# The Ferite Developers Guide - Extending and Embedding The Ferite Engine

**Chris Ross**

**chris@darkrock.co.uk**

**Eric Shorkey**

**The Ferite Developers Guide - Extending and Embedding The Ferite Engine**
by Chris Ross and Eric Shorkey

Copyright © 1999-2002 by Chris Ross

This documentation is released under the same terms as the ferite library.
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

# Table of Contents


**1. Introduction ..... 1**
**2. Creating Basic Modules ..... 3**
**3. Creating Native Modules ..... 5**
- Introduction ..... 5
- Builder ..... 5
- Ferite-C File Contents ..... 6
  - module-header ..... 6
  - module-init ..... 7
  - module-deinit ..... 7
  - module-register and module-unregister ..... 8
  - Native Functions, the builder way ..... 8
  - Classes and Namespaces ..... 10
  - Finally ..... 11
- Without Builder ..... 11

**4. Accessing Ferite Internals ..... 13**
- Introduction ..... 13
- The Memory Manager ..... 13
- Working With Variables ..... 13
  - Accessing a Variable's Data ..... 13
  - Changing a Variable's Type ..... 15
  - Creating and Destroying Variables ..... 15
- Working With Namespaces ..... 16
- Working With Objects And Classes ..... 18
  - Creating Classes ..... 19
  - Creating Objects ..... 19
  - Accessing Variables ..... 20
  - Accessing Functions ..... 20
- Calling Functions ..... 21
  - Namespace Functions ..... 21
  - Object and Class Functions ..... 22
  - Function Shortcuts ..... 23
- Raising Exceptions and Reporting Errors ..... 24
- Executing Code Snippets ..... 24

**5. Native Modules - By Hand ..... 25**
- Functions ..... 25
- The Rest ..... 27

**6. Embedding Ferite ..... 29**
- Getting The Engine Purring ..... 29
- Fake Native Modules ..... 31
- Cheating With Builder ..... 32

**7. Finally: Step By Step Examples ..... 33**

# Chapter 1. Introduction

Before you begin, make sure that you have read the Ferite user's manual. It will familiarize you with terms and concepts that are necessary for understanding the material that will be presented to you here.

This manual is designed to serve several large purposes, and Ferite is presently quite a moving target. Therefore, this document is written to describe a very specific version of Ferite, and much of its content may not apply to previous or future versions. The exact version of Ferite that this document applies to can be found in the title. I do my best to keep this manual up to date, but if you find inconsistencies or statements that are no longer accurate, please report them. The latest version of this document, as well as any past version can be found on the Ferite web site (http://www.ferite.org).

The individual sections are arranged to first give you a quick understanding of how things generally work, and then go into further detail. The chapters are progressively arranged, each chapter building upon the those that preceded it. It is recommended that first time readers read straight from the beginning through to the intended section.

If you are simply interested in Embedding ferite, you could probably just skip to that section Then again you wouldn't be armed with the knowledge to do much more than simply loading a pre-built script and running it, having no connection between your application and ferite itself. In order to really make ferite do useful things, read everything leading up to what you want to know.

It is also suggested that you read this document with a copy of the C api documentation as it points out not only the mentioned methods but also the methods that are releated to them.

# Chapter 2. Creating Basic Modules

Creating a module is actually quite simple. At its base level, a module is really nothing more than a ferite script with a specific name that resides within ferite's module search path. By default, ferite will look for modules in the current directory and the system's ferite module directory (usually this is `/usr/lib/ferite`). The module must have a file extension of either `.fe` or `.fec` in order to be recognized by ferite. (`.fec` denotes a special kind of script that will be covered later)

So essentially any script that you write can be included as a module. A script can import modules and other scripts by using either the uses keyword, or the `include()` operation. When you import a module, you refer to it by it's filename, minus the .fe or .fec extension. So mymodule.fe would be imported by `uses "mymodule";`. Modules may also import other modules. If an application imports a module, which in turn imports other modules, then all of the functionality exposed by the implicitly imported modules will be available.

The following example shows a script importing a module and accessing an exposed function, and the module that is imported. They are in separate files residing in the same directory.

**File 1 (the importer); Name: myscript**

```
uses "mymodule";

foo.bar();
```

**File 2 (the module); Name: mymodule.fe**

```
uses "console";

namespace foo{
 function bar(){
  Console.println("Hello there!");
 }
}
```

**Execution and result:**

```
$ ferite myscript
Hello there!
$
```

In the previous example, the module had exposed a namespace (foo), and a function within that namespace (bar). However, this is not the limit of what can be exposed. Modules can expose functions, classes, namespaces, and global variables. Like regular scripts, modules can also modify existing namespaces and classes by using the modifies keyword.

There is nothing special that a module must do in order to expose functionality. When a module creates a namespace, it is automatically exposed. The same goes for classes, functions and global variables.

Something that should be noted, is that any code in the anonymous function of the module will be executed when the module is first imported. A module can be imported numerous times, but this will not cause the code to be executed more than once. You can safely put run-once initialization code in a module's anonymous function.

Here is an example of a module taking advantage of several abilities.

**Name: myothermodule.fe**

```
uses "console";

global {
 number gMyNumber = 7;
}

class myclass{
 string WhatISaid;

 function myclass(string WhatToSay){
  Console.println(WhatToSay);
  self.WhatISaid = WhatToSay;
 }

 function tryme(){
  Console.println("You called myclass.tryme()!");
  Console.println("When created, I said: " + self.WhatISaid);
 }
}

namespace mynamespace{
 function hellothere(){
  Console.println("Hello there!");
 }
}

function plainfunction(){
 Console.println("You called plainfunction()!");
}

Console.println("I could be a module initializer!");
```

This code would result in gMyNumber being exposed as a global variable. The class 'myclass' would be available, as well as all of it's class members. The namespace 'mynamespace' would also become available, and it would house a single function called hellothere(). You would also get a function called plainfunction() placed in the main namespace, accessible simply by it's name. And to top it off, upon the importing of the module, the Console.println statement would be executed. This is a very important feature to note, as it allows for module writers to place initialisation code that will be executed.

# Chapter 3. Creating Native Modules

## Introduction

Note: This is a particularly complex section, covering a lot of intertwined concepts. You should read it straight through at least once before attempting anything on your own. It may help to read it multiple times, as certain concepts are mentioned long before they are fully explained.

A native module is a ferite module that contains native code to interface with the surrounding system. This can be of two main forms, a mix of both native code and ferite script [which is how the base modules for ferite are written] or they can be completely made up of native code.

The first type of native modules are written in ferite-c files, which are denoted by the `.fec` extension. When they are fully built, they can create either C libraries or shared objects that contain the native C code. In most cases, the .fec file is still necessary, as it may contain non-native code. These modules are partially auto-generated by using the `builder` command line tool [which comes with ferite] and are discussed from the next section onwards.

Modules that do not require the ferite-c file at runtime are dicussed later, after the `builder` tool and accessing the internals of ferite has been dicussed. This is due to the fact that they are harder to write and require knowledge of the ferite internals.

## Builder

Ferite-c files (.fec) are compiled using a special tool, called the builder which is run on the command line. The builder is only used for the creation of native modules. It is not required in order to run pre-built native modules. Depending on your ferite installation, you may need to install a development package to have access to the builder.

**What does builder do?**

The builder reads a ferite-c file, and creates the necessary C source, header files, and automake file that will be needed to compile the module. It takes several command line parameters, only a few of which we will cover here. You can pass builder either `--help` or `-h` on the command line to see all of the available options.

