things to stagnate at that stage. A number of hardware add-ons and accessories are due to follow; cameras, screens, and other electronic devices that can be hooked up to your Pi. Most likely at that stage, your current non-cased, 256Mb Pi which lacks mount holes is going to be practically worthless. *[Ed: Not true! See "Peter Lomas Interview" in Issue 12].*

What we have to remember is that the purpose of the Pi is to learn the principles of programming, interfacing, or just general computing. Although many things change, the fundamental principles do not. The Raspberry Pi is a device for learning, and for experimenting with. It is not a fashion accessory or a luxury gadget. The Pi is more of a cheap disposable item. There are people that sell their iPhone 4 just because there is now an iPhone 5 available. There will be Pi owners that fall into the same trap, but the people who will get the most from the Pi are those that stop worrying about these things and spend more time tinkering with whichever version they happen to have, and learning from the experience.

With this in mind, I realised that my own SD card was out-of-date and thought I had better take a peek at 'Raspbian Wheezy 18/09/2012 Edition'.

Back in issue 5, I wrote an article comparing Debian Squeeze and Wheezy, and one of the major downfalls I discovered was that although Wheezy was quite a lot faster than Squeeze, it scored very poorly on multimedia playback. So I decided I would first see if anything had improved with the updated image.

Firstly I downloaded the 439MB zipped image file from http://raspberrypi.org/downloads. This took about 45 minutes using the direct download link from my Windows computer. Then another 45 mins to write the image to the SD card using Win32 Disk Imager. I have a class 4 MicroSD card which is rather slow, but it does the job.

Next I plugged the MicroSD card (inside a normal SD card adapter) into my Pi and booted up to the Raspi Config screen.

Interestingly, I didn't have to change the overscan values for my monitor this time (as I have always had to do with previous versions of Debian).

I chose to expand the rootfs so that I could make use of the 16Gb card. I also set the timezone to London and configured the overclock to 1000MHz Turbo Mode, then exited the menu and rebooted.

After logging in I started LXDE by typing 'startx'. The desktop looked almost identical to the previous version, although the picture looked sharper. I opened an LXTerminal window and typed 'sudo amixer cset numid=3 1' to direct audio to the analogue output.

Then I used 'sudo apt-get update' and 'sudo apt-get install xine-ui' to get the xine media player. Xine is a media player I had looked at some time ago (during the making of The MagPi issue 3). It looked promising, but was really too sluggish to be useful. I thought perhaps with the newer OS and the overclock that things might be better this time around.

Fortunately my new Pi had finally arrived from RS Components (after a six month wait), and it was able to handle the full 'Turbo mode'. Unfortunately it arrived a week before the 512MB version came out and it also lacked the mount holes.

I did have an older revision of the Pi board that Antiloquax had kindly sent me, but this one couldn't handle the overclock - it failed to boot in Turbo mode and was unstable with any of the lower overclock levels.

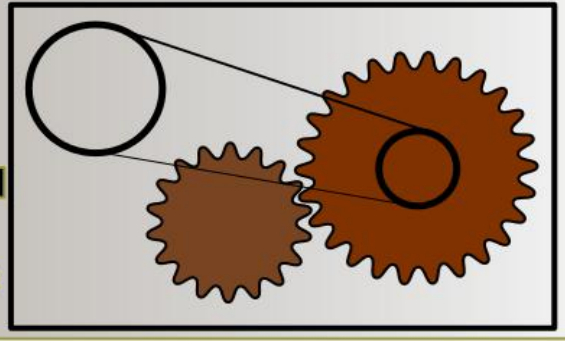
Xine was able to open and play many formats that I tested, although there was something odd about getting the files to open. Only after clicking the 'play next chapter >|' button would it select the correct media to play, otherwise a message stating 'There is no MRL' kept appearing. Xine managed to play avi, mp3, mp4, mov as well as wma, wmv and... mpeg - YES MPEG. However, it started dropping frames if you attempted to play a video at anything larger than 100% zoom. I haven't paid for the codec for MPEG. Xine was rather buggy and unstable, however the command line player 'omxplayer' worked better, but supported far fewer formats. Still a massive improvement over the previous version of Raspbian Wheezy, and I think this one is worthy of putting Squeeze finally to rest once and for all.

I took a break from media players and installed Scribus - the desktop publishing program we use to make The MagPi. I loaded up one of the Python Pit documents from issue 6, and noticed that it was significantly quicker thanks to the overclocking. It was particularly better when switching between layers, zooming and right-clicking to bring up the properties dialogue box.

