Building for RISC OS, Online

and what makes it tick

Gerph, November 2020

Building for RISC OS, Online

and what makes it tick

0. Introduction

November
2020

0. Introduction.

Good evening!

I don't know how this talk will go over. But I hope that it will be interesting, and maybe a little surprising. So, let’s go...

Introduction

How I'll do this talk

- Lots of technology, some of which may be alien to you.
- The talk is split into 5 sections, with a chance for some questions between them.
- Slides will be available at the end, together with some other resources.
- I'll take questions at the end for as long as people want.

Introduction

How I'll do this talk

There are a lot of technologies and features that I'll talk about. I'll call them out as I come to them. Some of what I'm talking about may not be familiar to RISC OS users. They may be much more common to users outside of RISC OS. If there's something you don't understand, and you feel you needs clarification, ask a question in the chat if you can. Someone else might be able to answer, or I may be able to address it immediately.
At the end of each section I'm going to have a short break to answer questions on that section. This should give us a chance to talk about that section's topics without getting too far ahead and people being completely lost.
At the end of the talk, these slides and a bunch of links to resources will be made available.

Introduction

What we'll talk about

1. Some background.
2. What JFPatch-as-a-service is.
3. How it works.
4. What powers it.
5. Conclusions.

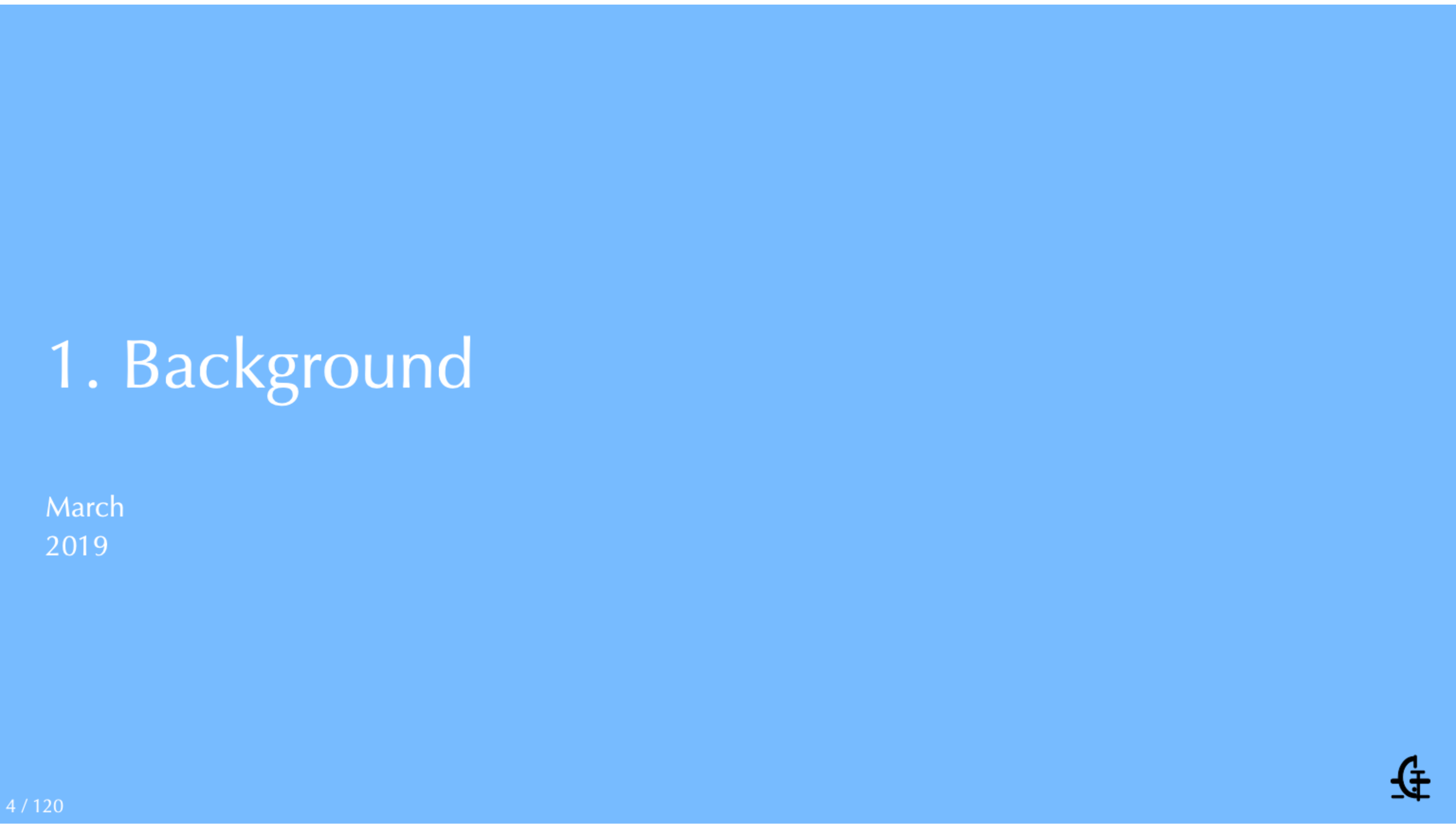
Introduction

What we'll talk about

There are 5 sections to the talk -

- a little bit of background
- a discussion of what JFPatch-as-a-service is and what you can do with it
- a very light dive into how that service works,
- a look at the system that powers everything,
- and finally some conclusions.

At the end of the talk I'll take questions on the subjects that have been covered here, for as long as people want to stick around.



1. Background

March
2019

1. Background

To look at the background we jump back about a year and a half, which will set the scene...

Background

Who am I?

- A RISC OS architect and engineer, who's been away from the community for about 15 years.
- I used to do a lot of things with RISC OS, which you can read about on my site if you're interested - gerph.org/riscos
- I'm not going to talk about that past here.
- I would like to think that I probably know RISC OS in design and execution better than anyone.

Background

Who am I?

You might know me as Justin Fletcher, or gerph. I changed my name a few years ago, and I use gerph where I want some continuity.
I'm a RISC OS architect, and in my day job, I'm a software engineer. I've worked with all of RISC OS from chip initialisation through to the applications and development tools. I'm pretty confident I know it better than most, even after 15 years away. Plus, as I will talk about, I've worked on many of the RISC OS interfaces more recently, which makes them more familiar.
If you want to know more about my past, the RISC OS Rambles on my website talk about it in much more detail.

Background

Dear gosh, why?

- What do I want to do with RISC OS and why?
  - Let's make something for me, because I can.

Background

Dear gosh why?

So why did I start doing RISC OS things again?
Well, about a year and a half ago, I began to understand that I was very unhappy. I didn’t have anything that focused me. I didn’t work on RISC OS things because when I do, I get angry and upset. And then I’m useless for days. So I stayed away.
I’ve tried to deal with that, but things still set me off. Even little things can be a problem for me - I wrote the Rambles to try to be a cathartic release, but it wasn’t completely successful. So I stayed away from the RISC OS world.
I have regular counselling, and I believe that it’s been working - because I realised, about a year and a half ago, that there was something I could do.
I realised that I enjoy working with RISC OS, but I don’t enjoy the RISC OS community. I’m not intending to offend anyone here by that, but that’s what’s debilitating for me. Stopping myself from doing things with RISC OS, because I understand how badly it affects me, has been a way of coping. But I realised that I can do RISC OS things for myself, and not care about whether anyone else sees them. Just doing it because I enjoy it.
Well, wow. This was lightbulb moment for me.
That’s why I’m doing this talk - because I’m really proud of that realisation. I can do things because I enjoy them, not because I feel I have to prove things to people.
That’s what this talk is. It’s my way of saying "this is what, most nights, makes laugh when I go to bed, chuckling ‘this is f’ing nuts’, ‘look how cute that is’.
JFPatch-as-a-service is the bit I've made available to the world, but the rest ... we’ll get to that.

Background

So, you want to use RISC OS, but…

- Development on RISC OS is tedious
  - The tools aren't great but they only run on RISC OS… and I don't have a RISC OS system (other than RPCEmu)
- RISC OS testing is awful
  - Most RISC OS projects do ad-hoc testing, rely on users; no automation
- RISC OS is awful for testing
  - If something goes wrong, you need to hard reboot; no isolation; no security

Background

So you want to use RISC OS but...

So I decided I would dust off some old source code and try to build some things. But there's some problems with that.

- Firstly, RISC OS development tools are a bit poor, and they only run on RISC OS. I don't have a RISC OS system. Not only that, but if you've worked with DDT, the debugger, you'll know how much like russian roulette that is - if you see the insides of it you'll find out why.
- The next issue with RISC OS development is that projects on RISC OS have awful testing in my experience. That includes my own.

Ad-hoc testing is the way to that most things were done and the mantra of 'it's compiled? ship it!' seems to apply. Maybe that's not true for developers these days, but that was how it was. I wanted to be able to address that, if I was to do anything substantial.

- The final issue is that RISC OS itself is not a good system for running tests on. It's cooperative and single tasking. Generally in testing the 'System Under Test' does not cooperate - it will hang, crash, and leak memory like it's going out of style. And on RISC OS that commonly means its time to reboot.

Plus there's no isolation, so even if it doesn't look like the system is broken, it might have corrupted something important. I needed to do something about that.

Background

How does the real world do things?

- Source control !
- Cross compiling
- Managed development environments
- Automated testing of changes
- Feature and regression testing
- Fleets of systems available for use

Background

How does the real world do things?

Ok, so we've understood some problems; let's see how the real world does things.

These are just a few of the things that in the real world we expect to have, and which I'm going to look at. There's:

- Source control
- Compiling code for RISC OS from other systems
- Controlling your development environment
- Automated testing in different ways
- And having fleets of machines to make things faster

Let's look at each of these and see how I addressed them.

Background

How can I do this? (1)

Source control:

- Move things to Git, because CVS is so very painful.

Background

How can I do this? (1)

My own source had been in CVS for many years, and in about 2016 I moved everything - even my own ancient RISC OS projects - into git. I have a large number of python and perl projects managed in git as well. It's so much easier to work with than CVS. And even for that one little script you're just tinkering with, you can create a repository for it.
Git is a source control system that is intended for large scale distributed use, but which works even for a single user working with a single file.
It doesn't need a server, but if you want to share code that's one way you can do it. GitHub is one of the best known systems for storing repositories, which people will almost certainly have heard of. However, at the time GitHub wasn't so friendly to private projects, and even still, I didn't want my projects going off site. So I chose to use GitLab - it's installed on my server, and... it's pretty great. Oh, and it's free.
...

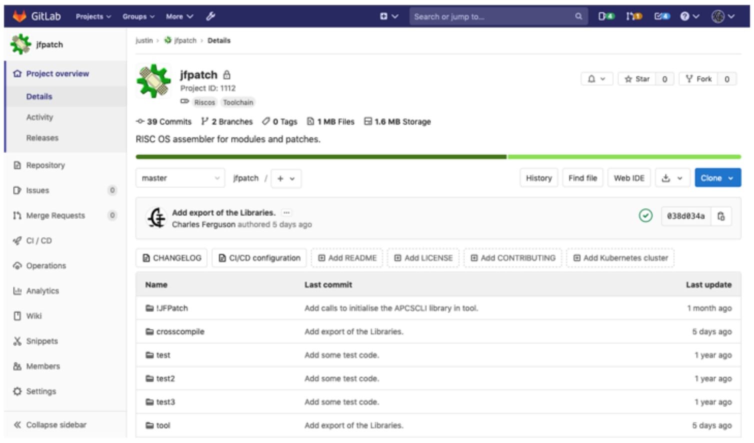
Background

How can I do this? (1)

Source control:

- Move things to Git, because CVS is so very painful.

Tech: GitLab, running on my linux server - it's publicly accessible, but most of the 1000-odd projects are private.



justin > jfpatch > Details

jfpatch
Project ID: 1112
Riscos Toolchain

39 Commits 2 Branches 0 Tags 1 MB Files 1.6 MB Storage

RISC OS assembler for modules and patches.

master jfpatch /

Add export of the Libraries.
Charles Ferguson authored 5 days ago
038d034a

Name	Last commit	Last update
!JFPatch	Add calls to initialise the APCSCLI library in tool.	1 month ago
crosscompile	Add export of the Libraries.	5 days ago
test	Add some test code.	1 year ago
test2	Add some test code.	1 year ago
test3	Add some test code.	1 year ago
tool	Add export of the Libraries.	5 days ago

...
So GitLab is the tech that I use for my source control.
I was amused - but not at all surprised - that ROOL selected GitLab as their server system, a few years ago.

Background

How can I do this? (2)

Cross compiling:

Background

How can I do this? (2)

So I've got a way of storing and managing the source, but I want to at least be able to compile and assemble things if I'm going to work on them.
It's all well and good being able to build RISC OS components for testing on macOS by stubbing libraries, but I want to be able to build things that will run on RISC OS.
...

Background

How can I do this? (2)

Cross compiling:

- Already had the toolchain ported to 32bit Linux and Windows, back in 2005.

Tech: Port the toolchain to 64bit Linux and 64bit macOS.

```
charles@laputa ~/pro/RO/mod/ris/Sou/Des/TaskWindow (master)> rm o*/*; riscos-amu
BUILD32=1 ram
riscos-objasm   -Stamp -quit   -I@ -predefine "BUILD_RAM SETL {TRUE}" -apcs
3/32/fpe2/swst/fp -predefine "BUILD_ZM SETL {TRUE}" -predefine "No26bitCode SETL {TRUE}"
-predefine "No32bitCode SETL {FALSE}" -predefine "APCS SETS \"APCS-32\"" -o oz32/Taskman
s/Taskman
ARM AOF Macro Assembler 3.32 (JRF:3.32.38) [07 Mar 2006]
Unrecognised APCS qualifier /fpe2
Unrecognised APCS qualifier /fp
MyDomain = 0000058C
Deprecated form of PSR field specifier used (use _cxsf)
riscos-link -rmf -rescan -C++ -o rm32/TaskWindow,ffa oz32.Taskman
TaskWindow: Module built {RAM}
```

...

Back in 2005 I had cross-compiling working for RISC OS, and most of the last version of RISC OS that I worked on were actually built on Linux.

However, that was all for 32bit Linux and Windows. So I ported the toolchain to work in 64bit compilers.

It wasn't actually 'simple', and I still see bugs now and then for cases I've missed. The authors loved their bitfields and packing structures into different types. But with a little work, the toolchain was made to compile RISC OS components on macOS.

All the cross-compiling tools have the prefix 'riscos hyphen' to make it easier to distinguish them. So you can see that 'riscos-amu', 'riscos-objasm', and 'riscos-link' were invoked in this example.

I needed to test the C compiler with some nice example code to see that it fared well. I had a lot of my own code, but using random stuff from other people to throw at the compiler highlights many problems you wouldn't otherwise find.

...

Background

How can I do this? (2)

Cross compiling:

- Already had the toolchain ported to 32bit Linux and Windows, back in 2005.

Tech: Port the toolchain to 64bit Linux and 64bit macOS.

Tech: Tool to extract example code from 'Rosetta Code' for testing (https://github.com/gerph/rosettacode)

...

So I wrote a small library that downloaded all the C code from the Rosetta Code website. If you don't know Rosetta Code, it's a website containing solutions for many problems, in many different programming languages.

And having downloaded all the programs, I ran them through the C compiler to see whether it crashed or not.

Background

How can I do this? (3)

Managed environments:

- How do I get my toolchain? find my libraries? store built components?

Background

How can I do this? (3)

So the tools for building exist, and I have headers and libraries that I can use to link against. But I need to get them to my machine.
Traditionally you downloaded and installed a package that gave you the build tools. And then you used them until the next time you needed to upgrade. In modern development, these tools and libraries change rapidly.
And even if they don't, you want more control over what exactly goes into building your latest product.
This is usually managed through a software repository that holds all those components that you might want - your toolchain, libraries, documentation, even full releases. These can be stored in the repository and retrieved in managed ways. This ensure that what you're doing is reproducible, because everything that went into making it is known, and can be used again.
...

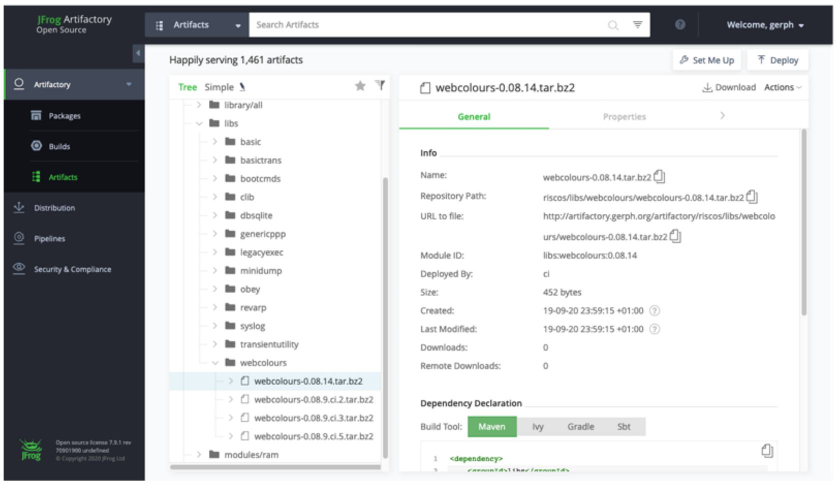
Background

How can I do this? (3)

Managed environments:

- How do I get my toolchain? find my libraries? store built components?

Tech: Artifactory for artifacts, and created some tools for pushing and pulling resources.



Happily serving 1,461 artifacts

```
library/all
libs
  basic
  basictrans
  bootcmds
  clib
  dbsqlite
  genericppp
  legacyexec
  minidump
  obey
  revarp
  syslog
  transientutility
  webcolours
    webcolours-0.08.14.tar.bz2
    webcolours-0.08.9.ci.2.tar.bz2
    webcolours-0.08.9.ci.3.tar.bz2
    webcolours-0.08.9.ci.5.tar.bz2
modules/ram
```

webcolours-0.08.14.tar.bz2

Info

Name:	webcolours-0.08.14.tar.bz2
Repository Path:	riscos/libs/webcolours/webcolours-0.08.14.tar.bz2
URL to file:	http://artifactory.gerph.org/artifactory/riscos/libs/webcolours/webcolours-0.08.14.tar.bz2
Module ID:	libs:webcolours:0.08.14
Deployed By:	ci
Size:	452 bytes
Created:	19-09-20 23:59:15 +01:00
Last Modified:	19-09-20 23:59:15 +01:00
Downloads:	0
Remote Downloads:	0

Dependency Declaration

Build Tool: Maven | Ivy | Gradle | Sbt

```
<dependency>
```

Open source license 7.9.1 rev 70901900 undefined
© Copyright 2020 JFrog Ltd

...
I chose Artifactory. It's not great. It's not terrible. It does the job. And it's free. And I created some small scripts that let me push and pull my RISC OS components in a structured way.

Background

How can I do this? (4)

Managed environments: (cont'd)

- What if I don't want to download my toolchain all the time?

