ESL

An Extensible Scripting Language

Version 0.2
June 1994

ESL has been developed as part of the Minerva Network Management Environment.

Table of Contents

Introduction

ESL, pronounced easel, is an extensible scripting language designed primarily for use in network management and monitor, yet with features well suited to systems administration tasks. The design objectives of ESL were to provide a scripting language that would be familiar to UNIX systems programmers, offer a convenient and efficient programming environment, and have low resource overheads and fast execution. This has been achieved by offering a C styled syntax in conjunction with powerful programming constructs and shell styled simplicity.

The decision to write yet another scripting language was born from a requirement of the Minerva Network Management Environment. Minerva is a user centric management environment with the primary goal of placing power and flexibility back into the hands of the network or systems manager. Historically, Minerva, as depicted on the cover of this manual, was the Roman Goddess of knowledge and information. The objective of the software package by the same name is to offer a central repository for information pertaining to all aspects of systems and network management.

One facet of Minerva's design is to offer a programmatic interface to SNMP based data acquisition so that the manager may perform customised and complex tests in a logical manner. To be successful in this goal, the programmatic interface had to be easy to use and offer a minimal learning curve to the network manager. It is viewed that the vast majority of UNIX programmers, UNIX systems administrators and network managers are familiar with the C programming language and as such, a scripting facility that offered a C styled interface would be comfortable and profitable.

To complete the C like environment, scripts are passed through the C preprocessor prior to interpretation. This caters for code inclusion using *#includes*, condition compilation or interpretation using *#define* directives, the use of text macros in scripts, and also C styled commenting.

ESL's syntax differs from that of C in only 2 key areas, those being variable identifiers and the names of predefined types. Variables names in ESL follow the convention set by the shells in which all names start with a '$'. This tradition has also been followed by the popular Perl programming language. As will be outlined in a later section, the range of types available to the ESL programmer differs significantly to those available in C. The difference is intentional with the aim of providing a simplified and more efficient programming environment.

Although not a syntactic difference, another major deviation from Kernighan and Ritchie C (K&R C) is the fact that ESL is a strongly typed language. All references made to values, either literal or variable, are checked for compatible types. This includes assignments, function parameters, mathematical operations and comparisons. As is outlined in the "Variables and Types" section of this manual, ESL does provide convenient mechanisms for type conversion to overcome the inflexibility that can be associated with many of the strongly typed languages available today.

For a language to be used for network monitoring it must have low system resource requirements and execute quickly. In a typical network environment in which large numbers of network nodes are being monitored, the number of data acquisition tasks running concurrently or in close proximity can be enormous. If a slow or resource hungry scripting language was used to drive the gathering of this information, the resource requirements placed on the platform performing the tasks would be very restrictive. ESL offers a mechanism by which the script can be compiled into a machine independent binary format that removes the requirement of source code parsing at runtime. This form of execution also removes the requirement for the script to be preprocessed by CPP prior to execution, hence reducing the time and resources required to load and execute the program.

The fact that the resultant binary is not host architecture specific offers many advantages. Firstly, the source code of the program need not be installed on the target machine. Similarly, any ESL library modules incorporated into the script using *#include* directives need only be available on the development machine. Although it is available on the majority of machines, CPP is no longer required for execution. On systems where a C development environment, including CPP, is an “optional extra” at a financial premium, this feature may be of great appeal. Finally, as a text based script is no longer being interpreted at runtime, the security problems associated with *setuid* shell scripts is reduced.

A major design goal of ESL was to provide a base language with a feature set rich enough to provide a rapid and powerful scripting environment, but to also offer a mechanism by which the language could be simply extended to suit a users particular requirements. One such custom situation is the requirement of SNMP support for network monitoring. Although executing commands externally in a manner similar to the Bourne shell provides great flexibility, it is a major performance overhead that conflicts with another of the original design goals. The solution has been to provide a simple, table driven mapping of internal ESL function names to external C language functions. External C libraries can then be linked into the ESL binary with access to the library functions being via the stub routines defined in the mapping table.

The stub routine definitions include details of the number and type of function arguments. These are used at runtime to check that external functions are called correctly and to enforce the strongly typed nature of the language. This mechanism has been used not only for unusual requirements, such as adding support for SNMP and aspects of Minerva, but also for the standard ESL file IO library and to provide access to the UNIX system calls. During execution, the ESL runtime module forms a typed, linked list of the data passed to the ESL function calls from within the script. The resultant linked list is passed to the stub routines to provide a simple and consistent mechanism for passing data from the ESL environment to the external C library routines. Similarly, there is a C function available for use in the stub routines that allows a value to be returned to the ESL script upon completion of the external function.

