Felix installation and tools

Release 2016.07.12-rc1

Mar 19, 2023

Contents

THIS IS A WORK IN PROGRESS!

Contents:

CHAPTER 1

Installation Guide

This is the Felix installation guide.

1.1 Linux: Ubuntu

1.1.1 Install Pre-requisites

The first step is to install the pre-requisites. The following list may not be complete. From command line or other tool install packages:

```
sudo apt-get install binutils
sudo apt-get install make
sudo apt-get install git
sudo apt-get install g++
sudo apt-get install python3
sudo apt-get install ocaml-native-compilers
```

These are optional components which provide graphics for the Felix GUI using Simple Direct Media Layer, version 2:

```
sudo apt-get install libsdl2-dev
sudo apt-get install libsdl2-ttf-dev
sudo apt-get install libsdl2-image-dev
sudo apt-get install libsdl2-gfx-dev
```

There are many other optional packages for which there is some level of support or interest. These include, without their apt-name at the momen (sorry, please help!):

- gmp: gnu big number library
- gmp++: C++ wrapper for gmp
- gsl: gnu scientific library

- botan: Crypto library
- icu: unicode library

1.1.2 Clone Felix

You first need to configure git, please see git docs. Then for a user without write access to the Felix repository, make a workspace directory *in which* the Felix repository directory will be created and *cd* into it. Now grab a copy of Felix:

```
git clone https://github.com/felix-lang/felix.git
```

1.1.3 Build Felix

Now you have to go into the repository clone and build Felix:

```
cd felix
. buildscript/linuxsetup.sh
make
```

What happens is that a build system called *fbuild* which is a Python program, will build a bootstrap version of Felix first.

Then, the bootstrap Felix will be used to build Felix again, this time using Felix own build tools, which are written in Felix.

Finally, a four test suites run. The first is a small number of bad tests that are supposed to all fail, ignore it.

Then the main regression test suite runs. Most of these tests should pass, or there's a bug, but as long as most pass, don't worry (at least, not yet!)

Then some tutorial examples are run as tests. Again, most should pass but don't worry about one or two fails.

Finally, some optional tests run, which exercise optional packages. Most of these are graphics and GUI tests and they WILL fail on Ubuntu. We will fix that in a moment!

1.1.4 Installation

Felix does *not* have to be installed to work. I recommend you do *not* install it, at least not yet. Installation is for enterprise users, rather than personal users.

The reason is that upgrades are frequent: Felix is typically upgraded every day. Rebuilding Felix is easy, but it is a pain reinstalling it all the time, it is better initially to run it in place. But here is how you would install it:

```
sudo mkdir -p /usr/local/lib
sudo mkdir -p /usr/local/bin
sudo make install
```

This will put most of the Felix system in */usr/local/lib/felix/felix-version* where *felix-version* is the version of Felix you're installing. You can install many versions of Felix all at once.

The install process *also* puts the *flx* command into */usr/local/bin*. This will overwrite any previous *flx*. For the installed Felix to work at all you will need to setup the *PATH* variable:

```
export PATH=/usr/local/bin:$PATH
```

The best place to do this is in your *$HOME/.profile*, if it is not set already.

For full plugin and dynamic library support, you will also need to set *LD_LIBRARY_PATH*. Normally, *flx* sets this for you, but if you want to run Felix built executables directly as standalone programs, *and* you want to link to Felix shared libraries, including plugins, then the system linker has to find the libraries so you will also need this:

```
export LD_LIBRARY_PATH=/usr/local/lib/felix/felix-version:$LD_LIBRARY_PATH
```

Felix does not put its shared libraries in the usual place, directly in a */usr/lib* or */usr/local/lib* directory. This is deliberate. You need to be able to delete a Felix version, or all of Felix easily, and for this reason *almost everything* lives under master directory */usr/local/lib/felix* and subdirectory *felix-version*, the main exception being the *flx* program, which is copied to */usr/local/bin*.

1.1.5 Running in Place

This option is prefered over installing Felix at the moment, although it is a little tricker to set up, it makes it very much easier to upgrade Felix.

After you have built Felix, you can use it in place, without installing it. First you need to do this: make sure you are still in the Felix directory, be very careful to use the correct quotation marks as indicated below!!

```
echo `export PATH=$PWD/build/release/host/bin:$PATH` >> $HOME/.profile
mkdir -p $HOME/.felix/config
echo "FLX_INSTALL_DIR: $PWD/build/release" >$HOME/.felix/config/felix.fpc
```

The *flx* executable looks to see if the file *$HOME/.felix/config/felix.fpc* exists, and if it does, it will set the variables seen in that file. You can also set the environment variable FLX_INSTALL_DIR in the Linux environment by adding this command to your *$HOME/.profile*:

```
export FLX_INSTALL_DIR=$PWD/build/release
```

where $PWD has to be replaced by the absolute path of the repository, and then the command above should be put into your $HOME/.profile. I personally use the first method.

1.1.6 Installing to an Alternate Location

PREFIX needs to be set and exported to the location to install, otherwise it defaults to */usr/local*, e.g.:

```
export PREFIX=/home/user/local
```

If you want to install it to the same location as previously (or to the configured location) you can just do:

```
export PREFIX=$FLX_INSTALL_DIR
```

1.1.7 Test it

This should work now:

```
flx hello.flx
```

CHAPTER 2

Basic Operation

Felix is designed to be used, in the first instance, like Python or other scripting language. You can just run a text file directly like this:

```
flx hello.flx
```

Behind the scenes, the *flx* command will first run the Felix compiler executable *flxg* generating some C++ files and some other information.

Then, it runs your chosen C++ compiler to compile and link those files to binary form.

Finally, it runs the binaries.

You do not need make, autoconf, or any kind of build script or configuration, it just works!

