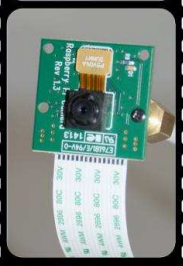
ISSUE 14 - JUL 2013

Get printed copies at themagpi.com

The MagPi™

A Magazine for Raspberry Pi Users

The camera module



I/O and logic expansion

Robotic arm control

Parallel processing

Bootcamp report

LED matrix

Charm

JAVA

Win a 512MB Raspberry Pi & interfacing goodies

ISSN 2051-9990

The MagPi

http://www.themagpi.com

Welcome to the 14th issue of The MagPi - another fully loaded guide to all things Raspberry Pi!

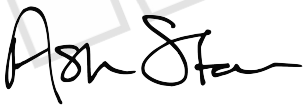
This month we begin our journey in introducing the newest module in the Raspberry Pi world, the camera, with a great guide written by James Hughes on setup and basic operation. The conclusion of this article will be found in next month's edition where James builds on these foundations with hints and tips on advanced usage.

We reload the Matrix with part two of the interesting Pi Matrix article, take a look at the Guzunty board and review the ever popular South West Raspberry Pi Boot Camps.

We bring you more on the programming languages Charm, Scratch and Python plus bring you the first MagPi article on the popular language, Java.

I am pleased to report that all pre-order Volume 1 MagPi bundles ordered via Kickstarter or themagpi.com have now shipped. We will soon have individual issues coming to print. We hope you enjoy the printed editions and thank you again for your support and patience in helping make the dream a reality.

Ash Stone

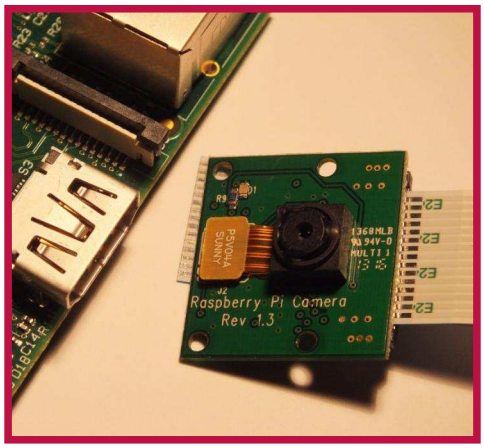


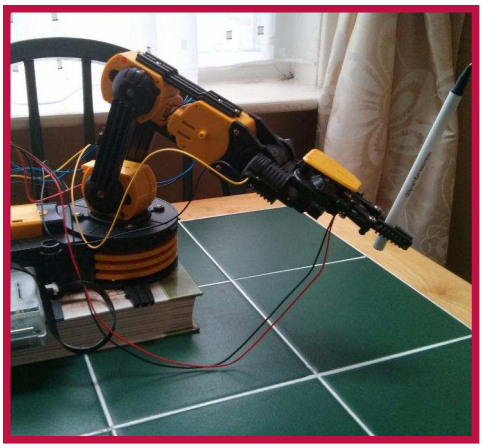
Chief Editor of The MagPi

The MagPi Team

Ash Stone - Chief Editor / Administration / Proof Reading
W.H. Bell - Issue Editor / Layout / Graphics / Administration
Bryan Butler - Page Design / Graphics
Ian McAlpine - Layout / Tester / Proof Reading
Chris 'tzj' Stagg - Tester
Colin Deady - Layout / Proof Reading
Matt Judge - Website / Administration
Aaron Shaw - Layout
Shelton Caruthers - Proof Reading
Meltwater - Proof Reading
Sai Yamanoor - Tester
James Nelson - Proof Reading
Adrian Harper - Layout
Paul Carpenter - Tester
Dave Allan - Tester
Claire Price - Proof Reading
Phil Tesseyman - Tester
Steve Drew - Layout

Contents





Boot Camp photographs on pages 3, 22 and 23 by Alex Sheppard
Prototype camera mount on the front cover is courtesy of Grasping Hand

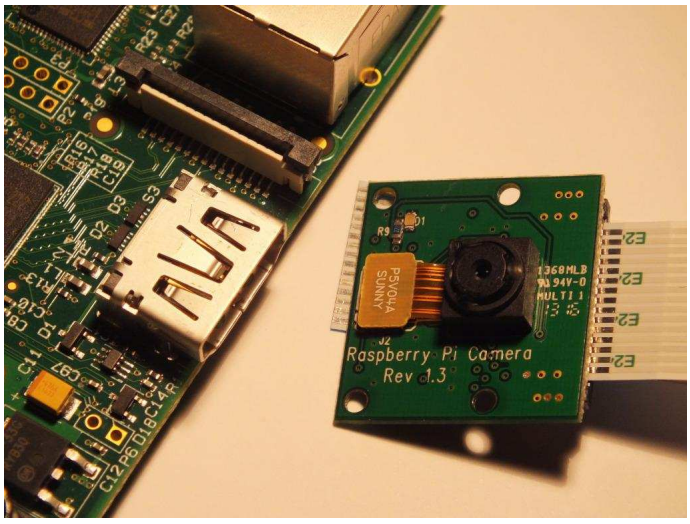
The Raspberry Pi camera - part 1

DIFFICULTY : BEGINNER

James Hughes

Guest Writer

A few weeks ago, the Raspberry Pi Foundation launched their first peripheral, a 5MP camera. Priced at $25, the same price as the model A, it's a very small PCB on which is an Omnivision OV5647 camera module. It connects to either the Model A or Model B Raspberry Pi using a 15cm 15 way ribbon connector.



Over the course of this series I will take you through initial connection of the camera to your Raspberry Pi and will show you some of the basic commands used to take both still and video imagery. At the end, some of the more sophisticated features will also be explained. Not every option will be covered (new options are being added all the time so it's difficult to keep up) but hopefully this article will give enough information for you to be able to cover most image taking tasks. Firstly though, a description of how the camera module came to be...

History

From its first launch the Raspberry Pi has had a connector on it to attach a camera to the GPU (the VideoCore 4 Graphics Processing Unit on the Raspberry Pi). This connection uses the CSI-2 electrical protocol and is a standard used in most mobile phones. It is an extremely fast connection, which on the Raspberry Pi is capable of sending 1080p sized images (1920x1080 x10bpp) at 30 frames per second, or lower resolution at even higher frame rates. It had always been intended at some point to release a camera module that could use this connection, as the ability to stream high speed video data through the GPU without any interaction with the ARM processor would always make the camera much more efficient than any USB attached webcam. It would also enable the use of the GPU's ability to encode H264 video, or JPEG images in hardware.

It turns out that productising a tiny PCB like the camera board is not a quick task! The prototype was re-designed to remove some unnecessary components, but more importantly, to move the camera crystal, used for timing, to the PCB itself.

Electromagnetic compatibility (EMC) testing had shown that the 25Mhz clock provided by the GPU caused too much interference as it passed up the ribbon cable to the PCB. Adding a crystal to the PCB itself stopped this interference. A couple more board designs later, to help with production line manufacture and testing, and eventually, about a year after the first prototypes, the production board was ready.

Meanwhile, work had been ongoing to write a couple of applications to make use of the camera, to update the GPU firmware to support the camera, and to improve the camera tuning as the basic tuning already in place had a number of obvious defects.

Camera tuning is a complex task, which has been covered on the Raspberry Pi website, so I won't repeat it here. Suffice it to say, the end results are well worth the extra effort put in by David Plowman while at Broadcom. Thanks David.

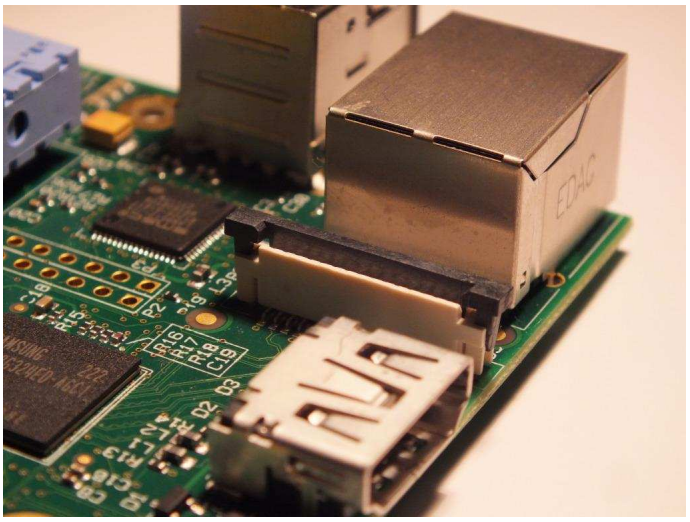
So, with the history out of the way, let's take a look at getting your camera going.

Setting up

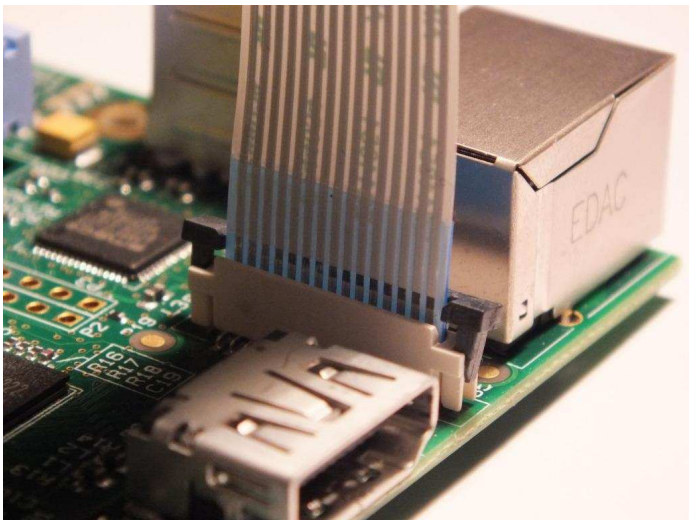
Firstly, it's important to start with a warning. Cameras like these are static sensitive. You should earth yourself prior to handling the PCB (a sink tap/faucet or similar should suffice if you don't have an earthing strap).

There are only two connections to make, the ribbon cable needs to be attached to the camera PCB and the Raspberry Pi itself. You need to get it the right way round or the camera will not work. On the camera PCB, the blue backing on the cable should be away from the PCB, and on the Raspberry Pi it should be towards the ethernet connection (or where the ethernet connector would be if you are using a model A).

Although the connectors on the PCB and the Pi are different, they work in a similar way. On the Raspberry Pi, you need to pull up the tabs on each end of the connector.



It should slide up easily, and be able to pivot around slightly. Fully insert the ribbon cable into the slot, ensuring it is straight, then gently press down the tabs to clip it into place. The camera PCB itself also requires you to pull the tabs away from the board, gently insert the cable, then push the tabs back.



The PCB connector is a little more awkward than the one on the Pi itself. There is a video at http://youtu.be/GImeVqHQzsE which shows the connections being made.

So, we now have the hardware all attached, we now need to make sure we have the correct software installed. As explained above, there are some command line apps to install, and a firmware upgrade with the camera driver and tuning to install. This process may not be necessary in future as newer distro's are

released which already contain the required files.

