Felix Linkage Architecture

John Skaller

April 8, 2020

Contents

Chapter 1

Introduction

This document describes the architecture of the Felix system. The Felix compiler generates C++ which implements the architecture in conjunction with the Felix run time system, which is implemented in C++.

Chapter 2

Linkage Concepts

2.1 Kinds of Linkage

Most OS provide two kinds of executable binary:

1. An **executable program,** often called an EXE.
2. A shared **dynamic link library** often called a DLL.

and three kinds of linkage:

1. **Static linkage,** which produces an executable program or shared library from object files produced by a compiler,
2. **Load time dynamic linkage,** in which a program or shared library automatically loads and links dependent shared libaries, transitively, and
3. **Run time dynamic linkage,** in which user code loads and links a shared library under program control, dependent libraries being loaded automatically

Libraries loaded under program control are sometimes called *plugins.* Symbols in a plugin library must be located and linked to by string name by the user code.

Some libraries on some systems are always linked at load time, often the system C library cannot be linked statically. This is to prevent duplicated static global data in the even a program statically linked to the C library also loads a shared library statically linked to a distinct copy of the C library, leading to unpredictable behaviour.

2.2 Symbol Visibility

During compilation, symbols may have one of three kinds of visibility:

1. Internal linkage, the symbol is visible only at compile time within the current translation unit,
2. External linkage, the symbol is visible to other translation units during static linkage, and,
3. Export visibility, the symbol is made visible from an executable or shared library. Export visibility usually implies external linkage.

2.3 Symbol Tables

There are three basic models for making symbols available to satisfy requirements:

1. A global namespace for all external symbols is used during static linkage. Behaviour is undefined or system dependent, if a symbol is duplicated or missing.
2. A global namespace can also be used for all exported symbols during load time dynamic linkage, this is called a *flat namespace*.
3. A local namespace can be used when an executable or shared library loads a dependent shared library. The exported symbols of the loaded library are made available only to the loading entity. In other words symbol imports are private.

Local namespaces follow the library dependency structure and form an acyclic graph which disallows mutual dependency.

Felix supports global namespaces when static linking and local namespaces when dynamic linking. Global namespaces are not supported when dynamic linking.

2.3.1 Linux Issues

Note that some linkers. including GNU ld, allow shared libraries to be linked with unsatisfied dependencies. The assumption is the symbols will be provided by the executable in a global namespace.

Felix does NOT support such behaviour.

Note this excludes the possibility of embedding CPython which tries to load extensions with unsatisfied dependencies. Luckily this only impact Linux.

Chapter 3

Linker Target Kinds

Felix can direct the system linker to produce several different kinds of object:

1. Executable (EXE)
2. Shared Library (DLL)
3. Object library (LIB)

The default is a shared library.

An object library, called an *archive* on Unix systems is a single file just containing a specified set of object files. The Felix terminology calls these libraries a *staticlib*.

The Windows linker does not support dynamic linkage directly. Instead, when producing a DLL, and LIB file called an *import library* is also produced. The LIB file is statically linked into executables and DLLs that depend on the DLL. In turn, at run time, the code in the LIB file causes the DLL to be loaded and symbols in the file located. For this to work the client code must specify the symbol is either `dllexport` to export the symbol from the DLL, or `dllimport` to import a symbol.

Chapter 4

Object Code Models

There are two kinds of object file supported by Felix:

1. Position dependent
2. Position independent

On most systems these are identical, however on some Unix systems and some processors, they differ.

During compilation Felix uses the `-fPIC` compiler flag to make the C++ compiler produce position independent code. On the `x86` family of processors the system ABI requires position independent code to link a shared library. Position independent code is less efficient, requires distinct instructions to access shared library exports, and requires additional support from global offset tables (GOT tables) to access library functions.

Position dependent code is the default for static linkage, whilst position independent code is used for dynamic linkage.

To ensure the object files are not confused, Felix universally addes the suffix `_dynamic` to object files containing position independent code, and `_static` to object files containing position dependent code.

Executables always use position dependent code. Shared libraries always used position independent code.

Archives always contains object files which are all either postion dependent or positition independent.

Chapter 5

Shell Commands

Felix generates libraries, not programs.

5.1 Linker Output Name and Location

If linkage of object files is required, the linker output will be placed in the Felix cache by default.

5.1.1 Default Location

The default cache location is:

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

The absolute pathname of the directory containing the primary file is appended. For Felix compiles, the primary file is the Felix file, ending with extension `.flx` or `.fdoc`.