2.1 Where is the executable?

Felix puts the executable here:

```
$HOME/.felix/cache/binary/$HOME/felix/hello.so
```

where *$HOME* is your home directory. In other words, Felix uses a cache, which by default is

```
$HOME/.felix/cache
```

to store temporary files. Text files go in:

```
$HOME/.felix/cache/text
```

and binary files go in:

```
$HOME/.felix/cache/binary
```

The file name used in the appropriate cache uses the prefix of the *absolute* pathname of the source file.

2.2 That's a library not an executable!

You're right! By default, Felix builds libraries, not programs, and it builds your program as a shared library. On Linux, this will have the extension *.so*, on OSX it will be *.dylib* and on Windows it will be *.dll*.

These library objects are loaded at run time by a small program executable, it will be called either *flx_run* or *flx_arun*, which can be found in *build/release/host/bin*. It will have extension *.exe* on Windows and no extension on unix like systems. This program is passed the absolute pathname of the library to load, which it does using *dlopen()* on unix like systems and *LoadLibrary()* on Windows.

2.3 How can I avoid rebuilding my program every time?

That's easy. Make a cup of coffee. There's nothing to do. Felix automatically checks dependencies and if it would compile the same program you have already compiled it just runs the already compiled one.

2.4 How can I make a standalone executable?

You can best do this like so:

```
flx --static -c -od . hello.flx
```

First, the *–static* switch says to do static linkage, instead of dynamic linkage. Second, the *-c* switch says to compile and link the program but not run it. Third, the *-od .* switch says to put the output of the build process into the current directory . using the basename of the source file *hello* as the basename of the executable.

On Unix systems *hello* will appear in the current directory and you can run it by

```
./hello
```

On Windows, *hello.exe* will appear instead, and you can run it by:

```
hello
```

You can copy this program wherever you like and it will work, it does not need Felix anymore.

Note that for some complex programs which uses plugins, the program will have to be able to find the plugins, which are shared libraries or DLLs, and you will need to also set up an environment where it can do so. The *flx* program, even though a statically linked executable, can still load plugins.

CHAPTER 3

Using Third Party Libraries

Felix is specifically designed to be able to use third party C and C++ libraries, and to be able to do so without any makefiles, linkage commands, or other scripts.

To make this happen, there is a configuration database which tells Felix where the relevant header files and libraries actually reside in your file system.

In fact, Felix itself uses the very same machinery to use its own native libraries.

3.1 Targets

Felix can build for any targets you can configure. The default, required, target is called *host*, and it is your normal programming environment.

Felix splits its data into two parts, *shared* and the target, usually *host*. The data in *shared* is common to all targets; it consists of all platform independent data, including Felix source libraries, and platform independent C++ header files.

In the target *host* you will find object files and executables compiled for your specific operating system by the C++ compiler toolchain selected as the default.

3.2 Configuration Database

Inside every platform dependent target, there is a directory called *config*, you can examine it here from the Felix install directory:

```
ls build/release/host/config
```

You will see a bunch of files ending in extension *.fpc*. Each of these files specifies how to use a package with the basename of the file. Lets look at one:

```
~/felix>cat build/release/host/config/sdl2.fpc
Generated_from: 3674 "/Users/skaller/felix/src/packages/sdl.fdoc"
Name: SDL2
Description: Simple Direct Media Layer 2.0
cflags: -I/usr/local/include/SDL2
includes: '"SDL.h"'
provides_dlib: -L/usr/local/lib -lSDL2
provides_slib: -L/usr/local/lib -lSDL2
requires_dlibs: ---framework=OpenGL
requires_slibs: ---framework=OpenGL
```

This is the one on my Mac, which runs OSX. The first line tells how the file got produced, the second is a common name for the library which is just documentation too, as is the third line, which gives more information.

But now come the important lines.

- *cflags* tells the C++ compiler how to find the SDL header file
- *includes* tells Felix what the header file name is
- *provides_dlib* tells C++ how to link the dynamic version of the SDL2 library
- *provides_slib* tells C++ how to link the static version of the SDL2 library
- *requires_dlibs* tells C++ about dependencies of SDL2 dynamic library
- *requires_slibs* tells C++ about dependencies of SDL2 dynamic library

Note that values of these attributes are specific to *your* computer, and also sometimes to the C++ compiler selected.

When you install a third party library, you have to create an entry similar to that seen above for the library in the configuration database for each target where you want to use that library.

On Ubuntu, you will normally only need three lines

```
includes: '"mylib"'
provides_dlib: -lmylib
provides_slib: -lmylib
```

because C++ will automatically look in */usr/lib* for libraries, and in */usr/include* for header files. Do not include the leading *lib* of library file names, nor the trailing *.so*!

3.3 Felix Source Code

Now you have installed your library, and added entries in the configuration database, you can use the library like this:

```
type mytype = "mylib::mytype"
  requires package "mylib"
;
```

This Felix code lifts the C++ type *mylib::mytype* into Felix, naming it *mytype* in Felix, and it tells Felix that the library providing that C++ types is in package *mylib* .. which of course is just the name of the file *mylib.fpc* in the configuration database.

This is how Felix maps abstract component name used in source code into the compilation and linker switches needed to use the C++ and binary library components.

The source code is therefore platform independent, and you can run programs which uses third party libraries without specifying any linker switches or other platform specific nonsense every again. You have to specify it once, in the configuration database.

Felix is smart, it will only link the library if it is actually required. If you do not use values of the type *mytype* then the library will not be linked, if you do, it will be.

CHAPTER 4

Platform Concept

Felix uses a powerful abstract platform concept which is designed to support both personal use and enterprise level operation.

To understand how this works, we need first to understand the notion of a *platform*, and the platforms involved with Felix.

A platform is basically a specific computer with specific environment configured. Roughly, a Windows box is one platform, a Mac is another, and Linux is a third.