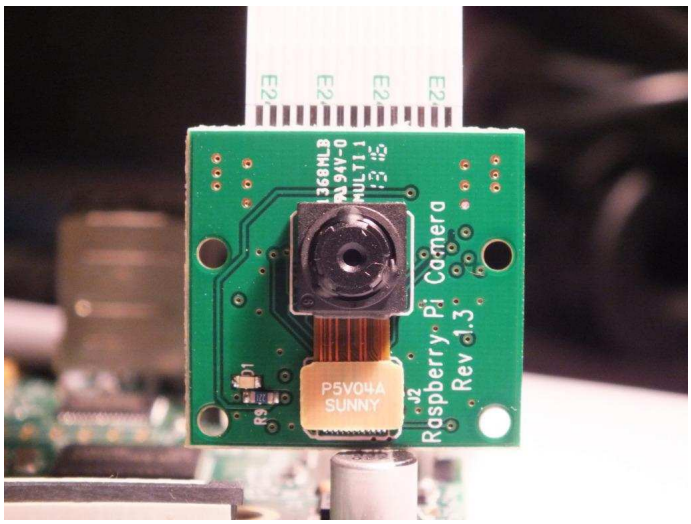
To update the firmware, libraries and applications, execute the following instructions on the command line.

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

Now you need to enable camera support using the raspi-config program you will have used when you first set up your Raspberry Pi.

```
sudo raspi-config
```

Use the cursor keys to move to the camera option and select enable. When you exit raspi-config it will ask to reboot. The enable option will ensure that on reboot the correct GPU firmware will be running (with the camera driver and tuning), and the GPU memory split is sufficient to allow the camera to acquire enough memory to run correctly. Running a camera and associated encoder does take up a big chunk of memory, as



the images are large, and there needs to be at least two of them in memory at any time in order to provide a double (in some cases triple) buffer which prevents 'tearing' of images during display.

Checking it works

OK, we have connected the hardware, and installed the software. Now we need to see if it is all working correctly! The quickest way is just to type in

```
raspistill -t 5000
```

This runs the stills capture program for 5 seconds (or 5000 milliseconds. Note that all times used are in milliseconds). You should see the whole display replaced with a moving image of whatever the camera is looking at. If you don't see the preview appear, or there are error messages displayed, go straight to the Troubleshooting section at the end of the article!

So, what can it do?

As with any camera, it can work in two ways. We can capture still images, just like a camera, or we can capture video, like a camcorder. The two demo applications provided handle these tasks separately. Although it would be quite possible to combine them into one application, that makes the code more complicated, and as demo code a lot of effort was put in to make it as easy to understand as possible. As an aside, the C language source code for the applications is publicly available from the Foundation's github repository. Anyone is welcome to play around with it and alter it for their particular purposes. The code is well commented and uses the Doxygen commenting scheme so you can produce HTML documentation directly from the source code.

So, on to the apps themselves. They are command line applications, and do not need to be run from a desktop environment since the preview output from the apps is superimposed over the top of any desktop or console being used. Typing just the name of the application, or adding --help or -? to the command line will produce a list of all the available options. The list is quite long, so you may want to pipe the output into 'less' so you can scroll up and down to read it.

```
raspistill | less
raspivid | less
```

Basic options

Both applications use the -t option to specify the length of time in milliseconds they should continue running.

So “raspistill -t 3000” will run the camera for 3 seconds.

In the stills case, and if a filename is specified, the capture of the still will take place at the end of the time period. In the video case, and again if a filename is specified, the capture starts straight away and will be as long as the time period.

To specify the filename of the output file, you use the -o option. The following command runs the camera for 2 seconds and at the end of the period will take a still image, saving it to the file image.jpg.

```
raspistill -o image.jpg -t 2000
```

This command will do the same but will save a 2 second H264 video file called video.h264

```
raspivid -o video.h264 -t 2000
```

You don’t need to specify a filename for the output - if you don’t then no capture is done, but the preview will still be run for the specified time. If you don’t specify a time, then 5 seconds is used as a default.

The following will take a picture after 5 seconds.

```
raspistill -o image.jpg
```

So we have made some files, but how do we see what they look like? Well, to view JPG images, the FBI program is quite useful. You can install it quickly using:

```
sudo apt-get install fbi
```

Then you can easily display your JPG images, zoom in and out (use - and +) etc.

```
fbi image.jpg
```

To view video, you can use the already installed omxplayer.

```
omxplayer video.h264
```

Preview options

You can specify whether you want the preview disabled (-n), or whether it is to appear full screen (-f) or in a window (-p). You can also specify what opacity (-op) the preview window is to have (so you can see the desktop underneath if necessary).

To display the preview in a window at position 100,100, width 500, height 400 and at 50% opacity (0-transparent to 255 -opaque), enter:

```
raspistill -p 100,100,500,400 -op 128 -o
image.jpg
```

To disable preview completely, enter:

```
raspistill -n -o image.jpg
```

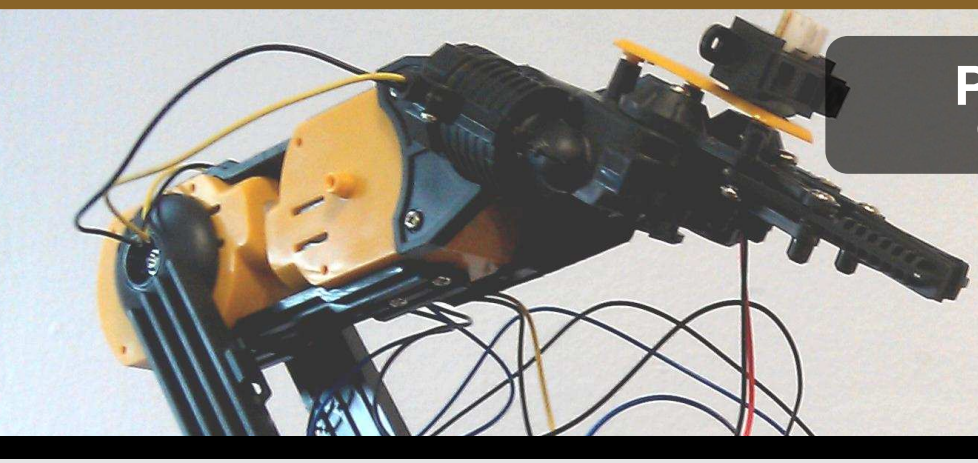
Join us in issue 15 of The MagPi where some of the more advanced features will be discussed!

Troubleshooting

If your camera is not working, there are a number of things to check to ensure it is set up correctly.

1) Are the ribbon connectors all firmly seated and the right way round? 2) Is the camera module connector firmly attached to the camera PCB? 3) Have you run sudo apt-get update, sudo apt-get upgrade? 4) Have you run raspi-config and enabled the camera?

So, if things are still not working, try the following:
Error : raspistill/raspivid not found. This probably means your update/upgrade failed in some way. Try it again.
Error : ENOMEM displayed. Camera is not starting up. Check all connections again.
Error : ENOSPC displayed. Camera is probably running out of GPU memory. Check config.txt in the /boot/ folder. The gpu_mem option should be at least 128.



Controlling the Maplin/OWIrobot robotic arm

Peter Lavelle

Guest Writer

DIFFICULTY : BEGINNER

Introduction

This article expands on the work by Jamie Scott, et al in the Wikihow article at http://www.wikihow.com/Use-a-USB-Robotic-Arm-with-a-Raspberry-Pi-(Maplin) and allows you to control the movement of the arm using a sequence of instructions read from a CSV (Comma Separated Variables) formatted file by a Python script.

This article assumes you are using the Raspbian distribution but you should also be able to get this working using the Occidentalis distribution from Adafruit. (More information about this distribution can be found at http://learn.adafruit.com/adafruit-raspberry-pi-educational-linux-distro/overview).

Configuring your Raspberry Pi

Before you can start using the script, there are a few things you will have to do to get your Raspberry Pi ready. The first thing you will need to do is add the user account you are using on the Raspberry Pi to the 'plugdev' group. Assuming you are using the default 'pi' user account, the command to do this will be:

```
sudo usermod -aG plugdev pi
```

The next step is to add a udev rule to allow the 'pi' user to send commands to the arm. To do this, create a file called /etc/udev/rules.d/85-robotarm.rules, using the command:

```
sudo nano /etc/udev/rules.d/85-robotarm.rules
```

with the contents below:

```
SUBSYSTEM=="usb", ATTRS{idVendor}=="1267",
ATTRS{idProduct}=="0000", ACTION=="add",
GROUP="plugdev", MODE="0666"
```

To save the file, press ctrl-x and then press ‘Y’ followed by Enter.

You’ll then need to install the Python USB libraries. I found that the one from the Apt repository was slightly out of date, so I had to install the Apt version first and then upgrade it using the pip command. The command below will do this:

```
sudo apt-get install python-pip -y
sudo pip install pyusb --upgrade
```

When the Python USB libraries have been installed, reboot your Raspberry Pi so that the changes to udev and the 'pi' user account take effect with the command:

```
sudo shutdown -r now
```

Connect the arm

When the Raspberry Pi has been rebooted, connect the arm to a free USB port and turn it on. A USB hub should not be needed here. To check that the Pi has successfully detected the arm, execute the command:

```
dmesg | grep usb | grep 1267
```

If the arm has been detected successfully you should see output similar to the line below:

```
[  252.790554] usb 1-1.3: New USB device
found, idVendor=1267, idProduct=0000
```

If you don't see the result above, turn off the arm using the power switch, unplug from the USB port, plug the arm back in and turn the power back on.

Getting the code

The scripts can be found on Github at https://github.com/peterlavelle/maplinarm

You can clone the repository directly on to your Pi if you have the Git client installed. Installing the Git client on Raspbian is as simple as running the command:

```
sudo apt-get install git-core
```

When you have the Git client installed, run the command:

```
git clone
https://github.com/peterlavelle/maplinarm
```

to pull down a copy of the Python code. This command will clone the repository into a subdirectory called 'maplinarm' and should pull down the files listed below:

```
git clone  https://github.com/peterlavelle/maplinarm
Cloning into 'maplinarm'...
remote: Counting objects: 35, done.
remote: Compressing objects: 100% (23/23),  done.
remote: Total 35 (delta 8), reused 12 (delta 2)
Unpacking objects: 100% (35/35), done.
```

```
cd maplinarm/
ls
commander.py  commands.csv  maplinrobot.py
README.md
```

Making the scripts executable

Now we need to make the scripts executable by the Raspberry Pi. To do this, change into the directory you cloned the repository down to using the cd command and run the command chmod 755 *.py This will give all files ending in '.py' executable permissions. You can double-check this with the command ls -la *.py and looking for the 'x' in the left-hand column of the output on each line. See example listing below:

```
drwxrwxr-x  3 pete pete 4096 May 19 14:11 .
drwx------ 37 pete pete 4096 May 19 14:11 ..
-rwxrwxr-x  1 pete pete 1726 May 19 14:11 commander.py
-rwxrwxr-x  1 pete pete 2388 May 19 14:11 maplinrobot.py
```

Programming the arm

Ok, now on to the good part. The script commander.py will read in commands from a CSV file passed to it as an argument.

There is an example file called 'commands.csv' in the Git repository to help you get started. So lets open this file up and take a look with the command:

```
nano commands.csv
```

Contents are included below for reference:

```
shoulder-up,1.00,1.00
elbow-down,1.00,2.00
base-clockwise,4.00,1.00
shoulder-down,1.00,1.00
grip-close,1.00,1.00
base-anti-clockwise,4.00,2.00
grip-open,1.00,1.00
elbow-up,1.00,2.00
```

As you can see here, the basic format for each command is:

```
command,duration,pause
```

`command` - any valid command found in the maplinrobot.py script
`duration` - the time in seconds to execute the command for. Make sure all values are entered to two decimal places. e.g, for 1 second specify a value of 1.00 here.
`pause` - the time in seconds to wait before executing the next command in the sequence. Make sure all values are entered to two decimal places. e.g, for 1 second specify a value of 1.00 here.