Background

How can I do this? (4)

Sometimes I don't want to download the toolchain and set up binaries. I just want to compile some code. Normally you would just install the toolchain and your libraries on the system in that case. Then they'd be there when you needed them.

Ok, you can do that. But modern build systems have been taking advantage of 'containers'. If you've not come across containers or 'docker' before, then I'll try to summarise what it is in a couple of sentences.

Containers are a way of bundling all the resources you need - and only those that you need - to do a task into a single package and let you run just that package. Docker allows this to be done within Linux systems, by packaging just the bits of the OS and the tool that you need, and then running that in isolation.

If you're thinking that's kinda like a RISC OS application that included the modules it wanted, it's a bit like that, but much more - if the application contained the bits of the OS and the boot or system resources, you're getting closer.

...

Background

How can I do this? (4)

Managed environments: (cont'd)

- What if I don't want to download my toolchain all the time?

Tech: Docker RISC OS development environment.

```
charles@laputa ~/pro/RO/mod/ris/Sou/Des/WindowScroll (master)>
docker run -it --rm -v $PWD:/riscos-source -v $PWD/build:/riscos-build --workdir /riscos-source
gerph/riscos-build riscos-amu
riscos-cmunge   -px -DCMHG   -IC:,RISC_OSLib: -26bit -o oz/modhead cmhg/modhead
CMunge 0.77 (JRF:0.77.47) [13 Jun 2006]
Copyright (c) 1999-2006 Robin Watts/Justin Fletcher
Norcroft RISC OS ARM C vsn 5.18 (JRF:5.18.119)  [Jun  7 2020]
ARM AOF Macro Assembler 3.32 (JRF:3.32.38) [07 Mar 2006]
0 Errors, 2 Warnings suppressed by -NOWarn
riscos-cc   -c  -Wc -fa   -IC:,RISC_OSLib: -za1 -apcs 3/26/fpe2/swst/fp -D__CONFIG=26 -zM -zps1
-o oz/main c/main
Norcroft RISC OS ARM C vsn 5.18 (JRF:5.18.119)  [Jun  7 2020]
"c/main", line 564: Warning: '=': cast of 'int' to differing enum
c/main: 1 warning, 0 errors, 0 serious errors
riscos-link -rmf -rescan -C++ -o rm/WindowScroll,ffa oz.main oz.modhead C:o.stubs
```

...

And that's the technology used here - it bundles the RISC OS development tools into a docker container, then lets me build my source with those tools. You can see the docker command is a little unweildy. I actually have a small script called 'riscos-build' which contains the bulk of that command, so I don't run the full command except when demonstrating it.

In this example, I'm building one of my old modules using the new toolchain.

Background

How can I do this? (5)

Automated testing:

Background

How can I do this? (5)

The next feature of modern development on my list was automated testing. This is very closely associated with the process of 'Continuous Integration' and the terms are often used interchangeably. The distinction to draw here is between ad-hoc testing, where you test what you think is going to be a problem by hand. And automated testing, where you define tests which are run automatically.

- In ad-hoc testing, you essentially sign off that you think it's good enough.
- In automated testing, your tests tell you whether it's good enough.

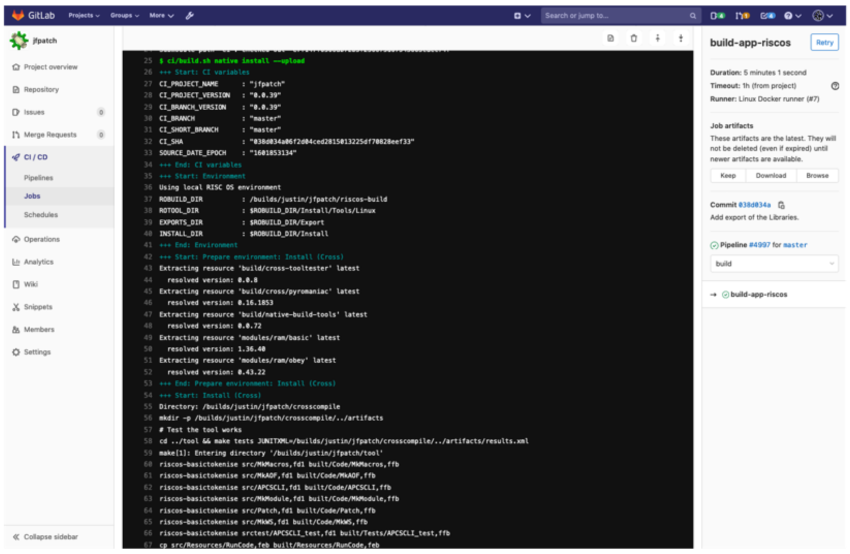
Because the tests are automatically run, you find out quickly that things have broken, and you don't reintroduce earlier bugs - assuming you've written tests for them. Specifically that's feature and regression testing - there are other types of tests that can be run automatically, and you can write your tests to do whatever you want.
...

Background

How can I do this? (5)

Automated testing:

Tech: GitLab CI triggers on every change - pulls resources from Artifactory, builds, pushes result to Artifactory.



```
$ ci/build.sh native install --upload
+++ Start: CI variables
CI_PROJECT_NAME      : "jfpatch"
CI_PROJECT_VERSION   : "0.0.39"
CI_BRANCH_VERSION    : "0.0.39"
CI_BRANCH            : "master"
CI_SHORT_BRANCH      : "master"
CI_SHA               : "038d034a06f2d04ced2815013225df70828eef33"
SOURCE_DATE_EPOCH    : "1601853134"
+++ End: CI variables
+++ Start: Environment
Using local RISC OS environment
ROBUILD_DIR          : /builds/justin/jfpatch/riscos-build
ROTOOL_DIR           : $ROBUILD_DIR/Install/Tools/Linux
EXPORTS_DIR          : $ROBUILD_DIR/Export
INSTALL_DIR          : $ROBUILD_DIR/Install
+++ End: Environment
+++ Start: Prepare environment: Install (Cross)
Extracting resource 'build/cross-tooltester' latest
  resolved version: 0.0.8
Extracting resource 'build/cross/pyromaniac' latest
  resolved version: 0.16.1853
Extracting resource 'build/native-build-tools' latest
  resolved version: 0.0.72
Extracting resource 'modules/ram/basic' latest
  resolved version: 1.36.40
Extracting resource 'modules/ram/obey' latest
  resolved version: 0.43.22
+++ End: Prepare environment: Install (Cross)
+++ Start: Install (Cross)
Directory: /builds/justin/jfpatch/crosscompile
mkdir -p /builds/justin/jfpatch/crosscompile/../artifacts
# Test the tool works
cd ../tool && make tests JUNITXML=/builds/justin/jfpatch/crosscompile/../artifacts/results.xml
make[1]: Entering directory '/builds/justin/jfpatch/tool'
riscos-basictokenise src/MkMacros,fd1 built/Code/MkMacros,ffb
riscos-basictokenise src/MkAOF,fd1 built/Code/MkAOF,ffb
riscos-basictokenise src/APCSCLI,fd1 built/Code/APCSCLI,ffb
riscos-basictokenise src/MkModule,fd1 built/Code/MkModule,ffb
riscos-basictokenise src/Patch,fd1 built/Code/Patch,ffb
riscos-basictokenise src/MkWS,fd1 built/Code/MkWS,ffb
riscos-basictokenise srctest/APCSCLI_test,fd1 built/Tests/APCSCLI_test,ffb
cp src/Resources/RunCode,feb built/Resources/RunCode,feb
```

build-app-riscos

Duration: 5 minutes 1 second
Timeout: 1h (from project)
Runner: Linux Docker runner (#7)

Job artifacts
These artifacts are the latest. They will not be deleted (even if expired) until newer artifacts are available.

Commit 038d034a
Add export of the Libraries.

Pipeline #4997 for master

→ build-app-riscos

...

The GitLab source control system has a test system built in. This lets you run whatever tests you need to. When a change is pushed to the GitLab server, the tests are triggered.

In my case, these tests will:

- download the toolchain and environment from Artifactory
- build the code
- and then push the built result back up to Artifactory if it was successful.

If it wasn't successful, I'll get an email, and the website reports the build log.

This example is showing a build of the JFPatch tool, which tokenises the BASIC code and collects the results into an artifact that can be used in the service.

Background

How can I do this? (6)

Feature and regression testing:

- Build programs and test code on other platforms.

Background

How can I do this? (6)

If I'm building, let's say, a RISC OS module, it'd be normal to write all the code in little components that you can try out separately, before bundling them together into the scary module enviroment. For a lot of code, you can test those components without needing the rest of the system. And for a lot of code, you can do that on non-RISC OS systems.
If you're just managing data structures, you don't need a real RISC OS system to check that the logic works right, for example. So in those cases you can make your code compile and test on a different platform. That's one way of doing things - and it's very effective.
In fact, just last month Jason Tribbeck was explaining exactly that process for his unit testing of the sound system.
...

Background

How can I do this? (6)

Feature and regression testing:

- Build programs and test code on other platforms.
- I need a way to test things on RISC OS, too…

Tech: … we'll come to that later …

…
But I really want to run things on RISC OS, because the environment that modules run in is a bit different, and you'll want to communicate with other parts of the system in situ. In particular, modules run in SVC mode - the mode with the highest priviledge and therefore the greatest power to make it a Bad Day for you.
We'll come to how you do that later.

Background

How can I do this? (7)

Fleets of systems for people to use:

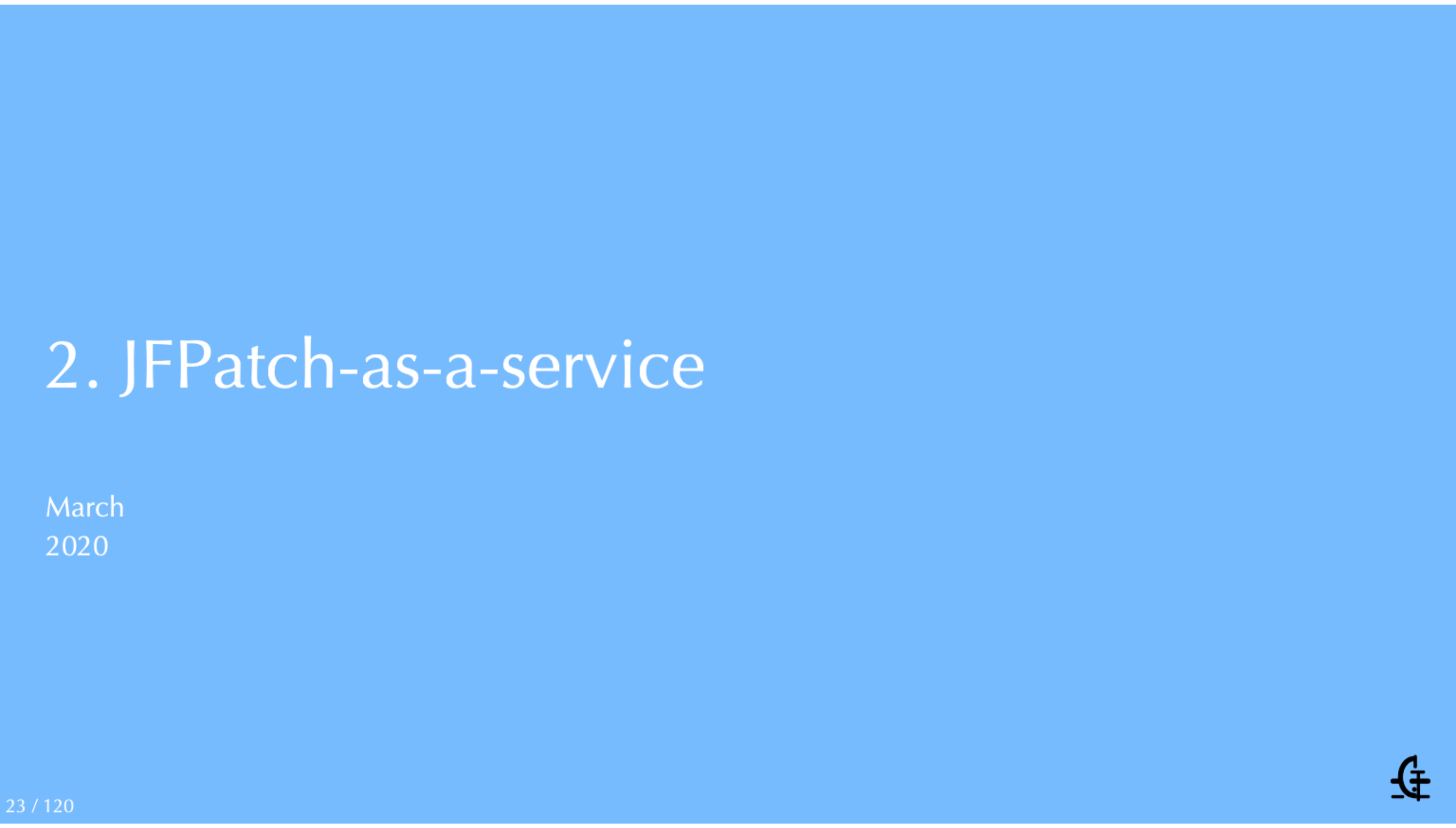
- That seems a stretch, but maybe it's not so hard...

Tech: JFPatch-as-a-service begins that process

Background

How can I do this? (7)

My final feature of modern development was to have a fleet of machines available to do your bidding - whether it be testing or processing. And maybe you don't need that, but if you're wanting to run a few thousand tests at a time then maybe you can save time by spreading them over multiple machines.
JFPatch-as-a-service isn't quite that.
But it's a step along the way.
Before I move on to talk about JFPatch-as-a-service itself, let's see if there any questions.



2. JFPatch-as-a-service

March
2020

2. JFPatch-as-a-service

Let's talk about JFPatch-as-a-service.
We spin forward a year - we'll cover the missing time later.

JFPatch-as-a-Service
Why?

A friend said to me…

> *“I can't wait until you csa.announce this and confuse the bejesus out of the RISC OS civilians.”*

To which my answer was…

> *“JFPatch as a service would be a doddle to do right now. A service that nobody asked for, or needed.”*

JFPatch-as-a-Service
Why?

In early March, David Thomas challenged me to make some of the things I'd been doing public. I was still nervous about that, and wasn't really comfortable making that kind of announcement.
But the idea of doing something for April 1st, which wasn't actually an April fool, was appealing. I genuinely expected the response to be a 'meh', but that wasn't the point.
The whole point of the work I was doing was to be enjoyable - and I like solving problems, particularly when those problems are strange. It'd been floating around in my head for a while to do 'something' like it, but it hadn't really solidified until then.
Running an on demand RISC OS build process on a cloud system as a service is probably unique. I've been away and I don't pay attention to the comings and goings, so maybe it's completely passe, but I liked the idea.
And I like the idea of surprising people with something they didn't expect.

JFPatch-as-a-Service

What is JFPatch?

- It's a pre-processor for the BASIC assembler.
- It has its own file format which describes things to patch, or modules to build.
- It converts these to BASIC files, then runs the BASIC, which does the heavy lifting of assembling.
- It is, itself, written in BASIC.
- It was used to write many of my early assembler modules.

JFPatch-as-a-Service

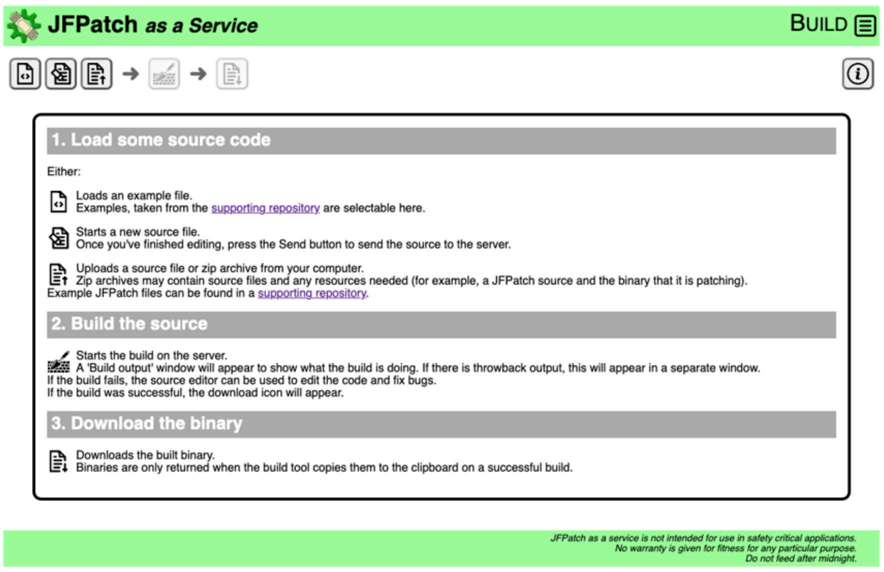
What is JFPatch?

JFPatch... it's an assembler I wrote when I was at school... something like 25 years ago. I learnt assembler by reading other people's code and patching it to do different things. Largely that was by writing a BASIC program that loaded the code, assembled some replacement code and saved it again.
That process is tedious. I wanted a shorthand for doing that boilerplate work. So I created a pre-processor that built those BASIC programs from a 'patch file'. I knew I shouldn't pollute the namespace, so the tool got a prefix of my initials.
It's actually not that clever, but it can construct a lot of the boilerplate that goes into writing modules. If you want a module that registers itself with a few vectors you can do that in just a couple of lines of code. It even goes so far to let you register a whole filesystem through some simple definitions in the header.

JFPatch-as-a-Service

What is the service?

- Takes its inspiration from Matt Godbolt's Compiler explorer - https://godbolt.org/



1. Load some source code

Either:

Loads an example file.
Examples, taken from the supporting repository are selectable here.

Starts a new source file.
Once you've finished editing, press the Send button to send the source to the server.

Uploads a source file or zip archive from your computer.
Zip archives may contain source files and any resources needed (for example, a JFPatch source and the binary that it is patching).
Example JFPatch files can be found in a supporting repository.

2. Build the source

Starts the build on the server.
A 'Build output' window will appear to show what the build is doing. If there is throwback output, this will appear in a separate window.
If the build fails, the source editor can be used to edit the code and fix bugs.
If the build was successful, the download icon will appear.

3. Download the binary

Downloads the built binary.
Binaries are only returned when the build tool copies them to the clipboard on a successful build.

JFPatch as a service is not intended for use in safety critical applications.
No warranty is given for fitness for any particular purpose.
Do not feed after midnight.

JFPatch-as-a-Service

What is the service?

What's the service?
The service is a front end that lets you upload JFPatch code, builds it on a cloud server, and then returns the result to you.
I wanted it to be a tool that could be used like Matt Godbolt's compiler explorer. If you've not seen it, Compiler Explorer is a website that lets you enter some code and examine what the compiler makes of it. What instructions it compiles to and how they're ordered.
It does a different thing to JFPatch-as-a-service, and my work isn't anywhere near as polished and professional as that, but... that's where the idea and structure came from. Matt's an ex-RISC OS user, too.

