Building a *Free Pascal* Cross Compiler for the Sinclair QL

Norman Dunbar

5th April 2021

Contents

1 Introduction

I already had the *Free Pascal Compiler* (FPC) installed on my Linux laptop, however, I don't use it that often – I'm not very good at remembering how to write Pascal code – so I decided that for this experiment in getting the bare bones of the Sinclair QL version of the *Free Pascal Compiler* I would set up a brand spanking new VM running the same Linux version as my laptop. This is Linux Mint 20.1, the 64 bit version.

The process of setting up the VM will not be discussed here, normally I would have used *VirtualBox* but as I had just done my 10 yearly wipe and refresh of the laptop, I hadn't yet installed it and I wanted to try out the `libVirt` system using *QEMU* and the kernel's KVM *stuff*[1].

The upshot of all this is, I had made things a little more complicated than they needed to be, but that's how I wanted it. If I messed things up really badly, I could easily wipe the VM and start again with little or no problem, whereas if I tried to do everything on the laptop, and messed up, I might need to be sorting things out for a while to get back to where I was. Experiments don't always work out fine!

1.1 Code Conventions

1.1.1 Line Numbers and Continuations

There are three main listing types in this document:

- Commands typed at a prompt, either the system prompt or an application's prompt;
- Output from commands;
- Text files, configuration files or source code files.

In the document, listings showing command lines will have line numbers. Hopefully you will see that the numbers are outside of the code block in the document, and you don't have to type in the numbers. Keep an eye out for long lines which have wrapped around. They will not have line numbers on the continuation lines and those will be indented to show that they are continued from above. This is a (contrived) example of a set of commands to be typed at the prompt:

[1] That's a technical term.

```
1 echo "This is a short line."
2 echo "This is a long line of what should be code and it is
     all on a single line, but has wrapped around."
3 echo "This is another short line."
```

Output from commands will be shown thus:

```
Hello World!
```

There will not be any line numbers, unless absolutely necessary, and long lines will be split and indented as before.

Source code, text or configuration files will be shown in this style:

```
1 #IFDEF CPUM68K
2 -Fu/where/I/have/installed/the/compiler/lib/fpc/$fpcversion/
     units/$fpccpu-$fpcos
3 -Fu/where/I/have/installed/the/compiler/lib/fpc/$fpcversion/
     units/$fpccpu-$fpcos/*
4 #IFDEF SINCLAIRQL
5 -FD</path/to/vasm-and-vlink>
6 -XPm68k-sinclairql-
7 #ENDIF
8 #ENDIF
```

Line numbering will usually be present and long lines will be split and indented.

1.1.2 Privileges

Unless otherwise noted, all commands will be executed as my local user, *norman*, and not as *root*. I may need to obtain root privileges from time to time, and to do that, I'll prefix the appropriate commands with `sudo` rather than logging in as the root user. When using `sudo`, you are initially prompted for a password and that password will be cached for a short period of time. During this time, any other `sudo` commands will not prompt.

1.1.3 Free Pascal Compiler Versions

As I worked on this document, there were many changes made to the sources of the *Free Pascal Compiler*, this included the occasional version number change. I started at version 3.2.0 and at some point, it had risen to 3.3.1. To avoid confusion, I have avoided hard coding the version numbers into paths where it is included. Those file and directory names will be noted as "n.n.n" regardless of the version in question.

1.2 Building the Development VM

The plan of action, after installing the VM and Linux, is to:

- Install any development software required.
- Install the *Free Pascal Compiler* (FPC) for Linux 64 bit.
- Install the FPC Source Code.
- Install and build the required assembler, *vasm*.
- Install and build the required linker, *vlink*.
- Build and install the QL version of FPC.

That would, hopefully, give me a working cross compiler so that I could try and write QL programs, compile them under Linux, copy them to my QL then test them. Easy stuff, no? My own QL system is Marcel Kilgus' excellent *QPC*[2] so I'm able to run the cross compiled programs without any problems. However, if you are using an actual QL, then you may need to compile the Pascal programs using the option to add an XTcc trailer record, and use my little `XTcc_bin` utility[3] to convert the files to executable on the QL.

As I worked through the experiment, it became obvious that some work would be needed on the existing, brief, system Unit for the QL, so I will also need to be able to edit and recompile the Sinclair QL Runtime Library (RTL) for FPC.

[2] *https://www.kilgus.net/qpc/*
[3] Issue_6 of my eMagazine has all the details you may need.

2 Development Software

Once the VM is created and Linux Mint installed, we need to install the software tools and packages which will allow us the ability to build the cross compiler.

2.1 Installing the Software

The following software is the minimum required to build the cross compiler and build Pascal programs for the QL:

- *Subversion*; the version control system used by the FPC developers. We need this to be able to download the source code for the compiler, and to keep the software up to date. We could avoid this step and just download a zip file, if necessary. *Subversion* comes in handy when changing the RTL as it allows a patch kit to be created to update the source code for others to use.
- *Build-essential*; a package on Mint that installs the various compiler tools and libraries necessary to compile stuff[1].
- The *ssh server*; to allow me to connect a terminal session from my laptop to the VM to do development work, and to copy down compiled programs.
- *Git*; which is not strictly necessary, but I use it myself and I wanted it installed, just in case. Feel free to leave it off if you don't use it.

```
1  # Update the Mint software database.
2  sudo apt update
3  [sudo] password for norman:
4  ... lots of output here.
5
6  # Patch the Mint system.
7  sudo apt upgrade
8  ... lots of output here.
9
10 # Install development tools and packages.
11 sudo apt install build-essential subversion git
12 ... lots more output here,
```

[1] Another technical term!

As I mentioned, I need to install and enable the ssh server, so that it runs now, and at startup:

```
# Install SSH server.
sudo apt install openssh-server
... More output here.

# Start ssh server now.
sudo systemctl start ssh

# Start ssh server at system startup.
sudo systemctl enable ssh
```

I need the current IP address of the VM to allow me to use `scp` to copy the file to my laptop from the VM, and upload it to QPC:

```
# What's my VM's IP address?
hostname -I
```

```
192.168.2.225
```

The above list sets up the VM ready to compile code. In addition to the above, I also need:

- To install FPC for the Linux host; the compiler is used to compile a Sinclair QL cross compiler version of itself, so it is required to be present.
- To build the QL version, we also need to install the source code for the compiler.

3 Installing the Host Compiler

The "host" is my Linux VM. It requires that a version of FPC be installed to compile and run programs for Linux 64 bit systems. Once installed it will be used to build the QL specific version of FPC that will run on the host, but create executables for the "target", the QL. Cross compiling can get a little bit confusing at times.

3.1 Download FPC

Point your favourite browser at https://www.freepascal.org/download.html and on that page, scroll down to where you find your particular system. Mine is X86-64 Linux, so I clicked that link.

On the next page, select a mirror – I used Source Forge – and click the link. If you click on one of the other mirrors, you get different options. Source Forge is *amusing* in that you have to choose your required version, again. There are quite a few and some are pre-build cross compilers, so be careful in what you choose. I required `fpc-n.n.n-x86_64-linux.tar` and that just happens to be the version that Source Forge decided was ideal for me and selected it as the "Download Latest Version" option, right at the top in a big green box.