The commander.py script will execute each line in the CSV file as a separate command for the duration you set and will wait for the number of seconds set in the pause value before moving on to the next command in the sequence.

Valid commands and a description for each can be found below:

`base-anti-clockwise` - Rotates the base anti-clockwise
`base-clockwise` - Rotates the base clockwise
`shoulder-up` - Raises the shoulder
`shoulder-down` - Lowers the shoulder
`elbow-up` - Raises the elbow
`elbow-down` - Lowers the elbow
`wrist-up` - Raises the wrist
`wrist-down` - Lowers the wrist
`grip-open` - Opens the grip
`grip-close` - Closes the grip
`light-on` - Turns on the LED in the grip
`light-off` - Turns the LED in the grip off
`stop` - Stops all movement of the arm

Try this out yourself by editing the commands.csv file in the repository and replacing the contents with your own instructions to the arm.

Running your program

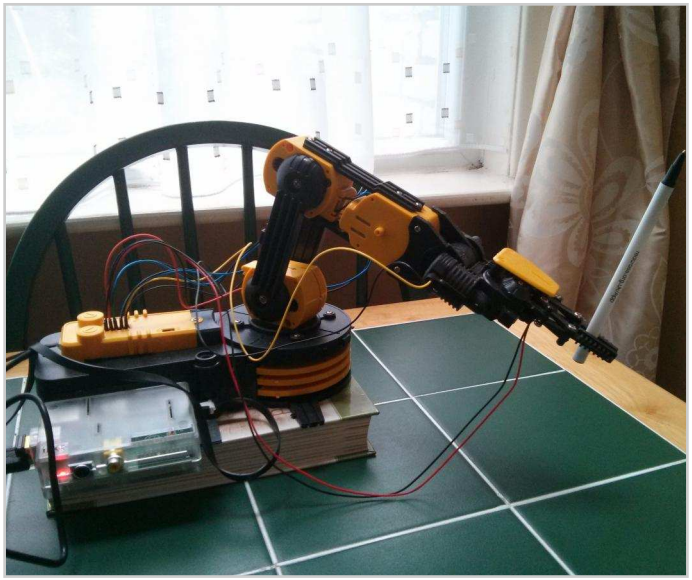
Running your program is as simple as running the command ./commander.py commands.csv from the command line on your Raspberry Pi. The script will output each step of your program as it runs it. You can find a example output of my terminal below as an example run using the commands.csv file in the repository:

```
 ./commander.py commands.csv
 Running command 'shoulder-up' for a duration
of 1.000000 second(s) with a pause of 1.000000
second(s)
 Sending command shoulder-up
```

The series of commands will then finish with:

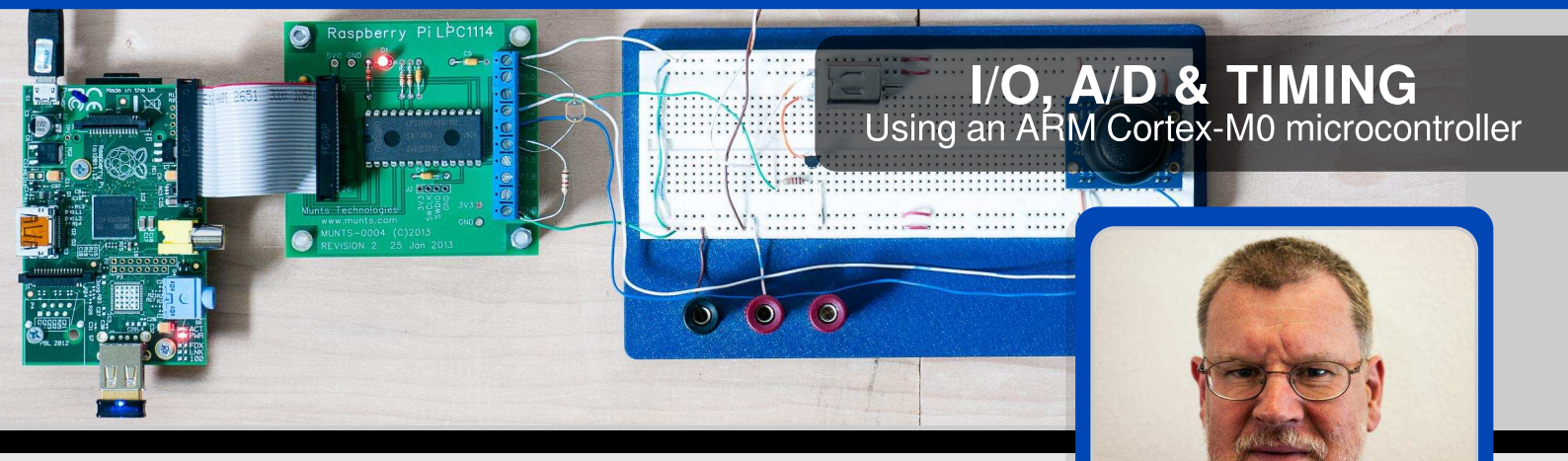
```
 Done.

 All commands executed. Stopping the arm
```

That's it. If you have any ideas on how this could be improved or expanded (maybe by adding some sort of control logic to the commands file, for example) then feel free to contact me via the contact form on my blog at http://solderintheveins.co.uk/contact-me/



Where to buy?

In the UK the robot arm is available from Maplin (http://www.maplin.co.uk - code A37JN) or elsewhere from Robotikits (http://www.owirobot.com - codes OWI-535 and 535-USB).



Input/Output Processor

DIFFICULTY : ADVANCED

Philip Munts

Guest Writer

As an embedded system developer, I awaited the release of the Raspberry Pi with great interest and anticipation. I ordered one early on and, like many others, waited for it to arrive. There are many, many embedded system possibilities for a $35 Linux computer.

Linux brings a great deal of capability to the party, especially with regards to networking and USB support. But one thing Linux systems, including the Raspberry Pi, lack: microsecond I/O (input/output) signal timing resolution. The multiuser, multitasking, virtual memory foundation of Linux simply prevents it from responding to or timing external events with predictable microsecond resolution.

Some interesting projects that would require microsecond timing resolution include:

- IR (Infrared) remote control protocols, such as the LEGO® Power Functions RC protocol;
- DCC (Digital Command Control) for model railroads;
- Ultrasonic ranging with a device like the Parallax ultrasonic distance sensor.

The other capabilities of Linux are so compelling that attempts have been made from time to time to graft in real time support. None of these have been particularly successful, at least in terms of "mind share". But fortunately there is another way: the I/O processor (Figure 1).

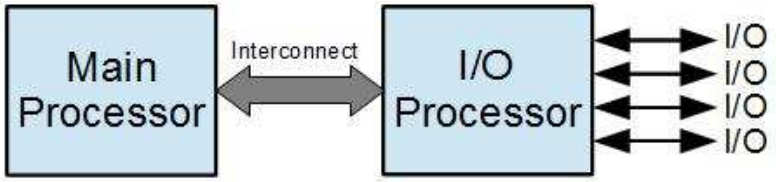


Figure 1 - I/O Processor

I/O Processor

An I/O Processor is simply a separate computer dedicated to I/O operations, usually acting as a slave to the "main" or "real" processor. The I/O processor runs its own software, separate from the main processor. Depending on the implementation, the I/O processor software may or may not be alterable from the main processor.

The I/O processor idea is not new. The IBM 7094 mainframe computer of 1962 could use a 7044 computer for all I/O. Processing was performed on the 7094 and I/O done on the 7044. Even the original IBM PC, released in 1981, had an 8048 8-bit microcontroller in the keyboard to handle key press timing. Today, the I/O processor idea has been pushed all the way down to single chip systems: The NXP LPC4300, itself a single chip ARM Cortex-M4 microcontroller, includes a separate ARM Cortex-M0 microcontroller, for real time processing, on the same chip.

The Raspberry Pi Model B offers several different I/O interfaces, for example: ethernet, USB, and the P1 expansion header. A case can be made for attaching an I/O processor to any of the three interfaces. Ethernet and USB offer good bandwidth and standard interfaces. An I/O processor board built to attach to either Ethernet or USB would also be able to connect to any other Linux, Windows, or Mac computer. But both Ethernet and USB impose significant cost and complexity penalties upon the I/O processor board. It would be difficult (though not impossible) to design an I/O processor board, with USB or Ethernet, costing less than the Raspberry Pi itself.

The P1 expansion header provides several different I/O interconnect: I2C, SPI, UART, and basic GPIO. Any or all of these may be used to communicate with a microcontroller.

LPC1114 I/O processor

I have designed and built an I/O processor board for the Raspberry Pi using the NXP LPC1114, an ARM Cortex-M0 single chip microcontroller. I selected this device for several reasons:

- **Low cost** - I paid USD $1.26 each for the LPC1114FN28 from Avnet Express. The LPC11xx family is essentially NXP's 32-bit attack on the 8-bit microcontroller market. They are priced less than almost any PIC or AVR of similar capability. The LPC1114 internal oscillator is also stable enough that no crystal or external oscillator is required, saving even more money.
- **DIP package** - I can do surface mount soldering, given proper equipment, but I can't say I enjoy it. As far as I know, the LPC1114FN28 and some PIC32MX devices (more expensive) are the only 32-bit microcontrollers available in a easily hand-solderable DIP package.
- **ISP** (In System Programming) - without additional hardware. NXP ARM Cortex microcontrollers have a built-in serial port boot loader that allows them to be reprogrammed without any additional hardware.
- **Good support** by free development tools. ARM microcontrollers, including the LPC1114, are very well supported by binutils/gcc/newlib/gdb compiler toolchain.

