Ember

A General Purpose Scripting Language for
Stand-Alone, Web Base, and Embedded Applications

Users Guide and Reference

Version 2.0

Feb 2017

Hughes Technologies

Preface

Intended Audience

This document has been prepared as a manual for the Ember programming language and it's associated libraries and modules. It is not designed as an introductory tutorial for procedural programming languages. The syntax and semantics of the Ember language are similar to those of the C language. A working knowledge of C will aid the reader in understanding the Ember language.

Integration of Ember and the World Wide Web is covered in the w3e section. It is assumed that the reader is familiar with the WWW, HTML, CGI scripts and the operation of a web server (http daemon).

Document Conventions

This manual has been designed to be printed on US Letter paper. While many parts of the world utilise the A4 paper size (Australia included), it is not possible to print A4 formatted documents on US Letter paper without loss of information. However, printing of US Letter formatted documents on A4 will result in a correct representation of the document with somewhat larger margins than normal.

Throughout this manual, parts of the text have been flagged with the symbol that appears in the margin opposite this paragraph. Such pieces of text are viewed as being important. The reader should ensure that paragraphs marked as important are read even if the entire manual section is only being skimmed. Important sections will include information such as tips on improving the performance of your applications, or areas where mistakes are commonly made.

Contact Information

Further information about Ember and its related software can be found on the Hughes Technologies Web site. The web site includes the latest version of Ember, documentation, support, example software, references to customer sites, and much more. Our web site can be found at

http://www.Hughes.com.au

Table of Contents

Introduction

Application developers are often faced with problems that are best solved using a scripting language. It may be a stand-alone application that can be rapidly implemented or prototyped in a scripting language such as Perl. It may be a web delivered application implemented using a scripts embedded within HTML such as PHP. It may be a need for macro facilities within an application, provided by an embedded scripting language such as VB Script. To the detriment of both the programmer and the application's user, a different scripting language must be used in each situation.

Ember provides a solution to all the problems outlined above. It is a lightweight procedural scripting language that was designed for use in embedded applications. The first two applications developed with embedded Ember were a front-end for running stand-alone Ember scripts, and a front-end for embedding Ember code within a web page. Both of those applications are provided within the software distribution.

Because the scripting language itself is identical whether it is being used stand-alone, within a web page, or within another application, most of this manual documents how to write scripts using the Ember language. Later in this manual, details of how to embed Ember in an application and how to extend Ember using a dynamic plug-in (an Ember module) is presented.

Basics

Ember has been designed to mimic the syntax and semantics of the C language while reducing some of the complexities and error prone features of C. This is intentional as most programmers working on UNIX machines have a working knowledge of C but look for a more "easy to use" language for scripting. The main changes from C are

- All memory management (i.e. allocation and deallocation of memory for variables) is taken care of by the Ember Virtual Machine. Your script does not need to perform any memory management operations.
- A variable has no fixed type. It will contain whatever is stored in it (e.g text value, numeric value). When you perform an operation on a variable, such as maths or comparisons, the contents of the variable are checked to ensure they are of the correct type. This concept will become clearer as we progress through this documentation.
- There is a dynamic array type. Each element of the array is a variable as described above. The elements are accessed as they are in C, i.e. variable[offset], but they need not be declared before use. That is, the array element is created when a value is stored in it without a pre-definition of the array.
- An associative array type is available to the programmer. As with other associative array implementations, the array index is actually a textual key value, and is specified between {} characters rather than the [ ] characters used to enclose a normal array index.
- Variables are not pre-declared. They are created when they are first used.
- Variable names must start with either a $ or @ character. The $ character signifies a local variable (within the scope of the current function) while the @ character forces a reference to a global variable.

Scalar Types

Variables are constructed from a $ sign followed by alpha-numeric characters and the '_' character. The only restriction placed upon the name of a variable is that the first character of a user defined variable must not be an upper case character. There is no need to pre-declare variables as you do in a language such as C. A variable is created the first time you assign a value to it. Similarly, the type of the variable is defined by the value that you assign to it. There are five types of scalar variables

- text
- integer
- unsigned integers
- real number
- money values (i.e. 2 decimal places)

Integer and unsigned integer types will be either 32 bit or 64 bit depending on the underlying system. If your system supports 64 bit integers then Ember will use 64 bit int and uint types.

The example code below illustrates the creation of variables

```
$int_value = 9;
$uint_value = (uint)240983;
$text_value = "Some text value";
$real_value = 12.627;
$money_value = 123.45;
```

At any point in time, the type of a value can be changed by using the type cast notation from the C language. If, for example, you wished to convert a numeric value into a text string, you would simply cast the value to the appropriate type. The sample below would result in the variable $text_val containing "1234" as a textual string rather than a number

```
$int_val = 1234;
$text_val = (text) $int_val;
```

Array and Assoc Types

Ember support two styles of arrays: normal and associative. A normal array is referenced (or indexed) using a numeric value. The elements of the array are referenced as element 0, element 1, element 2 an so on. In contrast, an associative array uses a textual key value to reference its elements. You may think of an associative array as a simple, in-memory database. An normal array is accessed by specifying the index value between [ ] characters. An associative array key is provided between { } characters. Examples are shown below.

A feature common to both types of arrays is their ability to handle an element of any available type. This allows you to create nested or multi-dimensional array. This is achieved by simply storing an array or assoc in an element of another array or assoc.

```
$arrayval[0] = "Foo";
$arrayval[1] = 5;
$arrayval[2] = 1.23 + 5.38;

$assocval{"Fred"} = "Flintstone";
$assocval{"Tom"} = "Hanks";
$assocval{"Array"} = $arrayval;
```

Array Handling push() pop() shift() unshift() #

Assoc Handling - keys()

Mathematical Expressions

Ember expressions are formed from mathematical equations incorporating literal values, values of variables and values returned from function calls. Ember is a little more flexible than other languages such as C. It will allow you to do maths operations on all data types including the text type. Adding two text values together results in the concatenation of the two strings. You can also perform maths on values of different types by casting the value to the correct type within the expression. Examples are given below.

```
$text_rval = "Hello" + " there!";
$int_val = 8 + 1;
$text_rval = (text)$int_val + " green bottles";
```

The first expression would result in the text value "Hello there!". The second would result in the integer value 9. The final expression would result in the text value "9 green bottles" using the text representation of the value of $intval from the previous line. Maths expression of any complexity, including any number of sub expressions enclosed in ( ) characters, are supported.

Ember provides the full range of "self modifier" operators such as increment, decrement etc. These operator modify the value of the variable on the left hand side of the operator. The complete list is shown in the table below.