Let the download run, it's about 85 Mb, and when completed, make a note of where you saved it, then in a file explorer session, navigate to the downloaded file, and extract it. Once extracted, it's a simple case of making sure that the file `install.sh` is executable, and running it. The extraction process is as follows:

```
1 # Switch to downloads and extract the compiler.
2 cd ~/Downloads
3 tar -xvf fpc-n.n.n-x86-64-linux.tar
4 ... Some filenames whizz by here
```

3.2 Installing FPC

```
1 # Install the compiler.
2 cd fpc-n.n.n-x86_64-linux
3 chmod ug+x install.sh
4
```

```
5 sudo ./install.sh
6 [sudo] password for norman:
```

The first prompt from the installer is for a location to install FPC and the RTL. This location should be on your path which invariably means that you will need root privileges, which is why we need use `sudo` to run the installer. I chose `/usr/local` as my install location, as follows:

```
This shell script will attempt to install the Free Pascal
   Compiler
version n.n.n with the items you select

Install prefix (/usr or /usr/local) [/usr] :  /usr/local

Installing compiler and RTL for x86_64-linux...
... Lots more here.

Running on linux
Write permission in /etc.
Writing sample configuration file to /etc/fpc.cfg
Writing sample configuration file to /usr/local/lib/fpc/n.n.n
   /ide/text/fp.cfg
Writing sample configuration file to /usr/local/lib/fpc/n.n.n
   /ide/text/fp.ini
Writing sample configuration file to /etc/fppkg.cfg
Writing sample configuration file to /etc/fppkg/default

End of installation.

Refer to the documentation for more information.
```

You may wish to make a note of those locations for the various configuration files, in case you need or want to change things. They should be fine for the host system – I didn't have to change mine.

After installing the compiler, we are then prompted to install the documentation and demonstration files. I chose not to, but if you wish to do so:

```
Install documentation (Y/n) ? Y
Installing documentation in /usr/local/share/doc/fpc-n.n.n
   ...
Done.

# For the demos location, just press ENTER.

Install demos (Y/n) ? Y
```

```
Install demos in [/usr/local/share/doc/fpc-n.n.n/examples] :
Installing demos in /usr/local/share/doc/fpc-n.n.n/examples
   ...
Done.

... Lots of extra text here.
```

3.3 Testing the Host Compiler

We now need to make sure that the compiler works. I created myself a `SourceCode` directory tree for all my source code, then changed into it ready for action:

```
1 mkdir -p ~/SourceCode/Pascal
2 cd ~/SourceCode/Pascal
```

Create the following `hello.pp` file:

```
1 program hello;
2 const
3     helloText = 'Hello FPC World!';
4
5 begin
6     writeln(helloText);
7 end.
```

Compile the test code:

```
1 fpc hello.pp
```

```
Free Pascal Compiler version n.n.n [2020/06/09] for X86-64
Copyright (c) 1993-2020 by Florian Klaempfl and others
Target OS: Linux for i386
Compiling hello.pp
Linking hello
8 lines compiled, 0.2 sec
```

Looks good, now test it:

```
1 ./hello
```

```
Hello FPC World!
```

So far the host compiler is looking good and can at least compile a simple Pascal program. We are ready to use FPC to rebuild itself as a Sinclair QL cross compiler.

3.4 Upgrading the Host Compiler

Form time to time, the host compiler will be updated by the Free Pascal Project. If you absolutely have to have the very latest version, then the instructions in this section are the ones to follow to upgrade the compiler.

You cannot upgrade using your distro's Package manager's commands – `apt`, `yum` etc – as you didn't install from your package manager. It must be upgraded manually.

Reapplying the install instructions will overwrite the existing files and upgrade the host compiler in place.

4 Installing the Compiler Source Code

Once we have the host compiler working, we can get hold of the compiler's own source code and use the host to build a cross compiler for the Sinclair QL. The first step is to grab the source code and to do this we need to use *subversion*.

```
1 cd ~/SourceCode
2 svn checkout https://svn.freepascal.org/svn/fpc/trunk fpc
3 ... lots and lots of stuff scrolling past here!
```

These commands will create a new directory, `SourceCode/fpc`, then checkout the main trunk of the FPC source code into the new directory.

5 Building the Cross Compiler

5.1 Build the Assembler and Linker

We need a certified assembler and a linker first. We must use the *vasm* assembler and the *vlink* linker, or things won't work. The two URLs where the source code for these utilities is to be found, are:

- http://sun.hasenbraten.de/vasm/index.php?view=relsrc
- http://sun.hasenbraten.de/vlink/index.php?view=relsrc

You need to get the latest sources and *vasm* version 1.8 or higher, and *vlink* version 0.16h are required.

In Linux, I was able to use the `wget` command, but it's a simple task to open the two URLs above, and click the link to download the source files. However you do it, it's probably wise to save the files into your `SourceCode` directory along with all the other source we are building.

```
1 cd ~/SourceCode
2 wget http://sun.hasenbraten.de/vasm/release/vasm.tar.gz
3 ...
4 tar -xvzf vasm.tar.gz
5
6 wget http://sun.hasenbraten.de/vlink/release/vlink.tar.gz
7 ...
8 tar -xvzf vlink.tar.gz
```

Now we need to extract and compile both utilities and copy the executable into a location on the path, first *vasm*:

```
1 tar -xzf vasm.tar.gz
2 cd vasm
3 make CPU=m68k SYNTAX=std
4 ...
5 sudo cp vasmm68k_std /usr/local/bin/
6 cd ..
```

The build process will create an assembler binary named `vasmm68k_std`.

Next, the *vlink* linker:

```
tar -xzf vlink.tar.gz
cd vlink
make
...
sudo cp vlink /usr/local/bin/
cd ..
```

5.2 Rename the Assembler and Linker

After this, both files must be either renamed or sym-linked (on Linux) as follows. If you are on Windows then a rename should be sufficient.

- The assembler must be named `m68k-sinclairql-vasmm68k_std`.
- The linker must be named `m68k-sinclairql-vlink`.

```
cd /usr/local/bin
sudo ln -s vlink m68k-sinclairql-vlink
sudo ln -s vasmm68k_std m68k-sinclairql-vasmm68k_std

# Optional, just rename:
#sudo mv vlink m68k-sinclairql-vlink
#sudo mv vasmm68k_std m68k-sinclairql-vasmm68k_std
```

5.3 Build the Sinclair QL Cross Compiler

Now we can build and install the cross compiler. I'm creating a new directory, named `bin`, in my home directory, for the installation. This will need to be added to my `$PATH`[1] at some point:

```
# Create a new bin directory for the cross compiler.
mkdir -p ~/bin
export PATH=~/bin:$PATH

# Get back to the source.
cd ~/SourceCode/fpc

# Clean everything  out.
make clean OS_TARGET=sinclairql CPU_TARGET=m68k
... Huge piles of scrolling stuff here!

```

[1] On Linux Mint, and probably Ubuntu as well, when using the *bash* shell, if a user's `$HOME` directory includes a directory named `bin`, it is automatically added to `$PATH` at login time.

```
# Build the cross compiler.
make crossall OS_TARGET=sinclairql CPU_TARGET=m68k
... More huge piles of scrolling stuff here!


# Install the cross compiler.
make crossinstall OS_TARGET=sinclairql CPU_TARGET=m68k
   INSTALL_PREFIX="/home/norman/bin"
... Guess what happens here!
```

This will create a file named `/home/norman/bin/lib/fpc/n.n.n/ppcross68k`, but I prefer to call mine `fpc-ql`, so:

```
cd ~/bin
ln -s /lib/fpc/n.n.n/ppcross68k fpc-ql
```

I've got a sym-link set up but on Windows you can easily copy or rename it. Now we can test it.

```
fpc-ql
```

```
Free Pascal Compiler version n.n.n [2021/04/05] for m68k
...
CTRL-C
```

We can see that we have a compiler for the M68K CPU.