Using the Interpreter

The command line interface to the ESL interpreter closely mimics that of a C compiler. Running ESL without any arguments will display the usage information as depicted below in figure 1.

```
Usage : esl [-vd] [-co outfile] [-Dmacro] [-Idirectory] sourcefile [args]

        -v     Show version information
        -d     Dump symbol and code tables
        -c     Compile code (code not executed)
        -o     Output file for compiled code
        -D     Define pre-processor macro
        -I     Include directory

        Options for debugging esl itself.

        -L     Enable internal logging
        -y     Run yacc parser in debug mode (code not executed)
        -l     Run lexical scanner in debug mode (code not executed)
```

Figure 1 - ESL command line usage

As can be seen by the usage message, CPP macro definitions are passed to the interpreter using the familiar *-D* flag while extra include directories are defined using *-I*. If ESL is executed with the *-c* flag, it indicates that the script is to be compiled rather than executed. This will produce a binary version of the script in a file with a name as specified by the *-o* flag or the name of the script suffixed with .out if no output file name is offered.

In keeping with CPP conventions, a standard directory is used for the initial search for *#include* directives. During installation, a directory heirarchy will be created, typically in /usr/local/Minerva/esl/include, to hold any system-wide ESL header files. The structure of the directory tree mimics that used for the C header files. Inclusion of local files can be achieved either by nominating a search path using the -I flag at runtime or by forcing an explicit path within the include statement.

The interpreter and binary format version number can be obtained using the *-v* option. It will become important to know the binary format version as future releases of ESL are made as this information is embedded into compiled scripts. To allow sane handling of modifications to the binary format, a compiled script will be rejected at runtime if it has not been compiled with a version of ESL with the same binary format as that being used during execution.

Another unusual flag offered by ESL is the dump option, *-d*. Dumping a script produces a textual representation of the variables and literals defined in the script as well as the pseudo code that has been generated during interpretation. The ESL interpreter utilises a stack based machine simulation during script execution that has a defined set of operations in the same way that a physical processor does. The instruction set offers familiar operations such as PUSH, POP, CALL, and RET, as well as some high level operations that are specific to ESL. This option is intended primarily for debugging purposes during development and serves no purpose to the programmer other than showing how the different ESL constructs are interpreted.

Other low level debugging information is available through the -L, -l and -y options. -L enables internal debug and logging output while -l and -y offer detailed debugging information for the scanner/parser combination. The -l flag causes generation of symbolic output from the lex based scanner. This information shows in symbolic terms how the scanner views the file and what tokens it will generate at runtime. The -y option shows the state trace generated by the YACC based parser. The -L, -l and -y options are of no use unless you are trying to debug or port the ESL interpreter.

Variables and Types

As was mentioned in the introductory section of this manual, all ESL variable names are prefixed with a ‘$’. Unlike the shells however, every reference to a variable, including assignment, requires the ‘$’ prefix. This allows a consistent use of variable names and removes possible conflicts between non-prefixed variable names and the names of user defined functions.

The range of types available to the ESL programmer, and the handling of those types, differs greatly to C. Although there are some common types, such as *int* and *char* that are handled in a similar manner, the similarity ends there. The intention behind defining a new set of type has been to offer greater simplicity and power than that offered by C. The most common of the new types is the *text* type. This replaces the *(char *)* construct to which we have become accustomed. The advantage of the *text* type over a pointer to a character is that memory allocation and management is automatic in ESL. The programmer is no longer concerned with *malloc()’ing* enough space for strings or using fixed length arrays of characters. Automatic allocation of memory is utilised for all types available within ESL.

One of the most used types within an ESL script is the *array* type. A variable defined this way is a dynamic array of *text* objects. As with the *text* type, automatic memory allocation removes the detailed concerns of watching array bounds and other memory related problems. An array of this type is indexed using numeric values in the same way an array is indexed in C, that is, the first element of an array is element number 0 rather than element number 1 as is the case in languages such as fortran.