Operator	Description	Example	Result
++	Increment	$val = 1; $val++	2
--	Decrement	$val = 5; $val--	4
+=	Self Add	$val = 1; $val += 5;	6
-=	Self Subtract	$val = 10; $val -= 5;	5
/=	Self Divide	$val = 10; $val /= 5;	2
*=	Self Multiply	$val = 10; $val *= 2;	20

A special operator supported by Ember is the count operator written as the # sign. The count operator is used to determine the size of certain variables. If you apply the count operator to a text value it will evaluate to the number of characters in the string. If you apply it to an array it will evaluate to the number of elements in that array. It is important that you are familiar with the count operator as many library functions return an empty string or array to indicate an error. You can catch such a situation by testing for a count of 0. In the first example below, $int_val would contain the value 5. In the second example, it would contain 3.   The third example shows how to test for an empty string.

```
$text_val = "Hello";
$int_val = # $text_val;

$array[0] = 0;
$array[1] = 1;
$array[2] = 2;
$int_val = # $array;

if ( # $textVal == 0)
{
        echo("String is empty!");
}
```

Conditions and Loops

Conditions are provided by Ember using the same syntax as C. That is, the conditional block is started by an 'if (condition)'. The blocks of code are defined using the { and } character. Unlike C, you must always wrap code blocks in { } characters (in C you don't have to if the code block is only one line long). After the initial code block, an optional 'else' block may be defined.

Multiple parts of the conditional expression may be linked together using logical ANDs and ORs. Like C, the syntax for an AND is && while the syntax for an OR is ||. As you will see in the example below, Ember provides more flexibility than C in conditions containing text values. You can compare two text values using the '==' equality test or the '!=' inequality test rather than having to use a function such as strcmp().

```
if ($int_val 5 && $int_val < 10)
{
        echo("The value is between 5 and 10\n");
}
else
{
        echo("The value is not between 5 and 10\n");
}
```

Ember supports only one form of looping - a 'while' loop. The syntax and operation of the while loop is identical to the while loop offered by the C language. This includes the use of 'continue' and 'break' clauses to control the flow of execution within the loop. The continue clause will force execution to return to the start of the enclosing while loop. The break clause will force the loop to be exited immediately.

```
while ($intval < 10)
{
        $intval = $intval + 1;
}
while ($charval != "")
{
        $charval = readln($fd);
        if ($charval == “Hello”)
        {
                continue;
        }
        if ($charval == "Bye")
        {
                break;
        }
        echo($charval);
}
```

User Defined Functions

As with most modern programming languages, Ember allows you to write user defined functions. The definition of a Ember function is

```
funct functName ( type arg, type arg ...)
{
        statements
}
```

As the definition dictates, a function must be started with the *funct* label. The remainder looks like a C function declaration in that there is a function name followed by a list of typed arguments. Any type may be passed to a function and any type may be returned from a function. All values passed to a function are passed by value not by reference. A few example functions are given below.

```
funct addition ( int $value1, int $value2 )
{
        return ( $value1 + $value2 );
}
```

```
funct merge ( array $values)
{
        $count = 0;
        $result = "";
        while ( $count < # $values)
        {
                $result = $result + $values [ $count ];
                $count++;
        }
        return ( $result );
}

funct sequence ( int $first, int $last )
{
        $count = 0;
        while ( $first < $last )
        {
                $array [$count] = (char) $first;
                $first++;
        }
        return ( $array );
}
```

It must be noted that function declarations can only be made before any of the actual script code of the file. That is, all functions must be defined before the main body of the script is reached.

Ember enforces a strict scope on variables used in user defined functions. Any variable referenced by the function is defined as a local variable in that function even if there is a global variable by the same name. Parameters are passed by value, not by reference, so any modification of the parameter variables is not reflected outside the scope of the function. The only way to modify the value of variables outside the scope of the function is by returning a value from the function or by explicitly referencing global variables as outlined below.

Ember supports the concept of explicitly accessible global variable by using a different syntax when referencing the variable. If a variable is referenced as $variable then it is a variable within the current scope (a local variable if it is referenced in a function, a global variable if referenced from the main code). If a variable is to be explicitly referenced as a global variable then it can be referenced as @variable rather than $variable (a preceeding “@” character rather than a “$” character). This will force the Ember symbol table management routines to access the global symbol table rather than the symbol table associated with the current execution scope.

User Defined Libraries

To help provide an efficient programming environment, Ember allows you to build a library of functions and load the library into your script at run-time. This allows for effective re-use of code in the same way the languages such as C allow you to re-use code by linking against libraries. The main difference is that the library is not "linked" into the script, it is loaded on request at run-time (a little like a C shared library). If the functions that were defined in the previous section of this manual were placed into a library called "mylib", a script could access those functions by loading the library as depicted below.

```
load "mylib.lib";

/*
** Now we can use the functions from the “mylib” library
*/
$array = sequence(1,10);
$count = 0;
while ($count < # $array)
{
        printf("Value %d is '%s'\n", $count, $array);
        $count = $count + 1;
}
```

The power and convenience of Ember libraries is most obvious when writing large WWW based applications using w3e (Ember embedded in HTML). Like any application, there will be actions that you will need to perform several times. Without the aid of libraries, the code to perform those actions would need to be re-coded into each Ember enhanced web page (because each HTML file is a stand-alone program). By placing all these commonly used functions into a library, each web page can simply load the library and have access to the functions. This also provides a single place at which modifications can be made that are reflected in all web pages that load the library.

Library files are not like normal Ember script files. A Ember script file is a plain ASCII text file that is parsed at run-time by Ember. A library file contains pre-compiled versions of the Ember functions that will load faster as they do not need to be re-parsed every time they are used. A Ember library file is created by using the -l flag of the Ember interpreter. If a set of functions was placed in a file called mylib.e, a compiled version of the library would be created using the syntax shown below.

```
ember –l mylib.e
```

The -l flag tells Lite to compile the functions and write the binary version of the functions to a file called mylib.lib. This is similar to the concept of using the C compiler to create an object file by using the -c flag of the compiler.

There are three points that should be noted about the use of Ember libraries. Firstly, it should be noted that a Ember library can only contain functions (i.e. it cannot contain any "main body" code that you would normally include in a script file). Secondly, like functions themselves, a library can only be loaded into a Ember script prior to the start of the main body code. Finally, the path given to the load command within the script does not enforce a known location for the library file. If you specify the library file as "mylib.lib" then Ember will expect the library file to exist in the current directory. You can of course provide a complete pathname rather than just a filename to the load command.

Dynamic Loading Modules

Included with Ember is a library of predefined functions known as the Standard Module. In Ember, a module is used to provide an interface between an Ember script and any external functionality. The Standard Module provides routines for string manipulation, maths, and access to operating system features such as file input/output. To provide further flexibility, Ember supports dynamic loading of modules at run-time. This allows features such as database access or network operations to be made available to an Ember script even though they are not part of the standard language.

Dynamic Modules are loaded into a script using the modload directive. Once loaded, all the functions provided by the module may be called from the Ember script. Included below is an example that loads the Mini SQL database access module and performs some basic operations against a database. Dynamic modules for Ember can be found on the Hughes Technologies web site at www.Hughes.com.au

```
/*
** Load the mSQL module using the modload directive
*/

modload “mod_msql”;

/*
** Connect to the server and insert a new staff record.  Note that there
** is no error checking done in this example.  It is not intended as an
** instruction into the use of mSQL – just an example of modload
*/
$sock = msqlConnect(“msql.mydomain.com”);
msqlSelectDB($sock, “my_database”);
$query = “insert into staff values (‘Fred’,’Smith’,’123 Hill St.  Southport, Qld 4217’)”;
msqlQuery($sock, $query);
msqlClose($sock);
```

Ember's Standard Module

The standard module is to Ember as the standard C library is to C. It is a library of functions that are available to all Ember programs. It provides basic functionality for string manipulation, file IO and other normal expectations of a programming language. Outlined below is a description of each of the functions available within the standard module.