However this is overly simplistic, because for example on Linux you can compile C++ with GNU g++ but you could also use clang. For C code, these produce the different files with the same interfaces, but this is definitely not the case for C++. Even though they share the same ABI, the library templates are different and compiled code using C++ standard library components is not compatible.

4.1 Platforms

When you use Felix there are four significant platforms. These are often the same, but they need not be.

4.1.1 The build platform

The *build* platform is the platform used to build Felix itself. If you build from source that's your computer, but it may be that one day there is a Debian package for Felix, and in that case the Debian autobuilder is the build platform.

Similarly, if you can get the GitHub posted tarball of Felix to actually work, the build platform for that system is the Travis Continuous Integration Server.

4.1.2 The host platform

The *host* platform is the system on which you write Felix programs and cause the Felix compiler *flxg* to be run.

This is usually your computer, the one you are sitting in front of when you edit and compile. However, there are complicated systems where this is not enough of a description. For example Windows 10 can run Ubuntu, and you can write programs for Windows on that Ubuntu.

4.1.3 The target platform

The *target* platform is the one where the C++ compiler runs. It is usually the same as the host platform, but it need not be.

You can run Ubuntu on Windows 10, and you can run *flxg* under Ubuntu to produce C++ files, which are then compiled by MSVC++ using the CMD.EXE shell with an environment set up for MSVC++.

4.1.4 The run platform

The *run* platform is the machine where your generated binary code actually runs.

Commonly on a Mac, the host platform is your Mac, the target is the same Mac, but the run platform is actually iOS because you're building an iPhone app.

In this case, your target C++ compiler will be clang configured to generate code for the run platform, which is an ARM processor, even though the Mac is an x86_64 processor.

4.2 Cross-Cross Compilation Model

What this means is that Felix is actually a cross-cross-compiler, not merely a cross-compiler!

This means generating platform dependent code requires two steps of platform adaption. In the first phase the host Felix system may generate C++ code specific to the downstream build tools.

In the second phase, a C++ cross compiler may compile that code to binaries, specific to the downstream run platform.

For most users, the build platform, host platform, target platform and run platform will all be the same.

4.3 Felix target directories

Felix represents the platform model primarily by allowings the user to create a number of distinct target directories.

After building Felix, in the build directory, there are two files:

```
build/release/share
build/release/host
```

The directory named *host* is the default target. It represents the platform chain for which the host, target, and run platformn are all the same and use the defaults set up during the build processes.

On Linux, for example, this will normally be that you are generating C++ code for Unix, using the *g++* compiler tool chain compile it, and targetting 64 bit Linux.

If you want to also be able to use the clang tool chain, you would make a new target directory:

```
build/release/clang
```

and use it like this:

```
flx --target=clang hello.flx
```

This will use the clang toolchain instead of g++, and it will link against libraries built with clang, instead of those built by g++.

In the *GNUmakefile* there are two ready made make targets for making Felix targets for the iPhone simulator and the iPhone. These look like:

```
iphonesimulator:
  # prepare directory
  flx_build_prep --target-dir=build/release --target-bin=iphonesimulator --source-
↪dir=build/release \
          --source-bin=host --clean-target-bin-dir --copy-compiler --copy-pkg-db \
          --copy-config-headers --toolchain=toolchain_iphonesimulator --debug
  rm -rf build/rtl-tmp
  # build rtl
  DYLD_LIBRARY_PATH=build/release/host/lib/rtl flx_build_rtl \
    --target-dir=build/release --target-bin=iphonesimulator --static --noexes


iphoneos:
  # prepare directory
  flx_build_prep --target-dir=build/release --target-bin=iphoneos --source-dir=build/
↪release \
          --source-bin=host --clean-target-bin-dir --copy-compiler --copy-pkg-db \
          --copy-config-headers --toolchain=toolchain_iphoneos --debug
  rm -rf build/rtl-tmp
  # build rtl
  DYLD_LIBRARY_PATH=build/release/host/lib/rtl flx_build_rtl \
    --target-dir=build/release --target-bin=iphoneos --static --noexes
```

These are advanced uses of the Felix build tools which create the extra targets

```
build/release/iphonesimulator
build/release/iphoneos
```

which can be used on a Mac to build code for the respective run platforms like

```
flx --target=iphonesimulator filename.flx
flx --target=iphone filename.flx
```

These use clang for building the code, with options set for it to cross-compile to the relevant target.

Our purpose here is not meant to explain how to create a new target, merely to show that the Felix architecture is designed to support code generation for multiple targets.

In an enterprise install, the system administrator of a large network would create and install targets for each kind of developer, and put all of them up on a server, so the developers could pick the target they need to use.

CHAPTER 5

The Package Configuration Manager Tool flx_pkgconfig

Felix uses a configuration database to locate resources required to build and run programs. The code to inspect this database is built in to the *flx* tool, however there is a standalone executable, *flx_pkgconfig* you can use from the command line.

5.1 flx_pkgconfig

To run the *flx_pkgconfig* tool, it needs to be on your *$PATH*. This should be the case if you have installed Felix, or, if you have set Felix up for in-place operation. You should try this to verify if the system can locate it:

```
flx_pkgconfig --help
```

which should output something like this:

```
flx_pkgconfig [options] pkg pkg ...
  returns code 1 if any packages are missing unless --noerror is specified
  prints package or field list to standard output on one line
options: (follows GNU conventions)
  --path=dirname        set database directory name
  --path+=dirname       append database directory name
  --extension=fpc       set resource descriptor extensions,
                          default 'fpc' use 'pc' for pkgconfig databases
  -h
  --hide                only process first package in path with a given name
                          default, process all occurences
  --list                list available packages from specified set
  --missing             list missing packages from specified set
  --noerror             do not return 1 because of missing packages
  -r
  --rec                 form transitive closure of specified set based on Requires␣
↪field
  --rec=field           form transitive closure of specified set based on specified␣
↪field
```

(continues on next page)

(continued from previous page)

```
  -b
  --backwards           process specified packages in reverse order
  --field=field         collate values of field in package set
  --keepleftmost        remove duplicate values in output keeping only leftmost␣
↪occurrence
  --keeprightmost       remove duplicate values in output keeping only rightmost␣
↪occurrence
  --keepall             keep duplicate values in output
  @filename             Replace with arguments from filename, one line per argument
```

flx_pkgconfig queries configuration databases for information about packages. From the repository root, try this:

```
~/felix>flx_pkgconfig --path=build/release/host/config --field=Requires flx_run
flx_pthread flx flx_gc flx_dynlink flx_strutil
```

This causes flx_pkgconfig to search the configuration database in the directory build/release/host/config, and find all the packages on which the package *flx_run* depends.