The basename of the primary file, that is, the name within its containing directory minus the extension (including the dot) is then appended.

Finally, an OS specific extenion for the target kind is added.

For example on my Mac:

```
~/felix>pwd
/Users/skaller/felix
~/felix>flx hello
Hello World!
~/felix>ls /Users/skaller/.felix/cache/binary/Users/skaller/felix/
build/          hello.dylib    hello.flx.par
```

(The par file is the parser output of the flx file.)

5.1.2 Specified Directory

You can specify the directory for the linker output using the `-od` switch followed by a space and the directory name. The directory will be created if it does not already exist.

For example on my Mac:

```
~/felix>flx -od fout hello.flx
Hello World!
~/felix>ls fout
hello.dylib
```

5.1.3 Output Bundle

You can also specify all related output goes in a specified directory using the `--bundle-dir=` switch followed immediately by a directory name.

```
~/felix>flx --bundle-dir=xout hello.flx
Hello World!
~/felix>lx xout
-bash: lx: command not found
~/felix>ls xout
hello.cpp        hello.rtti
hello.ctors_cpp  hello_dynamic.o
hello.dep        hello_dynamic.o.d
hello.dylib      hello_static_link_thunk.cpp
hello.hpp        hello_static_link_thunk_dynamic.o
hello.includes   hello_static_link_thunk_dynamic.o.d
hello.resh
```

5.1.4 Specified Cache

Finally you can also specify the cache location with the `--cache-dir=` switch followed immediately by a directory name.

5.2 Linkage Model

5.2.1 Dynamic Linkage

The default is to generate a shared, dynamic link library. The command:

```
flx hello.flx
```

generates a shared library in the Felix cache, then executes the program `flx_arun`. passing it the name of the library. `flx_run` loads the library, initialises it, and, if present, executes the procedure `flx_main`.

The programmer typically does not provide a `flx_main`. Instead, the side-effects of the mainline script is perceived as the program, although technically it is just the initialisation code for the library.

If you don't want to run the library, use the `-c` option.

```
flx -c hello.flx
```

You can also use `-c` with the other location controls.

5.2.2 Static Linkage

Produce Executable

To produce an executable instead of a shared library, use the `--static` option:

```
~/felix>flx --static -od sout hello.flx
Hello World!
~/felix>ls sout
hello
```

This produces a standalone executable instead of a library. The object file is produced in position dependent mode, and linked with a static link position dependent version of `flx_un` and a special object file called a *static link thunk*. The thunk has a fixed interface required by the static link version of `flx_run`, but contains variables whose values are the entry points of the generated object file.

```
~/felix>cat sout/hello_static_link_thunk.cpp
extern "C" void hello_create_thread_frame();
extern "C" void hello_flx_start();
void* static_create_thread_frame = (void*)hello_create_thread_frame;
void* static_flx_start = (void*)hello_flx_start;
```

The thing to note is the object file created imports the actual use symbols `hello_create_thread_frame` and `hello_flx_start` and exports fixed symbols `static_create_thread_frame` and `static_flx_start` which name variables containing the user symbols. All problems in computer science can be solved by introducing an extra level of indirection.

Produce Object Library

You can specify the `--staticlib` option to produce an object library instead of an executable:

```
~/felix>flx --staticlib -od statx hello.flx
~/felix>ls statx
hello.a
~/felix>ar -t statx/hello.a
__.SYMDEF SORTED
hello_static.o
```

Chapter 6

Library Model

The Felix compiler generates libraries not programs. The default library is a system shared library which acts as a plugin to the stub loader `flx_run` or `flx_arun` which loads the library dynamically. Special thunks are generated to also allow static linkage.

In the user program text, top level variables are aggregated in a C++ struct of type `thread_frame_t` called a *thread frame*. The top level executable code is gathered into a function called `modulename::_init_` which is the constructor code for the user part of the thread frame. The modulename is typically the base file name.

The thread frame also contains a pointer to the garbage collector profile object, the command line arguments, and pointers to three C `FILE*` values representing standard input, output, and error, respectively.

The main constructor routine `modulename_start` is an extern "C" function which accepts the garbage collector profile object, command line arguments, and standard files, stores them in the thread frame, and then calls the user initialistion routine to complete the setup of the thread frame.

The execution of the initialisation code may have observable behaviour. In this case the user often thinks of this as the running of the program.

Although the thread frame may be considered as global data, there are two things to observe. The thread frame, together with the library code, is called an *instance* of the library. More than one instance of the same library may be created.