Input Output Routines

echo ( )

```
echo ( string )
        text  string
```

echo() outputs the content of argument passed to it.. Any variables that are included in string are evaluated and expanded before the output is generated.

```
$name = "Bambi";
echo("My name is $name\n");
```

printf ( )

```
printf ( format [ , arg ... ] )
        text  format
```

printf() produces output in a manner similar to the echo function except that it allows for the output to be formatted based on the *format* argument. The semantics of the function are the same as those of printf() in C. The printf() format can include field width and justification information. Specification of a format field as "%17s" will generate a right justified value 17 characters wide. Prefixing the field width definition with the '-' character will produce a left justified result.

Example :

```
$name = "Bill";
printf("My name is also %10s\n", $name);
```

fprintf ( )

```
fprintf ( fd , format [ , arg ... ] )
        int fd
        text format
```

Like printf(), fprintf() produces text output based on the content of the *format* string and the arguments passed to the function. Unlike printf(), fprintf() sends the output to a file or other object referenced by the file descriptor passed as its first argument. The file descriptor must be created using the open() function. See the description of open() for more information.

Example :

```
$name = "Bambi";
$fd = open("/tmp/name","");
fprintf($fd, "My name is %10s\n", $name);
close($fd);
```

sprintf ( )

```
sprintf (format [ , arg ... ] )
        text buffer
        text format
```

Like printf(), sprintf() produces formatted text output based on the content of the *format* string and the arguments passed to the function. However, the output is stored in the text *buffer* provided as the first argument to the function call rather than being printed as output.

Example :

```
$name = "Bambi";
$outbuf = sprintf("My name is %10s\n", $name);
echo($outbut);
```

open ( )

```
int fd = open ( path , access )
        text path
        text access
```

open() opens the object (usually a file) pointed to by path for reading and/or writing as specified by the access argument, and returns a file descriptor for that newly opened file. The possible values for the access flags are:

Flag	Description
<	File is opened for reading
>	File is opened for writing
>>	File is opened for appending
<>	File is opened for reading and writing
<P	Create a named pipe in the file system and open it for reading
>P	Create a named pipe in the file system and open it for writing
<|	The contents of the path argument is a command string. The command is executed and the output of the new process is available to be read from the returned file descriptor
>|	The contents of the path argument is a command string. The command is executed and any data written to the returned file descriptor is passed as input to the new process

An error is indicated by a returned value of –1. In such a case, the system variable $ERRMSG will contain the error message.

It should be noted that both the named pipe related modes create the pipe prior to accessing it. If the pipe exists in the file system prior to the call, open() will fail.

Example :

```
$fd  = open("/tmp/output", ">");
if ($fd < 0){
        echo("Error : $ERRMSG\n");
        exit(1);
}
fprintf($fd,"This is a test\n");
close($fd);

$fd = open("ls -l /etc", "<|");
$line = readln($fd);
echo($line);
close($fd);
```

It should also be noted that the command string provided when reading from or writing to a process is executing using exec( ) not by passing it to system( ) or to the shell. As a result the failure to execute the specified command is trapped and a -1 is returned in that case. However, because the shell is not involved there is no file globbing performed on the command string. If you require file globbing then pass your command string to the shell using syntax as outlined below. Note that you will no longer receive an error of the command string doesn't execute (because Ember will see that /bin/sh executes properly and return that status).

```
$fd = open("/bin/sh –c 'ls -l /etc/*.txt'", "<|");
$line = readln($fd);
echo($line);
close($fd);
```

close ( )

```
close ( fd )
        int fd
```

close() closes an open file descriptor. If the descriptor relates to a file or a pipe, the file or pipe is closed. If the descriptor is a process, the standard input of the process is closed (and the process should terminate when it reads an EOF from its input).

Please note that if you do not close all file descriptors you open then your program will eventually run out of file descriptors. Naturally, all open descriptors are closed automatically when the script exits.

Example :

```
$fd = open("/tmp/input", "<");
close ($fd);
```

readln ( )

```
readln ( fd )
        int fd
```

readln() reads a line of text from the nominated file descriptor and returns the data. The end-of-line value is not removed from the data returned. An empty string will be returned when the end of file is reached or an error is encountered. If an error occurred, $ERRMSG will be set to a non-empty string on error.

Example :

```
$fd = open("/etc/passwd","<");
$line = readln($fd);
```

readtok ( )

```
readtok ( fd , token )
        int fd
        text token
```

readtok() reads data from the file descriptor until it finds the character specified as the token in the input data. Only the data read prior to the token is returned, the token character itself is not.

Please note that the token is a single character value. If more than one character is passed in the token argument, only the first character is used.

Example :

```
$fd = open("/etc/passwd", "<");
$username = readtok($fd, ":");
printf("Username is '$username'\n");
close($fd);
```

String Manipulation Routines

split ( )

```
split ( str , token )
        text str
        text token
```

split() splits the contents of a variable into multiple substrings using the value of token as the separator character. The result of splitting the string is returned as an array. If more than one character is passed as the *token*, all but the first character is ignored.

Example :

```
$line = "bambi:David Hughes:Hughes Technologies";
$info = split($line,":");
printf("Username = $info[0]\n");
printf("Full name = $info[1]\n");
printf("Organisation = $info[2]\n");
```

strseg ( )

```
strseg ( str , start, end )
        text str
        int start
        int end
```

strseg() returns a segment of the string passed as the *str* argument. The segment starts at *start* characters from the start of the string and ends at *end* characters from the start of the string. In the example below, $sub will contain the string "is a".

Example :

```
$string = "This is a test";
$sub = strseg($string, 5, 8,);
```

chop ( )

```
text chop ( str )
        text str
```

chop() removes the last character from the text string *str* and returns the new value. The primary use of this function is for chopping end-of-line characters off strings read from files with readln().

Example :

```
$line = readln($fd);
$line = chop($line);
```

tr ( )

```
text tr ( str , expr1 , expr2 )
        text str
        text expr1
        text expr2
```

tr() performs text translations on the string argument *str* based on the contents of *expr1* and *expr2* and returns the modified string value. *expr1* and *expr2* are sets of characters. Any character that is found in *str* that matches a character in *expr1* is translated to the corresponding character from *expr2*. The character sets can be defined by listing individual characters or by providing character ranges (such as A-Z to indicate all characters between A and Z). The example below will translate any upper case characters to lower case and translate any exclamation marks '!' found in the string with a full stop '.'