The switch we are currently most interested is -m. The -m switch allows you to specify the name of your module to the builder. This name will be used to determine the names of the files builder will create while reading the ferite-c file. If you do not specify a name using -m, it will default to `modulename`. For simplicity we will also use th -c and -f switches, which prevent the creation of a config.m4 and Makefile.am, respectively.

**Example of using the builder:**

```
$ builder -c -f -m mymodule mymodule.fec
```

When you run builder, it will create several output files, named according to the module name. The main files created are:

- *modulename*_core.c (holds native module init and deinit functions)
- *modulename*_misc.c (holds native code for the anonymous/_start function, if any)
- *modulename*_header.h (holds include statements that the various c files need)
- *modulename_classname*.c (you will get one of these for every class defined in the .fec)

- *modulename_namespacename*.c (you will get one of these for every namespace in the .fec)

These files will need to be compiled into a shared object or a DLL (depending on your platform, for simplicity we will simply refer to shared objects from here on, but they are interchangeable with DLL's). Both the resulting shared object and the original ferite-c file are needed for ferite to successfully import the module. You will need to place the ferite-c file in the module path, which was explained in the previous chapter. The shared object will need to be placed in the native search path. This is where ferite looks for all native modules. It is usually /usr/lib/ferite/*platform*, though the actual location may vary depending on the installation. (ex. /usr/lib/ferite/linux-gnu-i686)

**Note:** If you are interested in auto generation tools for standalone modules, you will probably be interested in the generate-module utility. The builder creates input files for automake and the like specifically tailored for modules that will be included with the ferite source. The generate-module utility is geared more towards auto generation for standalone modules. We will not cover generate-module in this guide.

## Ferite-C File Contents

Ferite-c files are very similar to basic modules. In fact you can quite easily run the builder on a basic module. You just would end up with a lot of source files that didn't have much content. In order to get some content into those files, we need to tell the builder what parts of our module are written in C, instead of ferite script. To do this, there are several new sections and keywords that we can place within our ferite-c file.

```
uses "modulename.lib"
```

One of the most important pieces of a ferite-c file, is a uses statement at the top that tells ferite at runtime to load the shared object file for the native module.

When you compile the files that builder creates into a shared object, ferite has no way of knowing the resulting file's name. Usually, people will compile it into a file called *modulename*.so, where *modulename* is the name of the module. However this is not required. You could quite easily compile a module from source obtained by building bob.fec, and call it jimmy.so.

The solution is to explicitly tell ferite to load a shared object by name. This is done with a special case of the uses statement. The syntax is much like the normal uses statement, only you place a .lib extension on the name of the module that is to be imported. This is such that ferite can know to load the native library for that platform without forcing the programmer to take into account specifics of that platform.

```
uses "bob.lib";
```

This will tell ferite to look in the native module path for a file called bob.so on `Linux` and bob.dylib on `Mac OS X`, and to import it. This type of a uses statement can also be used within a regular ferite script to load a native only module. (How to create native only modules is covered later.)

### module-header

The module-header section is where you will place any #include statements, or #define statements, or anything else that you expect your native code will need. The syntax for creating a module-header in a ferite-c file is much like defining a global section in a regular script. The code that is declared within the `module-header` is availible in all generated C files.

For example:

```
module-header{
 ...your headers go here...
}
```

Anything you place within the module-header section will be placed in the *modulename*_header.h file when builder parses the ferite-c file. This header is then included in every C source file that builder creates. You can have as many `module-header` blocks, the code will just be all placed together in the header file.

Here is an example of a module-header:

```
module-header{
 #include <stdio.h>
 #include "utility.h"
}
```

The builder doesn't do any validity checking in between the curly braces. So if you have typographical errors, you probably won't know until you try to compile the module.

## module-init

This section allows you to specify native code that is executed when the module is loaded into a script. It is an optional section, but builder will create an empty module-init function in the C source.

The module's anonymous function (sometimes referred to as the _start function) is also executed when it is first imported, but module-init code is executed first. Also, the _start function cannot contain native code. So if your module initialization requires multiple jumps between native and ferite code, you can use the _start function to call native functions where necessary and use ferite code for everything else.

The syntax for creating a module-init section is similar to module-header:

```
module-init{
 ...your code goes here...
}
```

This will cause all of the code placed within the curly braces to be placed in the module's init function. Incase you're interested, the build destination is the *modulename*_core.c file, in a function called *modulename*_init(). The function returns void and has 1 parameter, "FeriteScript *script", which is accessible to the code within the section.

## module-deinit

This section is syntactically almost identical to the module-init section. Like module-init, module-deinit is not a required section. Again, the builder will create empty module-deinit function in the C source for you if you do not specify one.

Code in this section is executed when the script that loaded the module is being deleted. More precisely, it is run by a call to ferite_script_delete(). However, you usually don't have to worry about the specifics unless you're embedding ferite in your application. For most purposes, just know that this code is run when the script has finished executing.

Here is an example of a module-deinit section:

```
module-deinit{
 ...your code goes here...
}
```

As you can see, it is basically the same as module-init. The return type is void, so you shouldn't try returning anything from this function. It also has the affected script passed into it, which is accessed exactly the same as you would for module-init.

## module-register and module-unregister

When a native module's shared object is loaded, it's register function is called once. This allows the shared object to setup any system specific things. Symetrically, module-unregister is only called once, and that is when the ferite module system decides to unload the shared object. They are both blocks of code like module-init and module-deinit and should be used the same way.

## Native Functions, the builder way

When developing a native module with builder it will be necessary to create functions that can be called by ferite scripts. To make this easy there are only two main differences between a ferite function and a native function. These are the keyword `native` and that the bodies of the functions are written in C.

First we'll start with an example of how to declare a simple native function:

```
native function foo(){
 ...your code goes here...
}
```

This would result in the C source between the curly braces being placed in one of the C source files. The exact file and the exact function name created depends on the namespace or class that the function is declared in. This might vary from version to version so I won't get into it here, but feel free to look at the source created. You'll probably be able to figure it out from there. To a scripter the function looks and tastes the same as a normal ferite function.

It should be noted that within each function the following variables are accessable:

- `script` - a pointer to the FeriteScript in which the function was called.
- `function` - a pointer to the FeriteFunction which owns the function executing.
- `params` - the null terminated list of parameters [see Calling Functions for more information].
- `self` - **Note!** only for object methods, a pointer to a FeriteObject on which the function is being executed.

### Parameters

The next step is to pass in some variables, and it is pretty easy to do. Simply declare the variables as you would normally do for any ferite script. When you get inside of the function, the values passed in will be converted to units that are workable in C with the same name. Complex objects will be presented to you in the form of pointers to different types of structs according to their type. All variables are available by the names you gave in the function declaration. Following is a quick breakdown of the different types and how they convert.

**Table 3-1. Parameter Types**

| | |
|---|---|
| number | double |
| string | FeriteString * |
| object | FeriteObject * |
| array | FeriteUnifiedArray * |

- number - Numbers are converted to doubles because doubles can represent LONG_MAX, and ferite numbers support floating point values anyways. If you expected to use the value as an integer in your function you can simply cast the double to a long. It is a good idea to check that the number passed in is not greater then LONG_MAX before you cast it to an long, otherwise you might end up with some funny looking results.
- string - Strings are converted to FeriteString *, and their C-string values are accessible by struct element 'data'. So you can retrieve the value of string mystring by mystring->data. Following is an example that accesses a string's value by using it in a call to strdup().

```
native function foo( string mystring ){
 char *mystring_copy;
 mystring_copy = strdup( mystring->data );
}
```

- object - Objects are instances of classes, which must be accessed by reaching into ferite's internals. This is covered in the next chapter "Accessing Ferite Internals".
- array - Arrays must also be accessed by reaching into ferite's internals. Again, this is covered in the next chapter "Accessing Ferite Internals".