NOTE: You should be aware that any time you update the source code by running `svn update`, there will be a need for you to clean and rebuild the cross compiler. When you do this, be aware that the version number may change. If, like me, you have sym-links to the complied cross compiler, then you will find those links becoming invalid. You will need to recreate them, pointing at the new version.

5.4 Create the Configuration File

We need to create a configuration file for the cross compiler.

```
cd ~/bin/lib/fpc
mkdir etc

cd etc
```

You may have to create the `etc` directory, I did. A file named `fpc.cfg` is required with the following content:

```
#IFDEF CPUM68K
-Fu<path/to/install>/lib/fpc/$fpcversion/units/$fpccpu-$fpcos
-Fu<path/to/install>/lib/fpc/$fpcversion/units/$fpccpu-$fpcos
   /*
#IFDEF SINCLAIRQL
-FD</path/to/vasm-and-vlink>
-XPm68k-sinclairql-
#ENDIF
#ENDIF
```

You will note the use of place markers for the installation directories. As my installation directory was the `bin` directory in my home folder, my file looks like this:

```
#IFDEF CPUM68K
-Fu/home/norman/bin/lib/fpc/$fpcversion/units/$fpccpu-$fpcos
-Fu/home/norman/bin/lib/fpc/$fpcversion/units/$fpccpu-$fpcos
   /*
#IFDEF SINCLAIRQL
-FD/usr/local/bin
-XPm68k-sinclairql-
#ENDIF
#ENDIF
```

If you are only interested in building Pascal programs for an *actual* QL, and have no intention of developing anything for Windows or Linux, then this configuration file is an excellent option:

```
#IFDEF CPUM68K
-Tsinclairql
-Fu/home/norman/bin/lib/fpc/$fpcversion/units/$fpccpu-$fpcos
-Fu/home/norman/bin/lib/fpc/$fpcversion/units/$fpccpu-$fpcos
   /*
#IFDEF SINCLAIRQL
-FD/usr/local/bin
-XPm68k-sinclairql-
-WQxtcc
#ENDIF
#ENDIF
```

You may have noticed that I added the `-WQxtcc` option? This is the option that tells the cross compiler to add an XTcc trailer record to the end of the compiled binary, this holds the data space required by the program. You can use this on a normal QL to convert the file into an executable[2]. The default is to write a special header at the start of the

[2]Using my own excellent(!) utility, XTcc_bin. Issue_6 of my eMagazine has all the details.

file and most/all the QL emulators understand this and can execute the file directly. If you are running on an emulator, like me, then omit the `-WQxtcc` option.

There is an interesting foible here. Whenever I ran the `fpc-ql` command, I got a warning that it couldn't find the unit files used, specifically, the *system* unit. Debugging the compilation command showed that it was looking for unit files in a subdirectory named `m68k/linux` instead of `m68k/sinclairql`. I eventually tracked this down, with the help of the `-vv` compilation option, to the presence of a rogue `.fpc.cfg` file in my `$HOME` directory. When I renamed this file, the problems went away. I have no idea how or why that file was present, it may have been there for quite some time.

5.5 Building Your First QL Program

Now, test the build. In the following, I've used the `-Tsinclairql` option to tell the compiler to compile for a QL. If you added the option to the configuration file, there's no need to use it here.

```
1 cd ~/SourceCode/Pascal
2 fpc-ql -Tsinclairql hello.pp
```

```
Warning 22: Attributes of section .text were changed from r-x
   - in Linker Script <link15795.res> to rwx- in hello.o.
```

You can safely ignore the warning if it appears, later versions of the linker no longer produce the message. On Linux it's easy to determine if the correct file has been created provided the option to use an XTcc trailer was specified in the configuration file, or at compile time with the `-WQxtcc` option:

```
1 file hello.exe
```

```
hello.exe: QDOS executable 'FPC_PROG'
```

The executable is `hello.exe`, it's a QDOS executable and since FPC version 3.3.1, the job name is "FPC_PROG" unless you give the source code a "Program xxxx" statement in which case the job name will be "xxxx". You can change the job name on the fly[3], if you wish, using the QL specific functions `SetQLJobName()`, and retrieve the job name with `GetQLJobName()` which returns a string or `GetQLJobNamePtr()` which returns a pointer. All we have to do now is get it over to a QL and try it out.

On the other hand, if you are using the default header option, `-WQqhdr`, explicitly or otherwise, then the following will show you that it is a QDOS executable (for the various emulators that can handle it of course!)

```
1 hexdump -c -n 18 hello.exe
```

[3]See https://wiki.freepascal.org/Sinclair_QL#Job_Name for details.

```
0000000   ] ! Q D O S    F i l e    H e a d
0000010   e r
```

6 Updating the Compiler Source Code

From time to time there will be updates to the FPC code. As we have used *Subversion* we can keep our local copy of the source up to date. This is as simple as:

```
1 cd ~/SourceCode/fpc
2 svn update
3 #Lots of changes scrolling on by...
```

Please be aware that it is seriously advisable to rebuild the cross compiler and reinstall it, any time that you do this. I have found that the following steps are best followed:

- If you have made any changes to the RTL or QL Units, then take a backup as a patch file:

```
1 cd ~/SourceCode/fpc
2 svn diff --patch-compatible > ~/FPC.patch.temp.txt
```

- Run the source code update:

```
1 cd ~/SourceCode/fpc
2 svn update
```

- Clean, build and reinstall the compiler. Obviously, replace my installation location with your own:

```
1 make clean OS_TARGET=sinclairql CPU_TARGET=m68k
2 make crossall OS_TARGET=sinclairql CPU_TARGET=m68k
3 make crossinstall OS_TARGET=sinclairql CPU_TARGET=m68k
     INSTALL_PREFIX="/home/norman/bin"
```

- If you created a patch file, reapply your changes to the newly updated source code:

```
1 cd ~/SourceCode/fpc
2 svn patch ~/FPC.patch.temp.txt
```

- And finally, if you did have a patch to apply, rebuild the RTL and/or the QL Units as necessary:

```
1 # Rebuild and install the RTL:
2 cd ~/SourceCode/fpc
3 make rtl_cleanall
```

```
4  make rtl_all RELEASE=1 FPC=fpc-ql OS_TARGET=sinclairql
       CPU_TARGET=m68k
5
6  cd ${FPC_INSTALL_DIR}/bin/lib/fpc/n.n.n/units/m68k-
       sinclairql/rtl
7  cp -v ${HOME}/SourceCode/fpc/rtl/units/m68k-sinclairql/*
       ./
8
9  # Rebuild and install the QL Units:
10 cd ~/SourceCode/fpc
11 make packages_clean OS_TARGET=sinclairql CPU_TARGET=m68k
12 make packages RELEASE=1 FPC=fpc-ql OS_TARGET=sinclairql
       CPU_TARGET=m68k
13
14 cd ~/bin/lib/fpc/3.3.1/units/m68k-sinclairql/qlunits/
15 cp ~/SourceCode/fpc/packages/qlunits/units/m68k-
       sinclairql/* ./
16
17 # Just in case:
18 cd ~/SourceCode/fpc
```

The procedures for compiling and installing the RTL and Units is described in detail below.

And yes, I know, there's a lot of `cd`'ing back and forth and I could have used `cd -` to swap back to where I came from, but it might get mistaken for `cd ~` in some fonts, so I thought I better be explicit.

