The Ferite Programming Manual

Chris boris Ross

chris@darkrock.co.uk

Blake Watters

blakewatters@nc.rr.com

The Ferite Programming Manual

by Chris boris Ross and Blake Watters

Table of Contents

Chapter 1. Introduction

1.1. What is ferite?

Ferite is a small robust scripting engine providing straight forward application integration, with the ability for the API to be extended very easily. The design goals for ferite are lightweight -IE small memory and CPU footprint, fast, and straight forward both for the programmer of the parent application and the programmer programming ferite scripts to learn the system.

1.2. What does this documentation provide?

This document is the official commentary on ferite including language information such as constructs and known issues, and an API guide for the standard objects provided with every ferite distribution. There is also information on writing external objects and classes, and also embedding ferite within your application.

1.3. Why should I choose ferite?

Ferite is designed to be added into other applications. With a constant API your application will be able to stay binary compatible with the latest ferite engine. This is very good because it allows you, the application programmer, to add powerful scripting to your application without having to worry about the actual internals. Ferite provides type checking, and does a lot of work for the programmer to keeps things as simple as possible.

Ferite provides a language very similar to that of C and Java with additional features from other languages (e.g. Regular expressions in the style of Perl). This means that the skill set acquired through learning these main stream languages can be instantly applied to the ferite scripts. Ferite is by no means a heavy language, it has kept the small language size of C which allows it to remain fast and lightweight. There is also the ability to push the language further with native classes, objects, namespaces, variables and methods.

Ferite also has a very small system memory and disk foot print making it ideal for it’s use.

Chapter 2. Getting Started

2.1. Obtaining ferite

Ferite can be obtained in a number of different fashions depending on your situation, it is also suggested that you only trust files downloaded from the ferite web site (http://www.ferite.org) We can not guarantee the validity of files downloaded from other locations (except fo occasions when it has been explicitly stated).

1. Source Code - ferite can be downloaded in a source tarball to be built on a system.

2. Binary Distribution - we provide pre-compiled packages for redhat and debian based systems. Debian users can automatically get ferite installed by typing

NOT WORKING JUST YET:

```
# apt-get install ferite ferite-dev
```

2.2. Compiling ferite

Ferite is very easy to build and should build out of the box on most platforms, the process is as follows:

```
$ ./configure
$ make
$ su -c 'make install; ldconfig'
```

2.3. Installing ferite

We should put something along the lines of setup etc etc once ferite has a .feriterc

Chapter 3. Language Reference

3.1. Scripts

Scripts are written as follows:

```
Class, Namespace, Function, Global and Use definitions.
Anonymous function.
```

The anonymous function is what is called when the script is run. All parts of a program have to be declared before they are used. This is to keep the code clean rather than a name resolution reason (all names are resolved at runtime within ferite). The Anonymous function is equivalent to the main() method within a C program.

An example script (the famous Hello World program):

```
uses Console;
Console.println( "Hello World from ferite" );
```

The 'uses' statement is used to import API either from and external module or from another script and is described in greater depth later on.

3.2. Comments

These are possibly the most important feature of ferite. Seriously. Ferite supports two methods of commenting code either the C style (/* */) or the C++ style (//). These can be placed anywhere within the scripts. All I can say is use them - comments make peoples life so much easier :-).

```
// This is a comment
/* This is another comment */
```

3.3. Types

Ferite is a semi-strongly typed language. This means, unlike in Perl or php, you have declare variables that have to be used and what type they are. Ferite has a number of types within it's system. The simple

types are **number** (automatically switches between integer or real number system), **string**, **array**, **object** and **void**, and are described below:

3.3.1. number

This type encapsulates all natural and real numbers within the 64bit IEEE specification. Ferite will automatically handle issues regarding overflow and conversion. Several things have to be said about the number type:

- All numbers start out as C longs (64bit integers). When the value goes over the maximum value allowed for long the type will switch over to a C double.
- Comparisons can be made between numbers but it should be noted that once a number has internally become a double comparisons are likely to give unexpected results.