## Return Values

So now you can pass variables into functions. Next you need to know how to return values from functions. Any time you don't specify a return value and your function runs off the end of its scope, ferite will assume you meant to return void. If returning void is not the desired effect, or you would like to specify a position to return from other than running off the end of the function's scope, you will need to specifically return a value using one of the following C macros.

Note! By default a function generated by builder will automatically return void. You only need to specify returns if you want.

- `FE_RETURN_VOID` - returns void to the caller, this is synonymous with not returning anything.
- `FE_RETURN_TRUE` - returns true to the caller.
- `FE_RETURN_FALSE` - returns false to the caller.
- `FE_RETURN_LONG( value )` - returns a number to the caller with the contents of the given long.
- `FE_RETURN_DOUBLE( value )` - returns a number to the caller with the contents of the given double.
- `FE_RETURN_STR( string, freeme )` - returns a FeriteString* to the caller. The parameter "string" is passed in as a FeriteString*. If freeme == FE_TRUE, string is freed using ferite's memory manager. if freeme == FE_FALSE, it is not freed at all.

- `FE_RETURN_ARRAY( pointer to array )` - returns an array to the caller.
- `FE_RETURN_OBJ( pointer to object )` - returns an object to the caller (objects are instances of classes).
- `FE_RETURN_NULL_OBJECT` - returns a null object to the caller (useful for functions that are expected to return an object, but need to signify an error condition).
- `FE_RETURN_VAR( variable )` - returns a FeriteVariable to the caller. This allows you to return a variable that you have created yourself to the engine. It will tag the variable allowing ferite to clear it up when it is finished with. If you want to return a variable, but keep hold of it, you must simply return the variable as you would an item from a normal c function. eg:

  ```
  return someVar;
  ```

All of these macros actually convert the given return values into a FeriteVariable * which is then returned to the caller. As a general rule, you should always use the available macros when mixing C and ferite to prevent your functions from breaking if the interface ever changes. These macros will be kept up to date, so you are safe to use them.

### And Finally

The previous few sections should give you enough information to get you up on your feet and writing native functions. To play with ferite's internals you will need to read on where this is dicussed in depth.

## Classes and Namespaces

Classes and namespaces in native modules work exactly like their non-native counterparts. You simply declare the namespace or class in the ferite-c file, and when a script tells ferite to import the module, ferite will parse the .fec and create the namespaces and classes as usual and link up the native functions from the shared object. There is absolutely no syntax change for creating classes and namespaces. Pretty easy isnt it?

There is, however, the added ability to place native functions within classes and namespaces. The syntax for doing so is no different that what you've already seen, just place them within the curly braces of the namespace or class that you would like them to be a part of.

Here is an example of a native function in a namespace:

```
namespace foo {
 native function bar() {
  ...your code goes here...
 }
}
```

And here is an example of a native function in a class:

```
class foo {
 native function bar() {
  ...your code goes here...
 }
}
```

You can also make functions in classes static, as was described in the user manual. So of course we can make those native functions as well. Simply place the keyword static in the function declaration.

```
class foo{
 static native function bar1(){
  ...your code goes here...
 }
 native static function bar2(){
  ...your code goes here...
 }
}
```

Both bar1() and bar2() are native functions that are static within the class foo. The order of the keywords does not matter.

### Object Data

When a native function belonging to an object is called, there is the `self` variable availible. This is a pointer to the FeriteObject which is currently executing. Now, to make life easier there is a part of the FeriteObject structure that allows you, the module writer, to attach any data to it. This is called [and refered to] as `odata` and is short for object data. Ferite does not and will never touch this, it is the job of the programmer to deal with it. It is very simple to use, simply access the odata member on self.

```
self->odata = get_some_resource();
```

Most of the time, the odata pointer is setup when the object is constructed and cleared up when the object is destroyed. This is simple to do as you merely write a native constructor and destructor. Most of the module code for ferite makes use of this feature and to make things more straight forward, a macro called SelfObj is declared casting odata into what ever form is stored there.

Example: In the 'Sys' module, odata is used to store a pointer to the FILE* pointer for file streams.

```
#define SelfObj (FILE*)(self->odata)
```

## Finally

This chapter should have helped you get off your feet and understand the way in which builder can help you not only rapidly develop modules but keep them very close to ferite code. You should look at the `.fec` files that ship with ferite to clarify any doubts you have.

## Without Builder

The natural flow of this document means that creating native modules without builder should be dicussed here. As most of the dicussion requires knowledge dicussed in the next chapter, this will be left until after the internals of ferite have been looked at.

# Chapter 4. Accessing Ferite Internals

## Introduction

This section is designed to teach you how to access, modify, create, and destroy various structures within ferite. It covers all of the different types of variables, functions, classes, and namespaces. It will first cover very basic memory management, then cover variables, namespaces, calling functions, calling object and class functions, and creating a class.

It should be noted that this chapter will cover the registering and accessing of methods, but wont tell you how to write one from scratch manually. That will be left for the next chapter where native modules by hand will be dicussed.

## The Memory Manager

Before we get started, we should cover ferite's internal memory manager. Under normal operation ferite uses it's own memory manager, which is basically a sub allocator, to achieve some significant performance gains over the s tandard malloc/free operations. This memory manager is used throughout ferite, and the data that is passed around in ferite is expected to be allocated under this manager. Using malloc to allocate memory for ferite internals will cause unexpected results. This is not to say that you cannot use malloc/free at all. That is not the case. Just don't use malloc/free for anything that you give to, or get from, ferite.

This new memory manager acts much like malloc/free in terms of how you use it. There are functions that mirror the malloc, calloc, realloc, and free calls.

- fmalloc( size )
- fcalloc( size, blocksize )
- frealloc( ptr, size )
- ffree( ptr )

These functions all look like and act like the functions they replace as far as the casual programmer is concerned. However, they smell different as far as the compiler is concerned, so don't mix calls on memory segments between malloc/free and fmalloc/ffree. They play well in the same sand-box, but don't ask them to swap tonka trucks with each other.

Unfortunately this means that there are only limited equivalents to the standard C functions that perform allocations such as strdup. More are likely to be implemented as time progresses, you can find the current available functions in the C apo. You can still use standard C library functions that don't attempt to free the memory you pass to them. So you're safe with printf. Basically, just keep track of who made gave you the memory, so you can give it back to the right one when you're done.

## Working With Variables

### Accessing a Variable's Data

Before we get into the specifics you should know that ferite internally represents all variables using FeriteVariable *'s, and they can safely represent any native type within ferite. The builder and the return macros you've already seen are in place to perform conversions for the sake of convenience. But sometimes you just need to stick your finger in the pudding, so here is how to do it without breaking ferite.

There are a few bits of general information you can get from a FeriteVariable * without looking specifically into one variable type. The internal variable name is accessible, although it really isn't as useful as it sounds. Usually the variable name has been automatically generated by an operator or a function. But if you'd really like to have it, you can access is by: [it is a null terminated c string]

```
var->name
```

Much more useful than the variable name is the variable type. This will tell you if the data held is a number (and what kind), a ferite string, an object, an array, or void (nothing at all). It is accessible by:

```
var->type
```

It is an integer that can be any one of the following values:

- `F_VAR_VOID` - a void variable, no value.
- `F_VAR_LONG` - a number variable as a c long.
- `F_VAR_DOUBLE` - a number variable as a c double.
- `F_VAR_STR` - a string variable.
- `F_VAR_UARRAY` - an unified array variable.
- `F_VAR_OBJ` - an object.