It does this by seeking the field *Requires* in *flx_run.fpc*, and then recursively examining all the packages required, as specified by that field.

Now try this:

```
~/felix>flx_pkgconfig --path=build/release/host/config --field=provides_slib -r flx_gc
-lflx_gc_static -ljudy_static -lflx_exceptions_static
```

Here, we want to find the contents of the field *provides_slib*. We have to use the *-r* switch to also make flx_pkgconfig recursively follow all the *Requires* fields. The result is the list of all the values of any *provides_slib* fields found in the transivitive closure of the recursive search.

You would use this command to generate the flags needed to pass to your linker, in order to link library *flx_gc* statically. On the other hand to link dynamically you would use this instead:

```
~/felix>flx_pkgconfig --path=build/release/host/config --field=provides_dlib flx_gc
-lflx_gc_dynamic
```

The reason is that shared libraries link their own dependencies. It is important *not* to specify the transitive closure, because that would pre-empt the linker, and might pick the wrong version.

flx_pkgconfig is *not* a special purpose program, unlike its precursor, pkgconfig. It is, in fact, a fully general, if simplistic, database query tool, for databases consisting of records represented by files with lines of fields, each field having a name and one or more values.

Here is a sample *.fpc* file:

```
~/felix>cat build/release/host/config/flx_gc.fpc
Generated_from: 2403 "/Users/skaller/felix/src/packages/gc.fdoc"
Name: flx_gc
Platform: Unix
Description: Felix default garbage collector (Unix)
provides_dlib: -lflx_gc_dynamic
provides_slib: -lflx_gc_static
includes: '"flx_gc.hpp"'
library: flx_gc
macros: BUILD_FLX_GC
Requires: judy flx_exceptions
srcdir: src/gc
src: .*\.cpp
```

This file contains more than information required to use the Felix garbage collector. It also contains enough information for the Felix build system to build it. Normally with third party libraries, you build it with the vendors build instructions, but for Felix own components, those build instructions are put in the *fpc* file to localise the information about the library in one place.

5.1.1 Path

The *–path=* and *–path+=* switches are used to set the search path used by flx_pkgconfig. The first switch specifies the first directory on the search path. The second can be used to add directories to the end of the search path. When looking for a package, the directories are searched in order.

The *–extension=* switch species the extension of the filename to use, excluing the .. This is *fpc* for Felix databases, or *pc* for those constructed for pkconfig. Examination of the latter may or may not succeed.

The *–hide* or *-h* switch specifies package hiding. This means, once a package is found, its contents are the final, total, record of fields. By default, flx_pkgconfig finds all occurences of a package, so that a field named *field*, if not found in the first record of the package, may still be found in the second.

By default, if a *field* is present in both packages, and the query is for a list of values, the values of the fields of both packages will be merged. This machinery allows extending a database with auxilliary information, without modifying it.

5.1.2 Recursion

By default, flx_pkgconfig only searches the specified list of packages. If the *-rec* or *-r* switch is given, however, the search extends to dependent packages, based on the field named *Requires*. This is the only field with a special meaning.

If the switch *-rec=* is used, the specified field name is used for recursion instead of *Requires*.

5.1.3 Action

The default action of flx_pkgconfig is simply to verify that a list of packages given exists. It returns 0 if all the packages exist, or 1 if one or more is missing.

The *–noerror* switch suppresses error checking.

The *–list* option causes flx_pkgconfig to list, on a single line, all the packages it finds. Used with the *-r* switch, this will provide the transitive closure over dependencies specified by the *Requires* field.

The *–field=* switch specifies to output the contents of the specified field, instead of the package name. Nothing is output if the field is missing from a package.

5.1.4 Order of output

If the *–keepleftmost* switch is specified, then the leftmost occurence of a value in the output is retained, and duplicates to its right are dropped.

If the *–keeprightmost* switch is specified, then the rightmost occurence of a value in the output is retained, and duplicates to its left are dropped.

If the *–keepall* switch is specified, then duplicates are kept.

If the *-b* or *–backwards* switch is specified, then the final list of values to be output is reversed just before printing it.

flx_pkgconfig accumulates field values as it sees them. It processes a file from top up to any *Requires* field. If recursion is enabled, then it will process any packages specified in the *Requires* field next, before continuing to process the

current file. This means field specified before a *Requires* field are gathered before dependencies, and fields specified after a *Requires* field follow those dependencies.

Unix linkers normally requires that a library A requiring a library B be specified first. This means that when processing object files or libraries, the linker gather external references as it progresses from left to right in the link order, and satifies them as it finds external definitions. Thus, a definition seen prior to a reference will not satisfy it.

Other linkers require that external definitions be given first, so that when a reference is seen it is immediately satisfied. If the definition comes later, it will not resolve the reference.

Some linkers allow gathering of all the definitions and references in any order and then satify the references from the currently gathered set.

Because the ordering can matter, you must order fields in a flx_pkgconfig data base carefully, and use the order control switches.