JFPatch-as-a-Service

What can you build with the service? (1)

- Any JFPatch code (now builds 32bit code)
- C code that compiles with the Norcroft compiler
- Pascal code (which will be converted to C and compiled with the Norcroft compiler)
- Perl code
- BASIC assembler
- Objasm assembler.

Tech: All the toolchain is built for 32bit RISC OS, automatically taken from Artifactory when the service is built.

JFPatch-as-a-Service

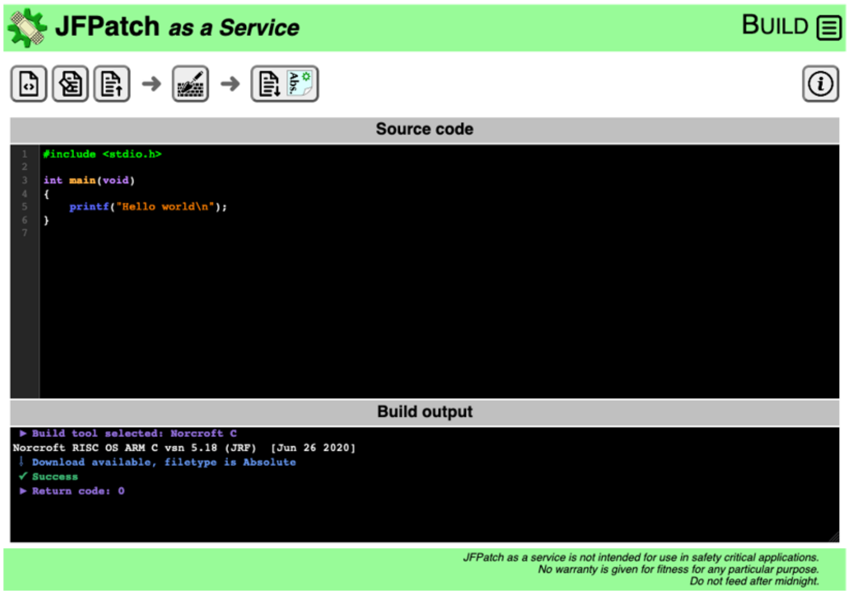
What can you build with the service? (1)

But what can the service do? Well, it can build any JFPatch code you throw at it, and now the module header is 32bit safe as well it can work on modern systems.
But, the service can also build C, and Pascal. It run BASIC programs, and build ObjAsm code. If you give it a zip archive containing an AMU Makefile, it would invoke it and return the binary it thought was produced by it.
You can try this out on the real site if you'd like. Here's what you see...

JFPatch-as-a-Service

What can you build with the service? (2)

C code…



```
#include <stdio.h>

int main(void)
{
    printf("Hello world\n");
}
```

```
 ▶ Build tool selected: Norcroft C
Norcroft RISC OS ARM C vsn 5.18 (JRF)  [Jun 26 2020]
 ⭳ Download available, filetype is Absolute
 ✓ Success
 ▶ Return code: 0
```

JFPatch as a service is not intended for use in safety critical applications.
No warranty is given for fitness for any particular purpose.
Do not feed after midnight.

JFPatch-as-a-Service

What can you build with the service? (2)

<use xc to turn on the cursor and point to bits>

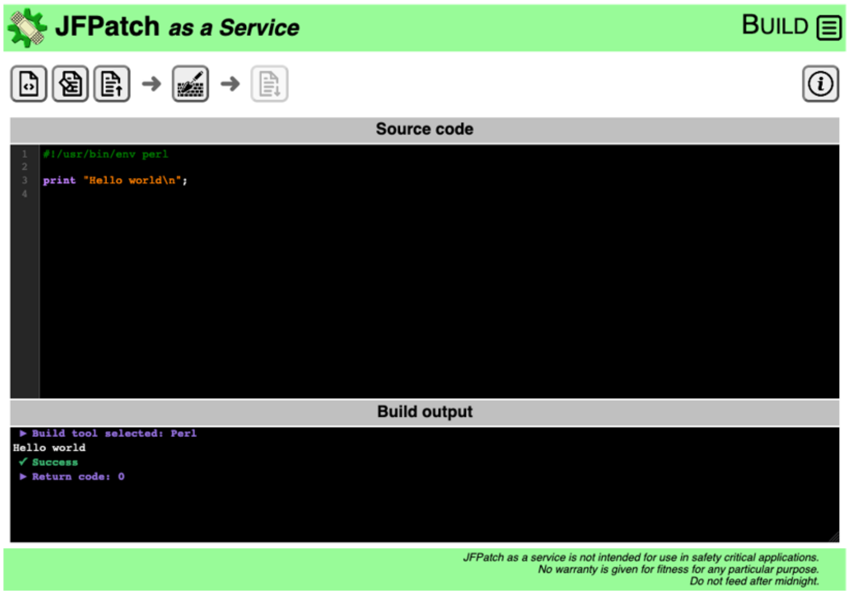
- The editor lets you enter the code you want to build.
- Your code can then be sent to the server for building, and the build output will be shown to you.
- Any output files produced will be available for download.

This is a small C file that I wrote, and then sent to the service. You can see in the Build output at the bottom that it has built, and the result was returned as an Absolute file.

JFPatch-as-a-Service

What can you build with the service? (3)

Perl code...



```
#!/usr/bin/env perl

print "Hello world\n";
```

```
 ▶ Build tool selected: Perl
Hello world
 ✓ Success
 ▶ Return code: 0
```

JFPatch as a service is not intended for use in safety critical applications.
No warranty is given for fitness for any particular purpose.
Do not feed after midnight.

JFPatch-as-a-Service

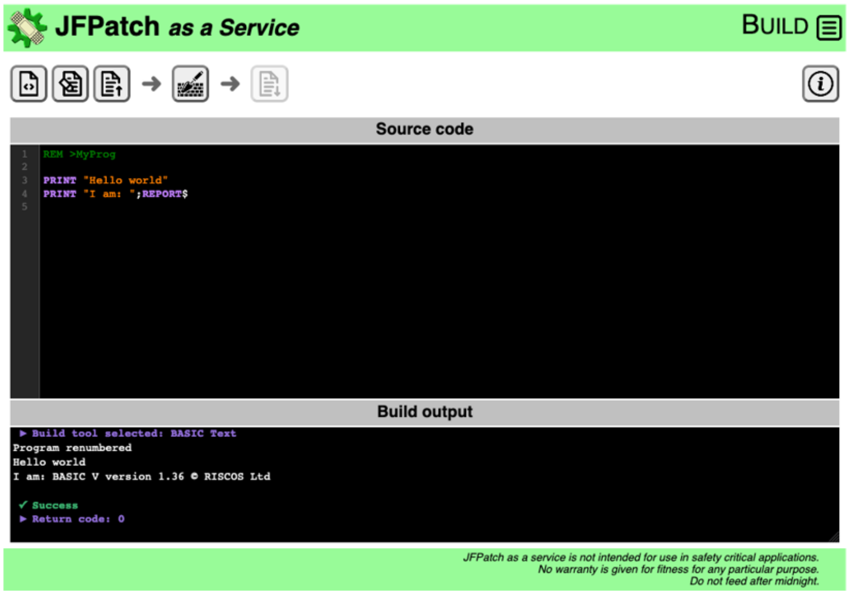
What can you build with the service? (3)

This is a very simple Perl program. There's no returned binary from this, because we just printed a message. You can see the 'Hello world' message in the build output.

JFPatch-as-a-Service

What can you build with the service? (4)

Plain BASIC...



```
REM >MyProg

PRINT "Hello world"
PRINT "I am: ";REPORT$

```

```
▶ Build tool selected: BASIC Text
Program renumbered
Hello world
I am: BASIC V version 1.36 © RISCOS Ltd

 ✓ Success
 ▶ Return code: 0
```

JFPatch as a service is not intended for use in safety critical applications.
No warranty is given for fitness for any particular purpose.
Do not feed after midnight.

JFPatch-as-a-Service

What can you build with the service? (4)

And finally a BBC BASIC program that also does very little.
And the service could actually do all this from April 1st. I had kinda hoped that someone would play with it and find that it did more things than just building JFPatch. I know I would have. It's not hidden, the HTML source on the website says explicitly that it supports these things, and describes which languages are recognised.
Nobody came to talk to me about it, so that 'easter egg' remained hidden.

JFPatch-as-a-Service

What might use the service?

Automated builds can use this:

- LineEditor (BASIC assembler) - https://github.com/philpem/LineEditor
- Nettle (C application) - https://github.com/gerph/Nettle/tree/ci
- CObey (C module) - https://github.com/gerph/cobey
- ErrorCancel (ObjAsm) - https://github.com/gerph/errorcancel
- Pico (C command line tool) - https://github.com/gerph/pico
- DDEUtilsJF (JFPatch module) - https://github.com/gerph/ddeutilsjf

JFPatch-as-a-Service

What might use the service?

In the time since then, I've converted a few other RISC OS tools to be able to be automatically built. They're on github with the label `riscos-ci`.
Some of the tools were very simple to make work - Rick Murray's ErrorCancel for example was just a simple matter of adding the right build commands to the repository.
Others were a little more involved, like Julie Stamp's CObey module, which I had to tweak the code, because I'm using an older compiler.
Running a build automatically on submitting the code is a great way of checking that it still works. Presumably you've built it yourself, but it never hurts to have an automated system check that.
It is also possible to do a 'release' of the code automatically in GitHub, so you can actually get a distributable archives out from the build.
If you check the DDEUtilsJF link, you'll see that there are 'releases' listed on the right of the page - those were automatically created after push. They just have to be approved by clicking a button in GitHub.

JFPatch-as-a-Service

How do you use the service?

Two interfaces, which are documented:

- JSON API.
- WebSockets API.

Documented on the website: https://jfpatch.riscos.online/api.html
Examples can be found at: https://github.com/gerph/jfpatch-as-a-service-examples

JFPatch-as-a-Service

How do you use the service?

How do you use it?
The service can be used 3 ways - either through the web interface, as you've seen, or by using the API. A service isn't really a service unless it can be used programatically.
There are two APIs for accessing the service - either a JSON HTTP API, or a WebSockets API.
The JSON API is more limited because you basically just post data to it and get back a response. It's also got a timeout of about a minute, which for some jobs may not be long enough.
The WebSockets API however, is an interactive system. If you're not familiar with WebSockets, then that's not especially surprising. They've been around since 2010, but unless you've been working on heavily interactive sites you probably wouldn't encounter them.
Unlike the regular HTTP request-response protocol, WebSockets allow bi-directional communication. That means you can send a message to the other end, and you can receive them. Packets are always delivered in order, and always delivered completely. Their content is application defined, so once established you can send anything you want.
JFPatch-as-a-service uses commands in JSON over WebSockets to request the server do builds for it.

JFPatch-as-a-Service

How do you use the JSON API?

Use your favourite HTTP request library. For example, `curl`:

```
curl -i -F 'source=@source-file'  http://jfpatch.riscos.online/build/json
```

Get a JSON response:

```
{
  "data": "... data goes here ...",
  "filetype": 4092,
  "messages": [
    "Build tool selected: JFPatch",
    "Return code: 0"
  ],
  "output": [
    "JFPatch ARM assembler v2.56\u00df (02 Mar 2020) [Justin Fletcher]\r\n",
    "Pre-processing...\r\n",
    "Assembling...\r\n"
  ],
  "rc": 0,
  "throwback": []
}
```

JFPatch-as-a-Service

How do you use the JSON API?

How do you use the JSON API?
Just use a URL library. The URLFetchers on RISC OS would do just as well, but probably you wouldn't need to invoke a cloud-based RISC OS build client from a RISC OS system. But you could.
And of course, you can parse the JSON response with any JSON library you'd like.
This is the sort of request and response you would have for building with the API.

JFPatch-as-a-Service

How do you use the WebSockets API?

Using the `wsclient.py` example gives a similar output.

```
welcome: u'Linking over Internet with RISCOS Pyromaniac Agent version 1.04'
response: u'Source loaded'
response: u'Started build'
message: u'Build tool selected: JFPatch'
output: u'JFPatch ARM assembler v2.56\xdf (02 Mar 2020) [Justin Fletcher]\r\n'
output: u'Pre-processing...\r\n'
output: u'Assembling...\r\n'
clipboard: {u'filetype': 4092, u'data': u'... data goes here ...'}
rc: 0
message: u'Return code: 0'
complete: True
```

Q: What about when you don't have, or can't use, Python?

A: `robuild-client` handles that.

JFPatch-as-a-Service

How do you use the WebSockets API?

The WebSockets system is a little harder to work with, though. You definitely need a client to talk to it.

Fortunately, in the JFPatch examples repository I supplied a python client that uses a package which does the work for you. This is an example of using 'wsclient.py' to build the same code.

But that's not always a suitable solution - it means you have to install other Python packages, and you have to have Python in the first place.

For those cases I created the `robuild-client`.

JFPatch-as-a-Service

What is the robuild-client?

- Created a build client that can be used to do the heavy work.
- Can be found at https://github.com/gerph/robuild-client
- Builds for Linux...

JFPatch-as-a-Service

What is the robuild-client?

The `robuild-client` tool is a little tool that's able to take a file, and give it to the service over the WebSockets interface. It can then report the output as it is produced by the build process. Finally, it can then save the output to a known location, or report an error through the exit status of the command in the normal way.
This makes it easy to work into a build pipeline.
The Github project for `robuild-client` itself builds a Linux version of the tool...
...

JFPatch-as-a-Service

What is the robuild-client?

- Created a build client that can be used to do the heavy work.
- Can be found at https://github.com/gerph/robuild-client
- Builds for Linux...
- ... then uses the tool it built to submit its code to the service, to build the RISC OS version.

Tech:

- robuild-client.
- port of JSON parse/creation library.
- WebSockets library.

...

... and then using that tool it submits its own code to the service, so that we get a RISC OS build out.

Why would you want to have run it from RISC OS? I don't know, but completeness says if it's a tool for RISC OS, maybe it should run on RISC OS. More specifically, I can't completely repress that feeling that people in the RISC OS world reject anything that isn't for RISC OS as irrelevant, so I have to make it work for RISC OS too.

I used two libraries to do this - a JSON parser, and a WebSockets library. The JSON parser was originally written by David Gamble, a friend who used to do RISC OS things, although I didn't discover this until afterward.

The repository is also an example of producing releases from an automated build, as well as building RISC OS code from a non-RISC OS code base using `sed` to fix incompatible things.

I didn't say it was a *good* example!

JFPatch-as-a-Service

How does the service know what to do?

- Simple files are recognised by their format.
- Zip files are recognised by their content.
  - The `.robuild.yaml` file can control what is actually run.

JFPatch-as-a-Service

How does the service know what to do?

How does it know what to do?
For simple files, it spots the language from its format. For example, if it's got BASIC tokens in then it's a BASIC file. If it has C comments in, it's a C file. The heuristic is pretty poor, but it works on pretty much anything but simple cases.
Zip archives can contain multiple files, and the services picks out ones that it can understand. If there's a Makefile in the archive that it recognises, it'll invoke AMU on the file. If there's a JFPatch file, it'll run that file - which allows you to patch other files.
Hoping the service guesses right isn't ideal, so there's more control available.
If the Zip archive contains a `.robuild.yaml` file, this will be used to control how the build runs.
...

JFPatch-as-a-Service

How does the service know what to do?

- Simple files are recognised by their format.
- Zip files are recognised by their content.
    - The `.robuild.yaml` file can control what is actually run.

```
%YAML 1.0
---

jobs:
  build:
    # Env defines system variables which will be used within the environment.
    # Multiple variables may be assigned.
    env:
      "Sys$Environment": ROBuild

    # Commands which should be executed to perform the build.
    # The build will terminate if any command returns a non-0 return code or an error.
    script:
      - dir riscos
      - !BuildAll
      - Clipboard_FromFile client.aif32.riscos-build-online
```

...

YAML stands for 'Yet Another Markup Language'. It's nicer to work with than JSON, so long as you avoid the many crazy cases it has. It's pretty nutty in extremis, but it's quite readable.

Although I say the file is called `.robuild.yaml` on RISC OS that would be `/robuild/yaml`, and it might have a filetype of hex F74.

In this example, from `robuild-client` itself, we set a single environment variable, and then give it 3 commands to run. Those are just RISC OS CLI commands, so you can do a lot more if you wanted - `!BuildAll` is an Obey file, which builds the libraries and then uses them to build the tool. The final copy to the clipboard is how we communicate the output to the service.

There's fuller documentation of the file format on the JFPatch-as-a-service website.

Before I move on to talk about how the service works, let's see if there any question on the front end of the service.

3. How The Service Works

March
2020

3. How The Service Works

I've talked a little about what the service offers and how you drive it. So now I'm going to talk about how it achieves that in the back end.

How The Service Works

What is the service made of? (1)

Tech:

- Infrastructure - AWS SSL, routing and linux server.
- Front End - Static site, websockets to talk to back end
  - Custom CodeMirror colouring - https://github.com/gerph/CodeMirror/tree/riscos-modes
- Back End - Python REST JSON API and WebSockets service
  - RISC OS Zip file decoding in Python - https://github.com/gerph/python-zipinfo-riscos
- Tools - JFPatch tool, compiler, assembler, linker, amu, etc.

How The Service Works

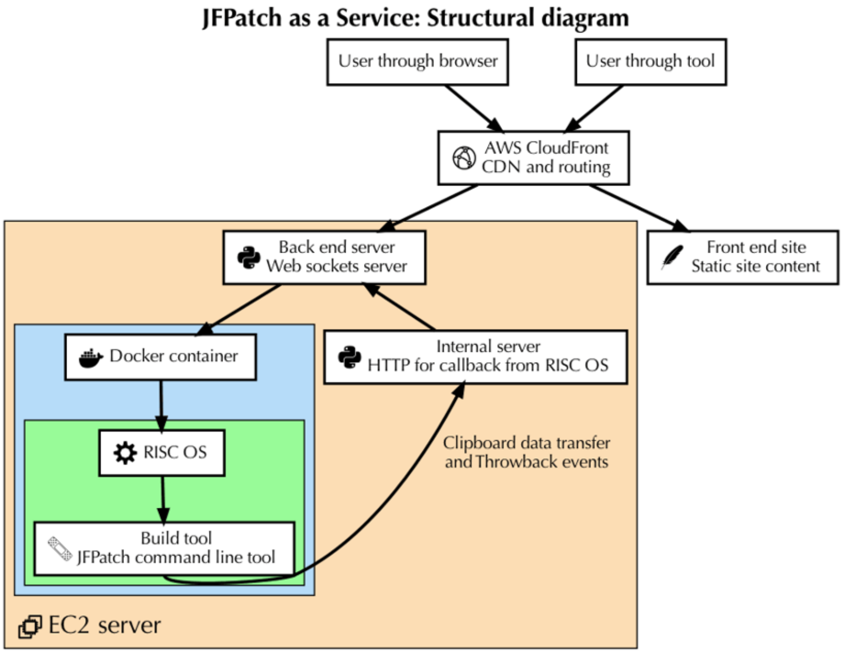
What is the service made of? (1)

What is the service made of?