3.3.2. string

Strings are specified using the " " deliminators and can contain control characters. The control characters are defined as in C by use of \'s. Individual characters within the strings can be accessed using a similar method to that of arrays - the result of the index is a string with only one character within it. Strings are null terminated but it is not necessary for you, the programmer, to worry about this as ferite handles it all (same as Perl).

3.3.3. array

Array's provide a method of random access of information. The variables can either be access by means of a hash key or indexed like C arrays. Any variable can be placed within an array - this in other terms means that you can mix numbers, strings, objects, and even other arrays. This is a very useful feature as it allows you to group together likewise data on the fly. Arrays are accessed using the "[ ]" notation as in other languages, between the brackets can be any expression. When this is used upon an index that does not already exist a void variable is returned which can, in turn, be assigned to (effectively adding the data to the array).

3.3.4. object

Objects are instances of classes (ferite's complex data structure). When first created they point to the null object (this allows the user to check to see whether the object has been instantiated). To create an instance see the new operator.

3.3.5. void

The void type is similar to Perl's and php's type when first created. I.E anything can be assigned to them but after having something assigned to them the void type is removed and the variable then becomes whatever type was assigned to it. e.g. If a void type is created and has a number assigned to it, it can't then have a string or object assigned to it.

3.4. Variables

Variables are simply instances of types and can be declared within a initializer and with other variables using the following syntax:

```
<modifier> <type> <name> [= <expression>] [, ...] ;
```

- <modifier>

  This is used to define properties of a variable. Currently the only property is **final** and this sets, whether or not after a variables first assignment, whether the variable is constant and therefore can't be changed. This is the same behavior shown by the final keyword within Java.

- <type>

  This is the type of variable that you wish to declare.

- <name>

  The name of the variable to be declared. This must be a unique name and doesn't use the $ as a prefix unlike Perl or php. The name must start with a alpha character but after that may contain underscores _, numbers [0-9] and other alpha characters.

- [= <expression>]

  Variables can be declared with a non-default value rather than the internal defaults. (number = 0, string = "", object = null).

  Please Note!

  When it is declared within a function you can specify any valid expression to be used as the varaibles initialiser - eg. a return from a function, the adition of two previously declared variables. When the

variable is declared globally, in a class or a namespace you can only use a simple initialiser by this I mean you can only initialise a number with a integer or real number (eg. 120 or 1.20), and a string with a double or single quoted string. It is **not** possible to initialise an array or object in a namespace, global, or class block.

- [, ...]

  Multiple variables can be defined using the above syntax (see below for an example)

- [;]

  This terminates the statement.

Example

```
number mynumber = 10, another_number;
final string str = "Hello World";
object newObj = null;
array myarray;
```

A variable's scope is as local as the function they are declared in with the exception of explicit global variables. This means that a variable declared in function X can only be accessed within function X. Global variables can be accessed anywhere within a script, and are declared using the following syntax:

```
global {
<variable declarations>
}
```

Unless explicitly defined a variable is considered local. There are a number of predefined global variables within a ferite script, these are null and err. Null is used to allow checking of objects and instantiating, and err is the error object used for exception handling.

3.5. Expressions

Almost everything written in ferite is an expression - they are the building blocks of a program - they are combined to build other expressions which are in turned used in others using operators. Expressions are built up using various operators, variables and other expressions, an example being say that adding of numbers, or creating an instance of a new object. Expressions are made clearer when discussing operators as these are what are used to build them.

3.5.1. Truth Values

What constitues a truth value?

- A number that is not zero is considered as true, this also means that negative values are also true. It has to be noted that if a number has switched into real format it is never likely to be considered false. Currently ferite deals with this by binding false to the range $\pm$ 0.00001 (NB. This is likely to change later).
- A string that has zero characters is considered false, otherwise it's true.
- An array with no elements is false, otherwise is considered true.
- An object is considered to be false if it doesn't reference any instantiated object.
- A void variable can't be true.

There are currently two keywords that can be used 'true' and 'false'.

3.6. Operators

Ferite comes with a whole bundle of operators to play with. They allow you to do basic things such as arithmetic operations all the way to on-the-fly code generation and execution. With each operator it's action on different types will be described. When an operator is applied to types it can not deal with, an exception is thrown and must be handled (see exception handling).