6.1 When the Cross Compiler Version Changes

From time to time the host compiler version may change. This change is pretty much invisible to you and the instructions in Chapter 3 Installing the Host Compiler, cover that possibility. However, just because the host compiler version has changed doesn't automatically mean that cross compiler versions change at the same time. How do I know this?

I was running with version 3.2.0 on my Linux host and version 3.3.1 for my cross compiler for the Sinclair QL. So my cross compiler is ahead of the host. However, I eventually updated the Linux host compiler to version 3.2.2 but the cross compiler is still ahead at version 3.3.1, even after I updated the source and rebuilt it.

Eventually, however, the cross compiler source will be brought up to date with the host compiler versions (or indeed, vice versa!) and things will start to go wrong for you after you run the `svn update` command. This is because the cross compiler executables are always located in a subdirectory which is named for the cross compiler version. In my

case it's `/home/norman/bin/lib/fpc/3.3.1/` so when the version number changes to 3.3.2, for example, that will change to`/home/norman/bin/lib/fpc/3.3.1/` but all my sym links will point at the old version.

6.1.1 Check Version After Updating

The solution is simple. After running the `svn update` command and rebuilding the cross compiler, check where the output has gone. Have a look in your installation directory (set by `INSTALL_PREFIX` on the `make crossinstall` command you used to install it), under `lib/fpc` and see what, if any, new subdirectories have been created.

```
1 cd ~/bin/lib/fpc
2 ls -l
```

```
1 drwxr-xr-x 4 norman norman 4096 May 29 11:09 3.3.1
2 drwxrwxr-x 2 norman norman 4096 May 29 11:24 etc
```

In this case, nothing new has been created, but let's say that there was a new `3.3.2` subdirectory found. We need to update our sym-links.

6.1.2 Checking Current Sym-links

```
1 cd ~/bin
2 ls -l fpc-ql
```

```
1 lrwxrwxrwx 1 norman norman 24 May 15 14:32 /home/norman/bin/
    fpc-ql -> lib/fpc/3.3.1/ppcross68k
```

So my cross compiler, which I wanted naming to *fpc-ql*, is obviously still using the old `3.3.1` version of the cross compiler binary.

6.1.3 Update Symbolic Links

Windows users would probably just copy the various files from the 3.3.2 subdirectory as opposed to changing sym-links. Linux users will need to do the following changes.

```
1 cd ~/bin
2 rm fpc-ql
3 ln -s lib/fpc/3.3.2/ppcross68k fpc-ql
4 ls -l fpc-ql
```

```
1 lrwxrwxrwx 1 norman norman 24 May 29 11:10 /home/norman/bin/
    fpc-ql -> lib/fpc/3.3.2/ppcross68k
```

So, I can now see that *fpc-ql* will execute the newest version of the cross compiler. This can be tested with:

```
fpc -ql
```

```
Free Pascal Compiler version 3.3.2 [2021/05/29] for m68k
Copyright (c) 1993-2021 by Florian Klaempfl and others
/home/norman/bin/lib/fpc/3.3.1/ppcross68k [options] <
   inputfile> [options]
...
```

Confirmed. We have the latest version.

Note: This section was only an example, the cross compiler, at the date of writing[1], is still at version 3.3.1. Please don't go looking for 3.3.2 just yet!

[1] 29th May 2021

7 Compiling QL Pascal programs

When writing your source code, standard Pascal requires that the first "executable" line in the file be the `Program ProgramName( ... )`, however, FPC doesn't require you to have this line at all. On the QL, the executable's job name will be determined from the `Program` line. If one is present, the job name will be as per the given program name, if that line is not present, the default job name will be "Program". There are some special QL functions that can be used to set or retrieve the job name. This may be useful if you are writing a file processing task of some kind, and you wish to add the current filename to the job's name. You are limited to 48 characters maximum though, so don't go too overboard!

The functions are:

- `Function SetQLJobName(const s: string): longint;` This function sets the job's name from a Pascal string and returns the number of characters successfully set as the Job name, or -1 if there was an error.
- `Function GetQLJobName: string;` This function returns the current job name as a Pascal string, or empty string if there was an error.
- `Function GetQLJobNamePtr: pointer;` This function returns a pointer to the Job name stored as a QL string (2 byte length + series of characters), or nil if there was an error.

To compile a QL program, all you have to do is:

```
1 cd ~/SourceCode/Pascal
2 fpc-ql -Tsinclairql hello.pp
```

As mentioned, if you added the `-Tsinclairql` option to the configuration file, you don't have to specify it here. I need to compile code for both Linux and the QL. I test my dodgy Pascal code on Linux first to be sure it compiles and runs correctly, then I repeat the operation for the QL. At least then I know that if it worked on Linux but doesn't on the QL, then it's the QL RTL that's most likely to be the cause.

You can safely ignore the warning about the attributes at the end of the compilation output.

The executable file for the QL will be created as `hello.exe` and will have a built in file header with details of the data space required unless you specified the `-WQxtcc` option either on the command line or in the configuration file. There are two options which control how the executable file for the QL gets it's data space information:

- `-WQqhdr` Set metadata to QDOS File Header style. A header record will be added to the file and QPC or other emulators can use this to execute the file directly, even from a `dos_` device. This option is the default and doesn't need to be specified.
- `-WQxtcc` Set metadata to XTcc style. This will be needed on an actual QL and you will need a utility to convert the file to an executable.

To compile the example program, there are a number of options:

- `fpc-ql hello.pp` this assumes that the `-Tsinclairql` option is to be found in the configuration file, if not, it will fail to compile. The executable will be created with header details embedded into the executable file ready for use on various QL Emulators.
- `fpc-ql -WQqhdr hello.pp` This is exactly the same as the variant above, however, it will override any `-WQ` option in the configuration file and force an embedded header to be used.
- `fpc-ql -Tsinclairql hello.pp` this variant explicitly specifies that the file must be compiled for the Sinclair QL and will overwrite any existing `-T` option in the configuration file. The executable will be created with header details embedded into the executable file ready for use on various QL Emulators.
- `fpc-ql -Tsinclairql -WQqhdr hello.pp` This is exactly the same as the variant above, however, it will override any `-WQ` option in the configuration file and force an embedded header to be used.
- `fpc-ql -WQxtcc hello.pp` The executable will be created with an XTcc trailer record holding data space details. This variant is suitable for use on an actual QL after the executable has been processed by an XTcc utility.
- `fpc-ql -Tsinclairql -WQxtcc hello.pp` This is exactly the same as the variant above, however, it will override any `-WQ` option in the configuration file and force an XTcc trailer record to be used.

8 Running Compiled Programs on the QL

8.1 Programs Compiled with `-WQqhdr`

These programs are best suited to the various QL emulators which know about the embedded header with the data space details. I use QPC and it copes happily with me executing these programs directly from the `dos_` drive. I don't have to make any changes, or somehow tell QPC that although the file is executing from a non-QL device, it still works perfectly.

```
1 ex dos1_hello.exe
```

And that's all there is to it! A 512 by 256 window will open and display the following:

```
Hello FPC World!
Press any key to exit.
```

8.2 Programs Compiled with `-WQxtcc`

The executable file which the cross compiler created can be copied over to a QL (or QPC or other emulator) and converted into a *proper* QDOS executable. In my case I can use my XTcc utility from my somewhat irregular Assembly Language eMagazine, Issue 6 – available from My GitHub[1], to read the details and write out an executable file.

```
1 ex win1_source_xtcc_xtcc_bin,ram1_hello.exe
```

The file, `hello.exe` is now ready to be executed.

```
1 ex ram1_hello.exe
```

A 512 by 256 window will open and display the following:

```
Hello FPC World!
Press any key to exit.
```

Success!