Finally I installed a few arcade games: 'beneath-a-steel-sky', 'geki2', 'geki3', 'pangzero' and 'xsoldier' all worked nicely.

Updated distributions are not necessarily better overall - sometimes it's worth holding back from upgrading until the bugs have been ironed out, but burying your head in the sand and ignoring the march of progress is also a bad idea. Always keep an eye on what's going on in the community to avoid missing out on the cool updates and performance tweaks.

Article by Jaseman



The basics of GNU Make

make is a central part of building packages of compiled code or documentation on a Linux system.

Open a terminal window. Then use the nano editor, described in the Issue 3 C Cave article, to create a file called Makefile containing:

```
# Create newFile.txt if it is not present
newFile.txt:
    touch newFile.txt
```

where there is one tab before touch. Save the file and in the same terminal window type make. The first time make is typed, make looks for the file newFile.txt. If the file does not exist it runs the touch command and the touch command creates newFile.txt. If make is typed again, it finds newFile.txt and does not run the touch command.

Similar to other scripting languages, comments start with a hash or pound character (#). The target newFile.txt has no spaces or tabs in front of it and is followed by a colon. Dependencies for the target can be added after the colon. The actions are given on lines after the target name and must be prefixed by a tab character. If white spaces are used, make will report an error. By default make looks for a file called Makefile. Other file names can be used by using the -f option. For example, make -f another.mk

Processing dependencies is where make becomes really useful. A small C program can be used to demonstrate this concept. Using the nano editor, create three files - main.c, printString.c, printString.h - and place them In a new folder.

main.c

```
#include "printString.h"
int main() {
  printString();
  return 0;
}
```

printString.c

```
#include <stdio.h>
void printString() {
  printf("Built with make!\n");
}
```

and printString.h

```
void printString();
```

Now create a new file called Makefile containing,

```
printString: main.o printString.o
    gcc -o printString main.o printString.o

main.o:
    gcc -c main.c

printString.o:
    gcc -c printString.c

clean:
    rm -f *.o
```

This time typing make will cause each .c file to be compiled into a .o file. Then the .o files are linked together to form an executable called printString. The printString target is the first target in the file and is therefore the default target. When make runs it checks the dependencies of printString, which are that the main.o and printString.o files exist and are not newer than the target printString. If the files do not exist then the target to make the file is run. Any target other than the default target can be run by typing the target name after the make command, e.g. make clean

Writing make files where each of the file names needs to be specified can rapidly become very time consuming. Automatic variables can be used instead of explicitly specified targets,

```
printIt: main.o printString.o
    gcc -o $@ $^

%.o: %.c
    gcc -c $< -o $@

clean:
    rm -f *.o
```

This Makefile has exactly the same action as the previous one. The automatic variable $@ is the target name, $^ are the names of all of the dependencies, and $< is the name of the first prerequisite. For each .o file required by the default target, make tries the wildcard %.c. If the .c file is missing make will report an error.

Wildcards can also be used to define a list of objects from the list of .c files in the present working directory,

```
OBJECTS = $(patsubst %.c,%.o, $(wildcard *.c))

printIt: $(OBJECTS)
    gcc -o $@ $^

%.o: %.c
    gcc -c $< -o $@
```

where OBJECTS is a variable. In this case, all of the .c files in the present working directory are used to build an executable called printIt. The command wildcard lists all files which match the pattern *.c. Then patsubst removes the .c ending and replaces it with .o. The resulting list is assigned to the OBJECTS variable. Try using touch to update each file time stamp and then re-run make to see what happens.

Makefiles can have many layers of dependencies. For software distribution on multiple platforms, the Makefiles are typically generated from templates using the autoconf tool.

Article by W. H. Bell

An introduction to the C++ programming language - one of the most popular used today.

C++ is a programming language, like C, Python and Java. It is a bit more advanced, but it is very popular and many of today's video games and many other programs are written in C++. It's fast and easily portable, which means the same code is mostly transferable between Linux, Windows and Mac OS machines. If you're up for the challenge, read on!

Try typing:

```
#include <iostream>
using namespace std;

int main()
{
  // Output a message.
  cout << "Hello, welcome to C++" << endl;
  return 0;
}
```

save it as "`hello.cpp`" and then compile by typing `g++ -o hello hello.cpp`

After compiling and running, you should see it print the message within the speech marks. Try changing this, recompiling and running, and see what happens. At first this looks quite daunting, but look below and on the next page, where there is an explanation of the unfamiliar terms.