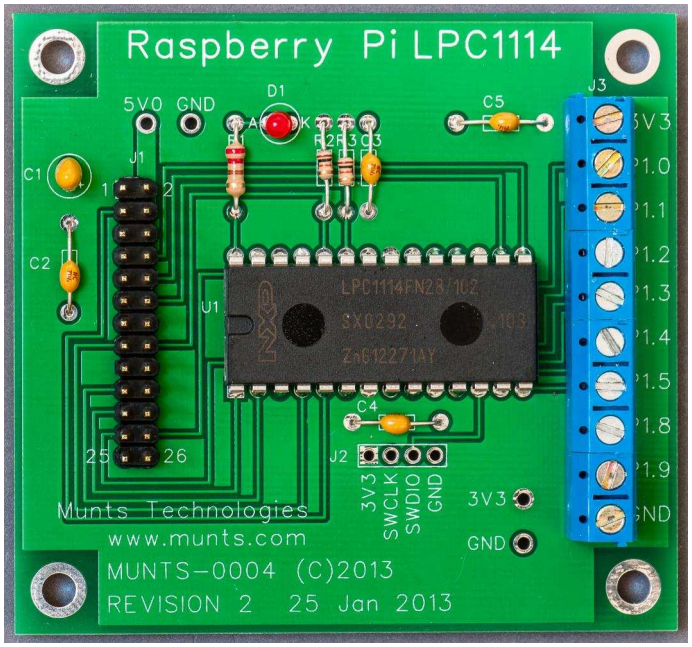


Figure 2 - Assembled I/O Processor board

Some other Raspberry Pi expansion boards contain a microcontroller (usually an AVR) as an addition or afterthought. The LPC1114 microcontroller is central to this board's design. Even though it only contains one integrated circuit, an LED, and 8 resistors and capacitors, this board offers:

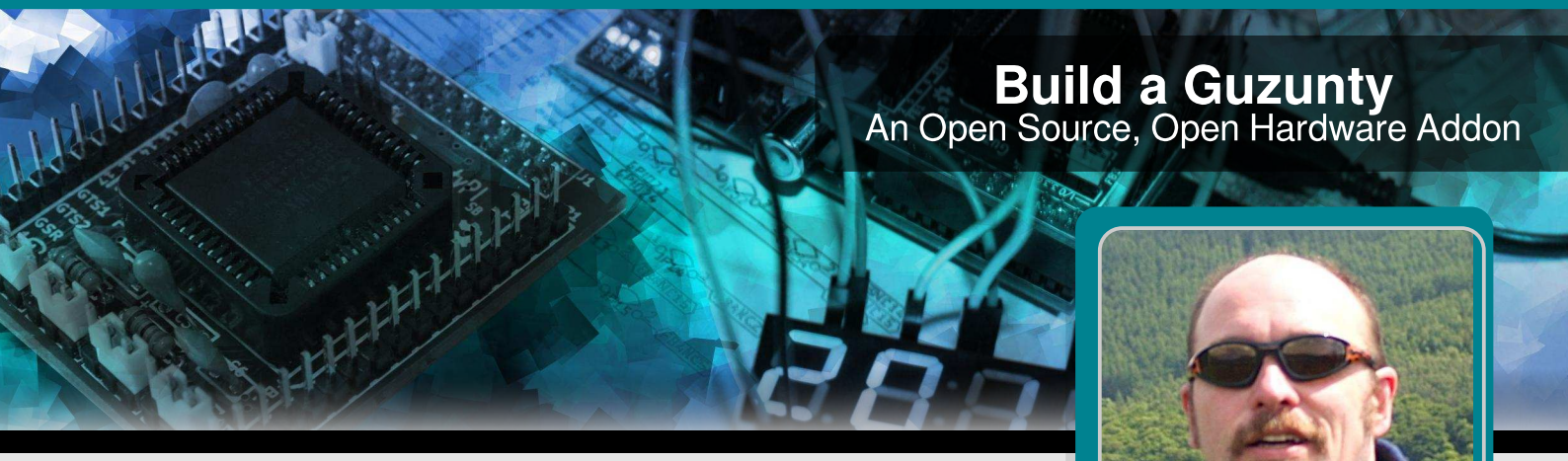
- 8 GPIO (General Purpose Input/Output) signals brought out to screw terminals.
- 5 terminals can be configured as A/D (Analog to Digital) inputs for measuring analog signals.
- 3 terminals can be configured as timer inputs, for counting or measuring incoming pulses.
- 5 terminals can be configured as timer outputs, such as PWM (Pulse Width Modulation) outputs for controlling motors.

Not all input/output functions are available at the same time; for example, P1.0 can be configured for any of digital input, digital output, analog input, timer/counter input, but only one function at a time.

For more information about the LPC1114 I/O Processor, including how to purchase one, visit: http://tech.munts.com/MCU/Frameworks/RaspberryPi/expansion/LPC1114/

Next time

In future articles I will describe in more detail how to use the LPC1114 I/O processor to build some interesting control projects with the Raspberry Pi.



Build a Guzunty
An Open Source, Open Hardware Addon

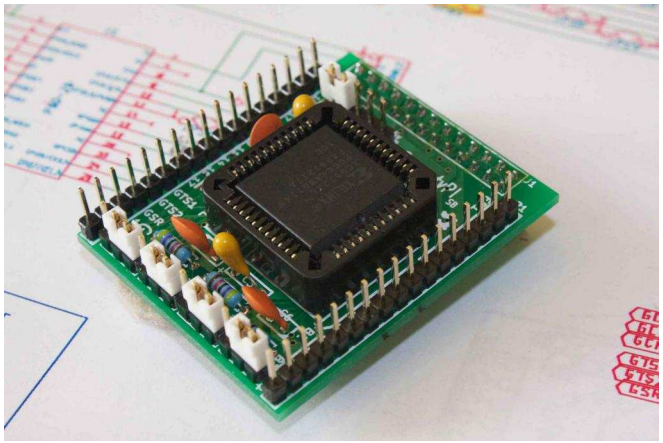
Make a low cost GPIO expander

DIFFICULTY : ADVANCED

Derek Campbell

Guest Writer

Do you want to connect electronics to your Raspberry Pi but are worried about overloading or damaging the GPIO (General Purpose Input Output) pins? Say "Hello!" to the GuzuntyPI-SB.



Guzunty is an Open Source, Open Hardware addon you can easily build yourself. This tiny little board will protect your Raspberry Pi while you learn, but it can do so much more.

The Guzunty is programmable using 'cores'. These files can handle repetitive tasks that slow down the Raspberry Pi's CPU and they can provide access to 25 more input/output pins than on the GPIO alone. Take it to the next level and learn to program a core to do new tasks yourself.

Protecting your Raspberry Pi

As standard, no Raspberry Pi signals are directly brought out to the pins on the top side of the Guzunty, so it is almost impossible to feed damaging voltages into your Raspberry Pi. Instead, all signals pass though the chip at the heart of this design, a Xilinx XC9572XL.

The XC9572XL is a Complex Programmable Logic Device or CPLD for short. Don't worry about the word 'Complex' in the name, the Guzunty makes it easy. The CPLD is tolerant of 5 volt signals which would damage your Raspberry Pi, so you can interface with more kinds of external devices, including Arduino.

NOTE: The CPLD will be damaged by voltages greater than 5.5 volts. You should be especially aware that mains voltages are dangerous and need special isolation circuitry. However, if you should make a mistake and fry the CPLD, a replacement chip can be purchased inexpensively and swapped into the socket.

Input / Output expansion

The CPLD is programmable, but unlike a computer program, a CPLD program defines hardware logic. To provide I/O (Input / Output) expansion, we program the CPLD to provide a Serial Peripheral Interface (SPI) on four of its general purpose I/O pins.
The System On Chip (SOC) of the Raspberry Pi has SPI, on GPIOs 8, 9, 10 and 11. We can thus easily make the Raspberry Pi send data to the

CPLD in any programming language that can open the SPI device. The CPLD then uses the data we send it to turn its output pins on or off. The SPI interface works in both directions, so we can also read from the CPLD whether its input pins are in a high or low state. The standard Guzunty board has a total of 25 free I/O pins. We need 4 Raspberry Pi GPIO pins talking to the CPLD and, if we need them, there are another 13 unused pins on the Raspberry Pi. Guzunty takes your Raspberry Pi from 17 GPIO signals to 40 I/O's (25 plus 13). Not bad!

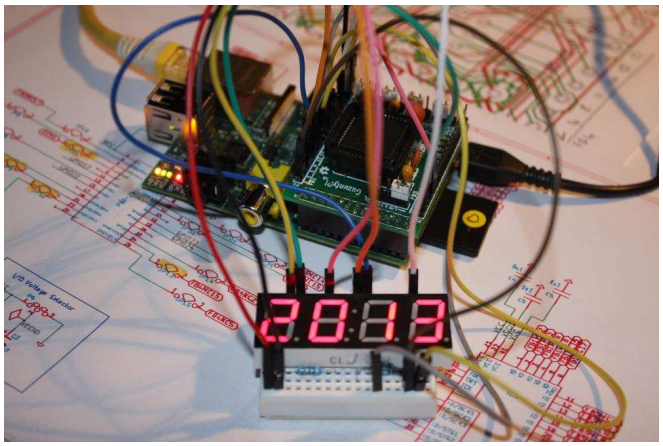
Doing repetitive tasks

These tasks will be simple jobs where we need to turn logic signals on and off quickly and repetitively as with Pulse Width Modulation (PWM) for example. With PWM, we can control the brightness of a LED or the position of a servo by controlling the length of time a logic output is held high. Another example is driving a LED or LCD display. The display looks clear and stable to the human eye, but in reality, the different display elements are being switched on and off hundreds of times a second. These tasks can put a considerable load on the CPU of the Raspberry Pi, but a CPLD can easily handle them leaving your Raspberry Pi to get on with the real work.

NOTE: Handling an independent task like this does require one extra pin on both the CPLD and on the Raspberry Pi to provide a clock signal so that the CPLD can keep track of time.

Design your own core

To make it as easy as possible to get started, the Guzunty website already contains a suite of cores to do a variety of useful tasks when plugged into your Raspberry Pi. However, you don't have to stop there. You can create your own cores using a free tool chain downloadable from the Xilinx website. You program cores using either schematics or a Hardware Description Language. How you do that is beyond the scope of this article, but in the meantime check out the cores already available at: https://github.com/Guzunty/Pi/wiki/Available-cores.

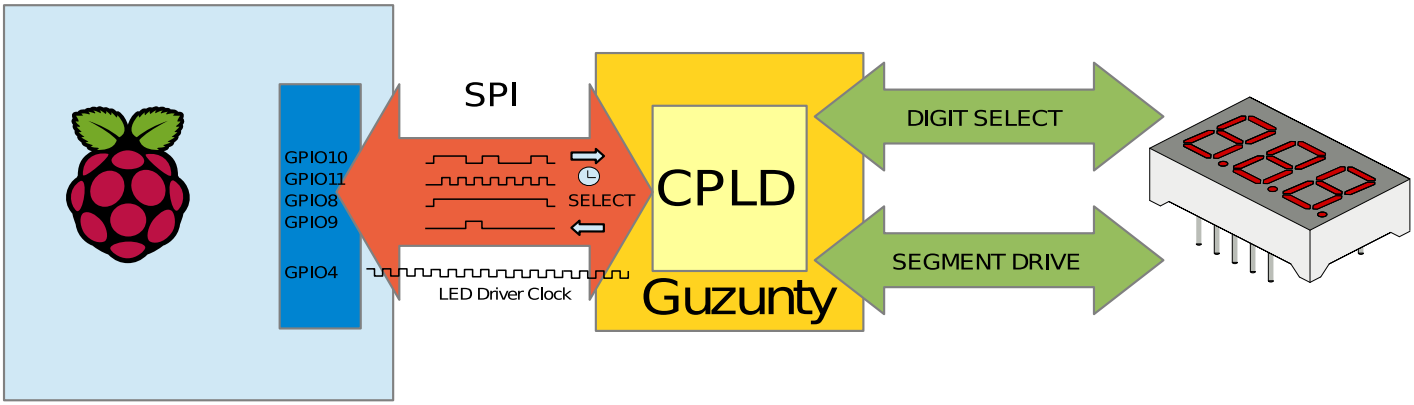


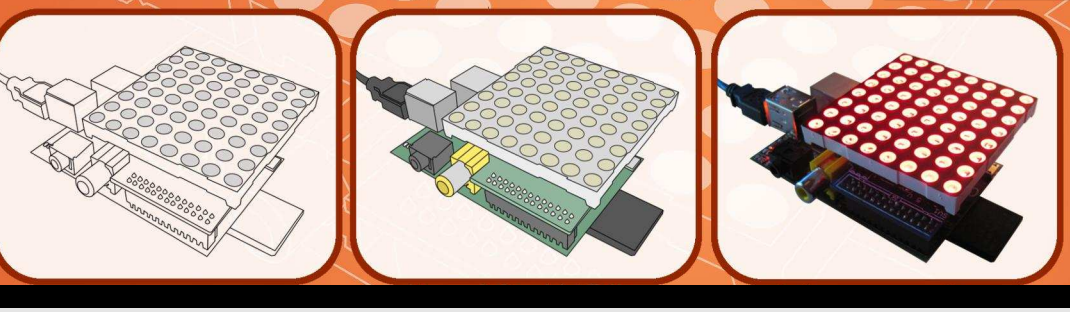
Can Guzunty power my projects?