5.1.5 Batched requests

Because searches can be long, you can put any sequence of switches and package names that would appear in a particular order into a control file. In this case, newlines may also separate the options. The sequence can then be includes by specifying the control filename prefixed by @. Control files can also refer to control files. Take care that inclusion is acyclic. Unlike *Require* fields, flx_pkgconfig follows inclusions without checking for cycles.

5.1.6 Merging

By default, if *-keepall* is not specified, duplicate field values are merged so a value will only occur once in the output.

5.1.7 Field specification

A field specification consists of a field name, followed by a colon *:* and then a list of values. If the list of values is too long for your taste, the list can be split over any number of field specifications, that is, the field name can be repeated.

Values are usually separated by spaces. If the value need to contain spaces, it can be quoted with single quote makes ‘ or double quote marks “. Quoting is also necessary if the value must contain quote marks. This is often the case for include file names, so you will see:

for example. Without the outer single quotes the first entry would lose the inner quotes which are part of a C include file specification.

5.1.8 Comments and ignored lines

Fpc files can contain blank linkes anywhere. You can also put lines starting with # as comments.

5.1.9 Substitutions

Flx_pkgconfig also allows substitutions. A line consisting of an identifier, followed by an = sign, followed by a value, defines a macro.

When processing field values, flx_pkgconfig replaces *${macroname}* with the specified value of a macro, if it exists.

5.1.10 Special Field handling

On some systems, options are given by two successive arguments. For example on OSX, frameworks are specified by

```
-framework OpenGL
```

Unfortunately, flx_pkgconfig cannot be given two values, *–framework* and *OpenGL* because the result would usually be a single *–framework* value and a lot of frameworks. On the other hand, you cannot specify a single value with quotation, because although this will collate correctly, the output to the *flx* tool, unlike the command line output, will be a single word which is passed to the shell, causing the compiler to get one argument, when it needs two in order.

The convention is to code this as:

```
`---framework=OpenGL`
```

that is, a single value, with a triple leading - character, and spaces replaced by = character. This is the format sane systems would use. Then, the two leading dashes are stripped off, and the remainder split by the equals character. *flx* uses this convention when processing results returned by *flx_pkgconfig* internally under program control. This is not part of the *flx_pkgconfig* system or tool, but a way to post-process the results if necessary.

CHAPTER 6

The flx tool

The executable *flx* is the primary tool used to interface components of Felix for the purpose of building, testing, and running Felix programs.

To work, *flx* must be on your *$PATH* or must be invoked using an absolute or relative pathname.

It is your responsibility to choose how your OS finds the *flx* program.

Once *flx* starts, it should be able to find everything else it needs automatically. By default, it uses the hard coded installation directory to find things, which is */usr/local/lib/felix/felix-version* on Unix systems or *C:usrlocallibfelixfelix-version* on Windows, where *felix-version* is the version of Felix installed.

You can override this hard coded default in several ways. The best way is to create a configuration file named:

```
$HOME/.felix/config/felix.fpc
```

on Unix systems or

```
$USERPROFILE\.felix\config\felix.fpc
```

on Windows. This file just needs to specify the installation directory:

```
FLX_INSTALL_DIR=/home/users/me/work/felix
```

You can also use an environment variable with the same name.

The *flx* tool needs to know the location of a lot of components, but it will derived most of these correctly from the single location given by the *FLX_INSTALL_DIR*.

If you build Felix from the git repository, then from the top level of the repository the installation directory is *build/release* on Unix or *build/release* on Windows. That is a relative file name, so you should set the variable to the corresponding absolute filename, which would be given by

```
echo $PWD/build/release
```

on Unix.

6.1 Basic Usage

Once it is all set up, you should be able to run felix programs like this:

```
flx hello.flx
```

This should work no matter what you current directory is, no matter where the target file is, and it should not write anything important anywhere except in $HOME/.felix/cache. You may find a couple of files like

```
flxg_stats.txt
```

floating about, that just contains some performance stats for the compiler, you can delete it freely, or complain so I can stop polluting your system.

6.2 Stand alone executable

To make a stand alone executable you should do this:

```
flx --static -c -od . hello.flx
```

and you will find the executable afterwards in the current directory. It will be named *hello* on Unix like systems or *hello.exe* on Windows. You can then run it as usual by *./hello* on Unix, or just *hello* on Windows.

The `--static` switch tells Felix to use static linkage, which is required for a standalone executable. Normally Felix uses dynamic linkage.

The `-c` switch tells Felix not to run the program.

The `-od .` means to set the output directory to the current directory ..

You can also use `-ox myhello` to set the output pathname to *myhello* without specifying the extension. If you want to specify the extension you can use `-o myhello.exe` for example. The reason not to do this should be clear, it makes the command OS dependent.

The reason for the `-od` switch is more subtle: *flx* has a batch mode which can run a whole set of programs based on a regular expression. In this case you don't know the name of the file to output, since it is determined by matching the regular expression against a whole directory of felix programs. When you build Felix, the batch mode is used to run all the regression tests.

6.3 Compiling Felix with added C++

Felix is specifically designed to work with C and C++. To this end, you can write Felix programs which requires C++ code you also supply.

It is not the purpose of this document to describe how to embed C++ into Felix. However let us assume you have Felix code which depends on the C++ file *mycxx.cpp*. You can then use this command to compile the C++ as well as the Felix code:

```
flx mycxx.cpp myfelix.flx
```

This will compile the mycxx.cpp file using the same C++ compiler Felix uses, generate C++ for myfelix.flx, and compile than, and link the compiled object files together and run them.

Felix does dependency checking on the C++ file. So it will not recompile the file if you do not change it.

Felix recognises the extension *.c* as C code, and *.cpp*, *.cxx*, and *.cc* for C++ code. We recommend you follow the convention that *.cxx* is used for translation units containing *main()* function, and *.cpp* for all others. This is the convention that Felix itself uses, and it has an impact when autobuilding Felix itself, using the specialised build tools.

