Felix Architecture

John Skaller

March 2, 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

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.

2.1 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
);
```

2.1.1 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
```

2.1.2 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 2.1 and the body in 2.2. The macros used are from the Felix run time library, and are shown in 2.3 and 2.4.

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.

2.1.3 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.

2.1.4 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 2.1: 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 2.2: 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 2.3: 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 2.4: 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;\
}
```

2.2 Exports

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

2.2.1 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.

2.2.2 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`.

2.2.3 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.

2.2.4 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.

2.3 Static linkage

All Felix libraries can be statically linked. If static linkage is selected, 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.

2.4 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.

Chapter 3

Routines

Felix has several different kinds of routines.

3.1 Displays

A display is simply a list of pointers to the last activation record of the calling parent, grand parent, and all ancestors, up to and including the top level thread frame object.

An activation record containing a display is called a closure. Note that closures capture activation record addresses not the values in them, so that if an activation record contains a mutable variable which is changed, the closure will see the changed value too.

3.2 Functions

Felix supports three function types:

```
D -> C      // standard function
D ->. C     // linear function
D --> C     // C function
```

3.2.1 Standard Function

The standard and linear functions are represented by an object containing a display, and containing a non-static member named `apply` which accepts as an argument a value of the function domain type, and returns a value of the function codomain type.

The caller address for a function is stored on the machine stack.

The representation of a Felix function is a C++ class derived from an abstract class which represents the function type. The apply method is virtual. Closures of functions are just pointers to an instance of the function. The C++ class constructor of a function accepts the display values and saves them in the function object, this forming an instance object.

The run time function pointer can be safely upcast to the type of the function, thus allowing a higher order function to accept and return function pointers. Note that function objects are heap allocated by default although the optimiser can stack allocate function objects if it is safe.

3.2.2 Function example

Given the following Felix function

```
fun f(x:int)=>new (x + 1);
var k = f;
println$ *(k 42);
```

the C++ header generated includes this code:

```
//TYPE 66321: int -> RWptr(int,[])
struct _ft66321 {
  typedef int* rettype;
  typedef int argtype;
  virtual int* apply(int const &)=0;
  virtual _ft66321 *clone()=0;
  virtual ~_ft66321(){};
};
```

which defines the function type, int a read-write pointer to int. The actual function is derived from this type:

```
struct f: _ft66321 {
  thread_frame_t *ptf;      // display: thread frame only

  int x;                    // parameter copy
  f(FLX_FPAR_DECL_ONLY);    // constructor
  f* clone();               // clone method
  int* apply(int const &);  // apply method
};
```

The contructor and clone method are given by:

```
//FUNCTION <64762>: f: Constructor
f::f(_ptf):ptf(_ptf){}

//FUNCTION <64762>: f: Clone method
f* f::clone(){
  return new(*PTF gcp,f_ptr_map,true) f(*this);
}
```

The macros in the constructor just pass the thread frame pointer.

The implemetation of the apply method is:

```
//FUNCTION <64762>: f: Apply method
int* f::apply(int const &_arg ){
  x = _arg;
  return (int*)new(*ptf->gcp, int_ptr_map, true) int (x + 1 );
}
```

The calling sequence is evident in the init routine:

```
//C PROC <64766>: _init_
void _init_(thread_frame_t *ptf){
  ptf->k  = (FLX_NEWP(f)(ptf)); //init
  {
    _a17556t_66323 _tmp66332 =
      ::flx::rtl::strutil::str<int>
      (
        *(PTF k)->clone()->apply(42)
      ) + ::std::string("\n")
    ;
    ::flx::rtl::ioutil::write(stdout,((_tmp66332)));
  }
  fflush(stdout);
}
```

3.2.3 Optimisation

The standard model for a function can be replaced by a more efficient representation. Felix can use an ordinary C function sometimes, or even eliminate the function by inlining. The display can also be reduced to contain only pointers that are required. In particular special binders can force the elimination of the display so that the resulting function, if it compiles, is not only C callable, but also has no direct access to Felix specific resources such as the thread frame or garbage collector.

3.2.4 C function type

For the C function type, a trick is used. Since C has no concept of a tuple, if the domain has a tuple type, the components are passed separately.

A C function can be used wherever a Felix function is required, this is done by generating a Felix function wrapper around the C function. The wrapper generation is automatic.

3.2.5 Subtyping rule

C functions and linear functions are subtypes of standard functions. Linear functions promise to use their argument exactly once.

Felix functions are not permitted to have side effects. This is not enforced, however the compiler performs optimisations assuming the rule is followed.

3.2.6 Generators

A special kind of function called a generator, which has the same type as a standard Felix function, is permitted to modify internal state.

The difference between a Felix function and generator is as follows: when a Felix function closure is stored in a variable, an invocation of the function calls a `clone` method to copy the object before calling the `apply` method on that copy. This is to ensure recursion works by ensuring the function get a separate data frame from any other invocation.

If the function was a generator, the clone method is still called but simply returns the this pointer of the object, ensuring that all invocations of the `apply` method use the same data frame. In particular state data is retained between applications, since the object is stored in a variable, and even more particularly generators can save program locations, allowing them to return a value and control in such a way that calling the `apply` method again resumes execution where it last left off.

3.3 Procedures

A procedure is Felix is radically different to a function. Procedure are given the type

```
D -> 0
```

where the 0 suggests that procedures do not return a value. The representation of a procedure consists of three methods.

1. constructor, creates closure

2. call method, binds arguments
3. resume method, steps procedure a bit

The procedure constructor, like a function, accepts the display to form a closure object. The call method, applied to the closure, binds the procedure argument into the data frame. The result is a new kind of object called a continuation object.

3.3.1 Continuation Object

Continuation objects all have the same type, `con_t` which is a base class for the procedure type abstraction, which in turn is a base for the actual procedure. The call method is a member of the type object, the resume method is a member of the continuation object.

```
namespace flx {namespace rtl {
struct con_t                    // abstract base for mutable continuations
{
  FLX_PC_DECL                   // interior program counter
  union svc_req_t *p_svc;       // pointer to service request

  con_t();                      // initialise pc, p_svc to 0
  virtual con_t *resume()=0;    // method to perform a computational step
  virtual ~con_t();
  con_t * _caller;              // callers continuation (return address)
};
}} //namespaces
```

Continuation objects all contain a variable called the program counter which tells where execution is up to within the procedure. Initially, it is set to the start of the procedure. When the resume method is called, some work is done, and then the procedure returns a pointer to a continuation object. It may be the same object, however if the procedure is returning, it could be the callers continuation, and if the procedure is calling another, it could be the called procedure's continuation object. A NULL is returned when the original procedure is finished.

Continuation objects are suspensions, the resume method is a function that in effect accepts a suspension and returns another suspension. The role of suspended computations will soon become evident!

3.3.2 Routines

For completeness, a routine is a procedure for which the current continuation is not passed implicitly. Felix currently uses procedure representation for routines and passes the caller address anyhow.

3.3.3 Coroutines

A coroutine is a procedure which, directly or indirectly, does channel I/O. It has the same representation as a procedure.

Technically, all procedures in Felix are coroutines. This includes the mainline!

3.4 Procedure operation

Procedures are run by repeatedly calling the resume method of a continuation object until NULL is returned. In Felix, the loop is also sometimes enclosed in a try/catch block to implement special semantics. The basic run loop is extremely simple and lightning fast:

```
while (p) p = p -> resume();
```

3.4.1 Resume Method structure

The resume method of a procedure for sequential flow is very simple:

```
con_t *resume() {
   switch (PC) {
   case 0: ... PC = 1; return this;
   case 1: ... PC = 2; return this;
      ...
   case 42: ... return caller;
   }
 }
```

In each case, at the end, the current continuation is simply the next case, except at the end, when the current continuation of the caller is returned.

3.4.2 Procedure Call

A procedure call is implemented by constructing a new procedure object, binding its arguments, and setting its caller to `this`, then returning that new procedure pointer. The call operation binding the parameters also sets the PC to 0 to ensure it starts in the right place.

3.4.3 Procedure Return

This is implemented by simply returning the stored caller's this pointer. When the driver loop calls resume() that procedure will continue where it left off.

3.4.4 Other operations

Procedures can also do various tricky things. For example the driver above can be augmented:

```
again:
  try { while (p) p = p -> resume(); }
  catch (con_t q) { p = q; goto again; }
```

This allows a procedure to throw a continuation object which then replaces the existing continuation.

3.4.5 Spaghetti Stack

Continuations are linked in two ways to form a spaghetti stack: there is a list of callers, terminated by NULL for the top level procedure call, and there is notionally a list of ancestors, particularly the most recent activation records of the ascestors, maintained through the parent pointer. Felix actually uses a display, so that every procedure contains a pointer to every ancestor, including the base ancestor, the thread frame. This reduces access time at the cost of additional storage in nested frames, however unused pointers can be optimised away.

3.4.6 Utility

The technology described above has very little utility in itself. There is no good reason for ordinary procedural code to return control to the driver, only to be called again immediately. The utility will become evident in the next section!

3.5 Procedure example

This example is a bit more complex that for functions;

```
proc f (x:int) { println$ x; }
proc g (x:int, h: int -> 0) {
  h x;
  h (x+1);
}

var k = g;
k (42, f);
```

The procedure types are:

```
//TYPE 66319: int -> void
struct _pt66319: ::flx::rtl::con_t {
  typedef void rettype;
  typedef int argtype;
  virtual ::flx::rtl::con_t *
    call(::flx::rtl::con_t *, int const &)=0;
  virtual _pt66319 *clone()=0;
  virtual ::flx::rtl::con_t *resume()=0;
};
```

and

```
//TYPE 66321: int * (int -> void) -> void
struct _pt66321: ::flx::rtl::con_t {
  typedef void rettype;
  typedef _tt66320 argtype;
  virtual ::flx::rtl::con_t *
    call(::flx::rtl::con_t *, _tt66320 const &)=0;
  virtual _pt66321 *clone()=0;
  virtual ::flx::rtl::con_t *resume()=0;
};
```

the functions are declared as:

```
//------------------------------
//PROCEDURE <64762>: f int -> void
//    parent = None
struct f: _pt66319 {
  FLX_FMEM_DECL

  int x;
  f(FLX_FPAR_DECL_ONLY);
  f* clone();
  ::flx::rtl::con_t *
    call(::flx::rtl::con_t*,int const &);
  ::flx::rtl::con_t *resume();
};
```

```
//------------------------------
//PROCEDURE <64766>: g int * (int -> void) -> void
//    parent = None
struct g: _pt66321 {
  FLX_FMEM_DECL
```

```
  _pt66319* h;
  int _vI64768_x;
  g(FLX_FPAR_DECL_ONLY);
  g* clone();
  ::flx::rtl::con_t *call(::flx::rtl::con_t*,_tt66320 const &);
  ::flx::rtl::con_t *resume();
};
```

The procedure f is defined with methods:

```
//PROCEDURE <64762:> f: Call method
::flx::rtl::con_t *
f::call(::flx::rtl::con_t *_ptr_caller, int const &_arg){
  _caller = _ptr_caller;
  x = _arg;
  INIT_PC
  return this;
}

//PROCEDURE <64762:> f: Resume method
::flx::rtl::con_t *f::resume(){
  {
    _a17556t_66323 _tmp66332 =
      ::flx::rtl::strutil::str<int>(x) + ::std::string("\n") ;
    ::flx::rtl::ioutil::write(stdout,((_tmp66332)));
  }
  fflush(stdout);
  FLX_RETURN // procedure return
}
```

and the procedure g with methods:

```
//PROCEDURE <64766:> g: Call method
::flx::rtl::con_t *
g::call(::flx::rtl::con_t *_ptr_caller, _tt66320 const &_arg){
  _caller = _ptr_caller;
  _vI64768_x = _arg.mem_0;
  h = _arg.mem_1;
  INIT_PC
  return this;
}