Yes, the Guzunty provides ground, 3.3v and 5v outputs. These are taken from the power pins on the Raspberry Pi GPIO header. You must adhere to the Raspberry Pi's overall power drain limits.
For more details, see: https://github.com/guzunty/pi/wiki/Frequently-asked-questions.

That's all Folks!

I do hope you decide to build your next project with a Guzunty. I'm sure you will be glad you did. Please visit https://github.com/Guzunty/Pi/wiki to find example cores and the code to drive them.





Part 2: Control individual LEDs & give the Pi Matrix a workout

Bruce E. Hall
W8BH
Guest Writer

DIFFICULTY : MODERATE

Introduction

In Part 1 of this series we looked at how to build the Pi Matrix kit and how to configure I²C on the Raspberry Pi to communicate with it. The result was a simple Python script to turn all the LEDs on and off. In this month's tutorial we will follow on from Part 1 and do something more interesting. First we will look at how the Pi Matrix works, then we will write code for something more YouTube friendly!

How does it work?

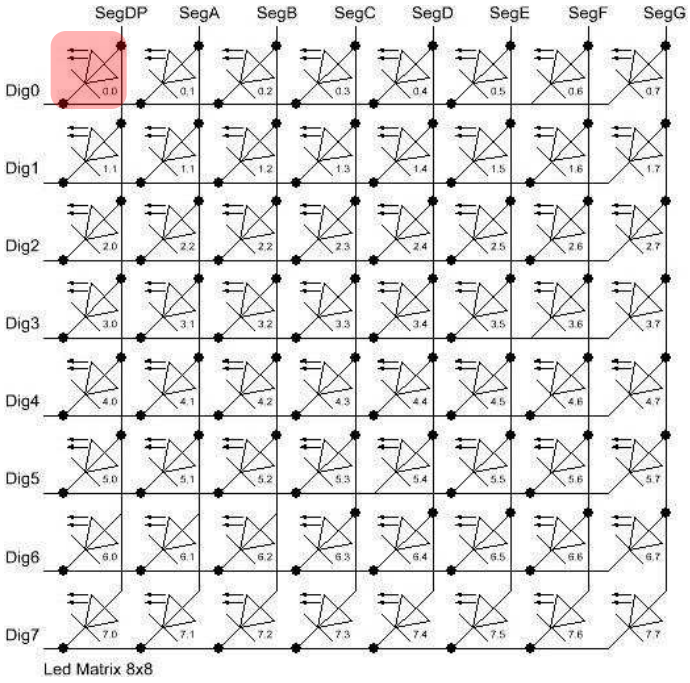
The heart of the Pi Matrix is the MCP23017 chip from Microchip. This is a wonderful device for creating hardware interfaces because:

- It 'speaks' I²C and only needs two communication lines from the Raspberry Pi
- It buffers and protects your Pi from the outside world
- It gives you 16 I/O data lines to work with
- It is dirt cheap! Less than US$1.50 at unit quantities

If you do something 'bad' with your connection to the outside world and fry the 23017, you've lost less than US$1.50 and still have a working Raspberry Pi.

It just so happens that the LED matrix requires 16 data lines - a perfect match. But how can you control 64 LEDs with only 16 lines? Don't you need 64 lines?

That's where the 'matrix' comes in. Take a look at a typical wiring diagram for these devices:



Led Matrix 8x8

The LEDs are wired in an 8x8 matrix with 8 rows (labeled 'Dig') and 8 columns (labeled 'Seg'). These 8 rows and 8 columns make up our 16 connections to the outside world. To turn on an LED, apply power and ground to the corresponding column and row connections. For example, to turn on the highlighted LED at position (0,0) apply power to the anode on left-most column 'SegDP' and ground to the cathode on the top-most row 'Dig0'. Notice that each LED shares a row connection with 7 other LEDs and a column connection with 7 other LEDs.

Let's see how our program from part 1 works. To light up all of the LEDs we send the hexadecimal value 0xFF to Port A of the 23017, which goes to the column pins of the matrix. Hexadecimal 0xFF is the same as binary 11111111. Each bit and therefore each column input will be at logic 1 (+3.3V). We also must send the value of 0x00 to Port B, which goes to the row pins. Hexadecimal 0x00 is the same as binary value 00000000. Each bit and therefore each row input will be at logic 0 (ground). Because all the columns (anodes) are high and all the rows (cathodes) are low, all the LEDs turn on.

Coding for columns

We can make more interesting displays by changing the row and column inputs. For instance, instead of making all of the column inputs high, if we only make one column high then only the LEDs in that column will turn on. The others will stay off.

To try this we will use the Python interactive mode. From the command line, enter:

```
python
```

The Python prompt is a tripple chevron '>>>'. From here we enter our Python code and the interpreter works as we go. Enter the 11 commands from the two code blocks on page 12 issue 13:

```
>>> import smbus
...
>>> bus = smbus.SMBus(1) #0 for Rev.1 Pi
...
>>> bus.write_byte_data(addr,portA,0xFF)
```

All LEDs should now be on. Now let's turn on a single column. To turn on the first column we just need to turn on the lowest bit. This is easy, just write the value of 00000001 (0x01) to Port A. Enter:

```
>>> bus.write_byte_data(addr,portA,0x01)
```

If we want to turn on the third column we write the binary value 00000100 (0x04) to Port A:

```
>>> bus.write_byte_data(addr,portA,0x04)
```

If we start numbering our columns at zero we can determine the value mathematically as 2^n, where n is the column number. So column 0 is 1, column 1 is 2, column 2 is 4, column 3 is 8 and so on. To code this value we use a neat trick, the left-shift operator '<<'. Enter the following and replace 'col' with any number between 0 and 7:

```
>>> bus.write_byte_data(addr,portA,
□□□□□0x01<<col)
```

Did what you expect happen? Depending on your orientation the result might have been the opposite of what you expected. This is because the column pins for the LED matrix are wired in the opposite direction to the Port A pins (see the table on page 13 in issue 13). To correct for this we'll use the right-shift operator instead of the left-shift operator:

```
>>> bus.write_byte_data(addr,portA,
□□□□□0x80>>col)
```

When you are ready to move on, turn on all the column LEDs with the following command:

```
>>> bus.write_byte_data(addr,portA,0xFF)
```

Coding for rows

Now let's do the same thing for rows. We can use the same technique but electrically we need to selectively bring the row to logic 0, not logic 1. For example, to turn on row 3 we'll need to send the binary value 11111011 to Port B. The third bit from the right is logic 0 and everything else is logic 1. Notice that all of the bits are flipped, or inverted, compared to what we used in the column method. We can write the binary value 11111011 as the inverse of 00000100.

In Python the inverse (or bit wise NOT) operator is the tilde '~'. Enter the following and replace 'row' with any number between 0 and 7:

```
>>> bus.write_byte_data(addr,portB,
□□□□□~(0x01<<row))
```

Coding for individual LEDs

It's easier than you think. All we have to do is

combine the code for selecting the row and selecting the column!

```
>>> bus.write_byte_data(addr,portA,
0x80>>col)
>>> bus.write_byte_data(addr,portB,
~(0x01<<row))
```

Finally, to turn off all LEDs enter:

```
>>> bus.write_byte_data(addr,portA, 0x00)
```

Demo code

We now have enough material to create a demo program. Each type of demo has been grouped into its own routine. Try typing this code using your favorite editor and save it as matrix-part2.py. Alternatively you can copy it from http://w8bh.net/pi/matrix-part2.py. You can run it from the command line with the following:

```
chmod +x matrix-part2.py
./matrix-part2.py
```

In Part 3 of this series we will create some intricate matrix displays. Have fun!

```
#!/usr/bin/python
# matrix-part2.py
# Selectively turn on and off each row, column, and pixel

import smbus                    #gives us a connection to the I2C bus
import time                     #for timing delays

#Definitions for the MCP23017 chip
ADDR   = 0x20                   #The I2C address of our chip
DIRA   = 0x00                   #PortA I/O direction, by pin. 0=output, 1=input
DIRB   = 0x01                   #PortB I/O direction, by pin. 0=output, 1=input
PORTA  = 0x12                   #Register address for PortA
PORTB  = 0x13                   #Register address for PortB
OUTPUT = 0
INPUT  = 1

# Lower Level LED Display routines
def Init23017 ():
      #Set up the 23017 for 16 output pins
      bus.write_byte_data(ADDR,DIRA,0x00);  #all zeros = all outputs on PortA
      bus.write_byte_data(ADDR,DIRB,0x00);  #all zeros = all outputs on PortB

def TurnOffLEDS ():
      bus.write_byte_data(ADDR,PORTA,0x00)  #set all columns low
      bus.write_byte_data(ADDR,PORTB,0x00)  #set all rows low

def TurnOnLEDS ():
      bus.write_byte_data(ADDR,PORTA,0xFF)  #set all columns low
      bus.write_byte_data(ADDR,PORTB,0x00)  #set all rows low

def SetLED (row,col):
      #Turn on an individual LED at (row,col). All other LEDS off.
      bus.write_byte_data(ADDR,PORTA,0x80>>col)
      bus.write_byte_data(ADDR,PORTB,~(1<<row))

def SetColumn (col):
      #Turn on all LEDs in the specified column. Expects input of 0-7
      bus.write_byte_data(ADDR,PORTB,0x00)
      bus.write_byte_data(ADDR,PORTA,0x80>>col)

def SetRow (row):
      #Turn on all LEDs in the specified row. Expects input of 0-7
      bus.write_byte_data(ADDR,PORTA,0xFF)
      bus.write_byte_data(ADDR,PORTB,~(1<<row))

def Pause():
      #Turn off LEDS and wait a while between cases
      TurnOffLEDS();
      time.sleep(0.5);
```

```
def FlashLEDS (delay):
      #Flash all of the LEDS on/off for the specified time
      TurnOnLEDS()
      time.sleep(delay)
      TurnOffLEDS()
      time.sleep(delay)

def LEDtest (delay):
      #Turn on all 64 LEDS for the specified time delay, in seconds
      print "\nLighting all LEDS for %d seconds" % delay
      TurnOnLEDS()                  #turn them all on
      time.sleep(delay)             #wait a while
      Pause()                       #then turn them off

def FlashTest (numCycles,delay):
      print "\nFlash all of the LEDS"
      for count in range(0,numCycles):
            FlashLEDS(delay)
      Pause()

def ColumnTest (numCycles):
      #Turn on each Column.
      #Keep PortB (rows) low & set each bit in PortA (columns) high
      #This will actually light LEDS in reverse order, col 7 to col0,
      #because PortA bit 0 is wired to Col7, A1 to Col6,..., A7 to Col0
      print "\nTurn on Columns 0 to 7"
      for count in range(0,numCycles):
            print "...cycle",count+1
            for col in range(0,8):
                  SetColumn(col)
                  time.sleep(0.5)
            time.sleep(1)
      Pause()

def RowTest (numCycles):
      #Turn on each row, from row 0 to row 7
      #Keep PortA (columns) high & selectively turn a bit in PortB (rows) low
      print "\nTurn on Rows 0 to 7"
      for count in range(0,numCycles):
            print "...cycle",count+1
            for row in range(0,8):
                  SetRow(row)
                  time.sleep(0.5)
            time.sleep(1)
      Pause()

def PixelTest (numCycles):
      #Flash each LED according to its (row,col) coordinate
      print "\nFlash each LED in (row,col) order"
      for count in range(0,numCycles):
            for row in range(0,8):
                  for col in range(0,8):
                        SetLED(row,col)
                        time.sleep(0.1)
      Pause()

# Here is the program body: call each of the text routines in turn.

print "\nPi Matrix test program starting"
bus = smbus.SMBus(1);               #Use '1' for newer Pi boards; 0 for oldies
Init23017()                         #Set all 16 I/O pins as output
LEDtest(5)                          #Turn on all LEDS to verify connectivity
FlashTest(25,0.1)                   #Flash them all for a little fun
ColumnTest(3)                       #Sequentially turn on Columns 0 to 7
RowTest(3)                          #Sequentially turn on Rows 0 to 7
PixelTest(1)                        #Flash each pixel in row,col order
FlashLEDS(0.1)                      #Visual end of test
print "\nDone."
```