There are a number of additional macros available for accessing the actual data within different variable types. You should use these macros as much as possible when working with ferite variables. Internal structures may change, but these macros should always be up to date and provide exactly the same semantics when it comes to value access.

- `F_VAR_VOID` - since this is a void variable, there really isn't any data to gain access to.
- `F_VAR_LONG` - the data can be accesses by VAI( var ) . This will make it act exactly like a c long. You can read its value, and set new values.

  Example:

  ```
  VAI(mynum) = 7;
  ```

- `F_VAR_DOUBLE` - the data can be accessed by VAF( var ). This will make it act exactly like a c double. You can read its value and set new values.

  Example:

  ```
  VAF(mynum) = 8.16;
  ```

- `F_VAR_STR` - using the VAS( var ) macro will get you a FeriteString *, which can then be used in API functions to perform various operations. You can access the string's length and data by using the original FeriteVariable * in the FE_STRLEN( var ) and FE_STR2PTR( var ) macros, respectively. FE_STR2PTR behaves like a char *, and FE_STRLEN behaves like an int. Whenever you change a string's content, you must always update it's internal size to reflect the new actual size.

  Example:

  ```
  ffree(FE_STR2PTR(var));
  FE_STR2PTR(var) = fstrdup("My new string!");
  ```

```
FE_STRLEN(var) = strlen(FE_STR2PTR(var));
```

There are a whole host of functions within the ferite engine for manipulating FeriteString*'s allowing you to do comparisons and replacements on the strings. It should also be noted that FeriteString*'s are designed to hold binary data.

- `F_VAR_OBJECT` - using the macro VAO( var ) will get you a FeriteObject *, which can then be used in a variety of api functions to access variables and functions within that object.
- `F_VAR_UARRAY` - using the macro VAUA( var ) will get you a FeriteUnifiedArray *, which can then be used in the unified array api functions to add, retrieve, and remove values from the array.

## Changing a Variable's Type

You can change a variable's type by changing the var->type value to the desired type, but converting the variable's contents is entirely up to you. You should be especially careful when changing the type of strings, objects and arrays, as they have a lot of extra data that goes along with them. However changing a number between types is quite easy. For example, to change a double to a long, you can simply do this:

```
VAI(var) = (long) VAF(var);
var->type = F_VAR_LONG;
```

And to change it back:

```
VAF(var) = (double) VAI(var);
var->type = F_VAR_DOUBLE;
```

It is considered bad for to simply change the type of a variable and is therefore not encouraged at all. It is therefore on your own head to keep things correct.

## Creating and Destroying Variables

Creating variables is quite simple. Each variable type has a 'create' function that returns a FeriteVariable *, which will be encapsulating the type you requested. The only thing difficult about creating variables is remembering that the returned object will always be a FeriteVariable *, rather than the specific struct type that you wanted. This is because the FeriteVariable * is encapsulating the variable type you had wanted. This is actually quite convenient, since you would have to stuff the specific variable type into a FeriteVariable * before giving it back to the ferite engine anyhow.

You already know how to manipulate these variables (or at least get to the information needed to manipulate them), so I'll just quickly run through the variable types available, and their creation functions. The parameters should be pretty self explanitary, if not please refer the the C api document. It should be noted though that they all take the same argument "alloc", this tells ferite whether or not the name of the variable should be allocated or whether it is static.

- `F_VAR_VOID` - FeriteVariable *ferite_create_void_variable(char *name, int alloc);
- `F_VAR_LONG` - FeriteVariable *ferite_create_number_long_variable(char *name, long data, int alloc);
- `F_VAR_DOUBLE` - FeriteVariable *ferite_create_number_double_variable(char *name, double data, int alloc);

- F_VAR_STR - FeriteVariable *ferite_create_string_variable(char *name, FeriteString *data, int alloc);
- F_VAR_STR - FeriteVariable *ferite_create_string_variable_from_ptr(char *name, char *data, int length, int encoding, int alloc); [As of the time of writing, the encoding value is always FE_CHARSET_DEFAULT. The reason for it being set now is so the in the future when the encoding of a string is important code will still work unmodified]
- F_VAR_UARRAY - FeriteVariable *ferite_create_uarray_variable(char *name, int size, int alloc);
- F_VAR_OBJ - FeriteVariable *ferite_create_object_variable( char *name, int alloc );

Deleting variables is actually easier than creating them, if you can believe it. To delete any ferite variable, you simply use the ferite_variable_destroy() function. This function takes the current script and a FeriteVariable * as parameters, and it returns void.

```
void ferite_variable_destroy( FeriteScript *script, FeriteVariable *var );
```

You can use this function on any type of variable, each will be handled in the appropriate manner according to it's type. Strings will have their c-string data freed by ffree() and will then be destroyed. Objects will have their destructor called before they are destroyed. Lastly, unified arrays will have the variables at each of its indexes destroyed in the appropriate manner according to their type and will then, themselves, be destroyed.

## Working With Namespaces

The next logical step would be to cover classes, but since classes are a combination of variables and functions and are, in a sense, much like namespaces, we need to do a little ground work before we delve into that subject. So in this section we'll cover how to create and delete namespaces, and how to create, access, and delete variables and how to register and delete functions, and how to find things within them.

Namespaces are created by registering them within the script. This can be done with the following function:

```
FeriteNamespace *ferite_register_namespace( FeriteScript *script, char *name, Ferite-
Namespace *parent )
```

The function takes three parameters; the script to register the namespace into, the name of the namespace you wish to create, and the parent in which to create the new namespace. The parent must either be a valid FeriteNamespace *. You can either find one with ferite_find_namespace(), or you can simply use script->mainns to use the top-level namespace of a script as the parent. If the register is successful, the FeriteNamespace * that refers to the new namespace is returned. The data it points to is internally allocated, so do not destroy it. If the register failed, it will return NULL.

Once you have a namespace created, you can delete it with this function:

```
int ferite_delete_namespace( FeriteScript *script, FeriteNamespace *ns )
```

This will destroy the namespace after recursivly destroying all of its children, this includes all variables, sub-namespaces, classes and functions. It currently always returns 1.

Creating and deleting namespaces is only fun for a short while. Eventually you'll want to put variables into your new namespace, and probably functions and classes as well. The next three functions will allow you to do j ust that.

```
FeriteVariable *ferite_register_ns_variable( FeriteScript *script, Ferite-
Namespace *ns, FeriteVariable *var )
```

This will register a variable into the namespace that you provide. If you've just recently created the namespace, you can use the FeriteNamespace * that the register function returned. Otherwise you will have to look up the FeriteNamespace * to the namespace you wish to place your variable in using the ferite_find_namespace() function. The value returned is always the same as the value passed in as the var parameter. The variable will be accessible under the new namespace according to its name stored in the FeriteVariable struct. So you might want to make sure you set it to something intelligent before you register it into a namespace.

```
FeriteFunction *ferite_register_ns_function( FeriteScript *script, Ferite-
Namespace *ns, FeriteFunction *f )
```

This functions registers a function into the given namespace. The return value is always the same as the value passed in as the f parameter. Again, the name of the element comes from the name field of the FeriteFunction struct. Set it before you register the function.

```
FeriteClass *ferite_register_ns_class( FeriteScript *script, FeriteNames-
pace *ns, FeriteClass *klass )
```

This will register a class into the given namespace. The return value is always the same as the value passed in as the klass parameter. Once again, the name of the element comes from the name field of the FeriteClass struct. Set the name before you register the class. Most of the time you will never use this as the standard way to create a class will also automatically register it, it is merely mentioned here for completness.