- Infrastructure - The infrastructure runs on AWS - Amazon Web Services - on their virtual machines. It has a load blancer and a small server which is running constantly. It could scale, but it's set up to not do so - the infrastructure costs about $35/month, which is about $30 too much, but I've not got around to streamlining it. It's a 'meh' at the moment.
- FrontEnd - Static site is built with HSC, and uses a custom colouring mode for ARM, BASIC and JFPatch. The new colouring modes were open sourced a while back, and you'll find them on GitHub.
- BackEnd - Two Python flask applications provide the back end - 1 for websockets, 1 for JSON. These accept the requests for builds, and dispatch them to RISC OS, providing the conduit for the output and throwback.
- The back end service uses a python module which handles RISC OS extension data in Zip files. I open sourced that a while back.
- Tools - The tool that runs is JFPatch itself. JFPatch was one of the easiest things to change for this environment. It needed a way to pass the built binary out to the caller - previously it just wrote to whatever file you supplied. Within the service it uses a new option to copy the output to the clipboard using the ClipboardHolder module. The back end code can then return the built binary to the user.
- JFPatch itself is written in BASIC which is tedious to work with on non-RISC OS systems, so all the source files are stored in text form and tokenised with a new tool I created. You saw an example of the tool being built in an earlier slide.

How The Service Works

What is the service made of? (2)



How The Service Works

What is the service made of? (2)

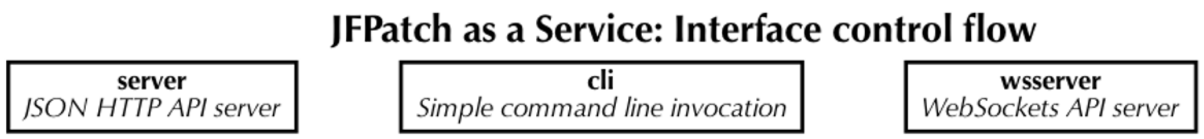
This is the basic structure of the system

- User accesses the site.
- AWS directs the request to the back end server.
- Back end server decides what we need to do.
- Server calls in to the RISC OS system.
- RISC OS runs the build.
- The result comes back to the server through HTTP.

It's probably worth noting at this point that no part of the service uses the cross-compiling tools - it's RISC OS all the way when it comes to running what you submit. That diagram is up on the JFPatch-as-a-service site, so let's look at what goes on below that.

How The Service Works

What runs those services? (1)


JFPatch as a Service: Interface control flow
server
JSON HTTP API server
cli
Simple command line invocation
wsserver
WebSockets API server

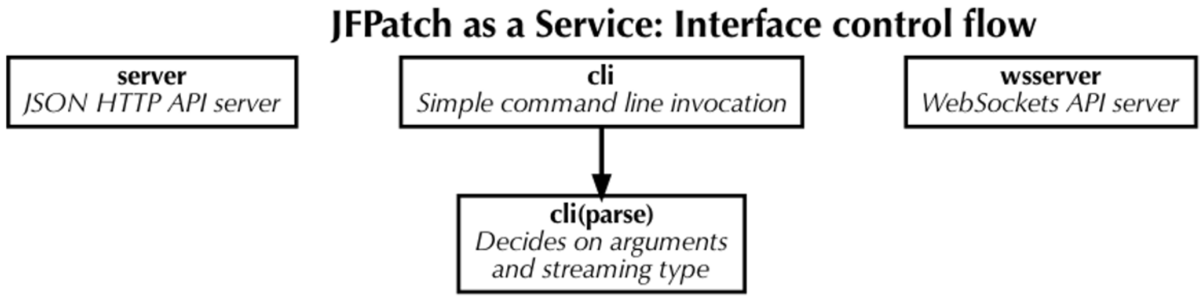
How The Service Works

What runs those services? (1)

There's actually 3 different ways that the JFPatch-as-a-service system can be invoked - as the JSON server, as the WebSockets server, and as a CLI tool.
The CLI tool is only ever used for debugging, so never gets used by the service itself, but it exists to make it possible to test the build process without having the JSON API or WebSockets API get in the way.
...

How The Service Works

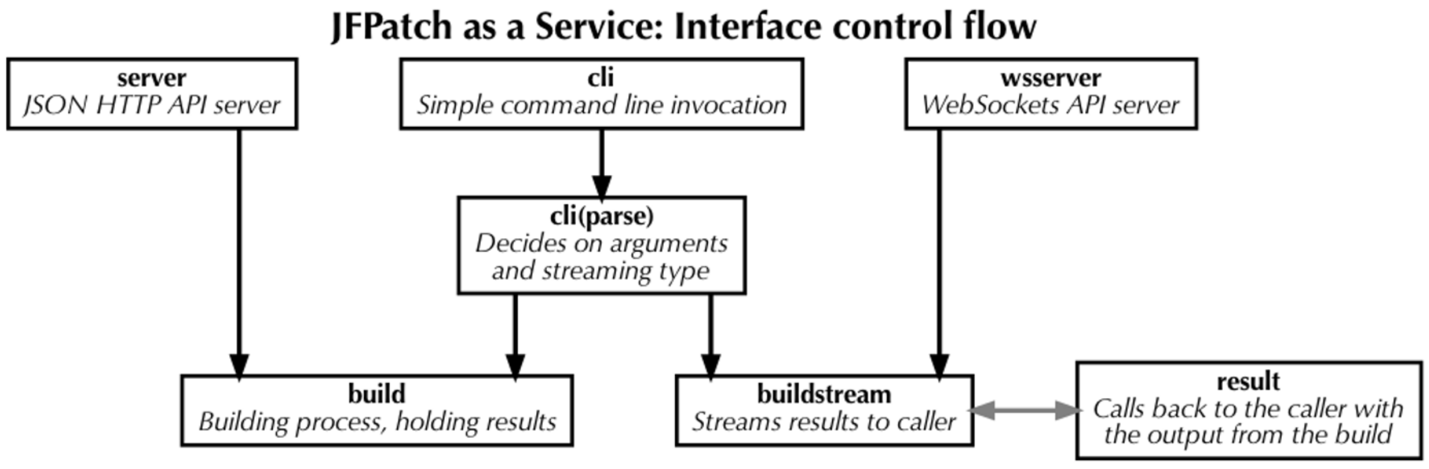
What runs those services? (2)


JFPatch as a Service: Interface control flow
server
JSON HTTP API server
cli
Simple command line invocation
wsserver
WebSockets API server
cli(parse)
Decides on arguments and streaming type

...
When the CLI tool is used, it has to manually parse out the parameters that would have been delivered to the WebSockets interface. The CLI system can use the simple interface, or the streaming interface like the WebSockets system.
...

How The Service Works

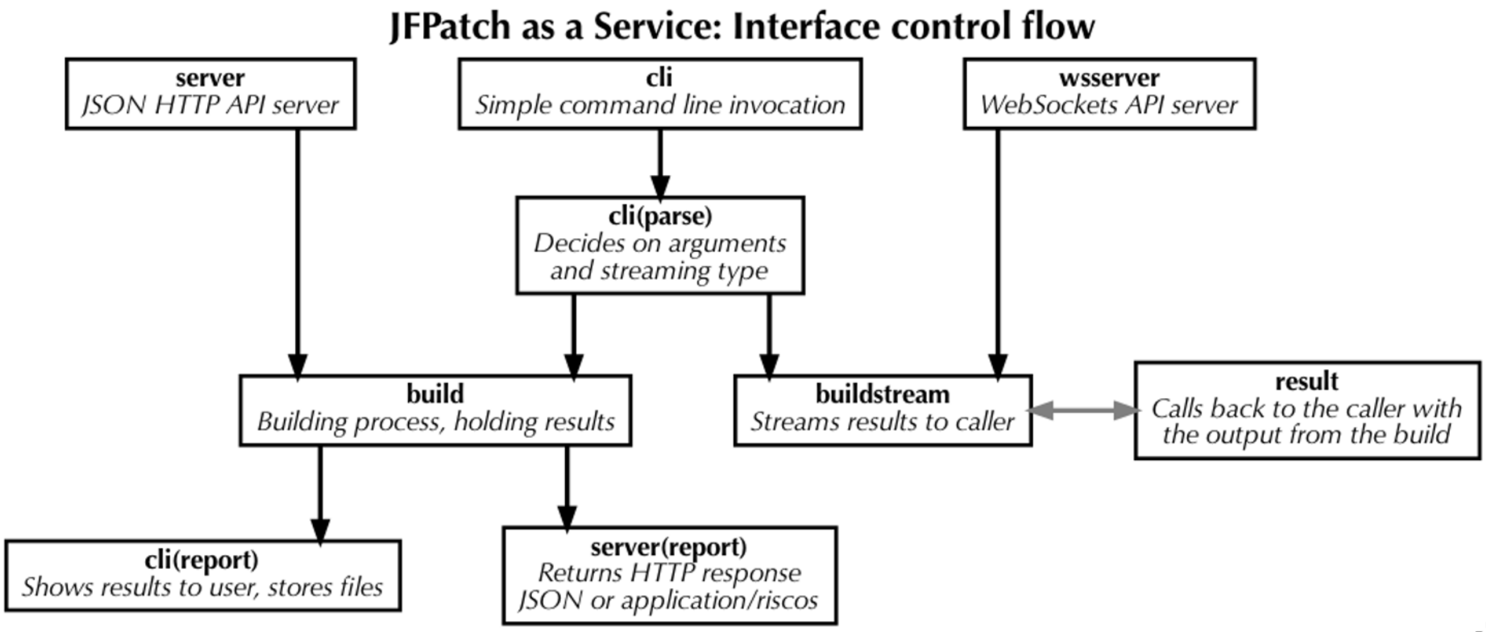
What runs those services? (3)


JFPatch as a Service: Interface control flow
server
JSON HTTP API server
cli
Simple command line invocation
wsserver
WebSockets API server
cli(parse)
Decides on arguments and streaming type
build
Building process, holding results
buildstream
Streams results to caller
result
Calls back to the caller with the output from the build

...
There's a 'build' object that knows how to run things. And there's a streaming version of that object - the difference is that the streaming version communicates the progress and results through a 'results' object that can talk to the caller. That's how the WebSockets system, and the CLI system in stream mode, get back information.
...

How The Service Works

What runs those services? (4)


JFPatch as a Service: Interface control flow
server
JSON HTTP API server
cli
Simple command line invocation
wsserver
WebSockets API server
cli(parse)
Decides on arguments and streaming type
build
Building process, holding results
buildstream
Streams results to caller
result
Calls back to the caller with the output from the build
cli(report)
Shows results to user, stores files
server(report)
Returns HTTP response JSON or application/riscos

...

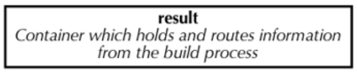
And once the build process is completed, we report back to the CLI or server with an appropriate final result.

Let's go look inside what happens in the build then.

How The Service Works

What runs those services? (5)

JFPatch as a Service: Builder control flow


result
Container which holds and routes information from the build process

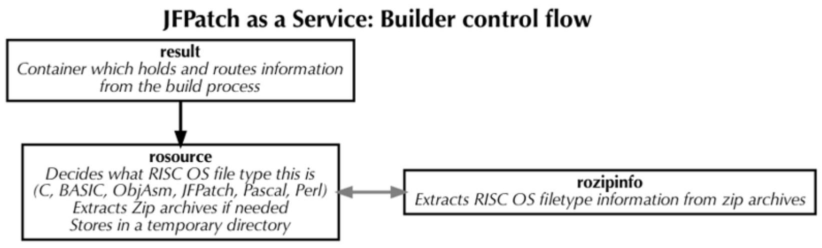
How The Service Works

What runs those services? (5)

As we saw from the last slide, we create a result object to hold information - even for the non-streaming case, we have one of these. It just holds the information, rather than routing it to anywhere.
...

How The Service Works

What runs those services? (6)


JFPatch as a Service: Builder control flow
result
Container which holds and routes information from the build process
rosource
Decides what RISC OS file type this is (C, BASIC, ObjAsm, JFPatch, Pascal, Perl)
Extracts Zip archives if needed
Stores in a temporary directory
rozipinfo
Extracts RISC OS filetype information from zip archives

...

The `rosource` module decides what filetype we're trying to handle. It uses some truly terrible heuristics to do this. Like 'is there an all caps PRINT in the file', or 'is there a `#include` in the file'. That only applies to single files, though - Zips are easy to recognise and extract.

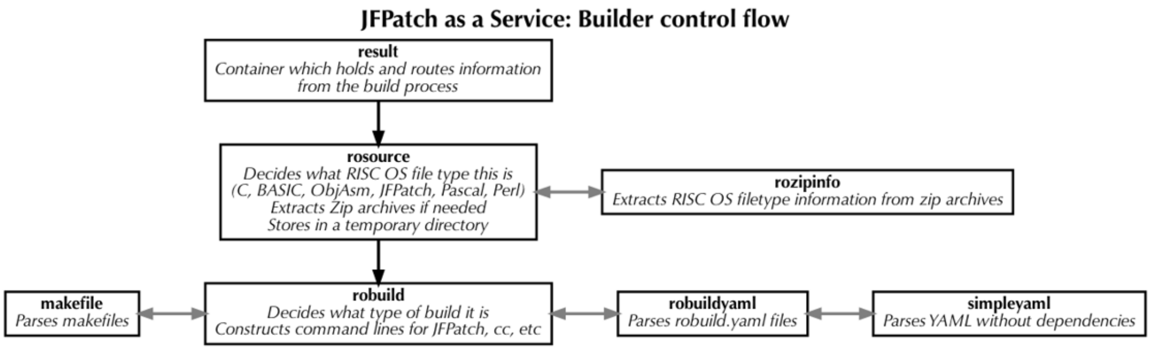
We use the RISC OS Zip library I mentioned earlier to extract the files from the archive. If they've got RISC OS types we use them. If they haven't got RISC OS types then we try to infer them where we can. All the files are stored in standard format for RISC OS on Unix - using the `,xxx` extensions.

Whether it's a single file, or many from a Zip, the result gets put in a temporary directory.

...

How The Service Works

What runs those services? (7)


JFPatch as a Service: Builder control flow
result
Container which holds and routes information from the build process
rosource
Decides what RISC OS file type this is (C, BASIC, ObjAsm, JFPatch, Pascal, Perl)
Extracts Zip archives if needed
Stores in a temporary directory
rozipinfo
Extracts RISC OS filetype information from zip archives
makefile
Parses makefiles
robuild
Decides what type of build it is
Constructs command lines for JFPatch, cc, etc
robuildyaml
Parses robuild.yaml files
simpleyaml
Parses YAML without dependencies

...

The `robuild` module looks at the files that were extracted and decides what type of build it is. It constructs a RISC OS command line that can be used to perform the build.

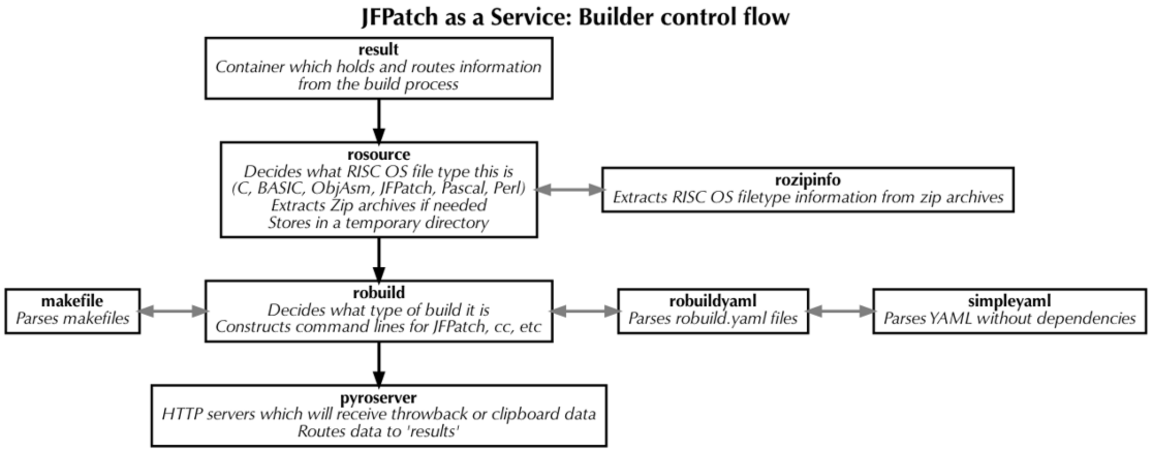
If we saw a `Makefile`, we use a small module which determines what the linkable targets are within it. These linkable targets will be returned to the user implicitly as artifacts.

If we extracted a `robuild.yaml` file, we parse the YAML file to give the commands and parameters for the build. I avoid some of the problems with YAML files being overly complex by not supporting them - the `simpleyaml` module is a package a wrote for just that purpose (that's open source too).

...

How The Service Works

What runs those services? (8)



JFPatch as a Service: Builder control flow
result
Container which holds and routes information from the build process
rosource
Decides what RISC OS file type this is (C, BASIC, ObjAsm, JFPatch, Pascal, Perl)
Extracts Zip archives if needed
Stores in a temporary directory
rozipinfo
Extracts RISC OS filetype information from zip archives
makefile
Parses makefiles
robuild
Decides what type of build it is
Constructs command lines for JFPatch, cc, etc
robuildyaml
Parses robuild.yaml files
simpleyaml
Parses YAML without dependencies
pyroserver
HTTP servers which will receive throwback or clipboard data
Routes data to 'results'

...