So now we've written our first program, how do we know what it will do, and what everything means? Here I will explain the important aspects of the program:

`#include <iostream>`

This is our include statement. This is used to include function, class and data definitions from the standard `iostream` library. The include statement is needed to use the `cout` and `endl` functions. The `iostream` library contains information on inputting and outputting.

`int main()`

This is the `main` function. All programs need a main function and anything within the `main` function is executed. All functions begin with '`{`' and end in '`}`'.

`// Output a message`

Anything beginning with '`//`' in C++ is a comment, and like comments in other programs these are ignored by the compiler. C-style comments using `/* */` are also allowed.

`cout` **and** `endl`

These are our commands. The cout command tells the program to output everything between the '`<<`' and the '`;`'. `endl` simply means 'end line'.

`;`
Some languages use these, semicolons. To explain them, think of the way we write normal, spoken languages like English. We end our sentences with a full stop. This is a similar idea - every time we want the program to move onto something new, we end it with a semicolon. Because of this we can use whitespace, which are spaces and new lines, however we like.

`return 0;`
This lets the program know that the `main` function is over, which means it is finished. It will stop running past this point.

With this information, lets move on to some more examples. Try the following:

```
#include <iostream>
using namespace std;

int main()
{
  // Create 2 variables
  int a, b;
  a = 1;
  b = 2;

  // Output the sum of these variables
  cout << “a + b = ” << a + b << endl;
  return 0;
}
```

Here, we have made two variables, `a` and `b`. They are integers, which means they are whole numbers. We have created the two, and then output the sum of the two.

Of course this is all well and good, but the output will always be 3 unless we change the code, which is not very useful. Instead, we could modify the program so we take a user input, and add those together. Try this:

```
#include <iostream>
using namespace std;

int main()
{
  // Create 2 variables
  int a, b;

  // Ask for and store user input
  cout << “Input the first number: ”;
  cin >> a;

  cout << “Input the second number: ”;
  cin >> b;

  // Output the sum of these variables
  cout << a << “ + ” << b << “ = ” << a + b << endl;
  return 0;
}
```

This will allow you to get the user’s inputs and add the two values.

THE SCRATCH PATCH

Defensive Bases

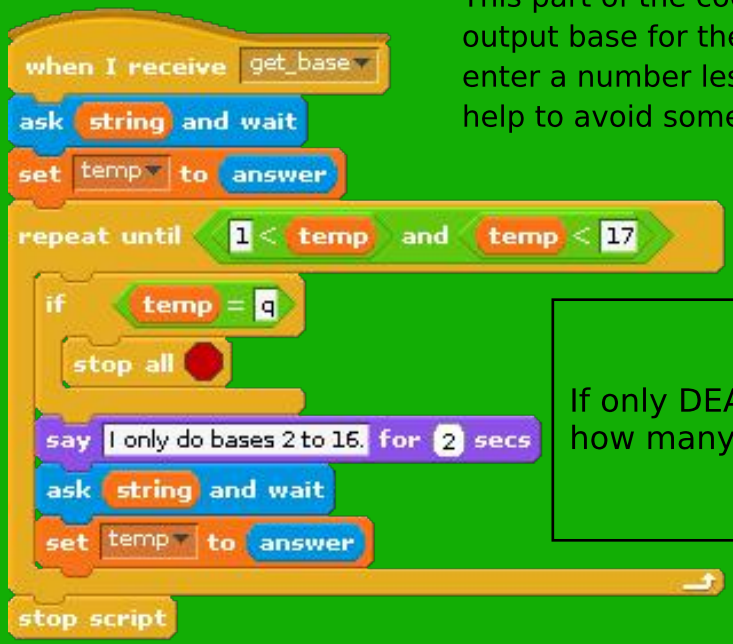
This month we are having a go at some "defensive programming"; in other words trying to write a program that can handle errors.

This converts a number from denary into the output base, getting the digits from the string "DIGITS" (see next page).

Sorry the scripts are in a funny order this month - space is tight!

The "get_base" Script

This part of the code allows you to set the input and output base for the program to use. It won't allow you to enter a number less than 2 or greater than 16. This will help to avoid some really strange (and incorrect) results.



Hex Joke!