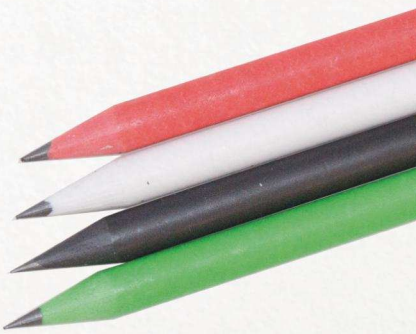
The Raspberry Pi is helping millions of kids write their first

print 'hello world'

We'd like to thank you for
making cool projects,
spreading the word,

...and also for buying sweet, sweet swag

Your generous support helps us do more*
http://swag.raspberrypi.org



AUTHENTIC SWAG EDUCATING SINCE '12

*Hello World is alright, but we've got to teach kids "20 GOTO 10" as well

The MagPi What's On Guide

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

Then this 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

Barnsley Hack-a-thon

When: Saturday 6th July 2013 @ 9.00am
Where: Digital Media Center, Bansley

This event will run from 9.00am until 9.00pm. The event is free. Further information is available at http://makedo.in/hackday

Liverpool Raspberry Jam

When: Saturday 6th July 2013 @ 9.30am
Where: North Liverpool Academy, 120 Heyworth St, Liverpool, L5 0SQ

The event will run from 9.30am until 4.00pm. Further information and free tickets are available at http://raspberryjam.org.uk/event/liverpool-raspberry-jam-rjam-rjamlpl-saturday-6th-july-2013/

Lima Raspberry Pi Meet-up

When: Saturday 13th July 2013 @ 11.00am
Where: Lima Public Library, 650 W Market St, Lima, OH

Attendees will receive an LRITA.org flash drive and a pizza lunch.
Register at http://www.lrita.org/events/bit-talk/july-13,-2013-raspberry-pi-meet-up.aspx

Singapore Raspberry Pi Training

When: Saturday 13th July @ 2.00pm
Where: Singapore Science Centre, Digital Design Studio, Singapore 609081

Runs from 2.00-4.30pm. Free for Science Centre members, S$10 for others. Further information:
http://www.itsc.org.sg/index.php?option=com_eventbooking&task=view_event&event_id=74&Itemid=41

What are the ingredients for a fun filled family Pi day?

Dr Mike Bartley & Caroline Higgins
Guest Writers

Raspberry Pi Boot Camp events are now being held on a regular basis in the south west of the UK. The MagPi asked Mike Bartley, one of the organisers of the Bristol event, to tell us all about the events and how they came about, hopefully to inspire similar events elsewhere.

In December 2012 the Bristol branch of BCS The Chartered Institute for IT held its annual talk to encourage children into computing. The venue was At-Bristol and the theme was the Raspberry Pi. With an inspirational talk from Rob Bishop of the Raspberry Pi Foundation, the event attracted over 450 attendees with at least 300 being children. Since the UK government proposed the planned updates to the IT curriculum last year there has been a growing interest in the subject from children, parents and teachers. It was clear we had hit a topic of high interest and we needed to understand how best to take it forward.

Both the University of Bristol and At-Bristol have outreach programs to encourage children into computing and engineering. A collaboration between the three organisations seemed like a perfect way to collectively achieve our goals; to create a series of events which support the learning of the next generation of innovators and engineers, whilst providing a fun day out for the family.

At-Bristol provides the facilities and logistics; the University of Bristol's students run a drop in clinic and workshops as well as loaning equipment. The BCS put a call out to their members to find volunteers who run workshops and demonstrations. All three made financial contributions of some sort. The voluntary "let's do this" attitude pervaded the organisation and the event and enabled the three organisations to work together to pull off the first two events, which proved to be hugely successful.

Over 400 children and 200 adults attended the first two Raspberry Pi Boot Camps to learn what they can do with the Pi. From controlling remote control cars and LEDs to setting up a home media server – the children got to grips with how they can use the Raspberry Pi to make things!

Marcus aged 12 commented "This was fantastic. My Pi has been at home in its box for a while but I've now learned loads of cool things to do with it. Me and my mates are looking forward to the next one already!" The events will now become a regular feature at At-Bristol on a Saturday in September, November, January, March and May.

The workshops ranged from beginner to advance level, and also included a show-and-tell section that gave visitors the opportunity to learn more about expert level projects. Pi-Cars proved to be very popular workshop; children had the opportunity to build remote control handset for a car using their Raspberry Pi. Other workshops included the Pi as a home media server, led traffic lights and how to log data to the internet. The Magpi team were on hand to answer questions. The show-and-tell section included a number of robots controlled by a Raspberry Pi and a drawing robot.

Obviously organising a large scale event of this nature throws up many challenges, finding enough equipment has been the main issue. We asked people to bring their own monitors, keyboards and power cables, which of course isn't always possible. We encouraged visitors bring their laptops and there were volunteers on hand to show them how to set up their laptop to control their Pi.

Future Boot Camps will continue the theme of workshops and demonstrations on the Pi and other platforms like Arduino. However, other aspects of computing will also be covered. For example, the second Boot Camp on June 15th featured a 'Rails Girls' workshop, hosted by local web agency Simpleweb. Female programmers were invited to spend the day learning Ruby on Rails. In addition to this we ran three 90 minute workshops which aimed to introduce children to application writing using the language Processing. The basics of variables, conditionals, loops and functions with the aim of developing a game by the end of the workshop. This will help the children prepare for a free one week programming course to be held at the University of Bristol as part of a national competition organised by the Young Rewired State.

Going forward the three organisations will continue to work together as all objectives remain completely aligned. It is clear that collaboration like this has been extremely beneficial in enabling us to achieve our goals. Events will be held over five Saturdays each year and there will be themes (such as Wireless, Robotics, and smart homes) to keep the events fresh. The first two events relied heavily on the goodwill of volunteers. Now that we have organised two successful boot camps we need to ensure the event is sustainable by asking for donations and sponsorship. We would like to buy dedicated equipment and cover the costs incurred by volunteers. If you would like to contribute to future events by providing a workshop, being part of the show-and-tell section or by making a donation, please contact caroline.higgins@bristol.ac.uk

Dr. Mike Bartley is Chair of BCS The Chartered Institute for IT in Bristol and of the High Tech Sector Group of the West of England Local Enterprise Partnerships.

Caroline Higgins is Outreach & Student Liaison Manager, Faculty of Engineering, University of Bristol.

An Introduction to CHARM

Programming in Charm part 3

DIFFICULTY : ADVANCED

Peter Nowosad

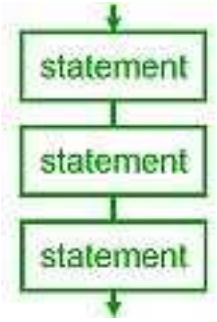
Guest Writer

Having covered RISCOS on the Raspberry Pi and Charm data types in the first two articles, it's my pleasure to welcome you to the third article in the series on the Charm programming language. If you've been following the series, you may well have already installed Charm under RISCOS and tried out the printed exercises. If not it's not too late to go back and do so!

In this article I am going to cover some of the syntax and semantics of Charm illustrated by simple examples which I hope will whet your appetite and arm you (no pun intended!) with enough knowledge to start writing your own Charm programs.

Charm is an object oriented structured programming language. The structured program theorem, which provides the theoretical basis of structured programming, states that three ways of combining programs namely sequencing, selection, and iteration are sufficient to express any computable function.

Sequence

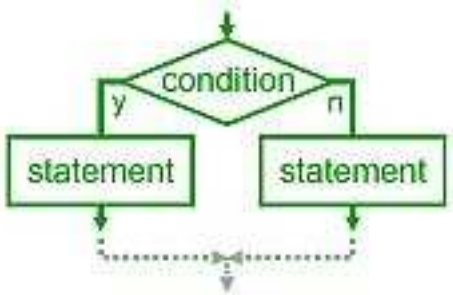


In the absence of flow of control statements related to selection and iteration, Charm programs execute a line at a time from top to bottom e.g.

```
int x := 1;
int y := 2;
int z := x + y;
do_this ();
do_that ();
```

however even in this simple example `do_this` and `do_that` cause call outs to named procedures that contain their own code (not shown here). Although white space is ignored, usually each line contains a single statement which ends with a semicolon, with one common statement type being the assignment statement in which the left hand side is assigned the value of the expression on the right hand side with the two being separated by the assignment operation `:=`.

Selection



Often programs want to take different courses of action under different conditions and this is handled by selection, which in Charm is provided by the `if` and `case` keywords.

`if` checks whether the following conditional expression evaluates to `true` in which case the next statement, or block of statements enclosed in curly brackets is executed, or `false` in which case it is not.

The `if` keyword has an optional associated `else` keyword that if present introduces the statement or statements that should be executed if the boolean expression is not `true` e.g.

```
if x = 1
    y := 0;
else
    y := 1;
```

Note the use of indentation to help highlight the flow of control logic.

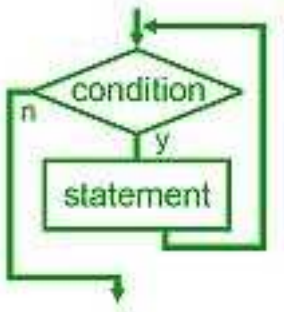
The `case` keyword introduces a series of clauses that match the value of an integer variable against a number of possible constant values and chooses a block of code to execute e.g.

```
    case x
    {
1:      handle_1 ();
2:      handle_2 ();
otherwise:
        handle_other (x);
    }
```

Note the use of the `otherwise` clause to catch any values that do not match.

Iteration

Sometimes the same code needs to be executed many times within a loop, often to process each member of an array. This is handled by iteration using the `while`, `repeat` and `for` keywords.



`while` checks whether the following conditional expression evaluates to true in which case the next statement, or block of statements enclosed in curly brackets are executed until the expression evaluates to false e.g.

```
while true loop ();
```

`repeat` is similar to while but the conditional expression is at the end of the loop and must evaluate to `false` for the loop to be repeated e.g.

```
repeat loop (); false;
```

`for` allows a control variable to be initialised, incremented and tested all in one statement e.g.

```
for (int i := 0 step i := i + 1 while i < 10) a[i] := 0;
```

initialises the contents of an integer array of size 10 to 0.

