Felix Linkage Architecture

John Skaller

March 22, 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
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. A symbols in a plugin library must be located and linked to by string name by the user code.

Most systems provide for mixed static and load time linkage. Felix does not support mixed linkage. However both statically linked executables and shared libraries can load plugins.

In addition, 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 Tables

Statically linked object files use a single global symbol table. Behaviour is undefined or system dependent, if a symbol is duplicated.

@tangler cstring.flx = share/lib/std/strings/cstring.flx Symbols in plugins are located under program control.

Symbol tables in shared libraries loaded as dependencies at the load time of another shared library or executable can be shared so they exist in a single flat namespace emulating delayed static linkage, or, they can be provide to the entity loading the library, this structure is called a two level namespace.

Felix does not support flat namespaces. External references in a library or executable can only be satisfied by that entities direct dependencies, conversely, the symbols in the loaded library are available only to the loading entity.

The two models Felix supports have distict architectures. Although Felix allows transparent selection of either dynamic or static linkage, it can only works because the Felix run time library (RTL) has been crafted to support both models, and if the user requires transparent selection of the model, care must be taken to design the linkage architecture appropriately.

Unfortunately some systems which one would like to embed in Felix use an inappropriate linkage model. In particular CPython extensions on Linux typically do not provide dependencies on the CPython run time and can only be loaded by an executable which includes the run time and exports its symbol table. This architecture is incoherent and is not supported by Felix. Python is linked using frameworks on MacOS which is the correct model.

Chapter 3

Shell Commands

Felix generates libraries, not programs.

3.1 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.

To generate the library without invoking it, use the `-c` option:

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

The library will be found at:

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

where the source extension is replaced by the extension used by the host OS. To change the location of the linrary use:

```
    flx -c -od directory hello.flx
```

which will put the library in the specified directory.

The library is generated from a single object file produced by the host C++ compiler. The base name of the object file is suffixed with `_dynmamic` and the host OS extensions. The object file will use `-fPIC` on Unix systems to produce position independent code. To produce only the object file and not perform linkage, use the `--nolink` option.

```
    flx -c --nolink hello.flx
    flx -c --nolink -od directory hello.flx
```

Note that this option simply skips linkage.

Chapter 4

Library Model

The Felix compiler generates libraries not programs. The default library is a system shared library which acts as a plugin to a stub loader 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` 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.

4.1 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
);
```

4.1.1 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
```

4.1.2 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.

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 4.1 and the body in 4.2. The macros used are from the Felix run time library, and are shown in 4.3 and 4.4.

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.

4.1.3 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.

4.1.4 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
2. `flx_run` is a restricted loader that cannot load the asynchronous I/O subsystem

Figure 4.1: Hello header

```
namespace flxusr { namespace hello {

//PURE C PROCEDURE <64762>: _init_ unit -> void
void _init_();

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(
  );

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

Figure 4.2: Hello body

```
namespace flxusr { namespace hello {

//Thread Frame Constructor
thread_frame_t::thread_frame_t() : gcp(0) {}

//------------------------------
//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 4.3: 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;\
}
```

Figure 4.4: 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;\
}
```

4.2 Exports

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

4.2.1 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.

4.2.2 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`.

4.2.3 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.

4.2.4 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.

4.3 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.

4.4 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