3.6.1. Arithmetic Operators

These operators are the basics, you should stop reading now if you don't understand them. The operators work as variable operator variable.

- Addition (+) - This operator add two variables together. Currently it only applies to numbers and strings. Adding strings together acts as concatenation, and adding a number onto a string will cause it to be converted to a string and concatenated. This operator acts as expected with numbers.
- Subtraction (-) - This operator only currently applies to number variables. It will be extended in the future to provide functionality with strings and arrays. It will allow removal of strings and deletion of keys within an array.
- Multiplication (*) - Currently only applies to numbers with plans to extend towards strings. It will multiply two numbers together and return the result.
- Division (/) - Applies to number variables. Guess what it does :-)
- Modulus (%) - Returns the remainder of integer division between two number variables. If the numbers are in real format they will be implicitly cast into integers and then the operation will be done.

3.6.2. Assignment Operators

The basic assignment operator is '='. This will make the left hand side variable equal to the right hand side. This is a copy value operator which means that the right hand side will be copied and then assigned to the left hand side. This is true with exception of objects where the left hand side will reference the object and it's internal reference count will be incremented.

It is possible to extend the operator by placing one of the Arithmetic operators in front al-la C. e.g. +=, -=, *=, /=, &=, |=, ^=, >>=, <<=. It will have the effect of taking the existing left hand side, applying the arithmetic operator with the right hand side and then assigning it back to the left hand side.

3.6.3. Comparison Operators

These are used to compare variables. It is only possible to compare like variable types, i.e you can only compare strings with strings, and numbers with numbers. They are all straight forward and act as would be expected from their name.

- Equal To (==) - true if both sides are equal.
- Not Equal To (!=) - true if both sides aren't equal.
- Less Than (<) - true if the left hand side is less than the right.
- Less Than Or Equal To (<=) - true if the left is less than or equal to the right hand side.
- Greater Than (>) - true if the left hand side is greater than the right.
- Greater Than Or Equal To (>=) - true if the left is less than or equal to the right hand side.

3.6.4. Incremental and Decremental Operators

These allow incrementing and decrementing of variables. Currently it only works with numbers.

- Prefix Increment (++<variable>)
- Postfix Increment (<variable>++)
- Prefix Decrement (--<variable>)
- Postfix Decrement (<variable>--)

If you have programmed within C or Java before you will know how these work. They both do what they say on the tin, but the difference between Pre and Post fix is subtle (but at the same time very very useful). With the prefix version the variable is in/decremented and the new value is returned, with the postfix the variable is in/decremented and the **previous** value is returned. e.g.

```
number i = 0, j = 0;
j = i++; // j = 0, i = 1
j = ++i; // j = 2, i = 2
```

3.6.5. Logical Operators

These apply to truth values and tend to be used for flow control.

- Not (!) - true if the expression it is applied to is false.
- And (&&) - true if both variables/expressions are true.
- Or (||) - true if either variable/expression is true.

3.6.6. Bitwise Operators

It must be noted that when numbers are passed to the bitwise operators their values are explicitly cast into a natural number if they happen to be floating point. This does not modify the variable being passed.

Example:

`10 & 11.1` `will actually be` **`10 & 11`**

- AND (&) - does a bitwise AND on the two variables passed to it.
- OR (|) - does a bitwise OR on the two variables passed to it
- XOR (|) - does a bitwise XOR on the two variables passed to it
- Left Shift (<<) - does a bitwise left shift on the two variables passed to it. It is equivelent to dividing the left hand side by two a number of times which is specified by the right hand side.
- Right Shift (>>) - does a bitwise left shift on the two variables passed to it. It is equivelent to multiplying the left hand side by two a number of times which is specified by the right hand side.

3.6.7. Complex Operators

These operators are individual and slightly more complicated that the other operators.

- Object or namespace attribute (., or ->) - To get an attribute or a method within a namespace or instantiated object you need to use either '.' Or '->'. It should be noted that they both perform exactly the same function, the reason for having two methods is so that people from Java direction can comfortably use '.' and people from a C background can use '->'. It is not bound to the type of variable(IE. namespaces and objects act the same) like C.