Example :

```
$str = "Hello There!";
$str = tr($str, "A-Z!", "a-z.");
```

sub ( )

```
text sub ( str , expr1 , expr2 )
        text str
        text expr1
        text expr2
```

sub() performs string substitutions on the string argument *str* based on the contents of *expr1* and *expr2*. If the string value passed as *expr1* is found anywhere in *str* it is substituted for the value if *expr2*. The example below would leave the value "This was a test" in $str. Note that unlike tr() the length of the string can be modified by sub() as there is no restriction on the content or length of the value of *expr2*.

Example :

```
$str = "This is a test";
$str = sub($str, "is", "was");
```

substr ( )

```
text substr ( str , regexp , pattern )
        text str
        text regexp
        text pattern
```

substr() extracts substrings from *str* based on the regular expression *regexp* and the extraction pattern *pattern*. Any parts of the string that are matched by parts of the regular expression enclosed in parenthesis are made available to the extraction pattern. The first such substring is available as $1, the second as $2 and so on. The string value created by expanding any such variables in *pattern* is returned. The example below would produce the string "Who's Jack?" as the regular expression enclosed in parenthesis will match a word containing a leading capital letter followed by lower case letter.

Example :

```
$str = "well, Jack is alright.";
$new = substr($str, ".*, ([A-Z][a-z]*) .*", "Who's $1?");
```

expandText ( )

```
text expandText ( str)
        text str
```

expandText() performs a variable expansion on the value provided as the *str* argument and returns the resulting text value.

Example :

```
$first_name = “David”;
$last_name = “Hughes”;
$info = “My name is $first_name $last_name”;
$new = expandText($info);
```

crypt ( )

```
text crypt ( str, salt)
        text str
        text salt
```

crypt() encrypts the string provided in the *str* argument using the first two characters of the *salt* argument as the key. If more than 2 characters are provided only the first 2 are used. The crypt implementation is compatible with the old-styled crypt included in UNIX distributions. This is a one-way crypt function so you cannot decrypt the resulting string. The first two letters of the encrypted value will be the *salt* used to generate it.

Example :

```
$mypw = crypt(“My secret info”, “AB”);
$salt = substr($mypw, 0, 2);
$testval = crypt($someValue, $salt);
if ($testVal == $mypw)
{
        echo(“test value matched password”);
}
```

char2ascii ( )

```
char2ascii ( character )
        text character
```

char2ascii returns the ASCII code of the first character of the given text string.

Example :

```
$ascii = char2ascii ( “A” );
echo(“The ASCII value of A is $ascii \n”);
```

ascii2char ( )

```
ascii2char ( ascii )
        int  ascii
```

ascii2char returns the character associated with the given ASCII code.

Example :

```
$ascii = char2ascii ( “A” );
$char = ascii2char( $ascii + 1 );
```

```
echo("The character after A is $char \n");
```

setchar ( )

```
setchar ( str., value, offset )
        text str,
        text value;
        int offset
```

setchar ( ) sets the value of the character located at the specified offset in the specified string to the character value provided. The modified string is returned.

Example :

```
$string = "Hello";
$string = setchar($string, "A", 0);
```

getchar ( )

```
getchar ( str., offset )
        text str,
        int offset
```

getchar ( ) returns the character value located at the specified offset of the specified string.

Example :

```
$string = "Hello";
$char = getchar($string,  0);
```

System Services Routines

chdir ( )

```
chdir ( path )
        text path
```

chdir() changes directory to the specified path.

Example :

```
if (chdir("/tmp/myDirectory") < 0)
{
        echo("ERROR : $ERRMSG\n");
}
```

exit ( )

```
exit ( status )
        int status
```

exit( ) terminates the execution of the script and returns the status value specified to the calling application..

Example :

```
if ($value < 0)
{
        exit ( -1 );
}
```

fatal( )

```
fatal ( message )
        text message
```

The fatal( ) function will generate a standard runtime error message on the standard error of the process and terminate the script.

Example :

```
If ($fd < 0)
{
        fatal("Can't open file : $ERRMSG");
}
```

getcwd ( )

```
text getcwd ( )
```

getcwd( ) returns the name of the directory in which the Ember script is currently executing.

Example :

```
$dir = getcwd( ) ;
```

geteuid ( )

```
int geteuid ( )
```

geteuid( ) returns the effective user ID of the user executing the Ember script.

Example :

```
$euid = geteuid( ) ;
```

getgid ( )

```
int getgid ( )
```

getgid() returns the process group ID of the process running Ember.

Example :

```
$gid = getgid( ) ;
```

getpid ( )

```
int getpid ( )
```

getpid() returns the process ID of the process running Ember.

Example :

```
$pid = getpid( ) ;
```

getppid ( )

```
getppid ( )
```

getppid() returns the process ID of the process that is the parent of the process running Ember.

Example :

```
$parent = getppid( ) ;
```

getuid ( )

```
int getuid ( )
```

geteid( ) returns the real user ID of the user executing the Ember script.

Example :

```
$uid = getuid( ) ;
```

kill ( )

```
kill ( pid , signal )
        int pid
        int signal
```

kill() sends the specified signal to the specified process.

Example :

```
if (kill(1, 9) < 0)
{
        echo("ERROR : $ERRMSG\n");
}
```

link ( )

```
link ( path , new )
        text path
        text new
```

link() will create a new link named new to the file specified by path. You cannot create a link over a file system boundary.

Example :

```
if (link("/tmp/foo", "/tmp/baa") < 0)
{
        echo("ERROR : $ERRMSG\n");
}
```

mkdir ( )

```
mkdir ( path )
        text path
```

mkdir() creates the directory specified by path.

Example :

```
if (mkdir("/tmp/myDirectory") < 0)
{
        echo("ERROR : $ERRMSG\n");
}
```

random ( )

```
int random ( )
```

random( ) returns a random number .

Example :

```
$num = random( );
```

rename ( )

```
rename ( old , new )
        text old
        text new
```

rename() renames the specified file from the old name to the new name. You cannot rename files over the boundary of a file system.

Example :

```
if (rename("/tmp/foo", "/tmp/baa") < 0)
{
        echo("ERROR : $ERRMSG\n");
}
```

rmdir ( )

```
rmdir ( path )
        text path
```

rmdir() removes the specified director from the file system.

Example :

```
if (rmdir("/tmp/myDirectory") < 0)
{
        echo("ERROR : $ERRMSG\n");
}
```

sleep ( )

```
sleep ( time )
        int time
```

sleep() will suspend operation of the script for time seconds.

srandom ( )

```
srandom ( seed )
        int        seed;
```

srandom( ) seeds that random number generator used to product the numbers made available via the random( ) function. By seeding the number generator with a semi-random number you can introduce further “randomness” into the number sequence

Example :

```
$seed = getpid( ) * getppid( ) ;
srandom ( $seed );
```

stat ( )

```
stat ( path )
        text path
```

stat() provides an interface to the stat() system call. The information from stat() is returned as an array. The elements of the array are:

Field	Description
0	Inode number
1	File mode
2	Number of links to file
3	UID
4	GID
5	Size of file
6	atime
7	mtime
8	ctime
9	Block size of file system
10	Number of file system blocks used

Example :

```
$sbuf = stat("/tmp/foo");
if ( #$sbuf == 0)
{
        echo("ERROR : $ERRMSG\n");
}
else
{
        echo("/tmp/foo is $sbuf[5] bytes long\n");
}
```

symlink ( )

```
symlink ( path , new)
        text path
        text new
```

symlink() will create a symbolic link called new to the file specified by path.