Boolean expressions

Boolean expression appear as part of many flow of control statements in Charm and can be composed of the following comparison operators:

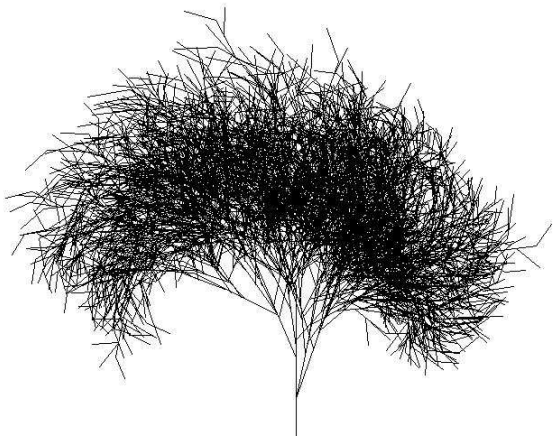
= - equal
- not equal
:=: - equal (pointer comparison)
:#: - not equal (pointer comparison)
> - greater than
< - less than
>= - greater than or equal to
<= - less than or equal to

If A and B are boolean expressions, they can be combined using the logical operators:

A and B - both A and B must be true
A or B - either A or B must be true
not A - A must be false

nil

The `nil` keyword occupies a special place in the Charm language and can be used to initialise and test pointers (ref variables). It indicates that the pointer is not currently setup to point anywhere, and an attempt to use it will result in an exception that stops program execution.



Tree recursion

The following program uses tree recursion and the language constructs to draw a tree on the screen

```
import      lib.Maths;
import      lib.Vdu;

module Tree
{
   const
      MODE       = 31,            | display mode          |
      OFFSET     = 30,              | display offset      |
      LEVELS     = 7,             | branch levels         |
      VARS       = 1000;          | variations            |

   int xmax, ymax, xshift, yshift; | screen parameters     |

   proc variation (real value, real deviation) real
   {
      return value * (1 + deviation * (0.5 - 1.0 * (Maths.random () mod (VARS + 1)) / VARS));
   }

   proc branch (int level, real length, real angle, int xs, int ys)
   {
      int xe := xs + length * Maths.fn.cos(angle);
      int ye := ys + length * Maths.fn.sin(angle);
      Vdu.plot (Vdu.P_POINT + Vdu.P_PLOT_F_ABS, xs l_sl xshift, ys l_sl yshift);
      Vdu.plot (Vdu.P_PLOT_F_ABS, xe l_sl xshift, ye l_sl yshift);
      if level > 1 + Maths.random () mod 2
      {
         real scale := variation (0.65, 0.25);
         int branches := level - Maths.random () mod 3;
         if branches <= 0 return;
         real range := variation (1.5, 0.2);
         for int i := 0 step inc (i) while i <= branches
         {
            real factor := variation (0.5, 0.8);
            if i = branches factor := 0;
            branch (level - 1,
                    variation (length * scale, 0.2),
                    variation (angle, 0.3) + i * range / branches - range / 2,
                      xe - (xe - xs) * factor, ye - (ye - ys) * factor);
          }
      }
   }
```

```
   export proc ~start ()
   {
      Vdu.mode (MODE);
      xshift := Vdu.mode_var (MODE, Vdu.MV_X_EIG_FACTOR);
      yshift := Vdu.mode_var (MODE, Vdu.MV_Y_EIG_FACTOR);
      xmax := Vdu.mode_var (MODE, Vdu.MV_X_WINDOW_LIMIT) + 1;
      ymax := Vdu.mode_var (MODE, Vdu.MV_Y_WINDOW_LIMIT) + 1;
      Vdu.g_colour (0, 7);
      Vdu.plot (Vdu.P_POINT + Vdu.P_PLOT_F_ABS, OFFSET l_sl xshift, OFFSET l_sl yshift);
      Vdu.plot (Vdu.P_RECTANGLE_FILL + Vdu.P_PLOT_F_ABS,
        (xmax - OFFSET - 1) l_sl xshift,
        (ymax - OFFSET - 1) l_sl yshift);
      Vdu.g_colour (0, 0);
      branch (LEVELS, (ymax - OFFSET) / 3, Maths.fn.asn(1), (xmax - OFFSET) / 2, OFFSET);
   }
}
```

To make the tree look more realistic some branch parameters have been randomised, for example their length and the angle between them. To compile this program, Charm version 2.6.5 or newer is needed.

For this month's exercise add some fruit (hint randomly draw some coloured circles at or near branch ends).

Next Time

Next time I intend to talk about writing containers in Charm, namely sets, lists and maps.

A Pi and a cup of Java, please!

DIFFICULTY : MEDIUM

Vladimir Alarcón & Nathaniel Monson
Guest Writers

You will need:

- **A Raspberry Pi with Raspbian.**
- **150 MB of free space in your SD card.**
- **Basic knowledge of programming.**
- **Basic command-line usage.**

Introduction

In this article I'll show you how to write and run Java programs on your Raspberry Pi.

Java is an object-oriented language designed to run on many operating systems without requiring recompilation of the source code. Java also includes a vast amount of libraries, which offer solutions to more advanced problems like running complex web sites or high-end mission-critical algorithms. In this article I will focus on the very basics of the language. Once you master the language there are plenty of web sites on the Internet with lots of details of libraries and many examples.

I'll first show you the steps to install the Java compiler and virtual machine on the Raspberry Pi. Then, we'll create a couple of basic Java programs... and we'll run them!

This article shows you the running examples first, and delves into concepts later. The idea behind this approach is that it will be easier for you to look at real Java programs and try to identify the new elements and their functions by yourself. Once you have written the program and run it, I will explain the novelties.

1. Installation

To write, compile and run your program, you'll need two things: a text editor, and a Java development kit (JDK). You can use any text editor to write a Java program. I prefer `Geany` because of its syntax colouring, but `Leafpad` or `GEdit` will work too. While there are several JDKs, I suggest OpenJDK 7. A JDK includes (mainly) a compiler and a Java virtual machine (JVM). The compiler generates platform independent bytecode and the JVM is able to run this bytecode.

To install `Geany` and OpenJDK 7 open a terminal window and type:

```
sudo apt-get install -y openjdk-7-jdk geany
```

It will probably take at least nine minutes to download and install everything, but it can take longer depending on the speed of your internet connection. Once the installation has finished, check both packages are installed correctly. You should be able to open `Geany` from the main menu, under "Programming". To test OpenJDK7, open a terminal window and type:

```
java -version
```

It should display a few lines starting with:

```
java version "1.7...
OpenJDK ...
```

2. Running our first program

Create a directory to store our programs. For example, open a terminal and type:

```
mkdir cupofjava
```

Now it is time to write our first program (a class) called "HiThere". Java is an object oriented programming langauge, where every program contains at least one class which in turn can use other classes.

Open the text editor (Geany in my case) and create a file called HiThere.java. Type in,

```
public class HiThere {

  public static void main(String[] args) {
    System.out.println("A Java Pi!");
  }

}
```

and save the file. Using the terminal, change to the directory "cupofjava" (where you created the program) using:

```
cd cupofjava
```

and then type:

```
javac HiThere.java
```

The javac command compiles the .java file into a .class file. The compiler analyses the source code in the HiThere.java file and generates the bitecode form. After 15 seconds or so, this command should silently finish. If you misspelled something it will show an error message that will display where the problem is. If this is the case, go back to the text editor, check your code and fix the misspelling, save the file, and then run javac again. Once the file has been successfully compiled, you'll find a new file in the same directory called HiThere.class. This is the compiled program that will be run.

To run the program type:

```
java HiThere
```

You don't need to specify the .class extension. The command javac compiles programs, and the command java executes them.

The program will run and will display:

```
A Java Pi!
```

Well... Congratulations! You have written and run your first Java program on the Raspberry Pi.

You probably noticed the program took a few seconds to write that message. Why so slow? Actually, Java is quite fast. The whole program took only a couple of milliseconds to run, but Java needs a few seconds at the beginning to load the JVM. The good news is that once the JVM is loaded, the program runs very fast.

Well, let's now look at the program in more detail. There's only one line in it that is actually executed. This line is:

```
System.out.println("A Java Pi!");
```

The other lines specify the name of the class "HiThere" (at line 1), and the name of the method main (at line 3). This class, similar to any other Java class, can have many methods but we are using only a single one in this example.

Challenge #1: Your turn now. Using the text editor, change the message in between double quotes in the source file from "A Java Pi!" to "My name is Name." (use your name) and save it. With the terminal, compile the program again and run it using the two commands (javac and java) shown before. If you do it right, the program will display your name now. Do it!

A note: The syntax of Java (the words and punctuation of the language) is very similar to the syntax of the C language. Any programmer with

knowledge of C will find the basics of Java very easy to understand.

3. Java variables and Control flow

The next example illustrates the use of variables and control flow statements. In the same directory where we stored the first program, create a new file called `DiceRoller.java`. Then append,

```
import java.util.Random;

public class DiceRoller {

  public static void main(String[] args) {
    Random generator = new Random();
    int d = 0;

    while (d < 4) {
      System.out.print("Rolling... ");
      int face = 1 + generator.nextInt(6);
      System.out.print("I got a "+ face
        + ".  ");
      if (face == 1) {
        System.out.print("Wow! An ACE!");
      }
      System.out.println();
      d = d + 1;
    }
  }

}
```

to the file and save it. Compile and then run the program.

```
javac DiceRoller.java
java DiceRoller
```

You'll see something similar to:

```
Rolling... I got a 2.
Rolling... I got a 1.  Wow! An ACE!
Rolling... I got a 4.
Rolling... I got a 5.
```

The program will roll four dice and will identify which ones are aces (the number 1). Do you see how it works?

There are quite a few things of interest in this example. This program uses two integer variables, named `d` and `face`. The variable `d` is used to make sure we roll four times, not three times or not five times. The `face` variable stores the die face after each run. The program also receives an array of strings in the variable named `args`. The `args` variable contains the command-line parameters present when the program is run. Finally, the program also uses an object called `generator`. Remember I told you a Java class can use other classes? This is an example. This program uses an existing class (called `Random`) that specializes in generating random numbers. A class can be used by calling its methods, either directly or by creating an object. In this case we only use one of its methods, the one called `nextInt()`, to get a random number.

Additionally, Java uses the brackets { and } to define groups of instructions, called blocks. Each block can be empty, or have one or more instructions. You can define sub-blocks inside existing blocks as needed. This is commonly used in control-flow statements.

Talking about this,... this example shows the use of two control-flow statements: `if` and `while`. An `if` statement executes a block only if the condition specified in parenthesis is true; a `while`, on the other hand, will execute the nested block multiple times as long as the condition is true. Other control blocks are `for`, do-`while`, `switch` and `if-else`.

In the example, the `if` statement checks the value of `face`. That's why the extra message appears only when an ace is rolled. The `while` statement, on the other hand, executes the included block four times. The `while` loop continues while the value of the variable `d` is less than 4. Notice that the variable is set to zero before the loop. Inside the loop it increases by one at the end of the block. Therefore, the first four times (0, 1, 2, 3) the `while` succeeds, but on the fifth one (when it has the value 4) it fails.