Example:

```
Console.println( "Hello World" );
Console->println( "Hello World" );
```

Both of the above are exactly the same internally although the syntax is different.

- Instantiate an object (**new**) - This operator is used to create an instance of a class (which can then be assigned to a an object variable. It is used as follows:

```
new <class name>( <parameters> )
```

<class name> The name of the class to be instantiated.

<parameters> The arguments to be passed to the constructor of the class.

It should be noted that multiple *object* variables can point to one object created using the new keyword. This is discussed later on within the *Classes* and *Objects* section.

Example:

```
object newObject = new SomeClass( "aString", 10 );
newObject = new SomeOtherClass( "James", "Taylor" );

(where SomeClass and SomeOtherClass have been defined elsewhere)
```

- Evaluate a string (**eval**) - This is a very powerful operator and can be very very useful. It also shows off the difference between a pre compiled language a scripting language. The eval operator allows you to on the fly compile and execute a script **and** get a return value. It is used like so:

```
eval ( <some string with a script> )
```

The string can be any value - but must be a valid script, if not an exception will be thrown.

Example:

```
eval( "Console.println(\"Hello World\");" );

This script is the same as:

Console.println("Hello World");
```

This is of course a very simple example and doesn't show what a useful operator it is, but it does allow you to at runtime modify the behavior of code. It should also be noted that there are potential security risks involved with this operator and it should be considered carefully.

3.6.8. Regular Expressions

Ferite features regular expressions with a similar syntax to that of Perl. Currently there is only one operator concerning regular expressions within ferite (although this is likely to change).

Apply regular expression (=~)

This operator works be applying the regular expression defined on the right hand side to the string on the left hand side. Regular expressions can only be applied to strings. For more information regarding regular expressions see the section later on in the manual.

3.7. Statements

Statements are basically a collection of expressions followed by a ';'. A block of statements is defined as multiple statements between braces {}'s.

3.8. Control Structures

Ferite contains methods for changing the flow of a program, these are called control structures and are detailed below. All the control structures that appear in C and Java behave in exactly the same way in ferite.

3.8.1. if-then-else

This allows for the conditional execution of ferite scripts based upon the results of a test. The syntax is as follows:

```
Type one:

if ( expression ) {
statements if the expression is true
}

Type two: (with an else block)

if ( expression ) {
statements if the expression is true
} else {
statements if the expression is false
}
```

```
It is also not necessary to place braces around the statement block if it’s only one statem
```

When execution is happening the expression gets evaluated and then it’s truth value is determined, if it’s true then the first block is executed. If an else block exists then it will be executed if the expression evaluates to false.

Example:

```
if ( a < b )
Console.println( "A is less than B" );

if ( b > c ) {
Console.println( "B is greater than C" );
Console.println( "This could be fun." );
} else {
Console.println( "It’s all good." );
}
```

3.8.2. while Loop

This construct provides a method of looping, and is used as follows:

```
while ( expression ) {
statements if the expression is true
}
```

The body of the while construct will only be executed while the expression evaluates to true. The expression is evaluated upon every iteration.

3.8.3. for Loop

This construct provides a more controlled method of looping and is also ferite’s most complicated loop. It’s syntax is as follows:

```
for ( initiator; test; increment ) {
statements if the expression is true
}
```

The initiator expression is executed unconditionally at the beginning of the loop. The loop will continue to loop until the test expression evaluates to false, and the increment expression is evaluated at the end of each loop.

As with C each of the expressions may be empty, this will cause them to evaluate to true (therefore causing the loop to continue forever if there is no test expression).

Example:

```
number i = 0;
for ( i = 0; i < 10; i++ )
Console.println( "I equals " + i ); // print out the value of i
```

3.8.4. do .. while Loop

The *do .. while* loop is a variation of the *while* loop, the one difference being that it guarantees at least one execution of it's body. It will only then complete looping until the expression evaluates to false. It's syntax:

```
do {
statements if the expression is true
} while ( expression )
```

3.8.5. break

break will end the current *for, while, do .. while* loop it is executed in.

3.8.6. continue

continue will cause execution flow to jump to the beginning to the current *for, while, do .. while* loop it is executed in.