[1] https://github.com/NormanDunbar/QLAssemblyLanguageMagazine/releases/tag/Issue_6

9 Amending the Run Time Library

The system unit lives in the run time library and is responsible for all the various startup needs of a compiled Pascal program. It opens the standard files, sets the program name and so forth. Sometimes it may be necessary to edit and recompile the RTL especially as the ability to cross compile programs for the QL only recently arose as a fun exercise for the November 2020 "*QLvember*" thingy![1]

9.1 Editing Source Code

The source for the RTL for the QL, lives in the `rtl/sinclairql` directory which you will find beneath the `fpc` directory created when you installed the source code previously. In my case, `~/SourceCode/fpc/rtl/sinclairql`.

The main files of interest here are:

- `System.pp` which contains the main system unit code. This unit is always included in all compiled programs and doesn't need to be specifically included with a `uses system` program line.
- `sysfile.inc` which contains the source code for QL file handling.
- `sysdir.inc` which contains the source code for QL directory handling.
- `qdosfuncs.inc` which contains various Pascal headers linking the QL specific code to the RTL.
- `qdos.inc` which contains interface functions for the QL's RTL.
- `sysos.inc` which contains an implementation of all the base types etc required for a minimal POSIX compliant subset required to port the compiler to a new OS. At present, it contains only the code to convert a QL error code to a, FPC error code.

9.1.1 Parameter Passing

You really need to beware of this because it has risen up and bitten me a couple of times! The default method of passing parameters around in Pascal is named *register*. This passed parameters using registers where it can, as follows:

- The first ordinal number (ie, a numeric value) is passed in register `D0`;

[1] I have no idea what it was actually called, so *thingy* it remains!

- The second ordinal number (ie, a numeric value) is passed in register `D1`;
- The first pointer or reference is passed in register `A0`;
- The first pointer or reference is passed in register `A1`;

So, when you see a Pascal procedure or function header that looks like the `mt_rechp` example from `qdos.inc`:

```
1  procedure mt_rechp(area: pointer); assembler; nostackframe;
      public name '_mt_rechp';
2  asm
3    movem.l d2-d3/a2-a3,-(sp)
4    move.l area,a0
5    moveq.l #_MT_RECHP,d0
6    trap #1
7    movem.l (sp)+,d2-d3/a2-a3
8  end;
```

You need to be aware that the `area` parameter, which is a pointer, will be passed in register `A0`. This means that the `move.l area,a0` instruction will be disassembled as `move,l a0,a0` and may appear redundant, but there are options to change the way that parameters are passed, so don't miss the line out!

In addition, if you are anything like me, I tend to almost always set my registers up in order, `D0-D7` then `A0-A7` as required when making calls to QDOSMSQ, that's fine in Assembly Language, but not so fine in Pascal when `D0` and `D1` are used for passing numeric parameter values. This code contains a hideous bug which took me a while to track down:

```
1  function fs_posab(chan: Tchanid; new_pos: dword):longint;
      assembler; nostackframe; public name '_fs_posab';
2  asm
3      move.l d3,-(sp)
4      moveq #_FS_POSAB,d0
5      move.l new_pos,d1
6      moveq #-1,d3
7      move.l chan,a0
8      trap #3
9      tst.l d0
10     bne.s @quit
11     move.l d1,d0
12 @quit:
13     move.l (sp)+,d3
14 end;
```

The problem is the `chan` variable is passed in register `D0`, but before we copy `chan` into `A0`, we have already overwritten it with the value for `_FS_POSAB`. This, as you can imagine, caused various channel not open errors.

9.1.2 Debugging

When compiling, you can see the assembly language that is created by the compiler if you use the `-al` (lower case 'a', lower case ell) command line option. This prevents FPC from deleting the Assembly Language source file(s) after the compilation is complete. If you use this option with the source file `hello.pas`, then your Assembly source will be found in the file `hello.s` afterwards.

Sometimes this is enough to give you a clue, other times may require debugging on the QL.

Over on the QL, however, you can debug a compiled Pascal program using *QMON2*. When I was debugging the `fs_posab` function listed above, I needed to jump into *QMON2* whenever I hit that particular section of code. I didn't want to have to trace through the entire program until I reached it, that would have been a nightmare as we (still) don't have a source level debugger for the QL. Yet!

What I did was simple. At the start of `fs_posab`, I added a `trap #15` instruction and recompiled and reinstalled the RTL as described later. On the QL, it was this easy:

```
1 lrespr win1_qmon2_qmon_bin
2 qmon dos1_test.exe
```

This loads up the executable, in my case from `dos1_test.exe`, and then stops to allow me the ability to enter commands:

```
1 tl 14
2 g
```

This neat pair of instructions says to *QMON2*, whenever you hit a trap higher than `trap #14`, break at that point and jump into the monitor. Now when the Pascal code reaches the `fs_posab` function, I get the ability to trace it, or set breakpoints etc and see if I can track down what is happening to make things not work.

NOTE: Don't forget, after fixing the problems, go back and remove the trap #15 instructions from wherever you put them.

It was through tracing this that I discovered that the `move.l chan,a0` was actually `move.l d0,a0` and as `D0` was already set to `_FS_POSAB`, I could see what was wrong. After reading up on FPC's default parameter passing methods of course!

10 Building the Run Time Library

In a similar manner to the Units (coming soon) , you cannot just compile a couple of code files that you changed. You have to make sure that all code files are recompiled any time you make a change. If you don't do this, you'll get warnings at program compilation times that some files cannot be found.

10.1 Compiling the RTL

Once any amendments have been made to the RTL source files, the whole lot needs to be recompiled and relinked. I should point out that the following compilation invocation[1] was extracted from the system by Marcel Kilgus when I was having troubles getting things to compile properly. I do not know if this is the official invocation or not, but it does seem to work.

The compilation should be done from the top level of the source tree, in my case this is the directory created by *Subversion* when I first downloaded the code, `~/SourceCode/fpc`. The process is simple:

- Change into the appropriate directory;
- Invoke the compiler;
- Install the newly compiled RTL.

The last step is covered in the following section and the first two are as simple as executing the following:

```
1 cd ~/SourceCode/fpc
2 make rtl_cleanall
3 make rtl_all RELEASE=1 FPC=fpc-ql OS_TARGET=sinclairql
     CPU_TARGET=m68k
```

You will need to watch out for error messages scrolling up the screen. On Linux, I have enough scrollback in my terminal session that I can review the entire compilation easily. If I remember correctly, on Windows, scrollback is limited unless yo set the properties for the command line session.

[1] Ie, magic spell!

10.2 Installing the RTL

Once we have a clean compilation, we can install the new RTL. This is where it gets slightly complicated as your RTL might not be installed where mine is. If you remember way back when we created and installed the cross compiler for the QL, I used an installation location of `/home/norman/bin`, that is my installation directory, yours will be whatever you used. You might wish to export that as an environment variable:

```
1 export FPC_INSTALL_DIR=/home/norman/bin
```

Now you can use that in the code that follows to be sure you have the correct locations for the RTL.

```
1 cd ${FPC_INSTALL_DIR}/bin/lib/fpc/n.n.n/units/m68k-sinclairql
     /rtl
2 cp -v ${HOME}/SourceCode/fpc/rtl/units/m68k-sinclairql/* ./
```

That makes sure that everything is copied across to the correct location.