The next logical step is gaining access to variable, functions, and classes that are registered to namespaces. This is done by retrieving a `FeriteNamespaceBucket` which contains the information you desire in it's data element. The following function is used for retrieving these buckets:

```
FeriteNamespaceBucket *ferite_find_namespace( FeriteScript *script, Ferite-
Namespace *parent, char *obj, int type )
```

This will return a FeriteNamespaceBucket * on success, or NULL on failure. It takes a script, and a starting point as the first two parameters. The third parameter is the dot-delimited name of the object you are looking for, relative to the parent namespace given. So if you are using the root namespace (script->mainns) as your parent namespace, and wish to access mynamespace.myothernamespace.myvar, then you would pass "mynamespace.myothernamespace.myvar" as the third parameter. However, if you already have a FeriteNamespace * that refers to 'mynamespace', then you could pass that in as the parent (2nd parameter) and then access myvar by passing "myothernamespace.myvar" as the obj (3rd parameter). Lastly, if you already have the FeriteNamespace * for 'myothernamespace', then you would simply pass "myvar" as the obj. Because you are only dealing with one level of depth, you do not place a period within the obj in that instance. The fourth, and last, parameter is the type of object you are looking for. It is always one of the following defined types:

- `FENS_NS` - retrieves namespaces
- `FENS_VAR` - retrieves variables
- `FENS_FNC` - retrieves functions
- `FENS_CLS` - retrieves classes

If you choose to pass 0 to the function, you will get back the named FeriteNamespaceBucket if it exists. Using the above defines allows you to tell ferite_find_namespace what type of bucket you are looking for guaranteeing that what you get back is the correct item and type.

Again, once you have the bucket, you can access the desired value by looking in the data element. Example:

```
FeriteVariable *myvar = NULL;
FeriteNamespaceBucket *nsb = NULL;

nsb = ferite_find_namespace(script, script->mainns, "mynamespace.myvar", FENS_VAR);

if( NULL != nsb ){ /* we found it! */
 myvar = (FeriteVariable *) nsb->data;
 /* we needed to cast because nsb->data is a void * type */
}
```

At this point I can use myvar just like any other FeriteVarible *, because it is one! When the value of this variable is changed it will be noticble straight away within the script. It is also important to note that you much not take these variables you have obtained and return them to the script via FE_RETURN_VAR. This will cause ferite to delete the variable and leave dangling pointers. If you wish to return the variable simply return it like you would a normal c variable:

```
return myvar;
```

To get a function is the same process. Example:

```
FeriteFunction *func = NULL;
FeriteNamespaceBucket *nsb = NULL;

nsb = ferite_find_namespace( script, script->mainns, "mynamespace.function", FENS_FNC )
if( NULL != nsb ){
 func = (FeriteFunction*)nsb->data;
 ....
}
```

It is good to note that within ferite's source, it is convention to call the namespace bucket variable 'nsb'.

As promised at the beginning of this section, here is how to unregister elements from namespaces:

```
void ferite_delete_namespace_element_from_namespace( FeriteScript *script, Ferite-
Namespace *ns, char *name )
```

This will delete the element name from the namespace ns within the script script. Be careful though, this function will not burrow down layers of namespaces to find the element you specify. So you cannot use the dot notation here, this is a deliberate design choice to stop accidental deletion of the wrong elements. You must first find the immediate parent of the element (using ferite_find_namespace()), and pass that in as the namespace ns. You can use this to delete namespaces from within namespaces as well, and in that case it will also recursively destroy the deleted namespace's contents.

So thats is all there really is to namespaces. They are an excellent form on container both in and out of scripts!

# Working With Objects And Classes

## Creating Classes

Registering classes is much the same as registering namespaces. You first reigster the class, then you add ] what variables and functions you wish to publish in them.

To register a class you use the ferite_register_inherited_class function call. This will create the class, setup the inheritence, register the class wihin a namespace for you and return it in one fell swoop.

```
FeriteClass *ferite_register_inherited_class( FeriteScript *script, Ferite-
Namespace *ns, char *name, char *parent )
```

The first parameter is the script, the second is the namespace in which you want to place the class, the third is the name of the class by which programmers can reference it and the fourth is the name of the class the new class inheirts from. The fourth argument can be in standard dot notation and is the name of the parent class. For instance it could be "Sys.Stream". The function will start looking for the class in the namespace that is passed to the function, and then start in the top level script namespace. For instance if the "Sys" namespace was passed to the function, you would want to specify "Stream". If you do not wish for your class to inherit from any existing class simply pass NULL and the new class will be automatically placed as a subclass of the base class "Obj".

Registering variables and functions with a class is much the same as registering them with a namespace, you simply pass an extra parameter to say whether or not the item is static (linked to the class) or an instance variable (linked to the object create from the class).

To add a variable you call:

```
int ferite_register_class_variable( FeriteScript *script, FeriteClass *klass, Ferite-
Variable *variable, int is_static )
```

The second argument is the class to add the variable to [which can be obtained from creating a new class or pulling one out of a namespace], the third argument is the variable to add, and the fourth variable is whether or not the variable is static.

To add a function you call:

```
int ferite_register_class_function( FeriteScript *script, FeriteClass *klass, Ferite-
Function *f, int is_static )
```

The arguments are almost identical except for the third one which is a pointer to a ferite function.

## Creating Objects

Creating objects is very straight forward. There are two main method calls that can be used.

The first is `ferite_build_object`, it's pupose is to simply allocate a FeriteVariable*, allocate the necessary structures [such as it's instance variables, and pointers to functions] and adds it to the ferite garbage collector. `ferite_build_object` does **not** call the new objects constructor. This is very useful for when you are doing manual setup of an object. The prototype for the function is:

```
FeriteVariable *ferite_build_object( FeriteScript *script, FeriteClass *nclass);
```

The second is `ferite_new_object` which does all the same things `ferite_build_object` does except it will call the constructor for the new object. It will return an FeriteVariable* that is ready to be cleaned up by ferite as and when it is returned to the engine and no longer wanted. It has the prototype:

```
FeriteVariable *ferite_new_object( FeriteScript *script, FeriteClass *nclass, Ferite-
Variable **plist);
```

The first two arguments are the same for `ferite_build_object`, the current script and the class you wish to instantiate. The third argument is the parameter list to be passed to the objects constructor. Read the next section on calling functions to find out how to create one and what they consist of.

## Accessing Variables

Firstly, we'll cover how to access variables within objects and classes. It is done essentially the same way for each. Both FeriteClass and FeriteObject structs have a variables element that is a hash of all variables within them. To make life slightly easier and code more understandable there are a couple of functions for retrieving the variables from either a class or an object.

```
FeriteVariable *ferite_object_get_var( FeriteScript *script, FeriteOb-
ject *object, char *name );
```

This is for getting the value out of an object. It should be noted that the second argument is not a FeriteVariable* but a FeriteObject*. This means that is it necessary, if you have a FeriteVariable* pointing to an object, to call it with VAO(nameOfVariable) - otherwise all sorts of issues will arise.

```
FeriteVariable *ferite_class_get_var( FeriteScript *script, FeriteClass *klass, char *r
```

Both the above functions take the name of the variable to obtain and will return a pointer to the variable if it exists, or will return NULL if it doesn't.