3.8.7. iferr-fix-else

This control structure provides the exception handling to within ferite. It is similar in a way to the try-catch-finally structure within Java but with one main difference. Within Java the finally block is always executed regardless of whether or not an exception occurs, in ferite the else block only gets executed if **no** exception occurs. The fix block is called when an exception occurs. This control structure is used as follows:

```
iferr {
statements
} fix {
statements to clean up incase of an exception
} else {
```

```
statements if no exception has occurred
}
```

It is possible to nest iferr-fix-else blocks. When an exception does occur a global variable called err is instantiated. This has two attributes, string errstr and number errno - these provide information on the error that occurred. Exceptions are propagated up through the system until a handler is found or the program has a forced termination.

3.9. Functions

Functions are made up of statements, variable declarations and normal program statements. Each statement is terminated by means of a ; - the same as C and Java. Apart from the anonymous function, functions are declared as follows:

```
function function_name( parameter declarations ){
variable declarations
statements
}
```

- *function_name* -- This is the name of the function to be called e.g. Print, Open.
- *parameter declarations* -- This is the signature of the arguments that can be passed to the function, and these are of the following form: <type> <name> (a comma seperated list)
- *variable declarations* -- See section *Variables*
- *statements* -- See section *Statements*

Example:

```
/*
  This function will add the string "foo" onto the end of the string it has been given and
  return it.
*/
function foo( string bar ) {
bar += "foo";
return bar;
}
```

Functions provide an easy way of grouping statements together to perform a task. It must be noted that all variables must be declared **before** any other code - it is not possible to declare variables within the other statements - it will cause a compile time error.

Functions can take a varaible number of arguments by placing a **...** at the end of the argument list. An array called **fncArgs** will be populated with **all** the variables passed to the function. They can be used as follows:

Example:

The following program listing shows how to access the array and make use of it.

```
uses Array, Console;

function test( string fmt, ... ){
   number i = 0;

   Console.println( "test() called with " + Array.size(fncArgs) + " args" );
   Console.println( fmt );

   for( i = 0; i < Array.size(fncArgs); i++ ){
  Console.println( "Arg[$i]: " + fncArgs[i] );
   }
}

test( "nice" );
test( "nice", "two", "pretty" );
```

If there is not an explicit return statement then the function will return a void variable. To return a variable it is as simple as using the **return** keyword:

Example:

```
return someValue * 10;
return 0;
return "Hello World";
```

3.10. Classes and Objects (and references)

A class is a collection of data and methods, where the methods are intended to work on the data. Classes are templates for variables they describe how complex data types work. To use a class it is necessary to create and instance of a class (see the new keyword) and assign it to an object variable. The syntax of a class is as follows:

```
class <name of class> {
    <variable and functions declarations>
}
```

An example class:

```
class foo {
string bar;

function foo( string str ){ // constructor
self->bar = str; // make bar equal to passed string
}

function printBar(){
Console.println( self->bar ); // print bar
}
}
```

```
This defines a class with a string and two methods. To create an instance of this class you
would do the following:
```

```
object someObj = new foo( "Hello World" );
someObj.printBar(); // will output Hello World
```

To reference variables and methods from within the class it is necessary to prefix the variable with **self->** or **self**. (remember that -> and . are the same in ferite). This merely tells ferite that you want the variable within the class (it is not necessary to do this for locally scoped variables within methods).

Classes can have constructors, these are within the form of a method with the same name as the class. The constructor will be called implicitly when an instance is created. It is suggested that you place you initialisation code here. (It should be noted that you can use all variables within a class in the constructor as they have already been created for you). An example of a constructor can been see above - method is called **foo**.

It is possible to extend classes by using inheritance, this is done using the **extends** keyword. There is no multiple inheritance and an example of inheritance is:

```
class Person {
string name;
number age;

function Person( string n, number a ){ // constructor
self->name = n;
self->age = a;
}
}

class Employee extends Person {
number salary;

function Employee( string n, number a, number sal ){ // constructor
super->Person( n, a );
self->salary = sal;
self->name += " - Employee"; // change the name
}
}
```

```
These classes are not usable in any fashion but merely highlight inheritance.
```

A couple of important facts need to be noticed:

1. When inheritance occurs and then an instance is made, the constructor of the super (parent) class is **not** called. It is up to the subclass (child) to explicitly call it.
2. To get the object as a cast of the super class the super keyword is used.