Whenever you change the RTL, unless what you changed isn't used by any existing QL Pascal code that you have already cross compiled, you will be best to recompile and test the programs. Mistakes happen – even though the compiler doesn't catch them, ask me how I know this?

NOTE: You may be wondering why there isn't a *make* target to install the RTL? Well, there appears to be a target `rtl_install`, and it appears to do what I want:

```
1 cd ~/SourceCode/fpc
2 make rtl_install OS_TARGET=sinclairql CPU_TARGET=m68k
```

Unfortunately, it always errors out. I'll hopefully be looking into this at some point soon:

```
make -C rtl install
make[1]: Entering directory '/home/norman/SourceCode/fpc/rtl'
make -C sinclairql all
make[2]: Entering directory '/home/norman/SourceCode/fpc/rtl/
   sinclairql'
make[2]: Leaving directory '/home/norman/SourceCode/fpc/rtl/
   sinclairql'
/usr/local/bin/fpcmake -p -Tm68k-sinclairql Makefile.fpc
Processing Makefile.fpc
Error: No targets set
make[1]: *** [Makefile:1649: fpc_install] Error 1
make[1]: Leaving directory '/home/norman/SourceCode/fpc/rtl'
make: *** [Makefile:2936: rtl_install] Error 2
```

10.3 A Useful Build Script

Rather than having to remember all those commands, I built myself a small script in the top level directory, `SourceCode/fpc`, and I use that to build and install everything as and when required. The code is:

```
1  #!/bin/bash
2
3  #
4  # Variables:
5  #
6  FPC_INSTALL_DIR=/home/norman/bin
7
8  #
9  # Builds the sinclair ql RTL for free pascal.
10 #
11 make rtl_cleanall
12 make rtl_all RELEASE=1 FPC=fpc-ql OS_TARGET=sinclairql
      CPU_TARGET=m68k
13
14 #
15 # Wipe out the existing stuff and copy the new stuff over.
16 #
17 cd ${FPC_INSTALL_DIR}lib/fpc/3.3.1/units/m68k-sinclairql/rtl
18 cp -v ${HOME}/SourceCode/fpc/rtl/units/m68k-sinclairql/* ./
19 cd -
```

10.4 Distributing Changes to the RTL

10.5 Creating a Patch File

So, you have done a lot of work in the RTL, and now you need to get this distributed to "all and sundry" so that they can take advantage of your hard work. How do you go about doing this? *Subversion* to the rescue.

It's a simple task to change into the top level directory and use Subversion to create a patch file that others can apply, easily, to bring their code up to the same state as yours. The process is:

```
1 cd ~/SourceCode/fpc
2 svn diff --patch-compatible | more
```

Once you have checked that it looks Ok on screen, you can write the changes to a patch file, as follows:

```
cd ~/SourceCode/fpc
svn diff --patch-compatible > ~/FPC.patch.txt
```

This command creates a patch file named `FPC.patch.txt` in the `$HOME` directory. The file contains all changed made to the files in `SourceCode/fpc` and all subdirectories, since the files were last downloaded or updated using *Subversion*. This file can be zipped up and distributed.

At present, I simply upload my patch files to the QL Pascal topic on the QL Forum. One of the FPC maintainers, Chain-Q, picks my patches up there and runs his careful eye over them before accepting or rejecting them for inclusion in the main source tree.

10.6 Applying a Patch File

If you also want my changes, you can grab the latest patch file, which may be zipped, and apply it to your source as follows. First, locate and unzip the patch file, if necessary:

```
# Unzip the downloaded patch file.
cd ~/Downloads
unzip FPC.patch.zip
```

```
Archive:  FPC.patch.zip
  inflating: FPC.patch.txt
```

Now there are multiple options on applying the patch. The first is to copy the unzipped patch file over top the top level of your source tree and apply it from there:

```
# Apply the patch file. Method 1.
# Copy the patch file to the code tree.

cd ~/SourceCode/fpc
cp ~/Downloads/FPC.patch.txt ./
svn patch FPC.patch.txt
```

The next common option is to leave the unzipped patch file where it is, and apply it from within the source tree:

```
# Apply the patch file. Method 2.
# Doesn't copy the patch file to the code tree.
# Just patches it from the Downloads area where it was
   unzipped.

cd ~/SourceCode/fpc
svn patch ~/Downloads/FPC.patch.txt
```

Finally, although there are other methods, simply apply the patch from the download area but tell Subversion where the source code lives:

```
# Apply the patch file. Method 3.
# Out of tree patching from Downloads area.

cp ~/Downloads/FPC.patch.txt ./
svn patch FPC.patch.txt ~/SourceCode/fpc/
```

Regardless of which method you decide to use, Subversion will bring the code up to date with the changes in the patch file. You should be aware that:

- Next time you run the `svn update` command in the source tree, the patches may be overwritten, or cause conflict;
- Some of the patches may cause conflicts if they patch areas of the code that you too are amending.

As the patch is being applied, you will see various prompts advising you on what has happened. For each patched file a single line will be printed with characters reporting the action taken. These characters have the following meaning:

- A = Added - this file has been added as a new file;
- D = Deleted - this file is now deleted;
- U = Updated - this file was successfully patched;
- C = Conflict - this file has a conflict. Both you and the path have updated the same part of the code. You will be prompted for an action to ignore or resolve the conflict.
- G = Merged (with local uncommitted changes)

Obviously, after patching the RTL, you will be required to rebuild and install it.

11 Amending the QL Units

The source for the various QL Units lives in the `packages/qlunits/src` directory which you will find beneath the `fpc` directory created when you installed the source code previously. In my case, `SourceCode/fpc/packages/qlunits/src`.

Files of interest here are:

- `qdos.pas` which contains the source code for the *qdos* unit – a collection of useful QL specific routines. Some of these are simply references to the code defined in the system unit in the RTL.
- `qlfloat.pas` contains some functions for converting to and from QL floating point numbers. This file makes up the *qlfloats* unit.

There are, currently, only two units for the QL cross compiler. We need a lot more. We are missing *sysutils*, *strings* and all the rest. We have some work to do!

The compiled units live in `packages/qlunits/units/m68k-sinclairql/` beneath the source code's main directory.

11.1 A Worked Example

This is a particularly silly example, but it shows what is required to add new stuff to the QL Units, which in most cases means the *qdos* unit. Let's say that we want to add a new procedure, `HelloNorm`, to the unit, what do we need to do?

- Edit the `qdos.pas` file and add the new procedure details to the interface section;
- Still editing the `qdos.pas` file, add the new procedure code to the implementation section;
- Recompile all the QL Units;
- Install the newly compiled QL Units;
- Compile and test a program which uses the new procedure.

The steps to compile and install the QL Unit files are described in detail in the following sections. The remainder of this section covers the editing of the unit source files, and compiling a test program.

11.1.1 Edit the `Qdos.pas` Unit file

In a Pascal unit file, there are normally two sections, the *interface* and the *implementation*. The interface describes the header of the procedure or functions – similar to how a C++ header file does, and the implementation is the full source code for the procedures and functions.

`SourceCode/fpc/packages/qlunits/src/qdos.pas`, is where the *qdos* unit keeps its source code, so that's the file to edit.