In addition, code can load additional libraries at run time. If these are standard Felix libraries, they too have their own initialisation function and a constructor which creates an initial thread frame.

All thread frames contain some standard data, in particular, a pointer to the system garbage collector. Thread frames are shared by threads.

6.1 Entry Points

The standard entry points for a Felix library are:

1. `modulename_thread_frame_creator: thread_frame_creator_t`
2. `modulename_flx_start: start_t`
3. `modulename_flx_main: main_t`

Where:

```
typedef void *
(thread_frame_creator_t)
(
  gc_profile_t*,      // garbage collector profile
  void*               // flx_world pointer
);

typedef ::flx::rtl::con_t *
(start_t)
(
  void*,              //thread frame pointer
  int, char **,       // command line arguments
  FILE*, FILE*, FILE* // standard files
);

typedef ::flx::rtl::con_t *
(main_t)
(
  void*               // thread frame pointer
);
```

6.1.1 Example Hello World

For example, given Felix program, found in top level of repository as `hello.flx`:

```
println$ "Hello World!";
```

we get, on MacOS:

```
~/felix>flx -od . hello
Hello World!
~/felix>llvm-nm --defined-only -g hello.dylib
00000000000018c0 T _hello_create_thread_frame
0000000000001910 T _hello_flx_start
```

6.1.2 Thread frame creator

The thread frame creator accepts a garbage collector profile pointer and a pointer to the Felix world object, allocates a thread frame and returns a pointer to it. The thread frame creator is library dependent, because the thread frame contains top level variables as well as the standard variables.

Start routine

The start routine accepts the thread frame pointer, command line arguments, and standard files, stores this data in the thread frame, constructs a suspension of the user initialisation routine, and returns it.

The client must run the suspension to complete the initialisation.

If Felix is able to run the routine as a C procedure, the suspension may be NULL.

Generated C++

The actual C++ generated with some stuff elided for clarity is shown below. The header is shown in 6.1 and the body in 6.2. The macros used are from the Felix run time library, and are shown in 6.3 and 6.4.

The thread frame is accepted by the external routine but is not passed to the init procedure because it has been optimised to a C procedure which doesn't use the thread frame.

6.1.3 main procedure

The `modulename_flx_main` entry point is the analogue of C/C++ `main`. It accepts the pointer to the thread frame as an argument. It is optional. If the symbol is not found, a `NULL` is returned.

6.1.4 Execution

Loading and execution of dynamic primary Felix libraries is typically handled by one of two standard executables:

1. `flx_arun` is the standard loader, it loads the asynchronous I/O subsystem on demand

Figure 6.1: Hello header

```
namespace flxusr { namespace hello {

//PURE C PROCEDURE <64762>: _init_ unit -> void
void _init_();
struct thread_frame_t :
  ::flx::run::base_thread_frame_t
{
  thread_frame_t();
};

}} // namespace flxusr::hello
```

where

```
struct thread_frame_t {
  int argc;
  char **argv;
  FILE *flx_stdin;
  FILE *flx_stdout;
  FILE *flx_stderr;
  ::flx::gc::generic::gc_profile_t *gcp;
  ::flx::run::flx_world *world;

  thread_frame_t();
}
```

Figure 6.2: Hello body

```
namespace flxusr { namespace hello {

//Thread Frame Constructor
thread_frame_t::thread_frame_t() {}

//------------------------------
//C PROC <64762>: _init_
void _init_(){
  {
    _a17556t_66120 _tmp66124 =
      ::std::string("Hello World!") + ::std::string("\n") ;
    ::flx::rtl::ioutil::write(stdout,((_tmp66124)));
  }
  fflush(stdout);
}

}} // namespace flxusr::hello

//CREATE STANDARD EXTERNAL INTERFACE
FLX_FRAME_WRAPPERS(::flxusr::hello,hello)
FLX_C_START_WRAPPER_NOPTF(::flxusr::hello,hello,_init_)
```

Figure 6.3: Frame wrapper macro

```
#define FLX_FRAME_WRAPPERS(mname,name) \
extern "C" FLX_EXPORT mname::thread_frame_t *\
  name##_create_thread_frame(\
    ::flx::gc::generic::gc_profile_t *gcp,\
    ::flx::run::flx_world *world\
  )\
{\
  mname::thread_frame_t *p = \
    new(*gcp,mname::thread_frame_t_ptr_map,false)\
      mname::thread_frame_t();\
  p->world = world;\
  p->gcp = gcp;\
  return p;\
}
```

2. `flx_run` is a restricted loader that cannot load the asynchronous I/O subsystem

Figure 6.4: Start function macro

```
// init is a C procedure, NOT passed PTF
#define FLX_C_START_WRAPPER_NOPTF(mname,name,x)\
extern "C" FLX_EXPORT ::flx::rtl::con_t *name##_flx_start(\
  mname::thread_frame_t *__ptf,\
  int argc,\
  char **argv,\
  FILE *stdin_,\
  FILE *stdout_,\
  FILE *stderr_\
) {\
  mname::x();\
  return 0;\
}
```

6.2 Exports

A Felix library may contain arbitrary user defined entry points. These are created by the `export` operator.