For example, for objects you would do this: (assume that some_object is of type FeriteVariable *, and it is a valid object)

```
FeriteVariable *myvar = ferite_object_get_var(script, VAO(some_object), "my-
var");
```

If myvar is not NULL, then it was successfully retrieved. If you want to do the same with a class, you do this: (assume that some_class is of type FeriteClass *, and it is a valid class)

```
FeriteVariable *myvar = ferite_class_get_var(script, some_class, "my-
var");
```

Again, if myvar is not NULL, it was successfully retrieved.

## Accessing Functions

Getting functions from objects or classes is easy if you can get a variable from them [Hint: make sure you read the last section!].

To get your hands on a function in an object you simply use the function call `ferite_object_get_function`. Suprised? You shouldn't be. It looks, feels and

tastes very similar to `ferite_object_get_var` except this time you get a function not a variable.

```
FeriteFunction *ferite_object_get_function( FeriteScript *script, Feri-
teObject *object, char *name );
```

To get your hands on a function tucked away in a class you simply need to use the function call `ferite_class_get_function`.

```
FeriteFunction *ferite_class_get_function( FeriteScript *script, Ferite-
Class *cls, char *name )
```

So there you have it. Easy.

## Calling Functions

Variables are fun for a while, but when you want to start doing things, you will want to play with calling of functions. For this you will need one vital ingredient a pointer to a FeriteFunction.

### Namespace Functions

Once you have a FeriteFunction *, the next thing you're probably going to want to do is call the function it to which it refers. This is one of the trickier things to do in ferite, but only because it involves several stages in order to complete.

Firstly, you need your FeriteFunction *, which can be obtained by using the ferite_find_namespace() function. Then you'll need to create a parameter list that you wish to pass to the function. This is done with the following function:

```
FeriteVariable **ferite_create_parameter_list_from_data( FeriteScript *script, char *fo
mat, ... );
```

This function does its best to make creating parameter lists simple. The first parameter is the script, the second is a format string that describes the types of variables that will make up the argument list, and the rest of the parameters are the values to be used as described by the format string. The format string must be zero or more of the following:

- n - a number, the value passed must be a C variable of type double
- l - a number, the value passed must be a C variable of type long
- s - a string, the value passed must be a pointer to FeriteString
- o - an object, the value passed must be a pointer to a FeriteObject
- a - an array, the value passed must be a pointer to a FeriteUnifiedArray

The function will return a parameter list (FeriteVariable **) which can then be used as a parameter in the next function to be dicussed. For your information, a parameter list is simply a NULL terminated c array of FeriteVariable* - these are easy to create by hand, but this function simply aids the creation.

```
FeriteVariable *ferite_call_function( FeriteScript *script, FeriteFunc-
tion *function, FeriteVariable **params );
```

This function will call the function and return a FeriteVariable *, which will be the returned value of the called function. It must be caught and destroyed, or you will leak memory. Even functions returning void will return a fully allocated FeriteVariable * of type F_VAR_VOID.

The first parameter is the script, the second is the pointer to the FeriteFunction you wish to call, and the last is the parameter list you had created with the previously described ferite_create_parameter_list_from_data() function.

When you are finished with the parameter list, simply delete it with this function:

```
void ferite_delete_parameter_list( FeriteScript *script, FeriteVariable **list );
```

So there you have it, three steps to calling another function within ferite. Here is a complete example which calls 'Console.println' with the string 'Hello World':

```
FeriteFunction *println = NULL;
FeriteVariable **params = NULL;
FeriteVariable *rval = NULL;

/* Create a string to pass to the function */
FeriteString *hello = ferite_str_new( "Hello World", 0, FE_CHARSET_DEFAULT );

/* Find the function in the scripts main namespace */
FeriteNamespaceBucket *nsb = ferite_find_namespace( script, script->mainns, "Con-
sole.println", FENS_FNC );

if( NULL != nsb ) /* Check to see if we have the function ... */
{
 println = nsb->data;

 /* Create the parameter list */
 params = ferite_create_parameter_list_from_data( script, "s", hello )

 /* Call the function */
 rval = ferite_call_function( script, println, params );

 /* And finally clear up after ourselves */
 ferite_delete_parameter_list( script, params );
 ferite_variable_destroy( script, rval );
 ferite_str_destroy( script, hello );
}
else    /* We dont.. lets print an error! */
 printf( "Cant find 'Console.println'! Is the console module loaded?\n" );
```

It should be noted that the above method for calling functions works for any function that is not an instance method within an object. This will be discussed further in the next section as there are a couple of things the have to be done on top of the above steps.

## Object and Class Functions

Calling methods of objects is much like calling regular functions. There are really only two differences.

Firstly, you look up the FeriteFunction * from within the object using the `ferite_object_get_function()` function, and you always pass the object itself as the last two parameters to the method, this must be done regardless of the number of arguments the object's function takes.

Example: (assume obj is of type FeriteVariable * and is a valid object)

```
FeriteFunction *func = NULL;
FeriteVariable **params = NULL;
FeriteVariable *rval = NULL;

func = ferite_object_get_function(script, VAO(obj), "function");
params = ferite_create_parameter_list_from_data( script, "oo", VAO(obj), VAO(obj) );
rval = ferite_call_function( script, function, params );
ferite_variable_destroy(script, return);
ferite_delete_parameter_list( script, params );
```

As you can see, it's very similar to calling normal functions. You may also want to know that there is a helper function (used in the example below) for ensuring proper placement of the object in the parameter list called ferite_object_add_self_variable_to_params().

Example: (assume obj is of type FeriteVariable * and is a valid object)

```
FeriteFunction *func = NULL;
FeriteVariable **params = NULL;
FeriteVariable *rval = NULL;

func = ferite_object_get_function(script, VAO(obj), "function");

/* Create an empty parameter list */
params = ferite_create_parameter_list_from_data( script, "" );

/* Add the 'self' variable to the parameter list */
params = ferite_object_add_self_variable_to_params( script, params, VAO(obj) );

rval = ferite_call_function( script, function, params );
ferite_variable_destroy(script, return);
ferite_delete_parameter_list( script, params );
```

Calling methods of classes (static methods) is sort of a hybrid between calling normal functions and object methods. With classes you look up the FeriteFunction * from within the functions element of the FeriteClass struct using the ferite_class_get_function() function, and then you call it like a standard function. You do not pass the object to class methods because there is no object associated with the method. For all intents and purposes class functions within ferite are treated the same as namespace functions.

Here is an example: (assume klass is of type FeriteClass * and it is a valid class)

```
FeriteFunction *func = NULL;
FeriteVariable **params = NULL;
FeriteVariable *rval = NULL;
FeriteString *hello = ferite_str_new( "Hello World", 0, FE_CHARSET_DEFAULT );

func = ferite_class_get_function(script, klass,"function");
params = ferite_create_parameter_list_from_data(script, "s", hello);
return = ferite_call_function(script, func, params);
ferite_variable_destroy(script, return);
ferite_delete_parameter_list(script, params);
ferite_str_destroy( script, hello );
```

And there you have it. You can now call class and object methods, and access variables within classes and objects.

### Function Shortcuts

## Raising Exceptions and Reporting Errors

There are times when things go wrong. It's a painful time, but it need not be. Ferite provides a means of raising exceptions to force a programmer to deal with errors but also a means of quietly setting the error information allowing the programmer to check for non-fatal things.