Currently there is no support for private members of a class. This is a planned addition in the future.

When an instance of a class is created it is added to the garbage collector so that is can keep an eye on it. Then a reference is returned - this is merely a pointer to the object within the system, this means that if you then assign one object variable to another - they both point to the same object.

Example:

```
class foo {
string name;

function foo( string n ){ //constructor
self->name = n;
}
}

object objA, objB;
objA = new foo("boris" );
objB = objA; // they both now point to the same object foo with name="boris"
```

Due to this and combined with the garbage collector, objects will automatically get cleaned up and removed from the system when they are not referenced anymore. It should also be noted that the garbage collector does work based on reference counting and is therefore susceptible to circular references.

3.10.1. Static Members

Ferite supports static members within classes. These act the same as within Java and allow to have functions and variables on a per class basis rather than a per object basis. Static functions and variables are to classes what functions and variables are to namespaces.

This is used as follows:

```
static function_name( parameter declarations ){
```

```
variable declarations
statements
}

or

static number nameofvar;
```

3.11. Namespaces

Namespaces are defined in the following manner:

```
namespace name of namespace {
variable, namespace, class, and function declarations
}
```

All languages have namespaces, Java's are governed by static members to classes but C has no means of explicitly defining them. The are a means of grouping likewise data and functions into a group such that there doesn't exist conflicts with names (hence namespace). Functions, Variables, Classes and even other namespaces can be defined within a namespace.

Example:

```
A standard error message for two different systems within the same program - Text and
Graphical. In C it would have to be done like so:

void text_print_error_message( char *msg );
void graphical_print_error_message( char *msg );

Whereas in ferite:

namespace Text {
function printErrorMessage( string msg ){}
// other stuff to do with text
}

namespace Graphical {
function printErrorMessage( string msg ){}
// other stuff to do with graphical
}

The ferite method is cleaner as it is more obvious what belongs to what, and also allows th
programmer using the Graphical and the Text API's to know that if they want to show an erro
message they merely call the printErrorMessage() function in which ever namespace they
want.
```

They promote clean and precise code. When a function is defined within a namespace it has to reference stuff within the namespace as code out side does, e.g. <namespace>.<resource>.

There is also an alternative syntax for namespaces allowing you to extend an already existing namespace or create a new one if it doesn't already exist. This is done like so:

```
namespace extends name of namespace {
variable, namespace, class and function declarations
}
```

When this extends the namespace it places all items within it in the block in the namespace mentioned. Eg:

```
namespace foo {
number i;
}

namespace extends foo {
number j;
}
```

In the above example the namespace **foo** has a number i and a number j. The main reason for this syntax was to allow module writers to easily intermingle native and script code within the namespace. There is also times when placing something in another namespace makes more sense. e.g. Placing a custom written network protocol within a Network namespace.

3.12. Regular Expressions

Regular expressions provide a very powerful method of matching and modifying strings. Using special syntax, code that would usually require line after line of special matching code can be summarised within a one line regular expression (from here on in referred to as a regex). They can either be found within the language, e.g. Perl or ferite, or as an add in library, e.g. Python, php and C. ferite's regex's are providied by means of PCRE (Perl Compatible Regular Expressions, a C library that can be found at http://www.pcre.org) and as a result are almost identical in operation to Perl's. Regex's look like this:

Example:

```
s/1(2)3/456/

This one will match all occurrences of the string "123" and swap them with "456"

s/W(or(l))d/Ch\1ris\2/

This is more complicated and will match occurrences of "World" and swap them with
"Chorlrisl". The reason being is due to back ticks which are discussed soon.
```

There are two types of regular expression support and that is match and swap. They are used as follows:

```
m/<expression to match>/
s/<expression to match>/<string to replace it with>/
```

To match an **m** is used, to swap an **s** is used. It is possible to capture strings within the regular expression using the same method as in Perl i.e. By using brackets. The captured strings upon each match are placed into **r<bracket number>** - this is equivalent to the $1, $2, ... $n strings in Perl. The maximum number of captured strings is currently 99, and example of captured strings is in the above expressions, i.e. (2) would cause "2" to be place within r1, in the second expression (or(l)) would cause "orl" to be placed within r1 and "l" to be placed within r2.