6.2.1 Export directives

Felix provides stand-alone export directives as follows:

```
export type typeexpr = "cname"; // generates a typedef
export fun fname of (domain-type) as "cname";
export proc fname of (domain-type) as "cname";
export cfun fname of (domain-type) as "cname";
export cproc fname of (domain-type) as "cname";
export python fun fname of (domain-type) as "pyname";
```

The type export creates an alias in the generated export header.

The `fun` and `proc` exports export an extern "C" wrapper around top level felix routines with name fname, domain type as indicated, giving the wrapper the C name cname. The domain type is required because Felix routines can be overloaded.

These wrappers accept multiple arguments, the first of which is the thread frame pointer. if the Felix routine has a unit argument, there are no further parameters in the wrapper. if the Felix routine has a tuple argument, there is an additional argument for each component of the tuple. Otherwise, there is one further argument.

An exported Felix procedure is run by the wrapper, so it acts like a C function. Such functions cannot perform service requests.

The `cfun` and `cproc` exports generate the same wrapper but without the thread frame.

The `python` variant exports a `cfun` but also triggers the generation of a Python 3.x module table, which contains an entry for the function under the specified name. The module table is made available by also generating the standard CPython entry point `PtInit_modulename`.

As a short hand, a function or procedure definition can be prefixed by word `export`, which causes a `fun` or `proc` export to be generated, using the same C name as the Felix name.

6.2.2 C libraries

Felix compiler can also generate plain C/C++ libraries. Such a library contains only the explicitly exported symbols, does not have the thread frame creator, initialiser, or main symbols, and cannot use any Felix facilities since it has no access to the garbage collector or `flx_world` control. The exports for the library must be all `cfun` or `cproc`.

6.2.3 CPython extensions

Felix can generate of CPython 3.x extensions. If any function is exported as `python` a module table is created automatically and all the python exports included in that table. The standard entry point `PyInit_modulename`

CPython extensions coexist with all other library forms.

6.2.4 Plugins

A Felix plugin is a special Felix library object. It contains the usual thread frame creator an initialisation routine and two additional routines. The first is an extra setup routine, which accepts a thread frame pointer and a C++ string argument and returns an int.

In general, plugins are written in Felix not C or C++. Plugin loaders are not currently type safe.

1. `modulename_setup`

of C++ type:

```
int setup_t
(
  void * // thread frame pointer
  std::basic_string<char>
);
```

and Felix type:

```
  string -> int
```

It is called by the plugin loader after the standard initialisation, and is used to customise the library instance.

Plugins also contain at least one additional function, which is typically a factory function that returns Felix object containing the actual plugin API as a record. The Felix library contains some polymorphic routines for loading plugins.

6.3 Static linkage

All Felix libraries can be statically linked. If static linkage is selected, @tangler cstring.flx = share/lib/std/strings/cstring.flx the compiler will generate an object file called a static link thunk.

The standard Felix loaders, `flx_run` and `\flx_arun` find shared libraries and entry points by using the string name of the library. Static link versions of these files must use fixed names instead. To make this work, they link to a static link thunk which in turn links to the actual symbols.

6.4 Dynamic loader hook

Felix commands to load libraries in general, and plugins in particular, do not actually load libraries or link to symbols directly. Instead, the commands are hooked to first look in a database of loaded libraries and symbols. If the library and its symbols are found in the database, the relevant addresses are used instead of loading the library, or searching for the symbols required in it.

Otherwise, the library is loaded dynamically and the symbols required searched for. The resulting symbol addresses are then stored in the database.

The purpose of this mechanism is to allow static linkage of the library or plugin, avoiding a run time search. Note that even statically linked primaries can still dynamically link plugins. If a program requires known plugins, pre-linking them makes the program more reliable and easier to ship.

Felix has special syntax for populating the run time symbol database. Once populated, attempts to load the library and symbols will transparently use the pre-linked version instead.

List of Listings