6.4 Compiling Felix with added object files

Sometimes you want to compile C++ code to object files yourself. In this case you can just add the object files to the command line. On x86_64 platforms in particular you need to take care that you compile the file for the same operational model as you will use with Felix. With static linkage, you can then run your program like:

```
flx --static mycxx.o myfelix.flx
```

On Windows, object files have the extension *obj* instead. If you leave out the `--static` switch like this:

```
flx mycxx.o myfelix.flx
```

you need to be sure you have compiled for relocatable code. With g++ you may need the `-fPIC` switch on Linux. So-called position independent code (PIC) is slower than position dependent code due to the ABI used by Linux, together with the x86_64 architecture. This problem may or may not arise on other platforms. Felix is very careful to distinguish object files generated for static linkage and those for dynamic linkage. When in doubt, use `-fPIC` because such code can usually also be statically linked.

You can also use the `--obj=` switch:

```
flx --static --obj=base myfelix.flx
```

This will add *base_static.o* to the link on Linux. The suffix will be *_dynamic* for shared library build. The extension will be *.obj* on Windows. This switch just removes platform dependencies from the command line.

6.5 Compiling Felix with added libraries

You can also tell Felix to link extra libraries into your program. The easiest way is to just put the filename of the library on the command line. Make sure you compile with the right model!

This method of linkage always works for static linkage:

```
flx --static libmylib.a myfelix.flx
```

should link your program against the give static link archive on Linux. On Windows you would use:

```
flx --static mylib.lib myfelix.flx
```

If you compile in dynamic mode, you can also give library names like this, they will just be passed as written to the C++ compiler. This is definitely NOT recommended because it probably will not work.

A better way is to pass specific linker switches:

```
flx --static -Llibdir -lmylib.a myfelix.flx
```

This should work for both dynamic and static linkage. On Unix, the switches shown are just passed directly to the C++ compiler in link mode.

On Windows, the toolchain drivers use ths *same* switches, but attempt to translate for MSVC++. For example:

```
flx --static -Llibdir -lmylib.lib myfelix.flx
```

should work on Windows. Note that on Unix, the system will look for libmylib.a whereas on Windows, it will look for just mylib.lib, without the *lib* prefix. MSVC++ uses different switches than Unix, but the toolchain knows what `-L` and `-l` mean and map these switches over to MSVC++ syntax.

Using specific switches like this is not recommended except briefly for experimentation. It is much better to register the library in the configuration database.

6.6 Compiling C++ only

flx can compile and run C++ programs, programs witten entirely without any Felix. For example:

```
flx --c++ --static needed.cpp mainline.cxx -- args
```

All you need is to add the `--c++` switch. When you run C++ like this you must remember that the Felix configuration data base will not allow automatic linkage, as it does for Felix programs, unless you modify the source.

We need to use the special symbol – above separate the list of C++ files and the arguments to the program.

6.7 Upgrading C++ for autolink

Felix can autolink C++ as well as Felix, using the Felix configuration database.

To enable autolink for C++, all you need to do is put the requirements in the C++ somewhere, usually in comments. For example

```
// @requires package mylib
```

will tell *flx* that this C++ file requires the package *mylib*. When linking, *flx* will lookup the configuration database for the file *mylib.fpc* and link against the binary library as specified in that package, the same as it would for Felix programs.

This also works if you're building mixed C++ and Felix from sources. The dependent packages are stored in a file associated with the C++ source file name in the Felix cache, the same way as for Felix packages specified by

```
requires package "mylib";
```

in Felix sources. The upgrade to your C++ code has no impact on your normal C++ compilation. The library will be linked against automatically only if *flx* drives the C++ compilation process.

Note that whilst the package requirements in C++ allow autolinking, as well as providing search paths for header files, you have to *#include* the header files in your C++ in the usual way for C++. *flx* cannot currently inject the header file includes into C++ you supply because that would mean the C++ would not be compilable by a C++ compiler, with any switches.

You do not need to do this if you embed the C++ inside Felix.

6.8 Specifying Header file search paths

In order to compile C++ code, or to compile Felix code which embeds C++ which requires header files, you can specify a search path on the *flx* command line by:

```
flx --static -Imydir myfile.cpp myfelix.flx
```

The `-Imydir` switch extends the search paths used for C++ compilation for the C++ source file *myfile.cpp* as well as for compiling the generated Felix C++ code. In addition it *also* adds the directory to the Felix library search path, so any Felix files in the specified directory will be found.

6.9 Output Object Type

The normal mode of operation of *flx* is to run specified program. Execution can be inhibited by using the *-c* switch.

By default, *flx* generates a dynamic library, this is a shared library on Unix with *.so* extension on Linux, or *.dylib* extenion on OSX, on Windows, you get a *.dll*.

The action of a Felix *program* is just the side effects of the initialisation of a library, that is, programs in Felix do not really exist. Thus, a generated dynamic library can act both as a program and also as an actual library.

Felix comes with two executables, *flx_run* and *flx_arun* which can be used to run any dynamic Felix library.

If the `--static` switch is set, then object files are generated for static linkage. Otherwise, object files are generated for dynamic linkage. Dynamic link object files on x86_64 Linux systems require position independent code. Shared libraries must be built from dynamic link object files.

If `--static` is set, then Felix links the object code for either *flx_run* or *flx_arun* together with a stub adaptor against the object file of your program, to produce a stand alone executable.

To generate a static archive, use the `--staticlib` switch. This produces an *.a* file on UNix systems and a *.lib* file on Windows. Note that this option implies `--static`. However, you can still make static link library from dynamic object files. You need to first compile a dynamic object file, and then on a separate command combine it with any other dynamic object files using `--staticlib`.