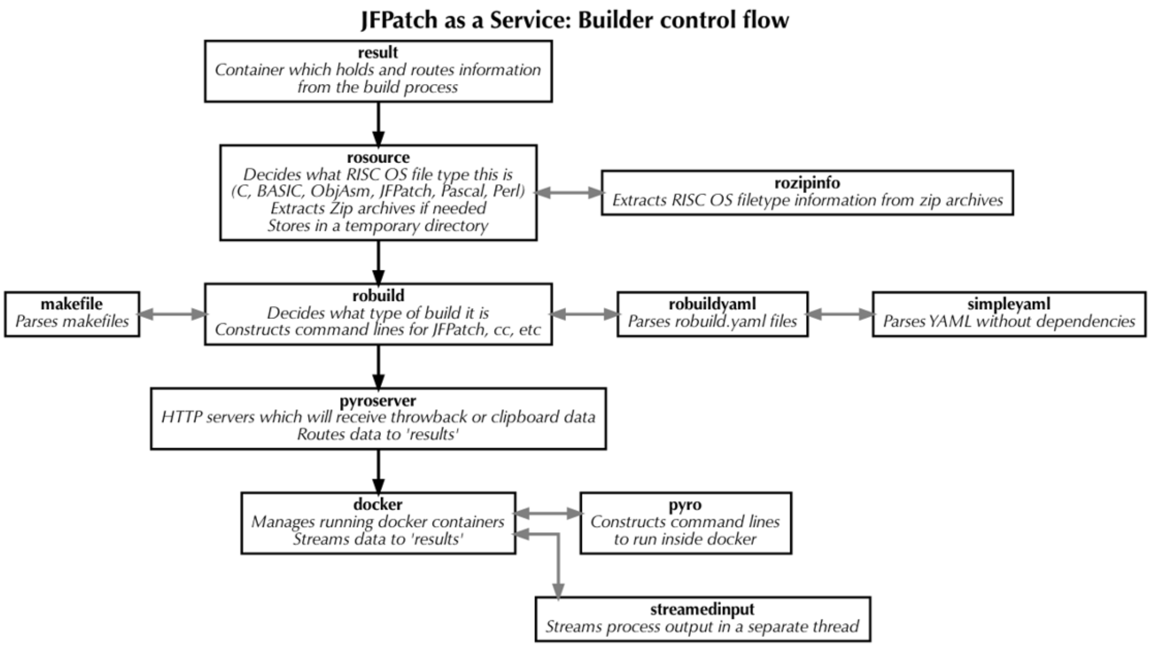
At this point we know what the build is going to look like, so we need to create a HTTP server to receive throwback and clipboard data from JFPatch (or whatever we end up running). The `pyroserver` module does this. Hopefully you remember that I'm using the clipboard to communicate the produced binary. But I've also got a version of DDEUtils that sends throwback data to a HTTP server as well. This way, we get structured information that can be fed back to the caller.

If those HTTP servers are called, they'll feed the data to the `results` object, which we created earlier on.

...

How The Service Works

What runs those services? (9)


JFPatch as a Service: Builder control flow
result
Container which holds and routes information from the build process
rosource
Decides what RISC OS file type this is (C, BASIC, ObjAsm, JFPatch, Pascal, Perl)
Extracts Zip archives if needed
Stores in a temporary directory
rozipinfo
Extracts RISC OS filetype information from zip archives
makefile
Parses makefiles
robuild
Decides what type of build it is
Constructs command lines for JFPatch, cc, etc
robuildyaml
Parses robuild.yaml files
simpleyaml
Parses YAML without dependencies
pyroserver
HTTP servers which will receive throwback or clipboard data
Routes data to 'results'
docker
Manages running docker containers
Streams data to 'results'
pyro
Constructs command lines to run inside docker
streamedinput
Streams process output in a separate thread

...

Finally we're ready to run the build. The `docker` module constructs docker command lines, which includes routing the HTTP ports properly, and setting up the commands to run RISC OS. The `streamedinput` module, which is also open source, handles receiving lines from the Docker process and feeding them to the results object.

Finally, there's the `pyro` module that builds the actual command to run RISC OS.

How The Service Works

What runs those services? (10)

- `robuild` has worked out the RISC OS commands to use.
- `pyro` is given those commands and constructs a command that can run RISC OS with those commands.
- `docker` is given that command, and builds a command to run RISC OS within a docker container.
- ... and the results of all of that gets fed back to the `results` object, which passes it back to the caller.

How The Service Works

What runs those services? (10)

There are 3 different command line sequences getting built up to run the RISC OS command in a safe an isolated way on the Linux server, and get the results back to the user.

- There's the commands we should run in RISC OS to the build
- There's the commands to run RISC OS itself.
- And the commands to run docker.

How The Service Works

Wait what?

> *“ Wait, RISC OS is running in Docker?*
> *But Docker runs on Linux?*
> *You're running RISC OS on Linux then? ”*

How The Service Works

Wait what?

It'd be nice if someone were to ask this question... but it's probably obvious from what I've been saying - and from the public documentation - that it's running RISC OS on a Linux system.
Before I move on to talk about how that works, let's see if there any on questions about this back end service.

• • •

June
2019

•••

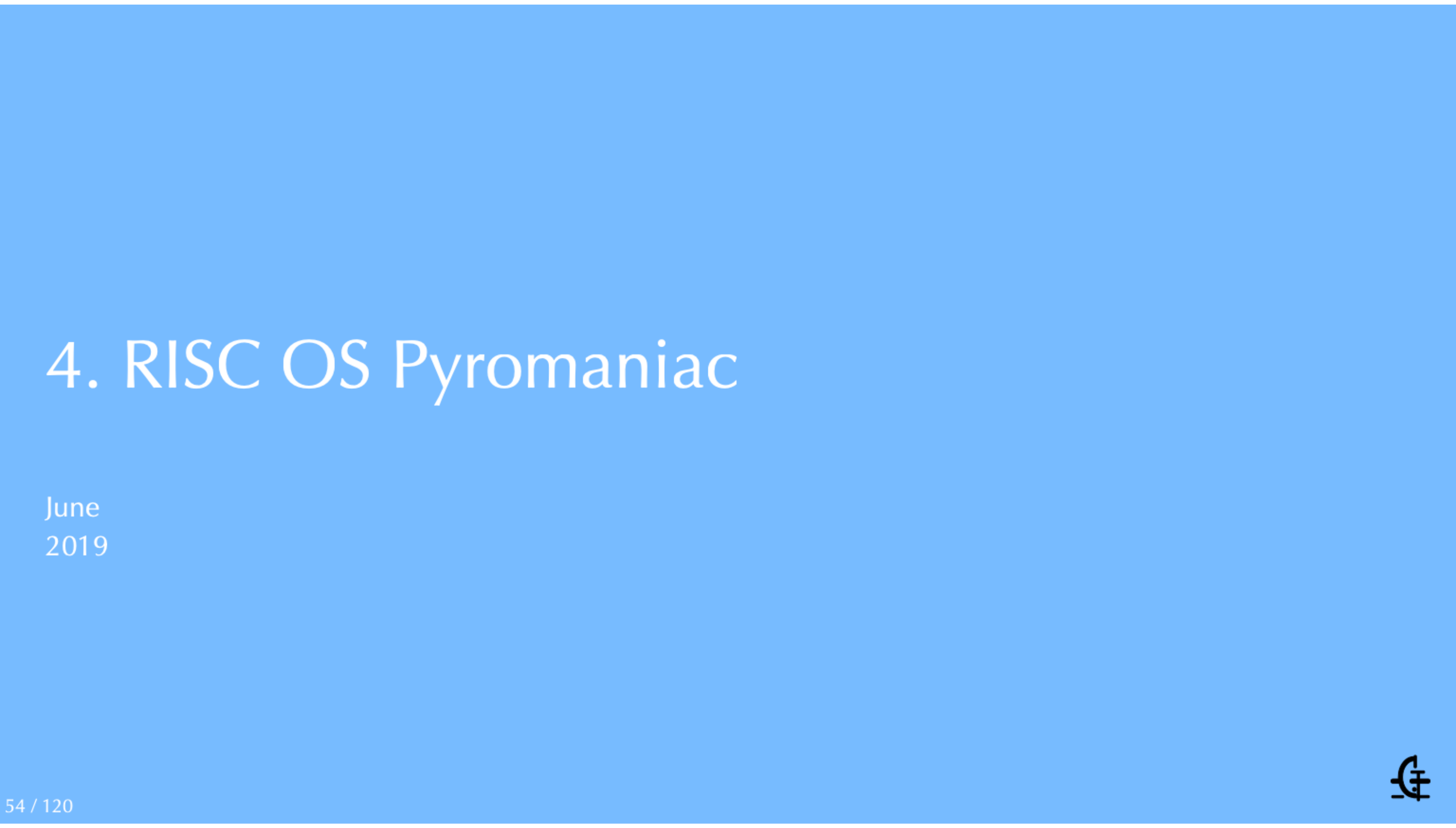
We're jumping back in time now - into that gap that I said we'd come back to.

I'd spent some time porting tools and making things build on macOS and Linux so that I could build a reasonable portion of the software that I had put into source control.

If you remember back in the 'Background' section, I talked about how important testing was. JFPatch-as-a-service is an amusing offshoot from the main goal of testing RISC OS things. It took about a month to create the service and site - just doing things in the evening after my day job.

The build client and automation came later, but the main service isn't that different to how it was originally released.

But the service was really just a bit of a distraction from the main work over the last year and a half.



4. RISC OS Pyromaniac

June
2019

54 / 120

4. RISC OS Pyromaniac

RISC OS Pyromaniac is the system that powers JFPatch-as-a-service. The name actually didn’t come until much later, but thanks to my friend Chris Johns for it, otherwise it might still be called `testrun`.

So let's talk a little about how it came to be...

RISC OS Pyromaniac

How do you test RISC OS software without RISC OS?

- My RiscPC is in storage.
- It's not good for testing.

RISC OS Pyromaniac

How do you test RISC OS software without RISC OS?

How do you test software without the operating system it runs on?
My RiscPC is all the way on the other side of town. It's not a good system anyhow - if something goes wrong, I'll have to reboot it that's bad for testing.
But...
...

RISC OS Pyromaniac

How do you test RISC OS software without RISC OS?

- My RiscPC is in storage.
- It's not good for testing.
- RISC OS was originally run semi-hosted from a BBC, using the BBC as the I/O and RISC OS as the main computer.
- That's what I want to be able to do - I want to be able to drive RISC OS from the CLI of a sane machine.

...

The early versions of RISC OS, when it was originally written, was semi-hosted from a BBC. The BBC does all the keyboard and disc and video, but the ARM system does the actual running of programs.

That's what I want to be able to do - I want to be able to drive RISC OS from the command line of a more sensible machine. In my case, the machine I use is my MacBook. Or my Linux server.

RISC OS Pyromaniac

Surely that's easy? (1)

- Surely that's easy? You just run an emulation system until it hits a SWI...
  ... and then you make the SWI do the I/O thing. Then you run some more?

RISC OS Pyromaniac

Surely that's easy? (1)

Surely that's easy? You just run an emulation system until it hits a SWI and then you make the SWI do whatever I/O it needs. Then you emulate some more?

...

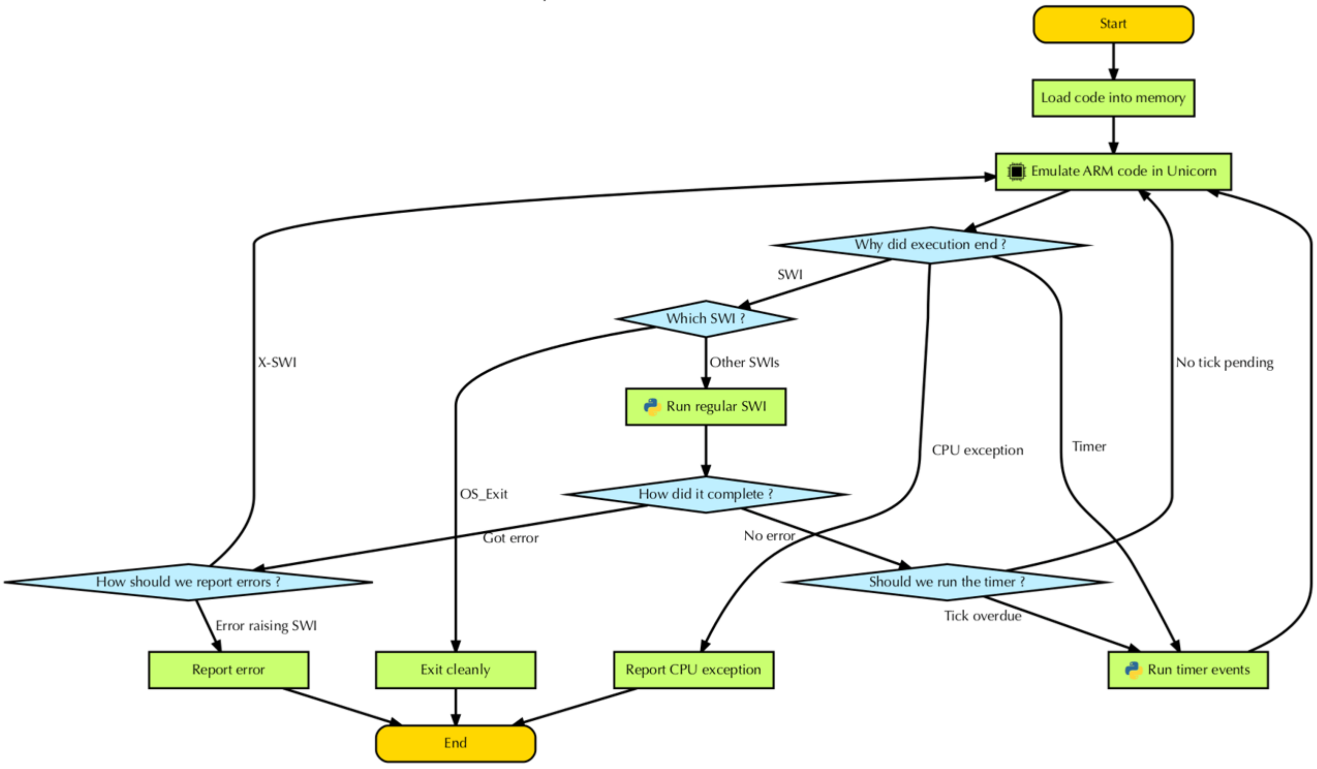
RISC OS Pyromaniac

Surely that's easy? (1)

- Surely that's easy? You just run an emulation system until it hits a SWI...
  ... and then you make the SWI do the I/O thing. Then you run some more?
- Yes - that's exactly what you do.
- The `IfThere` tool ran on June 10th - not well, but it ran.

...
Yup. That's the core of the Pyromaniac system.
The first thing it ran, was a tool that calls a few system calls to check if a file exists and runs a command if so - the `IfThere` command.
The 'I/O' part of the process is actually a pretty involved, so whilst I managed to get a tool running, what it ran wasn't actually that impressive.
The 2 SWIs that worked were OS_File 17 to read whether the file was there, and OS_CLI which just reported an error saying it wasn't supported. But it ran.
Lots has been developed since then.

Pyromaniac: Basic execution



RISC OS Pyromaniac

Surely that's easy? (2)

This is the basic execution sequence in Pyromaniac.
<Turn the pointer on with 'xc' to indicate things>

- You load some code.
- You emulate it until something causes you to stop.
- Decide why it ended - Timers don't do much, Exceptions just terminate, and SWIs decide where to dispatch to.
- SWI return has to handle errors in the way that RISC OS expects
  - if the SWI was error reporting, we need to exit with an error
  - if the SWI has the X-bit set, we just set the V flag and return back to our emulation.
- And repeat until you're done.

The devil is in the detail, so it's never quite that easy.
But let's not dive heavily into how it works; let's start from what RISC OS Pyromaniac actually is...

RISC OS Pyromaniac

What is RISC OS Pyromaniac?

- Pyromaniac is an alternative implementation of RISC OS for non-ARM systems.
- It is intended for use as a testing and prototyping environment which may be used during development and automated testing.
- Written in a high level language to make that possible.

RISC OS Pyromaniac

What is RISC OS Pyromaniac?

It's an alternative implementation of RISC OS. It doesn't use what went before in the core system. It can still run the ARM binaries, through emulation, but the OS is written in a high level language.
And that's so that it can be used to try things out, experiment and test. It's a lot easier in a high level language to replace an interface, or change the way that something works.
Similarly, it's a lot easier to reason about what the system is doing in a high level language. You're not bogged down in worrying about which flags are set, and what your registers hold.
And having been written from scratch, it's easier to make shortcuts for things you don't care about. There's a lot of the BBC OS heritage in there that you can just drop. Did you know that the VDU system can change the orientation so that characters proceed down the screen rather than across the screen? Did you know that the system scrolls text sideways if you do that? Do you care? That's something that you can drop when it's not part of the environment that you are trying to exercise.
If you want to try it, that's `VDU 23, 16`. And, actually, it's implemented in Pyromaniac as well. I just wanted an example of something obscure that most people would never have encountered.

RISC OS Pyromaniac

What's in a name?

- Pyromaniac is the system that runs ARM code.
- RISC OS is what's implemented on top of that.

So...

- A name to distinguish it from the ARM implementation.
  - RISC OS Pyromaniac
- A name for the ARM implementations.
  - RISC OS Classic

RISC OS Pyromaniac

What's in a name?

Strictly 'Pyromaniac' is the system for running ARM code in a useful way - that’s the ‘hardware’ if you like. 'RISC OS' is what's implemented on top of that, the whole being 'RISC OS Pyromaniac', but it doesn't really matter, and I refer to the whole as Pyromaniac.
You'll hear me refer to 'RISC OS Classic'; that's to distinguish the implementation of RISC OS. At one point I called it 'RISC OS Original', but that implies that it's been surpassed, which ain't the case!
‘RISC OS’ is the operating system interfaces. ‘RISC OS Classic’ is an implementation of those interfaces, and ‘RISC OS Pyromaniac’ is another implementation. Think of it like GNU/Hurd and GNU/Linux, if you know those systems.

RISC OS Pyromaniac

What makes up Pyromaniac? (1)

- Written in Python.
- Uses Unicorn (a QEmu derived package) for emulating 32bit ARM code.
- All other packages are optional.
  - Disassembly - needs `capstone`.
  - Graphics - needs `python-cairo`.
  - UI - needs `wxpython` or `gtk+3`.
  - Networking - more featured with `netifaces`.
  - Clipboard - interaction with system with `pyperclip`.
  - Sound - needs `python-rtmidi`.

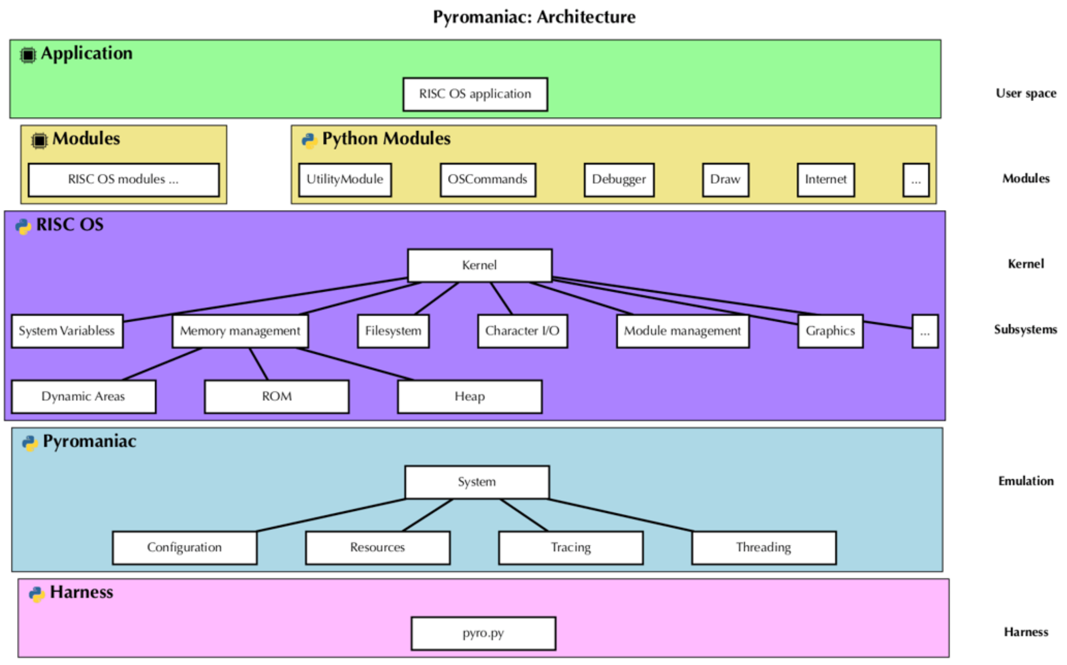
RISC OS Pyromaniac

What makes up Pyromaniac? (1)

If you haven't guessed, it's written in Python. It's largely modular internally, with subsystems separated into individual python files.
I use the Python Unicorn package to emulate the ARM code. I looked at a few different emulation solutions, but Unicorn is really nice.
Other than Unicorn, though, nothing else is required by the Pyromaniac system. If you need to run the UI, then you'll need GTK or Wx Python. If you want sound, you'll want the `rtmidi` system, and a MIDI synthesiser. But you don't /need/ them.
Without these modules, Pyromaniac will just track the current state. So if try to play a sound but the Python module isn't present, the call will say it was successful, but you won't get any sound. And if you're testing that the call works, that's good enough.