It is considered good form to return error values from a function call. This is the route you should take if you require the reporting of errors. For instance if you have a function that connects to a resource and returns an object to interact with that resource, it makes sense to return a null object [`FE_RETURN_NULL_OBJECT`] if that resource cant be obtained.

Sometimes this is not possible to return an error value. In these situations it is considered good form to use the function `ferite_set_error` [it's prototype is below]. This sets the `err` script object's values, but does not raise an exception. This allows the programmer to ignore things if needs be. It takes a number of parameters, the first is the script you are running in, the second is the error number and the last is the format of a string [same as `printf`] describing the error that has occured. It should be documented that this is the case such that the programmer knows what to look for.

```
void ferite_set_error( FeriteScript *script, int num, char *fmt, ... );
```

When all hope is lost, there are times when an exception needs to be rasied because some has gone completely wrong. This is done by calling `ferite_error`. You can pass it the error number and the message just like `ferite_set_error`.

```
void ferite_error( FeriteScript *script, int num, char *fmt, ... );
```

Sometimes it is nice to warn people about not so bad things, and as such there is a function `ferite_warning` which will place a warning on the script.

```
void ferite_warning( FeriteScript *script, char *errormsg, ... );
```

## Executing Code Snippets

Sometimes it is easier to execute a block of code, from within a function, written in ferite. For this you can use the eval mechanism. What this does is the same as the `eval` operator in ferite. It will compile and execute the script and then return the return value of the main function. For example:

```
rval = ferite_script_eval( script, "Console.println('Hello World');" );
```

You must destroy the return value using `ferite_variable_destroy` just as you would a function call.

# Chapter 5. Native Modules - By Hand

The aim of this chapeter is to combine information given in the previous chapters, add some more insight and show you how to write modules by hand. This chapter is also very useful for people wanting to embed ferite as it shows how to export an API by hand.

## Functions

Functions are as easy to write by hand as they are with builder. Infact, builder simply makes the following completely automatic, which is good 95% of the time, but sometimes you just have to take complete controll.

As with normal C functions we have to declare our native ferite functions. This is done in three stages, first we declare it, then we create our FeriteFunction structure and then we register it with the ferite engine. To declare the variable, you use the macro `FE_NATIVE_FUNCTION`, this is true for both object/class methods and normal namespace functions. This takes one argument, which is the name of the function you wish to create. After the macro, you simply write the body of your function as you normally would. For example:

```
FE_NATIVE_FUNCTION( printfnc )
{
 printf( "We are in our native function!\n" );
}
```

That is as simple as the functions are going to get. The next thing we need to do is create a FeriteFunction structure with which we can register the function [using the functions mentioned in the last chapter]. This is also easy, it is simply a function call to `ferite_create_external_function`. It's prototype is below:

```
FeriteFunction *ferite_create_external_function( FeriteScript *script, char *name, voic
Function*, FeriteVariable **), char *description );
```

This takes the current script, the name of the function, a pointer to the function, and it's signiture description. The first two are fairly obvious. The third means you simply pass the name of the native function, eg. in the above example it would be `printfnc`. The description is slightly more complicated, it is a null terminated string which takes a number of characters that describe what arguments the function can take.

- n - number
- s - string
- a - array
- o - object
- v - void
- . - variable argument list

Each character responds to each type and it allows ferite to make sure that the function gets passed the correct parameters. To make life slighty clearer, here are a few examples with the ferite function and what would be the equivelent description for a native function:

```
function ex1( string name, number age ){ } would be "sn"

function ex2( string format, ... ) { } would be "s."

function ex3( object res, string query, array args ) { } would be "osa"
```

To register the function you have back, you either use `ferite_register_ns_function` or `ferite_register_class_function`. You must be aware that you can only register each created function once! Otherwise ferite will certainly die when it tries to clean everything up at the end of execution.

So, lets assume that our above print function takes a string and a number and prints out the string the number of times it is told. The example below will show how to declare, create a FeriteFunction and register it in a namespace. The example will also allow us to touch on another couple of important areas.

```
FE_NATIVE_FUNCTION( printfnc ); /* Declare the prototype */

FE_NATIVE_FUNCTION( printfnc )
{
 FeriteString *print = NULL;
 double countd = 0;
 int i = 0, count = 0;

 /* Get the parameters */
 ferite_get_parameters( params, 2, &print, &countd ); /* #1 */

 /* Loop round printing */
 count = (long)countd;
 for( i = 0; i < count; i++ )
  printf( "%s", print->data );

 FE_RETURN_VOID; /* #2 */
}

void module_init( FeriteScript *script )
{
 /* Create the function */
 FeriteFunction *f = ferite_create_external_function( script, "printfnc", printfnc, "sr

 /* Now register it in the main namespace */
 ferite_register_ns_function( script, script->mainns, f );
}
```

Point *#1* is the main point to be covered. `ferite_get_parameters` is a helper function for getting the values of the parameters you have passed into C variables you can manipulate. It is very important that you do not delete or free the values you have because they point to the real values. This function takes two arguments at a minimum and any number of arguments more, the first is the parameter list you are given, when writing the native function, it is always called `params`. The second argument is the number of values from the parameter list that you want, and the rest of the arguments are pointer to the variables you wish to set. In our example above, the address of the `print` and `countd` variables were passed. This is exactly how builder gets the values from the parameter list - it is simply hidden from the programmer.

Point *#2* is just a highlight of a point made before. All functions must return something even if it is just a void. Builder hides point *#2* from you, but when writing functions from scratch, it is important you remember to return something.

To get the number of parameters that where passed to the function, you can use the `ferite_get_parameter_count`, this takes just one argument [`params`] and returns the number of variables in it.

```
int ferite_get_parameter_count( FeriteVariable **list );
```

When you have a method within an object there are always two extra parameters. `self` and `super`. To get at the object which self points to, you have to get at it the

same way you do other variables. The example below is almost identical to the one above except it assumes printfnc is part of an object.

```
FE_NATIVE_FUNCTION( printfnc ); /* Declare the prototype */

FE_NATIVE_FUNCTION( printfnc )
{
 FeriteObject *self = NULL, *super = NULL;
 FeriteString *print = NULL;
 double countd = 0;
 int i = 0, count = 0;

 /* Get the parameters */
 ferite_get_parameters( params, 4, &print, &countd, &super, &self );

 /* Self is now pointing to our object */

 /* Loop round printing */
 count = (long)countd;
 for( i = 0; i < count; i++ )
  printf( "%s", print->data );

 FE_RETURN_VOID;
}
```

So there we have it. How to write a function, it is not really that hard once you know how.

## The Rest

Now that you know how to write native functions by hand this section is a piece of cake. All you have to do to fullfill the requirements of a ferite module is write four functions. Yes it is that easy. These functions are the ones that builder creates for you from `module-init`, `module-deinit`, `module-register`, and `module-unregister`. Rather than talk too much, I will show you the code for a blank module:

```
void modulename_register()
{
 System wide setup. Called when the module is loaded from disk.
}

void modulename_init( FeriteScript *script )
{
 Per script setup. This is where you put the code to register
 namespaces, classes, functions and variables and setup anything the script
 needs.
}

void modulename_deinit( FeriteScript *script )
{
 Anything you need to shutdown per script. Ferite will clean up
 all structures you have registered so you do not need to clean those up
 yourself [eg. the namespaces you have registered].
}

void modulename_unregister()
{
 System wide shutdown. This gets called when the ferite engine is
 being deinitialised.
}
```

If you have these four functions exported from you module, it will find them without problem. One thing to note, the name of the module **must** be the same as the prefix for each of the functions otherwise ferite will not be able to find them. For instance in `foo.lib` the init function must be called `foo_init`.