The `--exe` switch tells *flx* to produce a static link executable. This is only necessary in special circumstances.

The `--nolink` switch inhibts linking so that the output object is now an object file. It can be combined with *–static* to produce a non-position independent object file. Unless overriden, *flx* produces static link object files with the source basename suffixed by *_static* and dynamic link, position independent object files with suffix *_dynamic*.

The `--nocc` switch inhibits C++ compilation.

The `--run-only` switch inhits all compilation and just runs the program, ignoring any dependency checking. Obviously this will fail if there is no program to run.

6.10 Output Location

By default, the output object of a *flx* operation will be placed in the cache as *$HOME/.felix/cache/binary/pathname* where *pathname* is the absolute pathname of the source file with the extension replaced depending on the output type and OS conventions, as well as the suffix for object files if the output type is an object file produced for a Felix source program.

The output pathname can be changed with the `-o pathname` switch. The given pathname is used instead of the default. This is discouraged because it is not platform independent.

The output pathname can be changed with the *-ox pathname* switch. In this case the pathname specified is used, except that the appropriate extension is added automatically. This is prefered over the plain `-o` switch because it is platform independent.

The output pathname can also be changed with the `-od dirname` switch. In this case, the output object is placed in the specified directory, with the name of the basename of the input file, and the appropriate extension. This option

is specifically design for use with batched compilations where the filename is not known, because the files to be processed are find by examining a directory and comparing filenames found in it with a regular expression. However this switch is also useful even if you know the filename because it avoid repeating it, and it is useful in a script, because it avoids the string processing required to remove the source extension.

When Felix translates a Felix program to C++ it normally puts the C++ files into the cache. You can override this with the `--output_dir=dirname` switch. This is primarily useful if you are cross compiling, where wish to capture the output files and ship them to another computer for C++ compilation.

The `--bundle_dir=dirname` switch bundles *all* the generated files for a program into a single directory. This includes resource control files, C++ output files, object files, executables, etc. This is sometimes useful when debugging, or when you need to ship some or all of the generated files to another computer.

The `--cache_dir=dirname` changes the location of the cache for this processing run. The cache is normally *$HOME/.felix/cache*. This is useful is you are running flx in a special mode, and it is *essential* if you are running *flx* simultaneously in multiple processes to avoid clobbering of cached files. Always use this if you are simultaneously building for different targets.

By default, Felix knows about targets and if you change targets the cache is cleaned automatically. Compiling from a clean cache takes considerable extra time, since the whole library has to be parsed and bound again.

6.11 Generic Performance Controls

flx provides several performance controls. The `--usage=level` control is a generic control over the compilation process. The level can be as follows:

6.11.1 hyperlight

Ultra fast performance, all run time checks stripped. Not recommended except for microbenchmarking tests.

6.11.2 production

For code to be shipped to clients. High performance run time at the expense of compile time, but includes run time checks.

6.11.3 prototype

For use developing a program, provides slightly faster compilation at the expense of some run time performance, and includes more run time checks and debugging controls.

6.11.4 debug, debugging

Provides the slowest output with the maximum debugging support. Object files should be produces with debugging information for debugger use. Comments in the generated C++ are expanded. Synthesised objects are reduced to make it easier to compare generated C++ with Felix sources.

Insecure run time debugging support is enabled. This includes run time UDP debugging traces on Unix platforms.

6.12 C++ compiler switches

Felix recognises certain switches and ships them to your C++ compiler. Only a fixed set of switches is recognised. In some cases, the switches may be translated by the underlying toolchain.

6.12.1 -Lword, -lword

Shipped to the linker.

6.12.2 -fword, -Wword

Shipped to the compiler. Sets warning controls and miscellaneous options. Compiler specific.

6.12.3 -Dword, -Dword=word

Shipped to the compiler. Sets macros. Translated for all compilers.

6.12.4 -O0, -O1, -O2, -O3

Shipped to the compiler. Tells the compiler which performance model to compile with. May interfere with instructions from Felix performance controls.

6.12.5 -Idir

Shipped to the compiler and also used by Felix. Specifies path for include file search.

6.12.6 –cflags=word

Shipped to compiler.

6.13 Debugging

The switch `--debug-flx` tells *flx* to emit progress and debugging information, especially about dependency checking.

The switch `--compile-time` tells the Felix compiler *flxg* to emit times for phases of execution. This is primarily useful to find exactly when a particular bug in Felix program is detected, since some error messages can be hard to understand.

The switch `--debug-compiler` turns on full debugging of the Felix compiler *flxg*. It is primarily for the developer of the compiler itself, not users.

The *–echo* switch tells *flx* to print commands it sends to the shell. You can also use the *FLX_SHELL_ECHO=1* environment variable to do this. That variable affects all Felix programs, including both *flx* itself and also any program it runs.

The `--force-compiler` switch forces *flx* to send Felix code to the Felix compiler *flxg* even if *flx* thinks the program and its dependencies are unchanged. This switch usually fails to achieve its intent because *flxg* also does dependency checking.

The `--clean` switch wipes out the entire cache forcing all compilation to run from scratch.

The `--nofelix` switch is used to inhibit translation of Felix code to C++. It does not prevent the other steps. This switch is used so you can add diagnostic prints to generated C++ code and rerun your C++ compilation, linkage, and execution steps, without *flxg* clobbering your edits.

6.14 Test Suites

The `--stdin=filename` switch tells *flx* to that when it runs a Felix program, to redirect standard input so it comes from the specified file.

The `--stdout=filename` switch tells *flx* that when it runs a Felix program, to redirect standard output to the specified file.

The *–expect=filename* switch tells *flx* that the expected output of a program is in the specified file. After the program has run, Felix checks the output agrees with the expected output.

The `--stdin`, `--stdout` and `--expect` switch are similar but they use the pathname of the Felix program with the extension replaced by *.input*, *.stdout* and *.expect* respectively. These switches are used to run test suites along with batch mode compilation.