RISC OS Pyromaniac

What makes up Pyromaniac? (2)



RISC OS Pyromaniac

What makes up Pyromaniac? (2)

This is the general architecture of RISC OS Pyromaniac at a high level.
<Turn the pointer on with 'xc'>
The bit you run is the harness, which just uses the Pyromaniac and RISC OS interfaces to start the system.
Pyromaniac itself provides the setup of the execution environment, and support functions. This is the only part of the system that communicates with Unicorn. So, were a different emulation system to be required, this could be substituted here.
The Kernel, and many python modules, sit above this and provide interfaces to RISC OS and its subsystems. There's a separate python file for most of the subsystems.
By default Pyromaniac initialises only 1 RISC OS module - the UtilityModule. In this form, it's very limited, but that's what you want if you're providing a test and debug system. The other internal modules can be initialised by a single option, which brings in many of the common functions that you expect from the OS.
These Python RISC OS modules, which I've termed PyModules to distinguish them from the ARM implemented RISC OS modules, have almost all the functionality that their ARM counterparts have.
PyModules and ARM modules have exactly the same interface - internally they're both descendants of the same class, which means that to all intents and purposes they work the same way. However, instead of dispatching the RISC OS calls to Python as the PyModules do, they execute code through the ARM emulation.
And finally there's the application that you're running, which is always in ARM.

RISC OS Pyromaniac

How is it different from other systems?

RISC OS emulation:

- RPCEmu, ArcEm - Hardware emulators.
- Riscose - OS interface replacement.
- Amethyst - ARM unit testing tool.
- Linux Port - Hardware / interface replacement.

Other systems:

- Wine - OS interface replacement.
- Docker - System isolation.
- Rosetta - Dynamic recompilation.

RISC OS Pyromaniac

How is it different from other systems?

So how is it different from other systems?
Probably most people know of RPCEmu and ArcEm. These are hardware emulators - they emulate as much of the machine as they can, and if they do their job right, the OS and applications are oblivious. Most of the work for that goes in to getting the hardware emulation right. There's no interface emulation at all - the OS that is running is the original OS with no changes.
Pyromaniac doesn't provide any hardware emulation at all. Pyromaniac works at the higher level of the OS interfaces.
Riscose may be a dim memory for some people. It was a project to provide an emulation at the SWI level for RISC OS applications and use an emulation system to handle the ARM code. That's quite similar to what Pyromaniac does.
There's a Linux Port of RISC OS which uses the host system's interfaces to trap execution on an ARM system, and emulation on non-ARM systems. Some of the interactions with the host system are through special interfaces, but if I've understood correctly, that's the exception rather than the norm. It's a hybrid system that emulates a hardware environment, with some interfaces to the host system.
Last month we were introduced to Amethyst, an ARM code execution system for unit testing on non-ARM systems. Amethyst isn't an OS emulator, but is intended to be used to check that individual units of code work properly.
Amethyst is incredibly similar to Pyromaniac in its goal of being able to test code on alien system, is the same, and it's using a very similar model. However, with Pyromaniac I wanted to be able to test the real interaction within the system.
The difference is one of intent really - Amethyst aims to unit test, and Pyromaniac aims to System test. They actually complement one another rather well.
If you haven't come across Wine, it's an x86 interface layer for Windows applications.
You take your Windows application and you run it on Linux or macOS with Wine. The Windows application thinks it is talking to Windows because that's the interface it calls, but those interfaces get translated to native operations on the native system. Because it's running the x86 code on an x86 system, it's not emulation, but it is replacing all the interfaces with those that have the same effect as the application was expecting.
That's kinda similar to what Pyromaniac does, although Pyromaniac is providing those interface replacements at the system call level, not the linker level.
Docker is the most different in terms of what it's doing - it uses the Linux kernel isolation and capabilities interfaces to isolate the execution of Linux applications. It's only superficially similar to Pyromaniac.
Rosetta is a technology used on macOS systems. It allows you to run the prior CPU architecture's applications on newer systems. The original version was to cope with

RISC OS Pyromaniac

How does it compare to a bare Operating System?

Has many of the same things:

- Address space management; memory allocation.
- System calls from applications.
- Heap management.
- I/O management.
- Device drivers.

But some are missing:

- Page table management.
- Hardware interrupts.
- Memory mapped devices.

RISC OS Pyromaniac

How does it compare to a bare Operating System?

How does it compare to a bare Operating System?
Pyromaniac has many of the things you might expect in a regular operating system...

- It still has to worry about allocating memory, and taking calls from user space.
- It has to be able to handle I/O for those applications, whether it be keyboard, file system, sound or whatever.
- And it still has to have device drivers so that you can talk to the outside world.

But it doesn't have to worry about many of the things that a real OS does...

- You have no idea how glad I am to not have to worry about page tables, TLBs and cache coherency.
- Without real hardware, there's no need to provide interrupt masking, prioritisation, and bus discovery.
- And with no physical chips you don't have memory mapped devices to poke registers with. There's no clocks to set, no instruction set features to worry about.

I'm very happy to not deal with those.

RISC OS Pyromaniac

What does it mean?

- A command line only version of RISC OS.
- A RISC OS which runs 32bit ARM binaries, on Windows, macOS, or Linux.
- A reimplementation, which uses none of the code that went before.
- Focused on being able to test software and diagnose issues more easily.

RISC OS Pyromaniac

What does it mean?

What does that mean?

It's command line focussed so that you can run most command line tools. Remember, the goal of Pyromaniac is to be a testing and experimentation system, so the command line and core system interfaces are vital to that.

Because it's written in Python, and uses Unicorn which runs on different systems, Pyromaniac runs on macOS, Linux and Windows.

The core system is all new code written from scratch, although it doesn't cover all the OS. So when you run BASIC for example, you have to use the ARM version of BASIC. But the code that is the Operating System is all new.

The point of the system is to be able to test and find problems. There's a lot of configurability for the system, and a large number of options for debugging and tracing code.

And...

...

RISC OS Pyromaniac

What does it mean?

- A command line only version of RISC OS.
- A RISC OS which runs 32bit ARM binaries, on Windows, macOS, or Linux.
- A reimplementation, which uses none of the code that went before.
- Focused on being able to test software and diagnose issues more easily.

Tech: RISC OS Pyromaniac, able to run RISC OS programs on other systems!

...

Yes, I should acknowledge that this is technology that I've created. A pretty niche technology, but it makes me laugh. A lot.

RISC OS Pyromaniac

Command line only?

- Command line is the primary interface.
- Graphics implementations exist - either 'headless' or using a window showing the screen - but command line is where it excels.
- For testing, you largely want to be able to exercise things without UI interactions, at least for the lower level tests.

RISC OS Pyromaniac

Command line only?

I asked a friend about my insecurities in doing this talk. I was worried about people saying "why are you telling us about this?", or not caring about an online service.
They replied "Nope, they'll be thinking 'Does it run Impression?'"
No. it does not run Impression.
There's no Desktop. There's a lot of command line, although there is a graphics system. And you can have graphics tools working without a desktop relatively well.
When you're running tests, you're isolating the functionality and the interactions, so command line is just fine.

RISC OS Pyromaniac

No graphics, then?

Different parts of the system:

- VDU - VDU4, text output.
- Graphics - VDU5, OS_Plot, Draw, Font.
- Frame buffer - Bitmap of the screen.
- UI - How you see the VDU and Graphics systems.

RISC OS Pyromaniac

No graphics, then?

There is a distinction between parts of the system.

- There's the VDU system, which handles character output and VDU4 content.
- There's the Graphics system, which is involved with the OS_Plot calls, VDU5 output, Draw, Fonts and the like.
- There's the usually a frame buffer which is the bitmap of memory that holds that content.
- There's the user interface, which is how you see the VDU and graphics.

...

RISC OS Pyromaniac

No graphics, then?

Different parts of the system:

- VDU - VDU4, text output: **Well supported**
- Graphics - VDU5, OS_Plot, Draw, Font.
- Frame buffer - Bitmap of the screen.
- UI - How you see the VDU and Graphics systems.

...
The VDU system is pretty well supported. It can output as a regular console application in the host OS with plain text. Or you can use the 'ANSIText' implementation, which translates the RISC OS VDU codes into ANSI terminal codes. This means that if you have yellow text in RISC OS, it comes out yellow in the terminal.
...

RISC OS Pyromaniac

No graphics, then?

Different parts of the system:

- VDU - VDU4, text output: **Well supported**
- Graphics - VDU5, OS_Plot, Draw, Font: **Well supported, but no sprites**
- Frame buffer - Bitmap of the screen.
- UI - How you see the VDU and Graphics systems.

...

The graphics system is quite well supported too, and it looks pretty reasonable most of the time. There's still holes - for example GCOL actions other than plain plotting aren't supported yet. But by and large it works well enough for simple things to work.

...

RISC OS Pyromaniac

No graphics, then?

Different parts of the system:

- VDU - VDU4, text output: **Well supported**
- Graphics - VDU5, OS_Plot, Draw, Font: **Well supported, but no sprites**
- Frame buffer - Bitmap of the screen: **Nope**
- UI - How you see the VDU and Graphics systems.

...
Normally on RISC OS you have a frame buffer - a region of memory that holds the bitmap of what you plotted. That doesn't exist on RISC OS Pyromaniac. In fact there may not even be a frame buffer on the host side either. Although normally you would have the Graphics implementation which used a bitmap on the host side, it doesn't have to.
Change a configuration option and instead of using a bitmap the graphics system creates an SVG - a vector graphic representation of the screen.
...

RISC OS Pyromaniac

No graphics, then?

Different parts of the system:

- VDU - VDU4, text output: **Well supported**
- Graphics - VDU5, OS_Plot, Draw, Font: **Well supported, but no sprites**
- Frame buffer - Bitmap of the screen: **Nope**
- UI - How you see the VDU and Graphics systems: **wxWidgets and GTK**

...

Finally there's the user interface - how you see the VDU and graphics. VDU output, as I've mentioned, can come out on the console, but the graphics output can be configured to be displayed in either wxWidgets or GTK - those are two different frameworks for applications. Or it can be configured to not be displayed at all.

...

RISC OS Pyromaniac

No graphics, then?

Different parts of the system:

- VDU - VDU4, text output: **Well supported**
- Graphics - VDU5, OS_Plot, Draw, Font: **Well supported, but no sprites**
- Frame buffer - Bitmap of the screen: **Nope**
- UI - How you see the VDU and Graphics systems: **wxWidgets and GTK**

What works...

- The VDU system, and the graphics system work, mostly.
- VDU and graphics are complex so not everything works as it does in RISC OS Classic.
- Not all of it works as documented - after all not all of it is documented!

...
And what you have to remember is that the VDU and graphics systems are pretty complicated. Remember that example for character direction from earlier? Well, there's a whole lot more where that came from.

RISC OS Pyromaniac

How do you use it? (1)

Command line invocation:

```
charles@laputa ~/pyromaniac> ./pyro.py --load-internal-modules --command 'gos'
Supervisor

*fx0

Error: RISC OS 7.16 (03 Oct 2020) [Pyromaniac 0.16 on Darwin/x86_64] (Error number &f7)
*time
Fri,09 Oct 2020 23:00:04
*quit
charles@laputa ~/pyromaniac>
```

RISC OS Pyromaniac

How do you use it? (1)

Ok, an example of how you actually use it.
This example runs the harness with the internal modules, and runs the command line.
The `*FX 0` command shows the OS version. There's no way to report the host system, because RISC OS wasn't really meant for that, so I tacked it on to the OS version.
And then, as you can see, I can run other commands, and exit the system - which takes us back to the calling shell.

RISC OS Pyromaniac

How do you use it? (2)

Running RISC OS programs:

```
charles@laputa ~/pyromaniac> echo '10PRINT "Hello world"' > myprog,fd1
charles@laputa ~/pyromaniac> ./pyro.py --load-internal-modules --load-module
modules/BASIC,ffa --command myprog
Hello world
charles@laputa ~/pyromaniac>
```

RISC OS Pyromaniac

How do you use it? (2)

You can load modules at the command line, and run programs.
In this case, I'm creating a `Hello world` program in the file `myprog`, and running it.
The 'myprog' file has filetype `fd1`, which is represented in the standard format for non-RISC OS systems by using comma followed by the filetype. Type `fd1` is BASIC Text, which by default will run in BASIC.
On the command line I need to load BASIC before I can use it - it's not a part of RISC OS Pyromaniac. And you can see that it does print `Hello world`.

RISC OS Pyromaniac

Graphics demo!

RISC OS Pyromaniac

Graphics demo!

Time for a demo of the graphics system.

That's what you have been watching for the past hour or so. This presentation has been executed entirely on RISC OS Pyromaniac.

...

RISC OS Pyromaniac

Graphics demo!

Graphics features:

- Fonts.
- DrawFiles.
- Images.
- Screen bank flipping.
- Mouse pointer.

Others:

- Key input

...

So that's demonstrating...

- Firstly, that RISC OS Pyromaniac is a real thing.

It demonstrates...

- The Font system. This isn't the Classic FontManager; it's using fonts from the Cairo graphics system, which uses the host system's fonts. All the Font calls are implemented in the standard way using the Font SWIs. So it works just as it would on RISC OS Classic, albeit with different fonts.
- In the corner, you'll see my 'G' logo. That's a DrawFile, being rendered through ImageFileRender. The DrawFile module just calls the Draw module to plot the content, and the Draw module has been implemented in RISC OS Pyromaniac to talk to Cairo's path primitives.
- The pictures you've seen to now are all PNGs. Like Drawfiles, those are rendered through ImageFileRender, with a custom ImageFileRender module which talks to Pillow, a Python image library.
- Hopefully you've seen no flicker as we moved between slides, because we use separate screen banks to render to the offscreen buffer before displaying it.
- You won't have seen the flicker, but you may have seen the mouse pointer appear when I've pointed to things, or in the form of the hourglass 'cog'. They're just standard RISC OS pointer definitions, albeit the hourglass is implemented entirely in Python.
- And there's key input - I press keys to move through the slides, or turn the pointer on and off.

The entire presentation tool was written just so that I could do this presentation.

RISC OS Pyromaniac
Graphics demo! (presentation)

Tech:

- Slide presentation system.
- Markdown parser.
- FontMap for font remappings.
- WebColours module for colour parsing.
- ImageFileRender for general image rendering, using DrawFile for vectors.

RISC OS Pyromaniac
Graphics demo! (presentation)

A small digression before I talk more about Pyromaniac, there's a bunch of tech here that I've used or created.
The presentation tool I created for this talk will be open source.
It uses an open source markdown parser - all these slides are written in Markdown.
The Font selection is all through the FontMap module - which I created years ago to give an easy way to find variants of fonts. You ask it for a named font, but made oblique, and it'll do it.
The WebColours module was created for my SVG converter - give it a colour name and it gives you the colour. One job, does it well.
And I'm using ImageFileRender for the image rendering because... well, it does what it says in the tin.

RISC OS Pyromaniac

Features - What works?

- System - Runs 32bit modules, utilities and applications.
- Interaction - Interacts with the host as its primary interface
- Video - Pretty good VDU and graphics support GTK/WxWidgets, or snapshots of state.
- Sound - SoundChannels mapped through MIDI
- Filesystem - Host filesystem by default, using `,xxx` filename convention.
- Network - Internet module provides limited support for IPv4 and IPv6 networking.
- Compatibility - Many simple programs work, if their support modules are loaded.

RISC OS Pyromaniac

Features - What works?

Back to Pyromaniac and what it can do...

- System - We can run command line tools and modules. That meets most of the needs of my testing and CI goals.
- Interaction - You just use the keyboard as you might expect. In the UI, you can use the mouse, too.
- Video - We can draw things with the VDU interfaces; Draw and FontManager are provided. But there's no frame buffer - direct screen access doesn't exist. We also don't have sprites.
- Sound - SoundChannels is provided, mapped to host MIDI devices. Each of the standard voice names is mapped to general MIDI instruments to approximate their sound. The sound system can actually work more like the BBC sound system if you'd prefer - I implemented the queueing system like the BBC because I had forgotten it didn't work like that on RISC OS. So now you have a choice.
- Filesystem - You can access the local filesystem, so you can get the files you are testing into the system. Or you can load FileSwitch and access regular RISC OS filesystems.
- Network - The host network stack can be accessed from RISC OS, and we try to map Socket calls through - if the host has IPv6, `*IfConfig` will show you, and you can connect using those addresses.
- Compatibility - It's RISC OS, and whilst not all the interfaces have been implemented, many things do work. It depends on how complete those interfaces are. Sometimes you don't care. Generally support for given interfaces varies between the level of Arthur and that of RISC OS Select. The actual 'features' document is about 70K, and I'll make that available in the resources at the end.

RISC OS Pyromaniac

Features - What doesn't work?

- Desktop - Not supported
- Filesystems - No registration of filesystems
- Sound - No wave output
- Graphics - No frame buffer, No Sprites, No true colour modes
- Many other things

RISC OS Pyromaniac

Features - What doesn't work?

What doesn't work?

- Desktop - although we don’t support the desktop, certain interfaces are provided to appease clients that expect them to exist. TaskWindow and TaskManager exist purely for this purpose.
- Sound - there’s no support for 8bit or 16bit sound. Although after Jason's talk last week, I was inspired to implement SoundDMA on an experimental branch. I can now play sounds with WaveSynth through Pyromaniac.
- Graphics - Sprites are harder because they’re direct access, and I’ve not got around to implementing them yet. Paletted modes are also hard, but they’re the more likely case for getting things wrong, so I implemented them first. I don’t have 24bit modes yet. That said, the paletted modes are actually implemented in the 24bit system, so you don’t get colour cycling with the current implementation either. Or flashing colours. The text cursor doesn’t flash, either.