3.12.1. Options

There are a number of options that can be used to modify the method that the regular expression's execution and processing, these are:

- **x** - This option allows the regular expression to be multi line, and also allows comments using the # character. This is useful for long regular expressions where it is important to remember what each individual part performs.
- **s** - This allows the **. (dot)** matching character to match newlines (\n's).
- **m** - This gets the ^ and **$** meta characters to match at newlines within the source string.

Example:

```
string foo = "Hello\nWorld\nFrom\nChris";
foo =~ s/^(.*)$/Foo/sm;

The above regex will be changed to "Foo\nFoo\nFoo\nFoo"
```

- **i** - This causes the regex engine to match cases without looking at the case of characters being processed.
- **e** - This causes the replace string to be evaluated as if it had been passed to **eval()**. The return value from the script will be used as the replacement text - the return needs to be a string.

Example:

```
string foo = "Hello World";
foo =~ s/Hello/return "Goobye";/ge
Console.println( foo );

foo will now equal "Goodbye World"
```

- **g** - This forces all matches along a line to be matched. Normally it is only the first occurance that is matched.
- **o** - This causes the regular expression to be compiled at compile time rather than runtime. This is useful for regular expressions that dont change but are used alot within a script.
- **A** - The pattern will only match if it matches at the beginning of the string being searched.
- **D** - This option allows the user to have only the **$** tie to the end of a line when it is at the end of the regular expression.

3.12.2. Backticks

Backticks are used within the swap mode of the regular expressions. It allows you to used captured strings within string that should replace the matched expression. There are used within the second example above. They are used as follows: a '\' (back slash) followed by the number that you want to use.

This is only a brief insight into regular expressions, and a suggested read is "Mastering Regular Expressions" by Jeffrey E. F. Friedl (published by O'Reilly), and that will tell you everything you need to know about regular expressions. :-) It is also suggested that the libpcre documentation is worth reading on http://www.pcre.org.

3.13. Uses and Include

Both the **uses** and the **include()** instructions tell ferite to include another script within the current one. The main difference is that **uses** is a compile time directive and **include()** is a runtime directive.

It is **VERY** important to note a specific behavior with these two language constructs. When a script is imported - the script importing it obtains: global variables, classes, functions, namespaces, but **NOT** and I repeat **NOT** the anonymous start function. By this I mean the code that gets run when running the script. The logic behind this apparent maddness is that it allows the anonymous start function to be used to write test cases or examples that go **with** the imported script. This means that if you modify something within the imported script (say it's from a library of scripts) - testing it is just a case of running it through ferite. Or if someone wants an example on how to use it's features - just look at the script being imported. This reduces the number of files that need to be distributed in libraries of scripts and allow distinct test cases/examples not to be lost (and stay current to the API they are against).

3.13.1. Uses

The uses keyword is used to import API from other external modules and scripts. The uses keyword is a compile time directive and provides the method for building up the environment. It can either pull in an external module, or compile in another script. The syntax is as follows:

```
uses name of module or script file, ...;
```

name of module or script file

Module is in the form of a name. (For more information on writing modules please see the document on embedding ferite) eg. **Console**.

Script file is in the form of a string, and it can either be an absolute or relative path. e.g. *"/usr/lib/ferite/scripts/test.fe"*

If either the script or the module can't be found the compilation of the script will cease with an error. It is suggested that these are placed at the top of the script (although this is not a requirement).

3.13.2. Include()

include() operates the same way as **uses**, except that it can currently only import other scripts. Once the call has been made - the facilities provided by the imported script can be used.

```
include ( "someScript.fe" );
```

Chapter 4. Application Interface

4.1.

Please see the document regarding embedding ferite in the developer section of the ferite website.

Chapter 5. Known Issues

The main known issue with version one of the ferite engine is that almost all bugs are found at runtime - there is only checking for correct data types at runtime, this is because of the way the engine operates and will be corrected in version two of the engine.