6.15 Batch Compilation

flx can run a command multiple times, replacing the primary Felix filename with a each name found in a directory which matches a regular expression.

The `--indir=dirname` switch sets the directory to be examined for filenames. The `--regex=regexp` sets the regular expression used to filter the filenames. This is a Google RE2 compilant regular expression. Make sure you get the command line quoting correct. The regexp must match the whole of the filename relative to the directory specified in `--indir` switch.

The `--nonstop` switch tells *flx* to run a batch of compilations without stopping. By default, it stops when an error is detected.

6.16 Felix compilation control

The `--nostdlib` switch prevents Felix from automatically including the Felix Standard Library. When you use this switch, Felix is said to be running *raw*. Raw operation is useful for teaching and experimentation because it removes types and functions defined in the standard library from consideration.

Note that the grammar, which is defined in the library, is *not* disabled by using this switch.

The `--import=filename` switch imports a file, as if you had written *include "filename"* in *every* file. This includes not just your main Felix program but every file it includes, directly or indirectly.

The import switch is used to import macros, because macros are lexically scoped to the file in which they are defined, and cannot be exported to another file. By using the import switch a file with macros in it is made available universally. It is used by *flx* itself, to import the macros which specify the host operating system, to enable platform dependent code to be generated.

By convention, macro definition files use the extension *.flxh*.

The switch may also be given in the form `--import=@filename`. In this case the named file contains a list of files to be imported.

6.17 Targetting

Felix has a number of switches to control targetting.

The `--target-dir=dir` switch sets the directory in which target subdirectories are located. It defaults to the Felix installation directory. The `--target-subdir=dir` switch sets the subdirectory of the target directory which contains the actual target configuration. It defaults to *host*.

If you have built a target for say *iPhone* you can build code for the iPhone by

```
flx --target-subdir=iPhone iphoneprogram.flx
```

This will compile the generated C++ with options and header files specified in the configuration database for that target, and link against libraries specified in that target. So for example, on a Mac, you will end up with the above setup with an ARM binary suitable for running on iOS, for the particular API for the version of iPhone you set up. Felix does not help you set up the environment, but once you have done so, *flx* will compile and link automatically for any target.

A popular target on Linux is *clang*. You will normally use the *host* target configured for *g++* however if you also have the *clang* family of compilers you can target them instead of, or as well as, *g++*.

Be aware, that changing targets clobbers your cache, so if you are building for multiple targets it is a good idea to have separate cache set up for each one.

The `--toolchain=toolchain` switch can be used to change your toolchain without changing your target. You must take care with this option, because code generated by one toolchain may or may not be compatible with code generated by another.

The *flx* tool keeps track of the toolchain which is used to compile C++ codes. However, if you are supplying already compiled object files, it cannot report a mismatch.

The `--pkgconfig_path+=dir` switch *prepends* the specified directory to the search path used by the internal *flx_pkgconfig* database query tool. It can be used to enable searching for third party libraries which have configuration data in a location outside Felix. This is strongly recommended practice, since rebuilding Felix destroys all data in the installation directory before installing a clean upgraded copy. However the command line switch is not the best way to provide the location of this configuration database, you should use the *$HOME/.felix/config/felix.fpc* file instead.

6.18 Miscellaneous

The `--help` switch prints a list of switches.

The `--where` switch tells the computed location of the Felix installation directory.

The `--time` switch causes *flx* to report how long a program takes to run. The time does not include compilation time, only execution time.

The `--version` switch reports the current version of Felix for which *flx* was compiled. Be warned, *flx* can run other version of Felix. To find out the version of Felix source library being used, you have to run a program which prints the library version.

The `--repl` switch runs a rudimentary line at a time psuedo interpretive loop. You can use this for one liners. The repl accepts multiple lines up to a blank line, then compiles the code and runs it. The next paragraph of input is appended to the code you already supplied and the resulting text is compiled and run again.

The `--felix=filename` switch loads a configuration control file with a set of configuration settings. This can be used to provide detailed customisation of the configuration of the Felix system.

By default, *flx* looks for the file *$HOME/.felix/config/felix.fpc* and loads that if it is found, as if it were specified with the `--felix` switch.

CHAPTER 7

File System Tools

Felix provides a number of tools to aid with portable file system operations. The primary driving concept behind these tools is that of a flat space of structured filenames which subsets of which can be identified by regular expressions.

The standard for regular expression is Google RE2.

All tools use *unix* standard pathname conventions in regular expressions. So / is the correct separator to use, even on Windows.

All tools use *native* conventions for directory names.

7.1 flx_ls

The simplest tool, *flx_ls* takes up to two arguments. The first argument is a directory name, the second a regular expression. It searches the directory and all subdirectories thereof recursively for pathnames exactly matching the regular expression, relative to the directory, and prints those that match, one per line, with names relative to the directory.

If the regular expression is omitted, `.*` is assumed, this matches all files. If the directory is omitted, `.`, the current directory is assumed.

Given the following directory structure:

Here are some examples:

```
>flx_ls top
leafA
leafB
nodeX/leafA
nodeX/leafC
```

```
>flx_ls top '.*leafA'
leafA
nodeX/leafA
```

```
>flx_ls top '.*/.*'
nodeX/leafA
nodeX/leafC
```

When using bash we strongly recommend enclosing the regexp in single quotes to ensure the regexp is treated as a single word and not interpreted.

Remember, the regexp must the pathname completely, from the first to last character inclusive.

This formulation for finding sets of files is much better than the common *glob* however you must remember that `*` means zero or more occurrences, and that `.` means any character. To find all files with an extension of three characters you would use:

```
>flx_ls top '.*\.[^.]{3}'
```

See: https://github.com/google/re2/wiki/Syntax for details.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search