So many other bits are missing as well. Etierh because they are less used cases, or I’ve just not got around to doing them yet.

RISC OS Pyromaniac

Networking

- Internet module supplied, using host interfaces.
  - Supports `AF_INET`, `AF_INET6`, `AF_UNIX`.
  - Many ioctls are supported, mapped to the host system.
- Resolver module provides IPv4 host name resolution
- EtherPyromaniac provides a DCI4 driver.
  - Provides a virtual network.
- EasySockets, which bypasses Internet.

Tech: Tap-Tun JSON server for Ethernet frames.

RISC OS Pyromaniac

Networking

The Internet module supports IPv4 and IPv6, which allows you to connect to, or serve to hosts on the local network or on the Internet. The interface enumeration, and socket operations are extended to take `sockaddr_in6` structures as you would expect. It's actually not that different for most applications - `ifconfig` has been extended to be able to display IPv6 addresses.
The Resolver module can do lookups for names and addresses although currently it only supports IPv4 - it's not been updated for IPv6 yet.
On the other hand, you can load the Classic MBufManager and Internet modules, and then the `EtherPyromaniac` module becomes useful - this provides a DCI4 driver which can be configured into a virtual network.
This virtual network is actually just a set of frames transmitted as JSON over a TCP connection. On the other end of the connection could be another Pyromaniac, or a 'virtual hub'. The virtual hub can connect many Pyromaniac instances together, and even connects to a tap which talks to a real network.
EasySockets is also supported, although it actually bypasses the Internet module entirely, and just uses the host network system. As a result, I connected to IRC servers quite early on in the development using the test programs, and without realising it, I had connected through IPv6. That would almost certainly have been the first use of RISC OS with IPv6.

RISC OS Pyromaniac

Draw module (1)

- Draw module supplied.
- Can render through the Cairo path system.
- Classic DrawFile works - the 'Gerph' logo is a Drawfile.
- Classic Draw module can be used too.

RISC OS Pyromaniac

Draw module (1)

I've mentioned that I can render DrawFiles, and that it goes through the Cairo path rendering system. But it's also possible to use the Classic Draw module with DrawFile.
In case you don't know, the Draw module puts lines on the screen by calling the `hline` interface - you can find that through `OS_ReadVduVariables`. On Select, it also uses the `polyhline` interface to work faster on accelerated systems.
In Pyromaniac though, `hline` is implemented as a call to the Cairo line drawing. This allows the Classic Draw module to render to the screen. This demo doesn't use the Classic Draw module.
Obviously it's not just the simple 'G' that can be rendered...

RISC OS Pyromaniac

Draw module (2)

You are now entering
SHIT CREEK
Twinned with Johnathan Creek, England
~~PADDLE~~ ~~SALE~~ ~~TODAY~~
Cancelled
Sold Out

RISC OS Pyromaniac

Draw module (2)

... my brother drew this about 25 years ago. It still amuses me.

RISC OS Pyromaniac

FontManager

- FontManager module supplied.
- Can uses Cairo 'toy' fonts.
    - Can be configured to use any 'fontconfig' discovered fonts.
- Supports different alphabets, including UTF-8.

But also

- Classic FontManager works...
- ... if you disable bitmap generation - it just uses Draw.

RISC OS Pyromaniac

FontManager

I've mentioned the FontManager, but specifically it only uses the 'toy' interface that Cairo provides. At some point I'll look at using Pango to make it a lot more flexible, but it works pretty well for the simple things.
There's a few limitations, like it doesn't really support font control strings. And transforms aren't supported. But I'll probably get to them eventually. It does support encodings, though. If you request UTF-8 encoding for your font, it'll use that. This presentation uses UTF-8 for all its content.
The Classic FontManager works as well, but you have to turn off the bitmap caching by configuring the FontMax values to 0. If you do that, it just uses Draw directly for all the operations.

RISC OS Pyromaniac
Configuration

- RISC OS Pyromaniac is highly configurable - over 240 directly configurable options, in 59 groups.
- Configuration can be on the command line or in configuration files.
- Example:
  - `./pyro.py --config modules.rominit_noisy=true --load-internal-modules --command gos`
  - `./pyro.py --config memorymap.rom_base=90000000 --load-internal-modules --command modules`

RISC OS Pyromaniac
Configuration

Ok, back to some core features...

Because Pyromaniac is intended as a debug tool, it needs to be configurable. It is not a case of one-size-fits-all.

Sometimes those configurations make the system ‘safer’ by removing access to the host systeem - the service uses these features to limit the host system access.

The first example here turns on the noisy ROM initialisation - this is akin to the option of the same name in the RISC OS Classic Kernel. It prints out more information about the ROM initialisation as the machine boots.

The second example changes the location in memory that hosts the ROM, and then shows where the modules were loaded.

RISC OS Pyromaniac

Configuration files

```
%YAML 1.1
---
# Configuration for loading the ROM for RISC OS 5

debug:
  - modules
  - traceregionfunc
  - podules
  - swimisuse

config:
  podule.extensionrom1: ROMs/riscos5
  modules.rominit_noisy: true
  memorymap.rom_base: 0x8800000
  modules.unplug: extrom1:Podule,ParallelDeviceDriver,TaskWindow,SpriteExtend,SystemDevices,...

modules:
  internal: true
```

RISC OS Pyromaniac

Configuration files

But you can also give it configuration files. These make it easier to specify groups of options without long command lines.
This example is a part of the configuration file that I used for testing the booting of a RISC OS 5 ROM.
The format is YAML, which makes it relatively easy to read and write.
The configuration files make it a lot easier to use a group of settings reproducibly- which is especially important in testing.

RISC OS Pyromaniac

Tracing and debugging

- Trace features:
  - Report all instructions.
  - Report basic block execution, function entries.
  - Report SWI entry and exit conditions.
  - Function, memory and SWI traps.
  - Exception and API misuse reports.
- Debug features:
  - Most modules have debug available.
  - Can be enabled at runtime (`*PyromaniacDebug +<option>`).

RISC OS Pyromaniac

Tracing and debugging

The point of Pyromaniac is to make it easier to debug and test. It has a few different ways of reporting the progress of the code.
Tracing instructions is easy, although it slows down the execution considerably. Cheaper operations are to trace SWI calls, or to trap specific memory accesses. The instruction trace will also try to elide loops as well, so that thousands of repetitions of a tight loop don't fill megabytes in the log file.
There's copious debug available, so you can turn on the debug for the subsystem you're seeing problems with.

RISC OS Pyromaniac

Tracing code (1)

Tracing SWI arguments (`--debug traceswiargs`):

```
>GCOL 0, 255,192,0
 383f848: SWI     ColourTrans_SetGCOL
         => r0  = &00c0ff00   12648192  colour
            r3  = &00000100        256  flags
            r4  = &00000000          0  action
         <= r0  = &00000000          0  gcol
            r2  = &00000002          2  log2_bpp
            r3  = &00000000          0  corrupted
```

Tech: OSLib parser and templating system

RISC OS Pyromaniac

Tracing code (1)

It really depends on what you're trying to find out as to what type of tracing or debugging you might want. The example here enables tracing of the SWI arguments when a SWI is called. This debug option uses a built in table of names and types for the SWI parameters to show what those registers mean.
Here we see that the BASIC command to select the graphics colour has triggered a call to the ColourTrans module's SetGCOL system call. It shows what the parameters were on entry and exit.
The information on what those SWIs are was extracted from the OSLib def files - I wrote an OSLib `def` file parser which could extract the information into a form that we can use. The parser also generates templates for PyModules and other things. The parser is being made open source.

RISC OS Pyromaniac

Tracing code (2)

```
$ pyro testcode/bin/word_time_string --debug trace
 700013c: ADR     r1, &07000174             ; -> [&00000000, &00000000,
                                                &00000000, &00000000]
 7000140: MOV     r0, #&e                   ; #14
 7000144: MOV     r2, #0
 7000148: STRB    r2, [r1]                  ; R2 = &00000000, R1 = &07000174
 700014c: SWI     OS_Word
 7000150: SWI     OS_WriteS
 7000164: MOV     r0, r1                    ; R1 = &07000174
 7000168: SWI     OS_Write0
 700016c: SWI     OS_NewLine
Time string: Sun,06 Sep 2020 08:22:38
 7000170: MOV     pc, lr                    ; R14 = &04107fe0
 4107fe0: SWI     &FEED05
```

RISC OS Pyromaniac

Tracing code (2)

It's also possible to trace the instruction execution explicitly. In this mode the system is disassembling as it executes the ARM code. It can report on the registers and memory as it is running.
This means that it can include relevant information about what the registers and memory contains at the time of the execution.
This is one of the tests for Pyromaniac which exercises the reading of the current time as a string.
`SWI &FEED05` at the end is the return from the code being run to the operating system.

RISC OS Pyromaniac

Debugging

```
charles@laputa ~/demo> pyro --load-internal-modules --command gos --debug cli,clialias,osfscontrol
CLI: 'gos'
CLI alias: Wildcard 'Alias$gos' start read from None
Supervisor

*.
CLI: '.'
CLI alias: '.' expansion
CLI alias: Expanded to 'Cat '
CLI alias: Execute: Cat
CLI: 'Cat '
CLI alias: Wildcard 'Alias$Cat' start read from None
Catalogue directory '@'
Canonicalise filename '@' using pathvar 0L, path 0L
Read boot option of '$'
Dir. $ Option 02 (run)
Read directory 0
CSD  NoFileSystem:$too
Read directory 3
Lib. NoFileSystem:$
Read directory 2
URD  NoFileSystem:$
example/py  WR/WR     example/pyc WR/WR     wimperror   WR/WR
*
```

RISC OS Pyromaniac

Debugging

This example shows a couple of the subsystem debug options - the command line interpreter and the FSControl interfaces.
You can see the debug output interleaved with the program output. This shows what the system was doing at each stage.
The command line system is surprisingly complicated. I think I've got most of it right, but there's bound to be some differences.

RISC OS Pyromaniac

What is it like to work with? (1)

- The Pyromaniac context is usually `ro`, containing...
  - registers (`ro.regs[#]`)
  - memory (`ro.memory[address]`)
  - configuration (`ro.config['group.option']`)
  - resource (`ro.resource['resource']`)
  - methods for execution (`ro.execute`, `ro.execute_with_error`)
  - trace functions (`ro.trace`)
  - the kernel (`ro.kernel`)
- The Pyromaniac layer is all about the lower level execution and setup of the system.

RISC OS Pyromaniac

What is it like to work with? (1)

What is it like to work with?
I think the code is pretty well structured. Every function or method has access to the `ro` object, which is a reference to the root Pyromaniac object. This holds all the information for the system execution, including the Kernel object, which is where most internal resources can be accessed.
The Pyromaniac object also handles the configuration options, so that all components are configured through the same interfaces.
The whole system is able to be multiply instantiated - you can run different Operating System instances just by creating a new Pyromaniac object.

RISC OS Pyromaniac

What is it like to work with? (2)

- The RISC OS Kernel context is `ro.kernel`...
  - dynamic areas (`ro.kernel.da`, `ro.kernel.da_rma`. `ro.kernel.da_appspace`, ...)
  - vectors (`ro.kernel.vectors[#]`)
  - modules (`ro.kernel.modules`)
  - vdu and graphics system (`ro.kernel.vdu`, `ro.kernel.graphics`)
  - input and mouse (`ro.kernel.input`, `ro.kernel.mouse`)
  - filesystem (`ro.kernel.filesystem`)
  - system variables (`ro.kernel.sysvars[varname]`)
  - program environment (`ro.kernel.progenv`)
  - system APIs (`ro.kernel.api`)
- The Kernel object is always referenced explicitly from `ro`.

RISC OS Pyromaniac

What is it like to work with? (2)

The Kernel object holds all the subsystems. Some of those objects are simple - for example, the vectors object is really just an registration interface with a dispatcher. Others, like the Dynamic Areas, are much more complex, as they allow many different operations to be performed on them through the RISC OS APIs.
RISC OS components implemented in Python can call any SWI through the System APIs object. This ensures that the calls get processed as if they had been issued by the classic system.
Let's have some quick examples...

RISC OS Pyromaniac

What is it like to work with? (3)

```
"""
OS_ReadEscapeState implementation.
"""

from riscos import handlers
import riscos.constants.swis as swis

@handlers.swi.register(swis.OS_ReadEscapeState)
def swi_OS_ReadEscapeState(ro, swin, regs):
    """
    OS_ReadEscapeState

    <= C flag is set if an escape condition has occurred
    """

    regs.cpsr_c = ro.kernel.progenv.escape_condition
```

RISC OS Pyromaniac

What is it like to work with? (3)

This is the implementation of the OS_ReadEscapeState SWI. It's a very simple system call which just returns whether the Escape key has been pressed in a flag.
As you can see from the implementation, that's what it does - it updates the C flag in the processor state to be reflect the escape condition from the program environment.
The registers object handles the numbered registers, but here the processor flags are being updated. In ARM these processor flags live in a virtual register.
Within registers object, there's a property setter which allows the flags to be manipulated as booleans without having to extract them from the flags word.
The function implementation is registered through a decorator on the function. This remembers the function as the handler for a given system call.

RISC OS Pyromaniac

What is it like to work with? (4)

Many commands are just a thin wrapper around a system call:

```
def cmd_rmload(self, args):
    """
    Syntax: *RMLoad <module-file> [<args>]
    """
    self.ro.kernel.api.os_module(modhand.ModHandReason_Load, args)
```

RISC OS Pyromaniac

What is it like to work with? (4)

This is a method within one of the PyModules, which implements the command to load a new RISC OS module.
Like the prior example it is very simple. It literally just calls the underlying system call.
Notice that there's no error handling present. There doesn't need to be - if an exception is raised by the code, it will be caught by the caller, and turned into a RISC OS error.
The function would only need to do more work if it had resources it needed to clean up.

RISC OS Pyromaniac

What is it like to work with? (5)

Context handlers can be used to make memory allocation easy:

```
def cmd_time(self, args):
    """
    Syntax: *Time
    """
    with self.ro.kernel.da_sysheap.allocate(128) as time_string:
        time_string[0].word = 0
        self.ro.kernel.call_swi(swis.OS_Word,
                                rin={0: osword.OsWord_ReadRealTimeClock,
                                     1: time_string.address})
        self.ro.kernel.writeln(time_string.string)
```

RISC OS Pyromaniac

What is it like to work with? (5)

In ARM, you might allocate memory on the stack to store some temporary strings. That's not as easy in Pyromaniac - although it is possible - so it's simpler just to allocate some memory from a heap.

In this example, the `*Time` command needs somewhere to store the string it gets back before it can print it. A context handler allows memory to be allocated, and when the code block is exited, the memory will automatically be freed.

The result is written to the screen through the Kernel method `writeln`. The Kernel object can actually be used as an i/o output, which means it is very easy to use like any other file handle.

RISC OS Pyromaniac

What is it like to work with? (6)

Context handlers can also preserve the output state:

```
def cmd_show(self, args):
    """
    Syntax: *Show [<variable>]
    """
    # Preserve and enable VDU paging
    with self.ro.kernel.api.vdupaging():
        # Enumerate and print variables
        for varname, vartype, value in self.ro.kernel.api.os_readvarval_enumerate(args):
            if vartype in (sysvars.TYPE_STRING, sysvars.TYPE_MACRO):
                # String returned parameters should have their value escaped GSTrans style
                value = self.ro.kernel.gstrans.escape(value,
                                                   escape_chars='|"<' if vartype != sysvars.TYPE_MACRO else '',
                                                   escape_control=True,
                                                   escape_topbit=True)

            suffix = ''
            if vartype == sysvars.TYPE_NUMBER:
                suffix = '(number)'
            elif vartype == sysvars.TYPE_MACRO:
                suffix = '(macro)'
            self.ro.kernel.writeln('%s%s : %s' % (varname, suffix, value))
```

RISC OS Pyromaniac

What is it like to work with? (6)

This is the implementation of the *Show command. We use a context handler to ensure that whilst it's running we're in paged mode. That means that when the screen gets filled, you have to do something to let the output continue.

We enumerate through the variables that match the argument. Regular variables are printed with GSTrans escaping - the GSTrans python module supports both decoding and encoding - and then we construct a suffix for the type of variable that's present. Finally, we send the line to the VDU system for printing.

Not everything is simple - some things that are made easy in RISC OS don't have easy translations into the Python code. But mostly it's just a matter of implementing what you need, as you need it. Fortunately, I know RISC OS pretty well.

A couple of months ago, I found myself saying "Dear gosh, MessageTrans_EnumerateTokens is tedious" as I found I needed to implement it. And then 10 minutes later, I had an implementation working. It's a nice to find that something that's a bit tedious on RISC OS actually only takes 10 minutes to implement. That's not quite how I remember it being on RISC OS Classic.

RISC OS Pyromaniac

What is it like to work with? (7)

Exceptions can be trapped in a pythonic way:

```
def cmd_unset(self, args):
    """
    Syntax: *Unset <variable>
    """
    try:
        self.ro.kernel.api.os_setvarval_delete(args)
    except RISCOSError as exc:
        if exc.errnum != errors.ErrorNumber_VarCantFind:
            raise
        # Lack of a variable is not an error
```

RISC OS Pyromaniac

What is it like to work with? (7)

I'd mentioned that errors are handled by the caller, but sometimes you need to trap them. In ARM, you would call the SWI with the X-bit set and then check the error codes.
In Python, it's pretty much the same pattern, except that Python exceptions are used so checking the error code is pretty simple.

RISC OS Pyromaniac

What good is it? (1)

- Writing my own software.
  - This presentation tool.
  - Other older components.
- Trying things out.
  - Sound system.
- Debugging other people's software!
  - https://asciinema.org/a/345766 shows an interactive session demonstrating that freeing the stack you're currently using may have bad effects.