//PROCEDURE <64766:> g: Resume method
```

```
::flx::rtl::con_t *g::resume(){
  FLX_START_SWITCH
      FLX_SET_PC(66331)
      return (h)->clone()->call(this, _vI64768_x);
    FLX_CASE_LABEL(66331)
      {
        ::flx::rtl::con_t *tmp = _caller;
        _caller=0;
        return (h)->clone()
      ->call(tmp, _vI64768_x + 1 );//tail call (BEXE_jump)
      }
      FLX_KILLPC
      FLX_RETURN // procedure return
      FLX_KILLPC
    FLX_RETURN
  FLX_END_SWITCH
}
```

This illustrates the derivation heirarchy of procedures, and also shows how a procedure is called. The first call sets the program counter to the next operation after the call, then invokes the procedure call method, passing the current object this pointer as the caller, along with the argument. The resulting continuation pointer is returned to the driver loop. The called procedure is resumed which causes it to print its argument, then it returns the callers pointer.

The second call has an optimisation. Instead of passing this as the caller continuation, it passes g's caller instead. When the f procedure returns it returns that instead of this. This is possible because the second call is the last thing done, the optimisation is known as tail call optimisation. In fact the compiler is generating four extra instructions after that which will never be executed.

This example was contrived to prevent the compiler optimising the procedures to ordinary C ones. We had to force the generation of closures by storing g in a variable k, and passing f to it as a parameter.

3.6 Fibration

Fibration is a method of interleaving control between fibres.

3.6.1 fibre

A fibre, or fthread (Felix thread) is an object containing a continuation pointer. in Felix, fibre objects also contain a pointer to another fibre, possibly NULL, which is used as decribed below.

3.6.2 synchronous channel

A synchronous channel, or schannel, is a set of fibres, possibly empty, all of which are either waiting to read or waiting to write. It is represented in Felix by linking fibres together through the special link, and storing the head pointer in the schannel object, setting the low order bit of the pointer to 1 if the fibres are waiting to write.

3.6.3 scheduler

A scheduler is an object with two variables, one which contains the running fibre, and the other of which represents a set of fibres called active fibres. These Felix representation uses the special link in the fibre to represent the set of active fibres, and may be NULL.

The scheduler is a C++ procedure, which returns control when both there are no active fibres left and the running fibre is exhausted.

3.6.4 Channel I/O

There are two operations for channel I/O, read an write. They are identical except that data is transfered from the writer to the reader. In Felix, the data is always a single machine word. Read and write are said to be matching operations.

Read operates as follows. If you read a channel which is empty, or contains only readers, then the fibre is added to the channel reader list, and is removed as the running fibre.

If the channel contains a writer, one is selected and removed from the list of waiting writers, data is transfered from the writer to the reader, the reader is removed as the running fibre, and both reader and writer are made active.

Write operation is dual, except the direction of data transfer is still from the write to reader.

3.6.5 Scheduler operation

The scheduler runs the current fibre using the driver loop. However in addition, it also checks a special code in the continuations called a service request. This is a request to do a channel read, channel write, suicide or spawn operation.

The read an write have already been described. Suicide simply removes the currently running fibre. It is usually implemented by setting the fibre's continuation pointer to NULL rather than using a service request.

Spawn takes a newly created fibre and makes it active, also making the currently running fibre active but no longer running.

The scheduler operation is to start with some fibre as running, and then, after a service request, the previously running fibre is no longer running, so the scheduler picks another one to run, removing it from the active set. If the active set is empty, it returns control.

Although the formal semantics allow the scheduler to randomly pick any active fibre to run, in Felix the operation is deterministic. When a reader and write match up on an I/O request the reader is always made active first, and the writer ends up on the top of the active list so it will run next if not displaced.

Spawn always pushes the running fibre onto the active list and runs the newly spawned fibre. This means that spawning and procedure calling are identical if the spawned fibre makes no service requests.

List of Listings