Challenge #2: Change the program to roll 7 dice, where each dice has 10 faces. Once you have saved your changes, go to the compile the program again and run it using the two commands (`javac` and `java`) shown before. If you do it right, the program will now display all seven dice. Go for it!

RasPiConnect

Connect your Raspberry Pi to the World!

Available on the App Store

RasPiConnect connects your Raspberry Pi to the outside world. It allows you to control virtually anything you connect to your Raspberry Pi from your iPad or iPhone.

- EASY to setup - no syncing required
- Buttons, gauges, webpages, webcam pictures and more!
- Build your pages on your iPad/iPhone
- Supports multiple Raspberry Pis
- Five pages of control panels
- Unlimited Controls
- Exchange your panels with friends
- Supports any computer that supports Python (windows, linux, etc.)

MiloCreek
Where Innovation Flows

Parallel calculations - part 3

DIFFICULTY : ADVANCED

W. H. Bell

MagPi Writer

Completing BatchCalculator

Welcome back to the Python Pit. This article is a continuation of the tutorial presented in Issues 10 and 13. If you have not already done so, it would be a good idea to read the previous parallel calculations articles in these Issues before proceeding.

The `FunctionCalculator.py` file was originally written to run a genetic algorithm, using several computers at once. The calculations for each point in the genetic algorithm were evaluated using C++ and ran for around eight minutes on high specification CPU cores. In this case, the time overhead of server and client processes written in Python was negligable. Since the cluster used was often used for other tasks, the implementation of `FunctionCalculator.py` provided a way of using as many cores as possible without requiring a rigid addressing structure.

There remain two steps left in this series of articles: (i) the completion of the `BatchCalculator` class and (ii) the use of `FunctionCalculator.py` to run some more serious calculations. Open the file `FunctionCalculator.py` from Issue 13. Then go down to function `shutdown` in the class `BashCalculator`. After `shutdown`, add the member function `evaluate`:

```
def evaluate(self, cmds):

  # Wait until at least one client thread is available
  while len(self.client_threads) == 0:
    print "Waiting for a client to connect"
    time.sleep(5)

  ncmds = len(cmds)

  # Create a buffer to collect the results
  results = [0.]*ncmds

  # If no commands were given return empty list of results
  if ncmds == 0:
    return results

  # Create a buffer to collect the status of the results and the results
  #   0 => not calculated, 1 => being calculated, 2 => done
  status_results = []
  icmd = 0
  while icmd < ncmds:
    status_results.append(StatusResult(0,0.))
    icmd = icmd + 1
```

```
# Loop until all of the calculations have finished.
finished = False
ithread = 0
icmd = 0
while not finished:
  print "Looping"
  print "Status and results = %s" % status_results
  time.sleep(1)

  # Check if all status flags are 2 and copy the results into the
  # results list at the same time
  jobsLeft = False
  for i in xrange(ncmds):
    sr = status_results[i]
    if sr.cmd_status != 2:
      jobsLeft = True
      break
    results[i] = sr.result

    if not jobsLeft:
      finished = True
      continue

    print "Have jobs to do"

    # Check the number of threads inside the loop in case more
    # threads are created during the loop
    nthreads = len(self.client_threads)

    print "Currently have %d threads to work with" % nthreads

    # If the index points at the last thread go back to the
    # first thread.
    if ithread == nthreads:
      ithread = 0

    print "Using thread index %d" % ithread

    # If the index points at the last cmd go back to the first cmd.
    if icmd == ncmds:
      icmd = 0

    print "Checking cmd index %d" % icmd

    # Check if this cmd has been submitted or not.  If the
    # command has already been submitted skip to the next
    # command.
    if status_results[icmd].cmd_status != 0:
      print "Command %d has status %d" % (icmd, status_results[icmd].cmd_status)
      icmd = icmd + 1
      continue

    print "Searching for an idle thread"

    # Find an idle thread
    foundThread = False
    while not foundThread:

      # Check the number of threads inside the loop in case more
      # threads are created during the loop
      nthreads = len(self.client_threads)

      # Keep looping round and round.
      if ithread == nthreads:
        ithread = 0
```

```
        self.client_threads[ithread].processingCmd().wait(1) # wait until finished or 1 sec.
        if not self.client_threads[ithread].processingCmd().isSet():
          foundThread = True
        else:
          ithread = ithread + 1
          time.sleep(1)

      # If there are no available threads
      if not foundThread:
        print "All threads are busy processing commands."
        time.sleep(2)
        continue

      # Submit the command and the target list element
      print "icmd %d" % icmd
      sr = status_results[icmd]
      self.client_threads[ithread].evaluate(cmds[icmd], sr)

      # Go to the next command
      icmd = icmd + 1

  # For debugging
  print "results = %s" % results

  return results
```

The `evaluate` function takes a list of commands and returns a list of floating point numbers which correspond to each command. The function performs the calculations by passing each command to a client process.

The `evaluate` function waits until a least one client thread has connected. Once one client has connected a `while` loop is used to evaluate each of the commands. The commands are given an associated status code of zero, which is set to one if the command is being evaluated or two if the command has been evaluated. To prevent a tight loop and an associated high use of CPU on the computer running the `BatchCalculator`, a `sleep` statement is used within several of the loops. There are a lot of `print` statements to show how the function works.

If a client process connects to the server while some commands are being evaluated, then it will be added to the pool of available clients. Therefore, the speed of calculation will increase as more clients connect. The `evaluate` function waits for one of the associated threads to become available and passes it a command. When a command has been evaluated the result is stored in the `status_results` list and then copied into the `results` list. The `status_results` list is of type `StatusResult`, which is mutable. Therefore, the `client_threads` function is passed a pointer to the element of the list, can write the results into it, and set the command status.

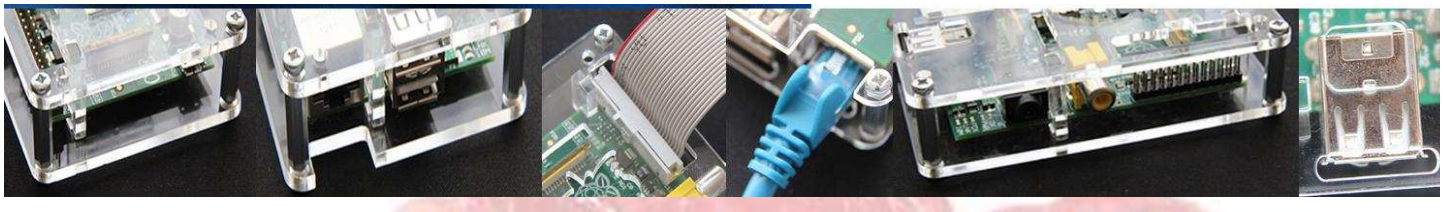
Testing the BatchCalculator

While the `launchBatchClient.py` program from the last tutorial in Issue 13 can be used to start the clients, the server program `launchBatchCalculator.py` needs to be updated to pass a list of commands to the `evaluate` function of the `BatchCalculator`. Look back at the first article in this series and see how the commands were passed to the `FunctionCalculator.py`. Then try to numerically solve,

```
y = 4*x**4 - (x - 4)**3/(6-x)**2 + x
```

for the value of `x` when `y` is 10. Choose 100 values of `x` at random between -1000 and 1000. Pass these equations to the `BatchCalculator`. Pick the best two points and select another 100 values for `x` within the second range. Repeat the procedure until a solution is found. While this calculation will run slower than using a single Raspberry Pi, the problem will demonstrate how to use the `BatchCalculator`. The solution to the problem will be given next time.

JULY COMPETITION



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

This month there is one MASSIVE prize!

The winner will receive a new Raspberry Pi 512MB Model B, an exclusive Whiteberry PCSL case, 1A PSU, HDMI cable, 16GB NOOBS memory card, GPIO Cobbler kit, breadboard and jumper wires!

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

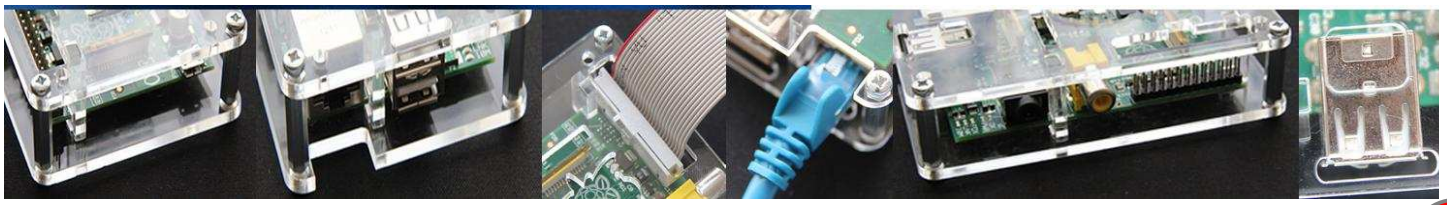
Closing date is 20th July 2013.
Winners will be notified in next month's magazine and by email. Good luck!

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

June's Winners!

The winner of a new 512MB Raspberry Pi Model B plus PCSL PiAngle case and GPIO Cobbler kit is **Surajpal Singh (Bristol, UK)**.
The 2nd and 3rd prize winners of a PCSL PiAngle case are **James Duffell (Preston, UK)** and **Kobitharun Kunasekaran (Scarborough, UK).**

Congratulations. We will be emailing you soon with details of how to claim your prizes!



Feedback & Question Time

Friday June 14, 2013. Escondido, California, USA. I got my magazines today! Hooray Hooray Hooray!

Homer Hazel

Wow! – what an incredibly professional production!

I have downloaded the 13 issues and have ploughed through the first two. As a retired IT teacher, this is just what we need and it is presented in such a way that anyone can get something out of it – young and old. My Pi takes me back to my Acorn Atom days and a soldering iron, and it is good to see young students taking this up again. I am currently working with a Year 7 young lady doing some programming and simple control circuits, and she is loving it – well, we both are, really!

I spend my life writing a complex engineering program in PowerBASIC – if only this was available for Linux (and it is not for the want of asking) – and getting to grips with the syntax of Python has been an interesting challenge. My only feeble complaint is that setting up a printer is a nightmare! I can get it to print from LibreOffice, but not the command prompt, though I am sure that the answer will be in the issues I have not looked at yet.

It is good to see Britain leading the world again in the field of microcomputers – well done Pi.

Iain Johnstone

[Ed: See issue 12 for command line printing]

288 pages of goodness just arrived. The mags look awesome, love the binder and stickers!

Its been a bit of a wait but its definitely been worth it! The whole bundle is fantastic and the magazine quality is perfect. They look and feel like a premium magazine you would buy from a newsagents. You can see right away the hard work that has gone into these and it is very much appreciated.

THANK YOU!! to the whole MagPi team!! Time to crack on and start reading!

Nial Pearce

First of all. GREAT magazine! Thanks for taking the risks to get it out there in the first instance. I bought the issues as soon as I saw I could and they turned up today (YAY!!)

Chris Burgess

Just received my binder and mags - they look great!

Thanks for all the effort you have put into this. And you just know that we all want Volume 2 now don't you :-)

Mark Pearson

[Ed: All Volume 2 issues to date will be available soon]

Just received my magazines and binder today. Wow! They are beautiful.

This whole project has been great. I had selected the Signature Kit and am I ever glad. The hardware kit came some time ago and the whole thing was top quality. It meant a lot to have those books signed by Liz and Eben. Congratulations for a well executed project.

Tony Guerich