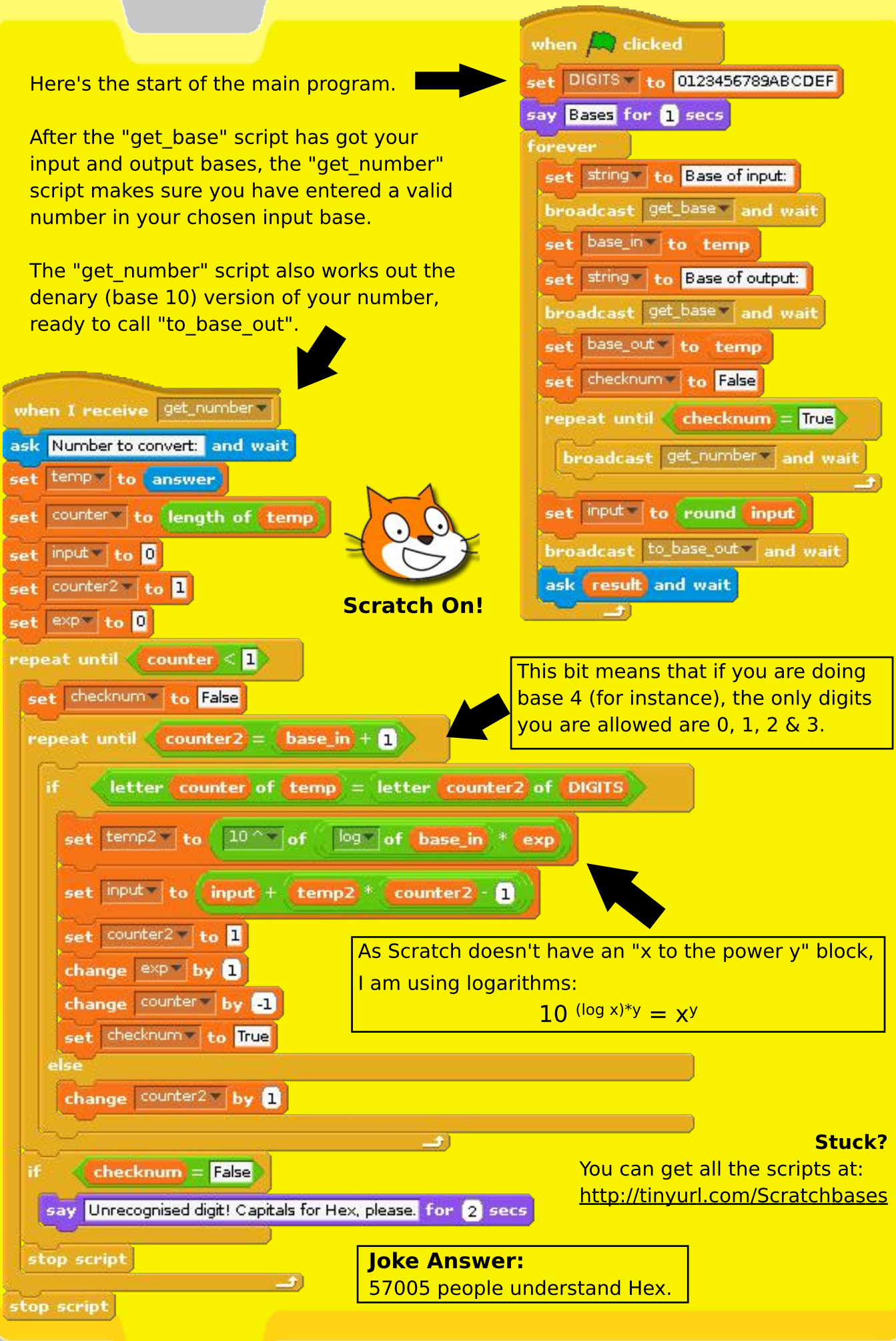
If only DEAD people understand hexadecimal, how many people understand hexadecimal?

(answer on next page!)

Here's the start of the main program.

After the "get_base" script has got your input and output bases, the "get_number" script makes sure you have entered a valid number in your chosen input base.

The "get_number" script also works out the denary (base 10) version of your number, ready to call "to_base_out".



Scratch On!

This bit means that if you are doing base 4 (for instance), the only digits you are allowed are 0, 1, 2 & 3.

As Scratch doesn't have an "x to the power y" block, I am using logarithms:

$$10^{(\log x)*y} = x^y$$

Stuck?
You can get all the scripts at:
http://tinyurl.com/Scratchbases

Joke Answer:
57005 people understand Hex.

the python pit™

Last month The Python Pit showed how to pull in configuration settings from an external text file. This month another option is presented.