RISC OS Pyromaniac

What good is it? (1)

So it's an Operating System, but can you actually do anything with it? I wrote this presentation software with it, without touching any part of the Classic system.
Does this presentation tool work on RISC OS Classic? Yes. But it didn't work first go. The FontMap module that I use was built wrongly. I had misconverted one of the build tools to 64bit. The result was that loading the module on RISC OS Classic broke ResourceFS and MessageTrans - and because the system is dependant on them, that meant you had to reboot.
The presentation tool itself didn't quite work when run from the desktop. I'd forgotten that there's the Wimp Command Window pending that you have to clear, otherwise you get odd effects.
Then I came to run it on RISC OS 5 and ... it locks up the machine. I'd already fixed the FontMap module so the lock up was surprising. Clearly I was doing something wrong, but what? It turned out that if you select an out-of-bounds screen bank, RISC OS 5 gets upset. There was a bug in my code that meant I was selecting the wrong bank, which should have been a benign operation, but was actually hanging the machine.
So those are two major problems resulting in reboots during development. That backs up my original assertion that RISC OS is bad for testing. But could these have been found within Pyromaniac?
Firstly, if it did get into a bad state, it probably would mean killing the Pyromaniac process, but that's pretty much expected. However, the FontMap problem didn't trigger because I haven't got around to implementing ResourceFS yet. It would have been found then.
The bank selection problem probably wouldn't have been found unless I went looking for it - you see, I've implemented the screen bank selection as a hash of bank numbers. You use one and it comes into existance. There is bounds checking, but it didn't report a misuse. There is now a warning for that case.
What other things does Pyromaniac help with? There's the build service, which was just a fun side project. It's pretty functional, and it can build things pretty well.
And I get a chance to try things out that I wouldn't otherwise be able to.
I mentioned Jason Tribbeck's Sound System earlier. During his talk last month, I knocked up a prototype using his documentation to see how well it fits together - I only spent about 3 hours on it, but the result was some hopefully useful feedback. The source to that 3 hours work is linked at the end of the talk.
And then there's the chance to debug other people's software.
Julie Stamp released an early version of an Obey module that they'd written. I was able to try it out and see how well it worked in a different environment.
Not just a plain 'this is what happened', but an actual recording in Asciinema of it.

RISC OS Pyromaniac

What good is it? (2)

```
Supervisor

*cobey:obey echosed
...
==== Begin exception report ====
Exception triggered: Exception 'Prefetch Abort'
  r0  = &a9a9a9a9, r1  = &a9a9a9a9, r2  = &00000000, r3  = &a9a9a9a9
  r4  = &00000191, r5  = &07001f28, r6  = &07008ac0, r7  = &00000001
  r8  = &07008aa8, r9  = &07007720, r10 = &07008d18, r11 = &a9a9a9a9
  r12 = &a9a9a9a9, sp  = &a9a9a9a9, lr  = &a9a9a9a9, pc  = &a9a9a9ac
  CPSR= &60000010 : USR-32 ARM fi ae qvCZn
Recently executed code:
    ---- Block &07006f4c, 1 instructions ----
     7006f4c: LDR     pc, &07007230            ; = &0382095c
    ---- Block &0382095c, 5 instructions ----
     382095c: {DA 'ROM', module 'SharedCLibrary'}
    Function: memset
     382095c: MOV     r12, sp                  ; Function: memset
     3820960: PUSH    {r0, r11, r12, lr, pc}
     3820964: SUB     r11, r12, #4
     3820968: SUBS    r2, r2, #4
     382096c: BMI     &038209E4
    ...
    ---- Block &038209a0, 4 instructions repeated 229 times ----
     38209a0: STMIA   r0!, {r1, r3, r12, lr}
     38209a4: STMIA   r0!, {r1, r3, r12, lr}
     38209a8: SUBS    r2, r2, #&20
     38209ac: BGE     &038209A0
    ...
    ---- Block &038209d4, 6 instructions ----
     38209d4: SUBS    r2, r2, #4
     38209d8: STRLT   r1, [r0], #4
     38209dc: STMGEIA r0!, {r1, r3}
     38209e0: SUBGE   r2, r2, #4
     38209e4: ADDS    r2, r2, #4
     38209e8: LDMDBEQ r11, {r0, r11, sp, pc}
==== End exception report ====

Error: Internal error: Abort on instruction fetch at &a9a9a9a8 (Error number &80000001)
*quit
```

RISC OS Pyromaniac

What good is it? (2)

This is the trace when it went wrong. Unfortunately they freed the stack that they were executing from. Then they returned, and had a bad day because they returned to invalid memory.

Pyromaniac is not a reversible debugger - that's my day job - so I have to deal with the aftermath of the crash.

But that's fine, because I run the command again with block tracing enabled - and because the system is deterministic I get exactly the same result. I can see that it hits the memset function to clear the stack, and then dies. Where the call comes from is easy to trace from its address.

RISC OS Pyromaniac

Problems...

- IRQs and timed events aren't handled well.
- Execution context is split between emulated system and Python code.
- Error handling is still a bit troublesome.
- Replaced writing device driver, with writing interface modules.

RISC OS Pyromaniac

Problems...

Lots of stuff works, but there's stuff that doesn't, and stuff I still have to fix.

- As I mentioned previously, I don't have any interrupts, but timed events are still important. Parts of the network system will not work if they can't time out operations. The spinning hourglass you see now and then wouldn't work without timers. They're actually implemented with a check at the end of execution to see if anything is outstanding. It's tacky, but it's good enough.
- Whereas on Classic RISC OS the execution state would be entirely in the memory of the machine, in Pyromaniac it's split between Python and the emulated system. This is a problem that most emulation systems have, but it's worse because the OS state is also split between the two.
- My error handling is still not quite right. There's a design decision that I made early on that needs fixing. Because errors are handled as Python exceptions, I haven't differentiated between a signalling and a non-signalling error. It hasn't hurt yet, but it will.
- And whilst I'm not writing C or ARM code to poke hardware registers, I'm instead writing interfaces to Cairo, or SoundDevice, or GTK. It's replaced one problem area with another.

RISC OS Pyromaniac

Other technologies!

Tech:

- RISC OS Alphabets in Python Codecs - https://github.com/gerph/python-codecs-riscos
- Non-RISC OS editor syntax modes:
  - SublimeText syntax for RISC OS command files - https://github.com/gerph/sublimetext-riscoscommand-syntax
  - NanoRC syntaxes for some RISC OS file types - https://github.com/gerph/nanorc-riscos
- Tool for building hourglass modules - https://github.com/gerph/riscos-hourglass-maker
- Tests for RISC OS APIs, and a tool for testing tools - https://github.com/gerph/riscos-tests
- PRM-in-XML documentation system rework.
- Miscellaneous toolchain updates.
- Changelog management system - https://gitlab.gerph.org/gerph/changelog-management

RISC OS Pyromaniac

Other technologies!

In doing this, I've created a few technologies, some of which I called out as I went along, but there are others:

- There's a Python package for handling RISC OS alphabets as Python codecs.
- I created syntax colouring modes for the two editors I regularly use - SublimeText and Nano. And I'd mentioned earlier that I created the syntax mode for some RISC OS types in CodeMirror.
- I built some hourglass modules. Largely that was so that I understood how those hourglasses worked on RISC OS Classic before I implemented the one in Pyromaniac.
- To make it possible to test the compiler, I wrote a tool that allowed me to test that the tools work properly - it runs the tool, captures the output, compares to what's expected. It's an integration testing tool, but it works on RISC OS and on Linux/MacOS. For the toolchain it has support for parsing bits of the AOF/ALF/ELF files. It's Perl 5.0.0 compatible, intentionally so that you can run it with last Perl that I've got for RISC OS.
- I created a public repository of RISC OS tests, which uses this tool. The repository has a selection of the tests from Pyromaniac in it. However anyone could use them with RISC OS Classic, if they wanted, or people could contribute new test code that exercises other parts of the system.
- To document some of the things that I've been working on, I revisited the PRM-in-XML documentation, and updated my systems to built documents like that.
- And I created a changelog management system because I was fed up with dealing with conflicts when I tried to merge different branches.

RISC OS Pyromaniac

What does it run on?

- macOS (console, GTK, wxWidgets)
  - Also a dedicated application.
- Linux (console, GTK, wxWidgets)
  - Also within a docker container.
- Windows (console, wxWidgets) [native and under Wine, also docker wine-py]
  - Also a dedicated application.

RISC OS Pyromaniac

What does it run on?

I've done this presentation on macOS using the wxWidgets UI. But it runs just as well on Linux and Windows.
I've built applications that you can run on macOS and Windows, and I've built docker containers with the OS inside.
JFPatch-as-a-service is a customised docker container. It has JFPatch and other tools in it. I have a private registry with my docker containers in, and the CI system updates and pushes to the registry so that they're usable by in other tests.
Automatic testing doesn't happen on Windows yet.

RISC OS Pyromaniac

“Releases”?

- Released once a month, just to myself.
  - October's version is 0.16.
  - Releases are a way to stop it being unusably 'half finished'.
  - Releases are a great incentive - I really have achieved a lot this month!
- Long lived development, for example...
  - Font Manager lived on a branch for about 6 months.
  - EasySockets is still on a branch.
  - PyromaniacGit is still be worked on.

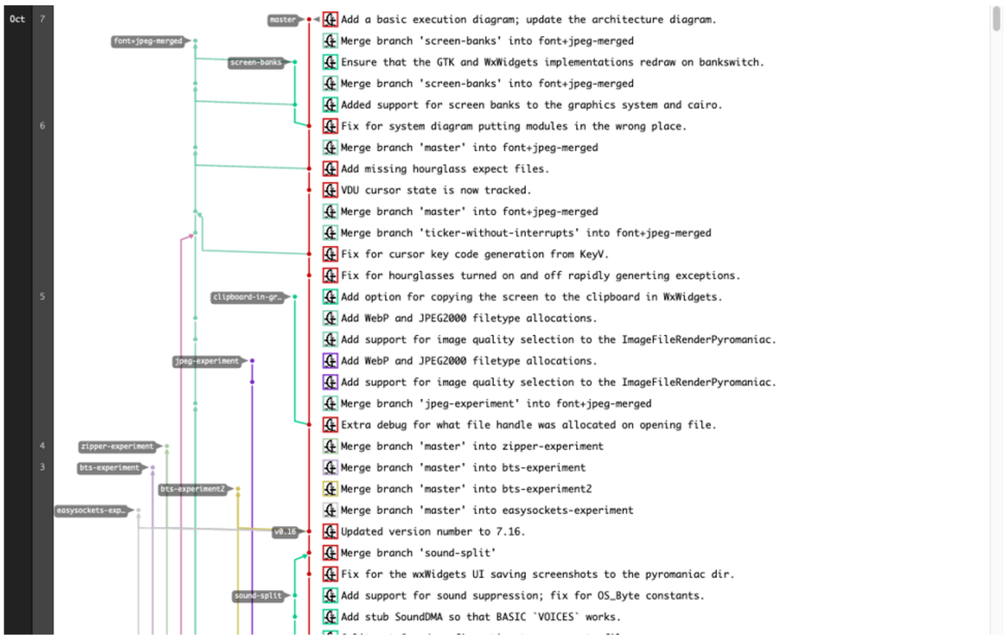
RISC OS Pyromaniac

“Releases”?

Once a month I go through a release process - updating the features documentation, the main product documents, do manual tests on the different platforms, update the version number, and so on.
If I’m only doing it for myself, why bother? It’s another thing that helps me feel good about what I've done - I can see what I’ve achieved.
There are some branches that have hung around for a while, because they're ok, but they don't make the cut for release. I have tasks on my board to review them, and raise issues when I find something is wrong.
There's a PyromaniacGit branch that's lets you use the host git client with RISC OS files; it's been there since June and still needs a lot of work.
...

RISC OS Pyromaniac

“Releases”?



```
Add a basic execution diagram; update the architecture diagram.
Merge branch 'screen-banks' into font+jpeg-merged
Ensure that the GTK and WxWidgets implementations redraw on bankswitch.
Merge branch 'screen-banks' into font+jpeg-merged
Added support for screen banks to the graphics system and cairo.
Fix for system diagram putting modules in the wrong place.
Merge branch 'master' into font+jpeg-merged
Add missing hourglass expect files.
VDU cursor state is now tracked.
Merge branch 'master' into font+jpeg-merged
Merge branch 'ticker-without-interrupts' into font+jpeg-merged
Fix for cursor key code generation from KeyV.
Fix for hourglasses turned on and off rapidly generting exceptions.
Add option for copying the screen to the clipboard in WxWidgets.
Add WebP and JPEG2000 filetype allocations.
Add support for image quality selection to the ImageFileRenderPyromaniac.
Add WebP and JPEG2000 filetype allocations.
Add support for image quality selection to the ImageFileRenderPyromaniac.
Merge branch 'jpeg-experiment' into font+jpeg-merged
Extra debug for what file handle was allocated on opening file.
Merge branch 'master' into zipper-experiment
Merge branch 'master' into bts-experiment
Merge branch 'master' into bts-experiment2
Merge branch 'master' into easysockets-experiment
Updated version number to 7.16.
Merge branch 'sound-split'
Fix for the wxWidgets UI saving screenshots to the pyromaniac dir.
Add support for sound suppression; fix for OS_Byte constants.
Add stub SoundDMA so that BASIC `VOICES` works.
```

…
I've not had a picture in quite a few slides, so these are the git branches I'm juggling. It shows the branches that I'm wrangling whilst I get Fonts and JPEG rendering to work together for this presentation.

5. Conclusion

106 / 120

5. Conclusion

Ok this concludes the section on Pyromaniac, but I'm not going to take questions just yet.
There's just one short section to finish up

Conclusion

Have I done what I set out to do?

Let's review what I saw as problems...

- Development on RISC OS is tedious
- RISC OS testing is awful
- RISC OS is awful for testing

Conclusion

Have I done what I set out to do?

If you remember back in the Background section, I said there were a few things that I set out to deal with. Let's look at each.

Conclusion

Development on RISC OS is tedious

- Source control
- Cross compiling
- Managed development environments
- Automated testing
- Feature and regression testing
- Fleets of systems available

Conclusion

Development on RISC OS is tedious

These are the things that I saw as features of modern development. Did I manage to achieve those goals?
...

Conclusion

Development on RISC OS is tedious

- Source control - ***yup, using GitLab, PyromaniacGit***
- Cross compiling
- Managed development environments
- Automated testing
- Feature and regression testing
- Fleets of systems available

...
(source control) - Yup, everything's in source control and I can access it.
...

Conclusion

Development on RISC OS is tedious

- Source control - yup, using GitLab, PyromaniacGit
- Cross compiling - ***yup, Linux and macOS***
- Managed development environments
- Automated testing
- Feature and regression testing
- Fleets of systems available

...
(cross compiling) - Yup, I can build RISC OS components on macOS or Linux.
...

Conclusion

Development on RISC OS is tedious

- Source control - yup, using GitLab, PyromaniacGit
- Cross compiling - yup, Linux and macOS
- Managed development environments - ***yup, docker, artifactory, applications***
- Automated testing
- Feature and regression testing
- Fleets of systems available

...

(managed development environments) - yup, I've got the multiple ways of controlling my builds.

...

Conclusion

Development on RISC OS is tedious

- Source control - yup, using GitLab, PyromaniacGit
- Cross compiling - yup, Linux and macOS
- Managed development environments - yup, docker, artifactory, applications
- Automated testing - ***yup, build.riscos.online, and GitHub and GitLab builds***
- Feature and regression testing
- Fleets of systems available

...
(automated testing) - yup, externally I created the JFPatch-as-service site, and internally I use CI on most things.
...

Conclusion

Development on RISC OS is tedious

- Source control - yup, using GitLab, PyromaniacGit
- Cross compiling - yup, Linux and macOS
- Managed development environments - yup, docker, artifactory, applications
- Automated testing - yup, build.riscos.online, and GitHub and GitLab builds
- Feature and regression testing - ***yup, thousands of tests, some public***
- Fleets of systems available

...
(feature and regression testing) - yup, I have thousands of tests.
...

Conclusion

Development on RISC OS is tedious

- Source control - yup, using GitLab, PyromaniacGit
- Cross compiling - yup, Linux and macOS
- Managed development environments - yup, docker, artifactory, applications
- Automated testing - yup, build.riscos.online, and GitHub and GitLab builds
- Feature and regression testing - yup, tests for the OS, and code coverage
- Fleets of systems available - ***well, no, not yet***

...
(fleets of systems available) - no, but mostly because I don't need to; I have a hefty server that can run multiple builds at once, and I've not had a need for more.

Conclusion
RISC OS Testing is awful

- RISC OS Pyromaniac has tests - about a thousand at present.
- Tests take about 18 minutes to run - and run on Linux and macOS in parallel.
- Code coverage hovers at around 65%.

Conclusion
RISC OS Testing is awful

You remember I said that RISC OS testing is awful? Well it is for RISC OS Classic.
But RISC OS Pyromaniac has tests for many of its interfaces. And runs them on multiple platforms. On each push.
Each of those 1000 tests starts RISC OS, executes the test and then exits it - so each environment is clean, and you're only testing the things you're interested in. Doing that with RISC OS Classic would be impractical - it's just not able to do that quickly and reliably.
I'm essentially booting RISC OS thousands of times per day to test its behaviour doesn't get worse. There aren't enough tests - there never are - but I try to make sure that at least some feature tests exists for each major change that happens.
And the code coverage for those tests is pretty decent too. Some things will always be hard to test, but it's looking good.

Conclusion

RISC OS is awful for testing

- Clarification: RISC OS Classic is awful for testing.
- Heavily used as part of the development of the `present` tool.
- JFPatch itself is tested.
- BASIC module has some tests that run programs.

Conclusion

RISC OS is awful for testing

The other part of the statements about testing was that RISC OS is awful for testing. After all, you may remember that the stated goal - the whole reason for starting doing this - was that I wanted to test things in a sensible way.
RISC OS Pyromaniac makes it possible to test in a less awful way. The environment can be controlled. Errors can be trapped. Execution time can be limited. And you can restart it.
JFPatch wasn't that bad to put through its tests. There are still some things to be improved, but it was important that I could see that I wasn't breaking things when I added the 32bit support. So I added tests first that it could parse the files, and then added the 32bit code. And then I could actually exercise some of the 32bit code on Pyromaniac.

Conclusion

Could it be better?

- More APIs.
- Better handling of corner cases.
- Sprites (sigh).
- Back Trace Structures.
- Finish the pending branches - Windows, Zipper, EasySockets, Git, DCI4, ...
- Using it for actual testing - that was what it was for!
- So many other opportunities.

Conclusion

Could it be better?

Could it be better? Well, of course!
There are huge swathes of things that I've not talked about here - subtleties, things that work, things that don't, ideas that might never pan out but would be cool.
I could talk for an hour on any part of RISC OS Pyromaniac, and there's so much more for me to do.
But I try not to feel too guilty about that, because the whole point of this was that I could do something I enjoy.

Conclusion

References

If you're wanting to know more, or review this talk, a site, https://pyromaniac.riscos.online/ has been created which contains support materials:

- Copies of these slides.
- Links to the technologies in these slides.
- Explanations of the CI examples.
- Development images and screenshots.
- Documentation from Pyromaniac (features, changelogs, configuration info).

There's also a demonstration site: http://shell.riscos.online/.

Conclusion

References

You'll find some reference material, including these slides, on pyromaniac.riscos.online. Feel free to try out the shell server too.
But there's one more thing that I set out to do when I started all this work...

Conclusion

Am I happy?

You can make whatever judgements you like!

Conclusion

Am I happy?

The great and liberating thing about all this is that none of it matters one jot whether it happens.
The only thing that matters is whether I feel it would be fun and it would make me happy to do it. That’s the thing that I learnt a year and a half ago, and that is what powers the service that you see - my happiness.
<pause>

6. Questions

Info site: https://pyromaniac.riscos.online/
Shell: http://shell.riscos.online/

6. Questions

That is the end of the planned talk… So I'll take questions now.