The hardest part of writing a module is probably getting it to compile and be installed. But that will be left as an exercise to the reader. [Hint: look at `generate-module`, a tool shipped with ferite for installing modules written using builder - but can be modified to handle home made modules].

You may also want to read the next chapter as a cunning secret is told that can make writing native modules easier.

# Chapter 6. Embedding Ferite

This chapter is split into three sections. The first deals with getting the engine up and running within your application so that scripts can be executed. The second section deals with the most effcient way of exporting the application's interface into a script so that useful things can then be done. The third is how to cheat with builder and applications.

## Getting The Engine Purring

Ferite is designed to be placed in pretty much anywhere. Therefore it is pretty easy to get the engine up and running, scripts compiled and then executed, and to clean everything up afterwards. To explain how to do this, an example is listed below and afterwards each line is discussed. It is a simple program that shows most of the functionality of the ferite command line program.

```
#include <stdio.h>
#include <stdlib.h>
#include <ferite.h>

int main( int argc, char **argv )
{
    FeriteScript *script;
    char *errmsg = NULL, *scriptfile = "test.fe";

    if( ferite_init( 0, NULL ) )
    {
        ferite_add_library_search_path( LIBRARY_DIR );
        ferite_set_library_native_path( NATIVE_LIBRARY_DIR );

        script = ferite_script_compile( scriptfile );
        if( ferite_has_compile_error( script ) )
        {
            errmsg = ferite_get_error_log( script );
            fprintf( stderr, "[ferite: compile]\n%s", errmsg );
        }
        else
        {
            ferite_script_execute( script );
            if( ferite_has_runtime_error( script ) )
            {
                errmsg = ferite_get_error_log( script );
                fprintf( stderr, "[ferite: execution]\n%s", errmsg );
            }
        }
        if( errmsg )
          ffree( errmsg );
        ferite_script_delete( script );
        ferite_deinit( );
    }
    exit( 0 );
}
```

And now, the explanation. It should be noted that only the lines that are critical to the operation of ferite will be dicussed, anything that is standard C will be left out.

```
#include <stdio.h>
#include <stdlib.h>
#include <ferite.h>
```

The above is all pretty standard issue. You dont need the `stdio.h` or `stdlib.h` headers to be honest. But with any program they are good practice. The one you do need is `ferite.h`. This will pull all the function prototypes and defines into the program so that the magic may begin.

```
if( ferite_init( 0, NULL ) )
```

This line initialises the engine. You must do this before you do anything ferite related. This is because this call will initialise the ferite memory system, the module system, the regex engine and potentially more things. If you dont call this then what happens is all your own fault. You may call this multiple times and it wont cause issues. It takes two arguments, the first is the number of arguments, the second is an array of strings. This is how options are passed into the engine. For a full list look at the command line program's help option.

```
ferite_add_library_search_path( LIBRARY_DIR );
ferite_set_library_native_path( NATIVE_LIBRARY_DIR );
```

Ferite does what it is told. One of the things that makes it very useful is the ability to control what modules are availible to be loaded. You can obtain the system wide defaults using the **ferite-config** shell script. If you do not call these then the ferite engine will be unable to load any modules and will only have the api that the application exports. This is useful for both controlling what the scripters can do and stoping people loading rogue modules into the system.

```
script = ferite_script_compile( scriptfile );
```

This line will compile the script that is in the file in the `scriptfile` variable. It will always return a script object. The return will either contain the error information or will be an executable script. It is also possible to compile a string into script. For this you call `ferite_compile_string`, it takes one argument which is the script to compile. There are also two more functions, `ferite_script_compile_with_path` and `ferite_compile_string_with_path`, they both take the same arguments as their respective counterparts, with the exception of an added argument. This is a null terminated array of search paths to add to the module system for the duration of the compilation. For more in depth information about these two functions please refer to the C api.

```
if( ferite_has_compile_error( script ) )
{
 errmsg = ferite_get_error_log( script );
 fprintf( stderr, "[ferite: compile]\n%s", errmsg );
}
```

This is how we check that everything is working. Or not. `ferite_has_compile_error` will return true if there was a compile error and false if not. If there is an error, the script will not be executable but you will be able to get the error logs from the script as shown above. You will need to, when finished, delete the script - this is dicussed later. You will also need to free the string returned using `ffree`.

```
ferite_script_execute( script );
```

Well, if you have got this far you will be wanting to run the script. This is easy. You simply pass it to `ferite_script_execute` which will execute the script. The return value is the return value from the script's main function. To find out whether a runtime error occured you will need to use the code below. **NOTE:** you can run scripts

multiple times but it is not recommended, as the state of the script can not be guaranteed. The way to run a script multiple times is to use `ferite_duplicate_script` and execute each duplicate - for more information please see the C api.

```
if( ferite_has_runtime_error( script ) )
{
 errmsg = ferite_get_error_log( script );
 fprintf( stderr, "[ferite: execution]\n%s", errmsg );
}
```

`ferite_has_runtime_error` will return true if there has been a runtime error on the script. To get the messages about the error you will need to use `ferite_get_error_log`. This will return the error log as a C string. You will need to free the string returned with `ffree`.

```
if( errmsg )
 ffree( errmsg );
```

Remember to free things!

```
ferite_script_delete( script );
```

Once you have finished with your script object you must delete it. Once again this is a simple and straight forward call to `ferite_script_delete`.

```
ferite_deinit( );
```

It's been a long day, you've been running scripts and it's now time to pack your bags and go home. There is one last thing to be done - tell ferite to deinitialise. This is done doing `ferite_deinit`. This will cause all allocated memory via `fmalloc/fcalloc/frealloc` to be deallocated, shutdown the module system and anything else that needs to be done. Once this has been called you can re-initialise the system with `ferite_init` and start all over again.

So there you have it. Thats how easy it is to get things up and running. It is suggest that you have a look at the command line program in the ferite distribution for more options availible or a more concrete example.

## Fake Native Modules

Fake native modules are a means of adding API to a script running within an application with exactly the same method as a module. Rather than let ferite load the module, you simply register the module, tell ferite to pre-load it, and away you go. The example below assumes a module compiled into the application.

```
ferite_module_register_fake_module( "theapp.lib", theapp_register, theapp_unregister, t
ferite_module_add_preload( "theapp.lib" );
```

The `ferite_module_add_preload` is important so that the module gets compiled into the script at compile time and therefore allows for initialisation code that gets executed to access the application. Please note that the **.lib** extension is very important. This is so that ferite knows that it is a native module and can handle it correctly [and also find it]. You should refer to the last chapter on writing native modules by hand for the information on how to write the native module. For those of you feeling slightly more lazy, read on, there is a cunning use of builder.

The above code should be placed before the calls to `ferite_script_compile` or `ferite_compile_string`.

## Cheating With Builder

You cant cheat with builder just yet, but the basic idea is this: Write your modules to embed using builder and a `.fec` file, compile the c code into your program, make sure you register the native module [as in the last section], and then tell ferite to preload the `.fec` file rather than the native library. This will cause ferite to parse the file just like a normal library but link to you application than to an external function. The only issue with this is that your source code for each function will be in the `.fec` file.

This is also the same method by which you would compile modules into a program.

# Chapter 7. Finally: Step By Step Examples