It should be noted that if the installation process determined that your operating system does not support the symlink() system call this function will not be available.

Example :

```
if (symlink("/tmp/foo", "/tmp/baa") < 0)
{
        echo("ERROR : $ERRMSG\n");
}
```

system ( )

```
system ( command )
        text command
```

system() will execute the command line specified by command in a subshell. Any output generated by the command is included in the HTML output. The exit status of the command is returned to the caller.

Example :

```
if (system("ls -l") != 0)
{
        echo("Error running ls! \n");
}
```

test ( )

```
test ( test, filename )
        text test
        text filename
```

test() offers functionality similar to the test program provided by the shell. Given a filename and a test, it will determine if the file matches the test specification. If it matches, 1 is returned otherwise 0 is returned. A table outlining the available tests is shown below.

Test	File Type
"b"	Block mode device
"c"	Character mode device
"d"	Directory
"p"	Named pipe
"s"	Non-empty regular file
"f"	Regular file
"u"	File is setuid
"g"	File is setgid

Example :

```
if (test("b", "/tmp/foo") == 1)
{
        echo("/tmp/foo is a block device\n");
}
```

truncate ( )

```
truncate ( path , length)
        text path
        int length
```

truncate() will set the length of the file to the specified length.

Example :

```
if (truncate("/tmp/foo", 0) < 0)
{
        echo("ERROR : $ERRMSG\n");
}
```

unlink ( )

```
unlink ( path )
        text path
```

unlink() removes the named file from the file system. If the file does not exist or another error occurs, a negative value is returned and the $ERRMSG variable is set to an appropriate error message

Example :

```
if (unlink("/tmp/foo") < 0)
{
        echo("ERROR : $ERRMSG\n");
}
```

Date / Time Related Routines

time ( )

```
time ( )
```

time() returns the time since 00:00:00 GMT, Jan. 1, 1970, measured in seconds as an integer value.

Example :

```
$time = time();
echo("The number of seconds since Jan 1 1970 is $time\n");
```

strftime ( )

```
time ( fmt, time )
        text fmt; int time;
```

strftime( ) returns a text representation of the UNIX time value time based on the format string passed as fmt. The available formatting options are

Option	Description
%a	day of week, using locale's abbreviated weekday names
%A	day of week, using locale's full weekday names
%b	month, using locale's abbreviated month names
%B	month, using locale's full month names
%d	day of month (01-31)
%D	date as %m/%d/%y
%e	Day of month (1-31 with single digits preceded by a space)
%H	hour (00-23)
%I	hour (00-12)
%j	Day of year (001-366)
%k	hour (0-23, blank padded)
%l	hour (1-12, blank padded)
%m	month number (01-12)
%M	minute (00-59)
%p	AM or PM
%S	seconds (00-59)
%T	time as %H:%M:%S
%U	week number in year (01-52)
%w	day of week (0-6, Sunday being 0)
%y	year within the century (00-99)
%Y	year including century (e.g. 1999)

Example :

```
$time = time();
$message = strftime("The time is %H:%M:%S on %A, %e %B", $time);
echo("$message\n");
```

ctime ( )

```
ctime ( time )
        int time
```

ctime() converts a value returned by time() into the standard UNIX text representation of the date and time.

Example :

```
$time = time();
printf("The date and time is '%s'\n",
ctime($time));
```

time2unixtime ( )

```
time2unixtime ( sec, min, hour, day, month, year )
        int sec , min , hour , day ,  month ,  year;
```

time2unixtime() provides a facility by which you can create a standard UNIX time value (i.e. the time since 00:00:00 GMT, Jan. 1, 1970, measured in seconds) for any specified date/time.

Example :

```
$time = time();
$time2000 = time2unixtime(0,0,0,1,1,2000);
printf("The number of seconds before the end of the century is %d\n",
$time2000 - $time);
```

unixtime2* ( )

```
unixtime2* ( time )
        int time;
```

The above functions take a UNIX time value (i.e. seconds since Jan 1, 1970) and return an integer value representing part of the time information. A list of the functionality provided by the individual routines is shown below.

- unixtime2year() - The year in which time falls
- unixtime2month() - 1 to 12 representing the month in which time falls
- unixtime2day() - 1 to 31 representing the day in which time falls
- unixtime2hour() - 0 to 23 representing the month in which time falls
- unixtime2min() - 0 to 59 representing the minute in which time falls
- unixtime2sec() - 0 to 59 representing the second from the start of the minute in which time falls

Example :

```
$time = time();
$year = unixtime2year($time);
$month = unixtime2month($time);
$day = unixtime2day($time);

echo("The date is $day/$month/$year\n");
```

Password file Related Routines

getpwnam ( )

```
getpwnam ( uname )
        text uname
```

Returns the passwd file entry for the user specified by uname. The result is returned as an array with the array elements defined as below.

Element	Contents
0	username
1	password
2	UID
3	GID
4	GECOS
5	home directory
6	shell

Example :

```
$pwinfo = getpwnam("bambi");
if ( # $pwinfo == 0)
{
        echo("User 'bambi' does not exist!\n");
        exit(1);
}
printf("Bambi's home directory is %s and his uid is %d\n",
$pwinfo[5], (int)$pwinfo[2]);
```

getpwuid ( )

```
getpwuid ( UID )
        int UID
```

getpwuid() returns the same information as getpwnam() but uses a UID to identify the user rather than a username. See the definition of getpwnam() above for details of the return format and usage.

Network Related Routines

gethostbyname ( )

```
gethostbyname ( host )
        text host
```

gethostbyname() returns an array of information about the specified host. Element 0 of the array contains the hostname while element 1 contains the hosts IP address.

Example :

```
$info = gethostbyname("www.Hughes.com.au");
if ( # $info == 0)
{
        echo("Host unknown!\n");
}
else
{
        echo("IP Address = $info[1]\n");
}
```

gethostbyaddress ( )

```
gethostbyaddress ( addr )
        text addr
```

gethostbyaddr() returns an array of information about the specified host. Element 0 of the array contains the hostname while element 1 contains the hosts IP address.

Example :

```
$info = gethostbyaddr("127.0.0.1");
if ( # $info == 0)
{
        echo("Host unknown!\n");
}
else
{
        echo("Host name = $info[0]\n");
}
```

isInCidrBlock

```
int isInCidrBlock( addr, block )
        text    addr;
        text    block;
```

isInCidrBlock( ) will determine if a specified IP Address or network is within a particular CIDR network block. Both arguments are specified in CIDR “slash” notation. If no network CIDR mask length is provided (i.e. the slash component) that it is assumed to be a /32, that is a single IP address.