The first edit for our new `HelloNorm` procedure is to add it to the end of the interface section. After this edit, my file looks like this:

```
procedure sd_line(chan: Tchanid; timeout: Ttimeout; x_start:
   double; y_start: double; x_end: double; y_end: double);

procedure HelloNorm;

implementation
```

You can see I added the procedure name just above the implementation. I tend to add stuff at the end as it makes it easier to find later! Next we need to add the source code for the new procedure, and again, I added mine right at the bottom of the file, just above the final end. (That's the end with a full stop in the source code – that line already exists.)

```
procedure HelloNorm;
begin
  writeln('Hello Norm!');
end;

end.
```

You won't need to type in that final line, it's already there, but any new code has to go above it.

Save the file and exit. We are now ready to compile the unit files.

11.1.2 Compile the QL units

The full details of compiling the units is described below, for now a quick couple of commands are all that is necessary:

```
1 cd ~/SourceCode/fpc
2 make packages_clean OS_TARGET=sinclairql CPU_TARGET=m68k
3 make packages RELEASE=1 FPC=fpc-ql OS_TARGET=sinclairql
     CPU_TARGET=m68k
```

Lots of messages will flash up the screen. Don't worry about it, just review the messages looking for errors and warnings. Once you have a clean recompile, it's installation time.

11.1.3 Install the QL Units

The units are compiled into a holding location which is not where they are actually found by the compiler. This was defined in the configuration file we created way way back! Remember?

```
1 #IFDEF CPUM68K
2 -Fu<path/to/install>/lib/fpc/$fpcversion/units/$fpccpu-$fpcos
3 -Fu<path/to/install>/lib/fpc/$fpcversion/units/$fpccpu-$fpcos
    /*
4 #IFDEF SINCLAIRQL
5 -FD</path/to/vasm-and-vlink>
6 -XPm68k-sinclairql-
7 #ENDIF
8 #ENDIF
```

This tells you where the units need to be placed for the compiler to find them, As you can see, there are no hard coded version numbers, CPU or OS names, so when we get a compiler update, we don't have to change anything and it all just works. The two lines in the configuration file with the `-Fu` options define where units will be searched for by the compiler. We need to be sure we store our compiled units in the correct place.

To copy the newly compiled units to the correct location, we use the following quickie:

```
1 cd ~/bin/lib/fpc/3.3.1/units/m68k-sinclairql/qlunits/
2 cp ~/SourceCode/fpc/packages/qlunits/units/m68k-sinclairql/*
    ./
3 cd -
```

Now we can compile and test the test program.

11.1.4 The Test Program

We need a test program. This is mine, it's called `hellonorm.pp`.

```
1 program helloMe;
2 uses
3    qdos;
4 begin
5    HelloNorm;
6 end.
```

WARNING: You cannot name the program with the same name as a procedure or function. If you do, and I did, you get this:

```
hellonorm.pp(5,13) Fatal: Syntax error, "." expected but ";"
   found
Fatal: Compilation aborted
```

First of all, we have to tell the compiler that we want to actually use the *qdos* unit. That's the purpose of the `uses qdos;` statement. Once we have done that, the compiler will hopefully find the new procedure and link it in to our executable.

We need to compile the code:

```
1 fpc-ql -Tsinclairql hellonorm.pp
```

Over on the QL, or in QPC in my case:

```
1 ex dos1_hellonorm.exe
```

```
Hello Norm!
Press any key to exit.
```

Easy peasy?

Compiling and installing the unit files are described fully in the next couple of sections.

12 Recompiling the QL Units

In a similar manner to the RTL, you cannot just compile a couple of code files that you changed. You have to make sure that all code files are recompiled any time you make a change. If you don't do this, you'll get warnings at program compilation times that some unit files cannot be found. So if you compile only the `qdos` unit, then recompile a program that uses it, you'll be told off because the `qlfloats` unit cannot be used.

12.1 Building the QL Units

The various units for the QL cross compiler are compiled from the usual top level source directory. Because they must all be compiled at the same time, it's advised to clean things out first.

```
1 cd ~/SourceCode/fpc
2 make packages_clean OS_TARGET=sinclairql CPU_TARGET=m68k
```

```
make -C packages clean
make[1]: Entering directory '/home/norman/SourceCode/fpc/
    packages'
make -C fpmkunit clean_bootstrap
make[2]: Entering directory '/home/norman/SourceCode/fpc/
    packages/fpmkunit'
/usr/bin/rm -rf units_bs
make[2]: Leaving directory '/home/norman/SourceCode/fpc/packages
    /fpmkunit'
./fpmake clean --localunitdir=.. --os=sinclairql --cpu=m68k -o -
    Tsinclairql -o -Pm68k -o -XPm68k-sinclairql- -o -Ur -o -Xs -
    o -O2 -o -n -o -dm68k -o -dRELEASE --compiler=/home/norman/
    bin/ppcross68k -bu -scp
make[1]: Leaving directory '/home/norman/SourceCode/fpc/packages
    '
```

There should be no problems with the output, so we are now ready to rebuild the units:

```
1 make packages RELEASE=1 FPC=fpc-ql OS_TARGET=sinclairql
    CPU_TARGET=m68k
```

```
make -C packages all
make[1]: Entering directory '/home/norman/SourceCode/fpc/
    packages'
./fpmake compile --localunitdir=.. --os=sinclairql --cpu=m68k -o
    -Tsinclairql -o -Pm68k -o -XPm68k-sinclairql- -o -Ur -o -Xs
    -o -O2 -o -n -o -dm68k -o -dRELEASE --compiler=fpc-ql -bu -
    scp
Start compiling package rtl-unicode for target m68k-sinclairql.
       Compiling rtl-unicode/BuildUnit_rtl_unicode.pp
       Compiling ./rtl-unicode/src/inc/graphemebreakproperty.pp
       Compiling ./rtl-unicode/src/inc/eastasianwidth.pp
[ 77%] Compiled package rtl-unicode
Start compiling package tplylib for target m68k-sinclairql.
       Compiling tplylib/BuildUnit_tplylib.pp
       Compiling ./tplylib/src/lexlib.pas
       Compiling ./tplylib/src/yacclib.pas
[ 82%] Compiled package tplylib
Start compiling package qlunits for target m68k-sinclairql.
       Compiling qlunits/BuildUnit_qlunits.pp
       Compiling ./qlunits/src/qdos.pas
       Compiling ./qlunits/src/qlfloat.pas
       Compiling ./qlunits/src/qlutil.pas
       Compiling ./qlunits/src/sms.pas
[ 99%] Compiled package qlunits
make[1]: Leaving directory '/home/norman/SourceCode/fpc/packages
    '
```

Check the compilation output for any errors or warnings, fix them as appropriate and recompile. Once you have a clean compilation, we are ready to install.

12.2 Installing the QL Units

As with the RTL, it appears that the *make* target which *should* install the units, fails miserably:

```
1 make packages_install RELEASE=1 FPC=fpc-ql OS_TARGET=
    sinclairql CPU_TARGET=m68k
```

Any time I've run the above command, I get this output:

```
make -C packages install
make[1]: Entering directory '/home/norman/SourceCode/fpc/
    packages'
```

```
./fpmake install --localunitdir=.. --os=sinclairql --cpu=m68k
    -o -Tsinclairql -o -Pm68k -o -XPm68k-sinclairql- -o -Ur -
   o -Xs -o -O2 -o -n -o -dm68k -o -dRELEASE --compiler=fpc-
   ql -bu -scp --prefix=/usr/local --baseinstalldir=/usr/
   local/lib/fpc/3.3.1
Installing package rtl-unicode
The installer encountered the following error:
Failed to create directory "/usr/local/lib/fpc/3.3.1/units/
   m68k-sinclairql/rtl-unicode/"
make[1]: *** [Makefile:1948: install] Error 1
make[1]: Leaving directory '/home/norman/SourceCode/fpc/
   packages'
make: *** [Makefile:3026: packages_install] Error 2
```

Bear in mind that I only know the *make* target name because I *grep*ped them all out of the top level `Makefile`. I am probably doing something incorrectly.