Using command line arguments gives fine control over a program at the moment of execution.

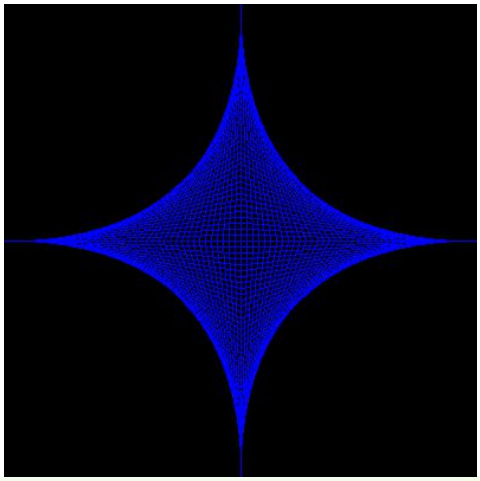
This Pygame line renderer uses Python's argparse:
http://docs.python.org/dev/library/argparse.html
At the LXTerminal run the command:

```
python lines.py -h
```

This will display all arguments available. For example:

```
python lines.py -s 3 -t 4
```

will generate a shape larger than the default (-s 2) and with slightly denser lines (-t 5). Experiment with the options available.



```
# line generator with command line arguments
# By Colin Deady - 03 October 2012

import os, pygame, argparse, sys
from pygame.locals import *

# initialise pygame (to render the image)
pygame.init()

# Define two functions that will be used:

# 1) fnAppend2Log will write a line to a log file
def fnAppend2Log( line2write ):
    logfile = open('lines.log', 'a')
    logfile.write(line2write + '\n')
    logfile.close()

# 2) fnPlotLines will render a quarter of the shape.
# Uses the previous co-ordinates as the new starting co-ordinates
def fnPlotLines(quarter, sX, sY, eX, eY, incSX, incSY, incEX, incEY ):
    fnAppend2Log(quarter + ' quarter co-ordinates:')

    # calculate and loop through line co-ordinates
    for i in range(0,iterations, args.step):
        nSX = sX + (incSX * i) # start X
        nSY = sY + (incSY * i) # start Y
        nEX = eX + (incEX * i) # end X
        nEY = eY + (incEY * i) # end Y

        # draw a line between the pair of co-ordinates.
        pygame.draw.line(screen,(lineColour),(nSX,nSY),(nEX,nEY),1)
```

PYTHON VERSION: 2.7.3rc2
PYGAME VERSION: 1.9.2a0
O.S.: Debian 7

TESTED!

```
        # construct a string for the window title and the log file
        coordText = '('+str(nSX)+','+str(nSY)+')-
                    ('+str(nEX)+','+str(nEY)+')'
        # render the image line by line (takes longer)?
        if args.renderlines == 'y':
            pygame.display.update();
            pygame.display.set_caption(coordText)

        # output co-ordinates to the log
        fnAppend2Log(coordText)

    # return the final calculated co-ordinates
    return (nSX, nSY, nEX, nEY);

# define the command line arguments:
parser = argparse.ArgumentParser(description='Render shape')
parser.add_argument('-s', action='store', dest='scale', type=int,
                    default=2, help='Render size, default=2, 200x200px)')
parser.add_argument('-t', action='store', dest='step', type=int,
                    default=5,
                    help='Lower step values for denser lines (default=5)')
parser.add_argument('-r', action='store', dest='renderlines',
                   choices=('y','n'), default='y',
                   help='Render line by line (Y) or finished object (n)')
args = parser.parse_args()

# Define the variables that will be needed
sz = 100*args.scale # size in pixels horiz x vert of a quarter image
iterations = sz +5 # number of lines to render per quarter
lineColour = 0,0,255 # the colour of the line to draw (blue)

# open a pygame screen on which to render our objects
# the image size is twice the object to be rendered as we render 4 quarters
screen = pygame.display.set_mode([sz*2,sz*2],0,32)

# Draw the lines, quarter by quarter, returning the co-ordinate pairs
# The starting co-ordinates equal the end from the last quarter rendered
sx, sy, ex, ey = fnPlotLines('Top left', sz, 0, sz, sz, 0, 1, -1, 0 )
sx, sy, ex, ey = fnPlotLines('Bottom left', ex, ey, sx, sy, 1, 0, 0, 1 )
sx, sy, ex, ey = fnPlotLines('Bottom right', ex, ey, sx, sy, 0, -1, 1, 0 )
sx, sy, ex, ey = fnPlotLines('Top right', ex, ey, sx, sy, -1, 0, 0, -1 )

# if rendering each line is suppressed then display the final image
if args.renderlines == 'n':
    pygame.display.update();

# save the rendered image to a file
pygame.image.save(screen, 'lineimage.png')

# display the result for 10 seconds
pygame.time.wait(10000)
```