Example :

```
isInCidrBlock( “101.1.2” , “10.1.1.0/24” );
isInCidrBlock( “10.1.1.0/24” , “10.1.0.0/16”) ;
```

HTML / WWW Related Routines

addHttpHeader ( )

```
addHttpHeader ( str )
        text str
```

addHttpHeader( ) adds the provided text string to the information returned to the client's browser within the headers of the HTTP response. There is no return value.

Example :

```
addHttpHeader("Pragma: no-cache");
addHttpHeader("Set-Cookie: $name=$value;");
```

forceHttpAuth ( )

```
forceHttpAuth ( area, path)
        text area
        text path
```

forceHttpAuth( ) generates the headers required to request username/password authentication from a browser. It then ignores the data received. This can be used to invalidate any cached authentication data etc. The area argument is used to uniquely identify the application requestion authentication. The *path* argument provides the location of a file containing HTML that is to be sent to the browser along with the authentication request. If *path* is an empty string a default page is returned.

Example :

```
forceHttpAuth( "My App","/auth_error.html");
```

httpAuth ( )

```
array httpAuth ( area, path )
        text area
        text path
```

httpAuth( ) is used to request authentication information from the browser for the specified *area*. An array is returned to the calling application providing the supplied username in element 0 and the password in element 1. As with forceHttpHeader( ), the path argument identifies and HTML file to be included in the response to the browser. If an empty string is passed as the argument then a default page is sent.

Example :

```
$data = httpAuth("My App", "/auth_error.html");
$uname = $data[0];
$passwd = $data[1];
```

includeFile ( )

```
includeFile ( filename )
        text filename
```

includeFile() may be used to include the contents of the specified file in the HTML output sent to the browser. The contents of the file are not modified or parsed in any way. If the first character of the file name is a / then the filename is an absolute path name from the root directory of the machine. If it is note, the filename is a relative path from the location of the script file.

Example :

```
includeFile("standard_footer.html");
```

setContentType ( )

```
setContentType ( str )
        text str
```

setContentType() can be used to override the default content type sent to in the HTML header of the generated HTML output. If it is to be used, it must be the first line of the script. Note : not even a blank line may preceed a call to setContentType().

Example :

```
setContentType("image/gif");
```

setHttpResponse ( )

```
setHttpResponse ( str )
        text str
```

set HttpResponse( ) can be used to override the default HTTP Response code sent to the browser. The protocol standard defining HTTP should be checked to determine the available valid response codes and the actions associated with them.

Example :

```
setContentType("302 Foundf");
```

urlEncode ( )

```
urlEncode ( str )
        text str
```

urlEncode( ) converts a normal text string into a URL encoded string. URL encoding translates various special characters, such as spaces and control characters, into a textual form. This allows them to be passed as data within a URL or when used with the GET method of a form.

Example :

```
$value = urlEncode($value);
$url = “http://some.host.com/test.ehtml?value=$value”;
```

Misc Routines

defined ( )

```
defined ( var )
        variable var
```

The defined( ) function will determine if the specified variable name or element of a complex t variableexists and has a value assigned to it. If the variable is defined, a non-zero value is returned.

Example :

```
If ( defined( $foo ) )
{
        echo("Yes, the variable exists\n");
}

If ( defined( $assoc{"first_name"} ) )
{
        echo("Yes, the assoc element exists\n");
}
```

Embedding Ember in HTML Pages

w3e is an application that processes Ember code that has been embedded within an HTML document. This allows a web developer to generate dynamic content directly from within their HTML documents. The full power of Ember, including shared Ember libraries and dynamic loaded modules,

Scripting Tag

To facilitate the w3e extensions to normal web pages, Ember code is included in your HTML code. It is differentiated from normal HTML code by including it inside <! > tags. As an example, a w3e version of the legendary Hello World program is provided below.

```
<HTML>
<HEAD>
<TITLE>Hello World from w3e</TITLE>
<HEAD>
<BODY>
<CENTER>
<H1>Introduction to w3e</H1>
<P>

<!
        echo("Hello World\n");
>
</CENTER>
</BODY>
</HTML>
```

As you can see, there is a line of code in the middle of the HTML page, enclosed in <! > tags. When the page is loaded through the w3e CGI program, anything enclosed in <! > tags is parsed and executed as an embedded ember script. Any output generated by the program is sent to the user's browser. In this case, the string "Hello World" would be sent as part of the HTML page to the browser. The remainder of the page is sent to the browser unmodified.

There can be any number of w3e tags within a single page and there can be any number of lines of code within a single w3e tag.

To execute the script depicted in figure do not specify the path to the file in the URL as you would normally do as your browser will just be sent the unprocessed HTML document. To execute the script you must specify a URL that executes the w3e binary and tells it to load and process your script. The w3e binary is called w3e and will usually be located in the /cgi-bin directory (if it isn't there contact your system administrator). If the normal URL of a w3e enhanced web page is /staff/lookup.ehtml, you would load it using the following URL:

```
/cgi-bin/w3e/staff/lookup.ehtml
```

This URL instructs the web server to execute the w3e binary and tells it to load the /staff/lookup.html script file. Some web servers can be configured to execute a CGI based on the suffix of the requested file. Such a server could be configured to automatically execute the w3e CGI program for every file with a suffix of .ehtml. To configure the popular Apache web server to automatically process all .ehtml files with w3e, simply add the following to your httpd.conf file

```
AddType application/ember .ehtml
Action application/ember /cgi-bin/w3e
```

Form Data

One thing virtually all CGI type programs have in common is that they process the contents of an HTML form. The form data is passed to the CGI program via either a GET or a POST method by the http server. It is then the responsibility of the CGI script to decypher and decode the data being passed to it. w3e greatly simplifies this process by converting any form data passed to a script into global Ember variables within the Ember Virtual Machine. These variables can then be accessed by your script code.

When an HTML form is defined, a field name is given to each of the elements of the form. This allows the CGI to determine what the data values being submitted actually mean. When the data is passed to w3e, the field names are used as the variable names for the global variables. Once a set of variables has been created for each form element, the values being passed to the script are assigned to the variables. This is done automatically during start-up of the w3e program.

As an example, imagine that the following form was defined in an HTML page.

```
<FORM ACTION=/cgi-bin/w3e/my_stuff/test.ehtml METHOD=POST>
<INPUT NAME=username SIZE=20>
<INPUT NAME=password SIZE=20 TYPE=PASSWORD>
<SELECT NAME=user_type>
<OPTION VALUE="casual">Casual User
<OPTION VALUE="staff">Staff Account
<OPTION VALUE="guest">Temporary Guest Account
</SELECT>
</FORM>
```

In the example we have defined three fields within the form, two text entry fields called username and password, and a menu called user_type. We have also specified that the action for the form is to call w3e and tell it to process /my_stuff/test.html passing the form data via the POST method. When the data is submitted, the values entered for the three form fields are passed to w3e. It then creates three global variables called $username, $password and $user_type, and assigns the user's data to those variables. The values can then be accessed within the Ember script code embedded in test.ehtml by referencing the variables.