Anyway, having extracted the desired unit location from the `fpc.cfg` configuration file, I have to run the following commands to copy the newly compiled unit files over to the correct place:

```
1 cd ~/bin/lib/fpc/3.3.1/units/m68k-sinclairql/qlunits/
2 cp ~/SourceCode/fpc/packages/qlunits/units/m68k-sinclairql/*
    ./
```

Once this has been done, you may be advised to recompile any existing Pascal programs which use the units, just in case you've managed to introduce a runtime bug which the compiler can't catch for you.

13 Creating New Units

The previous chapter introduced you to amending the existing QL Units. In this chapter, we shall see how simple it is to add a new unit ready for use by the programs developed for the QL. The steps involved are:

- Create the new unit's source file;
- Add the new unit to the `fpmake.pp` file, or it will not be added as a unit;
- Compile all the units;
- Install the newly compiled units.

Compiling and installing are already documented in Section 12.1 Building the QL Units and Section 12.2 Installing the QL Units and will not require further explanation.

13.1 Creating the Unit

13.1.1 Unit Skeleton

You create a new unit with the following skeleton layout:

```
1  unit UnitName;
2
3  interface
4
5  uses ...
6  const ...
7  type ...
8  ...
9
10 implementation
11
12 uses ...
13 const ...
14 type ...
15 var ...
16 ...
17
18 end.
```

13.1.2 Interface

The interface section is where you will be making all your forward references to procedures and such like which will be defined in this unit. This is effectively the same as the header files (`*.h`) as used in C or C++ development, and defines the public parts of the unit – the bits that programs or other units can use.

All the consts, types etc mentioned in this section will be visible to programs and other units which use this one.

13.1.3 Implementation

The implementation section is where the meat and bones of the unit are to be found. Anything here which is not declared in the interface section, cannot be seen outside of the unit. Any of the uses, const, type, var etc clauses here are purely for use in the unit itself.

13.1.4 An Example Unit

Details of writing units can be found at https://wiki.lazarus.freepascal.org/Unit so the following will be a brief overview.

Unit source code lives in `~/SourceCode/fpc/packages/qlunits/src` and must be saved to a file with the same name as the unit, with a '`.pas`' extension. The *qdos* unit is therefore named `qdos.pas`, for example. It is important that you follow this standard, or problems will arise.

In the time honoured tradition of writing example code, I've created a simple unit named *hello*, which lives in the source file, `hello.pas`. This is the code for the unit:

```
1  unit hello;
2  interface
3
4  procedure greet(prompt: pchar);
5  procedure helloWorld;
6
7  implementation
8
9  procedure greet(prompt: pchar);
10 begin
11   writeln('Hello ', prompt);
12 end;
13
14 procedure helloWorld;
15 begin
```

```
16    greet('World!');
17  end;
18
19  end.
```

As you can see from the interface section, this unit exposes only two procedures:

- `Greet` which takes a single parameter;
- `HelloWorld` which takes no parameters.

13.2 Update `fpmake.pp`

This is the easy part! Once the hard work of writing the unit's code and interface is done, you can easily add it to the full compiler build. All you have to do is add the new unit to the file `fpmake.pp`.

13.2.1 Adding New Units

The file, `fpmake.pp`, lives one directory higher up than the source files you have been working on. You simply need to change up one level and edit the file.

Somewhere in the file you will find the following lines of code:

```
1  ...
2  T:=P.Targets.AddUnit('qdos.pas');
3  T:=P.Targets.AddUnit('qlfloat.pas');
4  T:=P.Targets.AddUnit('qlutil.pas');
5  T:=P.Targets.AddUnit('sms.pas');
6  ...
```

All you now need to do is add another line, usually at the end, passing in the name of your new unit:

```
1  ...
2  T:=P.Targets.AddUnit('qdos.pas');
3  T:=P.Targets.AddUnit('qlfloat.pas');
4  T:=P.Targets.AddUnit('qlutil.pas');
5  T:=P.Targets.AddUnit('sms.pas');
6  T:=P.Targets.AddUnit('hello.pas');
7  ...
```

As this is a cross compiler, please make sure that your unit file names are case dependent – it's fine on Windows where case is ignored, but on other (proper[1]) systems, case is dependent and something that works on Windows may not work on other systems.

[1] Grins, ducks and runs!

13.2.2 Adding new Examples

Adding example programs to demonstrate the new units is equally simple. Source code for the examples lives in `~/SourceCode/fpc/packages/qlunits/examples` and there's no difference from writing a normal Pascal program. Just do it! Once compiled and tested in the usual manner, it can be added to the system as an official example. Once again, `fpmake.pp` is your friend.

This is my simple example for the new *hello* unit created above, it's called `helloEx.pas`:

```
1 Program HelloNorm;
2
3 uses
4   hello;
5
6 begin
7   helloWorld;
8   greet('Norm!');
9 end.
```

Once that compiled and executed happily in testing, I was able to add it to `fpmake.pp`. It's a simple case of editing the file, again, and looking for the following lines of code:

```
1 ...
2 P.ExamplePath.Add('examples');
3 T:=P.Targets.AddExampleProgram('qlcube.pas');
4 T:=P.Targets.AddExampleProgram('mtinf.pas');
5 ...
```

All that is required is to add another line, passing in the name of the new example program's source file within the `examples` directory. In my case, I added my example `helloEx.pas`, so here's what my change looks like:

```
1 ...
2 P.ExamplePath.Add('examples');
3 T:=P.Targets.AddExampleProgram('qlcube.pas');
4 T:=P.Targets.AddExampleProgram('mtinf.pas');
5 T:=P.Targets.AddExampleProgram('helloEx.pas');
6 ...
```

Once again, please make sure that your example's file name is case sensitive.

13.3 The Build

After adding a new unit, or an example program, you need to run a build of the units and a reinstall of the compiled unit files. This is covered above in Section 12.1 Building the QL Units and Section 12.2 Installing the QL Units.

13.3.1 Version Control

Once the new unit has been tested to destruction, and the example program(s) have been written and tested also, you *might* need to add the files to version control. I say might because, like me, you probably don't have write access to the main repository. My changes to the QL Cross Compiler have all been done with patch files created by *Subversion* as described in Section 10.5 Creating a Patch File.

Unfortunately, *Subversion* doesn't know about your new files, so they need to be added to your local copy of the repository. This is simply done:

```
1 cd ~/SourceCode/fpc/packages/qlunits/src
2 svn add hello.pas
```

```
1 A hello.pas
```

Also, for the examples:

```
1 cd ~/SourceCode/fpc/packages/qlunits/examples
2 svn add helloEx.pas
```

```
1 A helloEx.pas
```

13.3.2 Update `README.txt`

In the directory `~/SourceCode/fpc/packages/qlunits`, you will find the file `README.txt`. Please update this file with details of any new units and examples.

13.3.3 Create the Patch

Now that subversion knows about your new files, and the readme has been updated, running a patch creation will include the new files in the patch output, ready for use by other QL developers, or even, by the Free Pascal Project. See Section 10.5 Creating a Patch File for details on creating a patch file.