Try adding an extra argument to optionally disable writing to the log file. This will improve rendering time. Hint: as well as the argument you will need to add two `if` statements as the log is written to in two different places within the code.

Some other ideas for command line arguments:

- let the user choose the filename for the output image
- specify the background and line colours from a list of defaults (black, white,red, green, blue)
- enter a demo mode that loops the code, rendering many shapes in random colours

Feedback & Question Time

Q: Regarding the Skutter series, you mention a robot arm can be mounted. Could you please specify which robot arm and where I can get it?

Richard

A: Skutter is a long-term series, with the first article published in issue 1 of The MagPi. This contains some background information: "The robotic arm kit called the OWI Edge is currently available from Maplin electronics and it uses a simple USB interface to control it."
http://www.maplin.co.uk/robotic-arm-kit-with-usb-pc-interface-266257

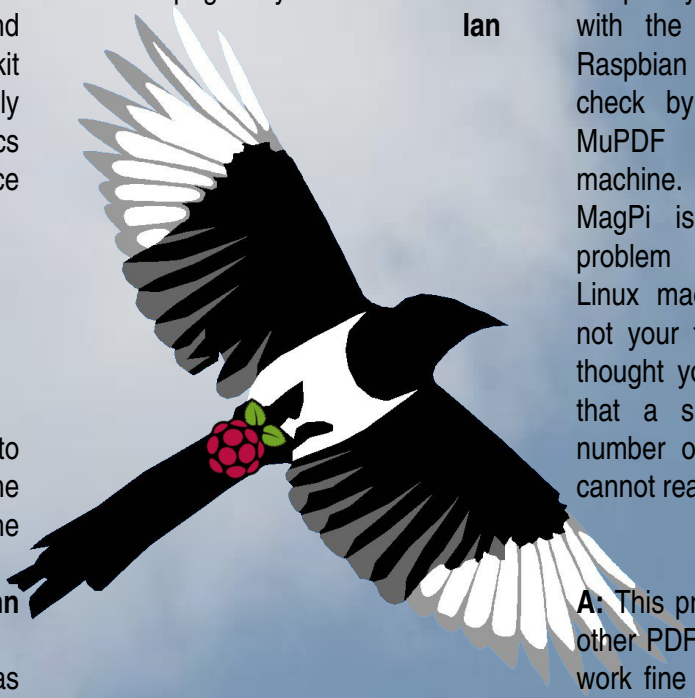
Q: Is it going to be possible to view the mag on an iPad in the near future or am I missing the point?

John

A: The Issuu site we use has recently started to support HTML5, so it should now work on iPads and iPhones etc. You can also download the PDF and view it within iBooks. We are currently working with a developer regarding a Newsstand app.

Thank you for making a printed edition of The MagPi available from http://www.modmypi.com. I always print out each issue because I prefer to read from paper rather than online. Imagine my surprise to discover that the cost is only £2.49. It almost costs that much for me to print out 32 colour pages myself!

Ian



Q: As a new user of the Raspberry Pi I'm very interested in your excellent magazine. I have had no difficulty reading your Issues 1 to 6 on my desktop computer that uses Linux Mint 13. Imagine my disappointment when I found that, of the six issues, I can only open Issue 3 on my Raspberry Pi. Not being familiar with the MuPDF program that Raspbian uses, I thought I'd better check by installing and testing MuPDF on my Linux Mint machine. The result is that every MagPi issue opens without a problem using MuPDF on my Linux machine. This is possibly not your technical problem but I thought you would want to know that a significant and growing number of Raspberry Pi owners cannot read your magazine.

Lloyd

A: This problem extends to many other PDF files. MuPDF seems to work fine for some users but not others. Personally I have removed MuPDF and use xPDF instead, which works with everything. You can do this with the following commands:

```
$ sudo apt-get remove MuPDF
$ sudo apt-get install xPDF
```

The MagPi™

editor@themagpi.com