Security Related Features

When building "real" applications with a package such as w3e, security of your application program itself can become an issue. Because the actual program code is embedded in the HTML code, anyone wishing to obtain a copy of your source code would just need to access the w3e enhanced web page directly rather than accessing it via the w3e CGI program (if you are web server doesn't auto-process as outlined earlier). If a user did this, the source code would not be processed and would appear in the HTML sent to the browser. If a user saved the source of the page from their browser they would have a full copy of your source code on their machine. Naturally, this is a major problem for people who write proprietary applications. To overcome this problem we suggest that you place as much of your proprietary code in a precompiled Ember library and load it into your embedded script. Details of library files are outlined in the ***Introduction*** section of this document.

W3e also provides facilities to limit it's capabilities on a per script basis. By using the runtime configuration you can limit or disable filesystem access external program execution. Please see the *Access Control and Runtime Configuration* section of this manual for full details.

Access Control and Runtime Configuration

w3e supports restricted system resource access via a runtime configuration file. The entries in the config file can be used to tighten the security associated with providing access to ember enhanced web pages.

The config file is located in the installation directory (usually /usr/local/ember) and is called w3e.conf. When w3e is called upon to execute a script, the filesystem path of the script (not the web page's URL) is checked against the contents of the config file. The file is split into sections where each section relates to a particular file path prefix. If the path of the script matches the path prefix of a section then the config entires of that section are loaded.

A default config section can be specified by using the * character as the prefix. The default section is loaded for all accesses. Loading of the config file terminates once the first section other than default section is matched against the script's path.

The config items available are defined below

Item	Description
Allow_Exec	Controls permission for execution of external programs via the system and open module routines.
Allow_File_Read	Controls file reading
Allow_File_Write	Controls file writing
Exec_Path_Prefix	If exec is allowed the execernal program's path must start with this prefix
File_Path_Prefix	Any file access must refer to a file that has the specified path prefix
Local_Files_Only	Only files in the current directory can be accessed. chdir is also disabled.

And example config file is included below (it is also available in the misc directory of the distribution)

```
# w3e.conf
#
# w3e supports a range of configuration options that can allow it's to be restricted
# based on the wishes of the system administrator.   Each option is associated with
# a file's path in the filesystem (not the URL path associated with it's web page).
# The first section that matches the file's path will be used.  If a default section
# is provided (using the [*] path) then it is loaded but the search for a specific
# path will continue.
#
# This file must be installed in the installation directory. By default
# it should be /usr/local/ember/w3e.conf
#
#########################################################################

# A path of * indicates the default settings.  A match on this section
# doesn't terminate the config file scan.
[*]
        Allow_Exec = False

# If the script's pathname (not URL path, this is the real filesystem
# path we are talking about) has a prefix of /foo then apply the
# following config options.  This will replace any of the hardcoded
# default settings of those loaded from the default section above
[/foo]
        Allow_Exec = True
        Local_Files_Only = True

# Load some settings for scripts located under the /baa directory
[/baa]
        Allow_Exec = False
        Allow_File_Read = True
        Allow_File_Write = False
        Exec_Path_Prefix = /usr/local
```

Embedding Ember in Applications

Overview

One of the main features of Ember is that it can be embedded into other applications. If your application requires features such as macros or any form of programmatic control, then embedding Ember into your application could be a solution.

Embedding Ember into your code is very simple and requires the following steps

- Create an Ember Virtual Machine
- Load your script code into the VM
- Add any variables to the VM that your script may use
- Parse / Compile the code and report an syntax errors
- Execute the script watching for any run-time errors
- Access the variables of the script to retrieve any result data

To create a new Ember virtual machine simply call the eCreateScript( ) function. It will return you a pointer to a newly created and initialized virtual machine structure.

Your ember source code can be located either in a file or in an array buffer. To load your script from a file, use the eFileSource(script, path) function, passing it a pointer to your Ember VM and a path to the file. To load your script from a buffer, use the eBufferSource(script, code) function passing it your VM pointer and a pointer to a char array containing your script code.

The source code will probably perform some programmatic operation on a set of pre-defined variables. If that is the case, those variables must exist in the VM before the script is compiled. You can create any variables you wish by calling the eCreateVariable(script,varName) function. The variable will be created in the VM's symbol table but it will not contain any value. To define a value for the variable, you must use one of the following functions. The *symbol* argument is the value returned from the call to eCreateVariable( ). Each function will store the provided value into the symbol table and set the type of the variable to the specified type.

- eSetIntValue(script, symbol, integerValue)
- eSetUintValue(script, symbol, uintegerValue)
- eSetRealValue(script, symbol, realValue)
- eSetTextValue(script, symbol, textValue)

Once the VM has been loaded with the script and any variables, the script can be compiled. To compile the script simply call eParseScript(script). If it returns a negative value then a syntax error or similar compile time error was encountered. Details of the error and its location can be found using the eGetLineNum(script), eGetSourceName(script) and eGetErrorMsg(script) routines.

If the script compiled properly then it can now be executed. The script can be executed by calling eRunScript(script). If a negative value is returned then a runtime error was encountered. As with the compile time errors mentioned above, the location and description of the error can be determined by using the eGetLineNum(script), eGetSourceName(script) and eGetErrorMsg(script) routines.

Once the script has executed, the value of the scripts variables can be accessed using eGetVariableValue(script,varName). The value is always returned as a textual representation of the value (i.e. a char string representation).

API Reference

eCreateScript ( )

```
ember *eCreateScript ( )
```

eCreateScript() returns a pointer to a newly created and initialised Ember Virtual Machine structure. The returned pointer is used by all other Ember API routines to control, modify, or access the contents of the virtual machine. A NULL pointer will be returned upon failure.

Example :

```
ember       *script;
script = eCreateScript ( );
```

eFileSource ( )

```
int eFileSource( script, path )
            ember       *script;
            char        *path
```

eFileSource() informs the VM that the script code is located in the file specified in the path argument. The file is opened and internal references to the file are initialised. If the file could not be opened, a negative value is returned to indicate and error. Otherwise a zero value return indicates success.

Example :

```
If ( eFileSource( script, "/tmp/test.e") < 0)
{
            printf("Couldn't open script file\n");
}
```

eBufferSource ( )

```
eBufferSource( script, buffer )
            ember       *script;
            char        *buffer
```

eBufferSource() informs the VM that the script code is located in an in-memory character array. Because a pointer to the buffer is stored within the VM structure, it is important that the contents of the buffer are not modified prior to the script being parsed.

Example :

```
char        source_code[ ] = "$foo = 1 + 2; echo(\"$foo\n\");";
eBufferSource( script, source_code);
```

eCreateVariable ( )

```
e_stab *eCreateVariable( script, varName )
            ember       *script;
            char        *varName;
```

eCreateVariable( ) creates an uninitialised global variable in the VM. The varName argument specifies the name of the new variable. A pointer to the symbol table entry is returned.

Example :

```
e_stab      *symbol;
symbol = eCreateVariable( script, "test_value");
```

eSetIntValue ( )
eSetUintValue( )
eSetRealValue( )
eSetTextValue( )

```
eSetIntValue ( script, symbol, intVal )
        ember       *script;
        e_stab      *symbol
        int         intVal;
```

The eSet*Value( ) set of routines allow you to store a value of the specified type into a variable. The variable is referenced by the symbol argument (usually the result of a call to eCreateVariable). In each variant of this routines, the type of the third argument is the same as the type specified in the functions name (i.e. an int value for eSetIntValue, a text value ro eSetTextValue etc).

Example :

```
eSetIntValue( script, symbol1, 32);
eSetTextValue( script, symbol2, "This is a test");
```

eParseScript( )

```
int eParseScript ( script )
        ember       *script;
```

eParseScript( ) parses and compiles the script code into the VM's internal byte code format. If there is a problem compiling the code, such as a syntax error or reference to undefined variables, an error will be returned. Errors are indicated by a negative return value.

Example :

```
If (eParseScript( script ) < 0)
{
        printf("Error compiling script");
}
```

eRunScript( )

```
int eRunScript ( script )
        ember       *script;
```

eRunScript( ) executes the compiled script code that is stored in the VM. Errors are indicated by a negative return value.

Example :

```
If (eRunScript( script ) < 0)
{
        printf("Error executing script");
}
```

eGetErrorMsg( )

```
text  eGetErrorMsg ( script )
        ember       *script;
```

eGetErrorMsg( ) returns a char pointer to a textual representation of the last error encountered. It may indicate a compile time error or a runtime error depending on the last actions undertaken by the VM.

Example :

```
printf("Error is : %s\n", eGetErrorMsg( ) );
```

eGetLineNum( )

```
int  eGetLineNum ( script  )
        ember      *script;
```

eGetLineNum( ) returns the line number on which the last error was encountered by the virtual machine. If the error was a runtime error that occurred in a loaded library file, the line number will reflect the line number of the library files source code on which the error occurred.

Example :

```
printf("Error is at line  %d\n", eGetLineNum( ) );
```

eGetSourceName( )

```
text   eGetSourceName ( script  )
        ember      *script;
```

eGetSourceName( ) returns the name of the script source. If the source was loaded from a file, the source name will be the name of that file. If the script was provided by a buffer source, the source name will be "input buffer". Please note that the source name does not always indicate the file in which an error occurred as the script may have loaded a precompiled library in which the error was encountered. See eGetCurFileName( ) for more information.

Example :

```
printf("This script was loaded from %s\n", eGetSourceName( script ) );
```

eGetCurFileName( )

```
text   eGetCurFileName( script  )
        ember      *script;
```

eGetCurFileName( ) returns the name of the file in which the currently executing script code was original located. If the current code was loaded from a precompiled library, the returned file name will be the name of the library file. Otherwise it will be the name of the script source. In conjunction with other routines above, this routine can be used to determine the location of an error.

Example :

```
printf("Error found at line %d of file %s while processing script %s\n",
        eGetLineNum( script ),
        eGetCurFileName( script ),
        eGetSourceName( script ) );
```

eGetVariableValue( )

```
text   eGetVariableValue( script, varName  )
        ember      *script;
        char       *varName;
```

eGetVariableValue( ) extracts the contents of the specified global variable from the virtual machines symbol table. The result value is always returned in text form rather than the native type of the variable.

Example :

```
Printf( "The value of the $result variable is '%s' \n",
        EGetVariable(script, "result") );
```

Example

```
/*
** This is a simple program that demonstrates how an ember script can be embedded in a c program and
** used to programmatically evaluate expressions within the c code.  The script code is in an array buffer.
*/

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include "ember.h"

char        source_code[ ]  = "$result = $x + $y; ";

int main(argc, argv)
        int         argc;
        char        *argv[];
{
        ember       *script;
        e_stab      *symbol;
        char        *result;

        /*
        ** Setup the script and point it to our script code buffer (the source_code array)
        */
        script = eCreateScript();
        eBufferSource(script, source_code);

        /*
        ** Add the required variables to the VM
        */
        symbol = eCreateVariable(script,"x");
        eSetIntValue(script,symbol, 10);

        symbol = eCreateVariable(script,"y");
        eSetIntValue(script,symbol, 17);

        /*
        ** Parse/Compile the script reporting any syntax errors
        */
        if (eParseScript(script) < 0) {
                fprintf(stderr,"Error at line %d of script '%s'.\n",
                        eGetLineNum(script), eGetSourceName(script));
                fprintf(stderr,"Error is '%s'\n\n", eGetErrorMsg(script));
                exit(1);
        }

        /*
        ** Execute it and report any runtime errors
        */
        if (eRunScript(script) < 0) {
                fprintf(stderr,"Runtime error at line %d of script '%s'.\n",
                        eGetLineNum(script), eGetSourceName(script));
                fprintf(stderr,"Error is '%s'\n\n", eGetErrorMsg(script));
                exit(1);
        }


        /*
        ** Grab the value of the variable called "result" from the script
        */
        result = eGetVariableValue(script,"result");
        if (result)
                printf("\nResult = '%s'\n\n",result);
}
```

Ember vs Lite

Lite was the initial implementation of the language and functionality outlined in this manual. It was developed as the scripting language that was distributed with the Mini SQL database system. Lite and it's HTML embedded counterpart called w3-mSQL have been replaced by Ember and w3e. When compared to Lite, Ember code executes several times faster and uses less memory. The internals of the virtual machine are also much cleaner and less prone to errors. Ember can also be embedded in other applications – a feature that was missing from Lite.

To port your existing Lite or W3-mSQL application over to Ember there are a few things that must be considered. Firstly, Ember is much more strict about accessing variables that do not exist. Lite allowed you to access nonexistent variables and it simply returned an empty string as their value. Ember however views this as an error and reports it as such. This is an important design change and one that can greatly increase the reliability of your Ember application.

If you require to check for the existence of a variable, Ember provides the *exists( )* function in the standard module. This function returns a positive value of the variable exists and a zero value if it doesn't. See the module documentation for further information.

Ember allows some C styled "short cuts" that Lite didn't understand. All the fragments below are legal in Ember but would have generated errors in Lite

- if ($foo)
- $foo++
- $foo—

As Lite was distributed with mSQL, it included bindings for the mSQL API in the standard distribution. Ember is being distributed as a stand-alone language and as such only includes basic string and system functionality in the base distribution. Access to mSQL is still available but requires the use of the mSQL module. A dynamic module for mSQL access is available from www.Hughes.com.au.

To remove confusion, the *char* type provided by Lite has been replaced by a *text* type in Ember. Ember will compile and execute code that utilises the old *char* type but it will generate a warning message for each use of *char*.

Ember allows an array element to contain an array, providing multi-dimensional arrays. Lite provided no such feature. Similarly, Lite did not provide the Associative arrays available to the Ember programmer.

To simply the use of Ember in applications that make reference to currency, a money type has been added. This new type is simply a fixed precision real value offering two decimal places of precision (e.g. 1.23). Although designed for use with currency values it is also a useful type for many numeric quantities.

W3-mSQL provided a simple and error prone authentication system called W3-Auth. Support for W3-Auth is not available from Ember. Web based Ember applications can manage their own authentication needs using the HTTP Authentication routines provided in the WWW related module.