ESL also offers an *assoc* type that allows the definition of associative arrays. An associative array is an array of *text* objects that is indexed using another *text* object. As such, an associative array can be viewed as a simple, keyed database. To provided efficient access to data stored in an *assoc*, ESL utilises a 16 way hash table to find information in the array. Available through the *binddb()* function (see the section labelled "ESL Specific Functions") is a mechanism via which an associative array may be "bound" to an indexed database on disk. ESL uses the NDBM database library for handling of bound-assoc's. The most convenient feature of assoc binding is that access to the array is identical whether the array is in core or bound to disk. Once the variable is initialised, the programmer need not concern him/herself with details of where the database is being stored. This allows variables within a script to appear to retain their value between invocations of the program. This is a handy feature if values acquired at two points in time require comparison (e.g. an interface byte count obtained via SNMP). The programmer need not worry about writing and re-reading text files or other such traditional mechanisms for script based data storage.

Type	*Description*
int	Signed integer value
text	Textual string
char	Single character value
array	Array of text objects
assoc	Associative array of text objects

Figure 2 - Standard ESL Types

Unlike C, ESL supports the notion of undefined or uninitialised variables. Many standard functions return an undefined value as an indication of error rather than returning a negative integer value or an empty text object. As will be described in detail in a later section, a function is available to the programmer to determine the current state of a variable prior to accessing it. An access to an undefined variable will result in a runtime error as this is viewed as an error in the program (eg. passing a variable to a function prior to assigning it a value).

Limited support is available for a construct similar to the C *'&'* address operator. Currently, it is only available for use in external C functions, not in internal ESL functions. Passing a variable to an external function call as *&var* rather than *$var* allows the C code to gain access to the variable's symbol table entry. Although this is not a commonly used feature, it has been required during implementation of some of the standard ESL library functions, in particular, the associative array binding and variable status routines.

To overcome the restrictive nature that is imposed by strongly typed languages, a convenient type conversion mechanism has been included in the design of ESL. It was viewed that the automatic type conversion offered by the shells did not fit the strongly typed mold, while the programmatic methods available to a C programmer were too inconvenient. A middle ground that would be convenient to use yet familiar to C programmers was required. The result has been to use the C type casting syntax to perform type conversions. As such, prefixing an integer variable with *(text)* will generate a text string containing a textual representation of the numeric value contained in the integer variable. Type casting can be used in assignments, maths operations, conditional tests, function parameters and function return values.

A Gentle Introduction

Let us follow in the traditions of language programming manuals and illustrate the language by example. Presented in *sample 1* is a more complicated than usual version of the world's most coded program. As you can see, the style and syntax of the language should be very familiar to any C programmer. This simple program also illustrates some of the shell-like convenience features of ESL. In particular, automatic memory handling, type casting, and "symbolic maths" (ie. maths operation on non-numeric data) have been used to craft this program.

Programe

```
/*
** An ESL version of Hello World
*/

print_string($msg)
     text  $msg;
{
     text  $outmsg;

     $outmsg = $msg + (text) ‘!’;
     printf(“$outmsg\n”);
}


main()
{
     print_string(“Hello World”);
}
```

Output

```
Hello World!
```

Sample 1 - "Hello World" ESL Style

The most notable difference in syntax from that of C arises from the new types available within ESL. Associative arrays and “ordinary” arrays both offer a mechanism via which they can be bulk initialised. This is a form of array literal if you like. Both forms use a comma delimited list where each list element corresponds to a member of the array at the defined index. The fact that arrays are indexed numerically allows a one to one mapping of the list data into the array at the respective position. The first element of the list is mapped to index 0 while the second element of the list maps to index 1 and so on.

Associative arrays on the other hand are indexed using a text value rather than a number. This requires an assoc literal to present the data in a keyed format. This can be achieved by defining a list in which each entry is a grouping of the key and datum pair. It should be noted that the syntax used to access data in an assoc is also slightly different to that used for ordinary arrays. Where an array element is referenced using ‘[ ]’ as syntactic bounds for the index value, an assoc uses ‘{ }’. This is intentional as it highlights the access to assoc’s as the complexity of scripts increases.

As with all ESL variables, memory allocation is automatic. This not only includes the assigned array element itself, but also any missing parts of the array structure. As such, if an assignment is made to element 6 of an array that contains no other data, the first 5 elements of the array will be created and assigned the undefined value.

```
assoc       $assocVal,
            $boundAssoc;
array       $arrayVal;
text        $textVal,
            $keyVal;

/*
** In-core array usage
*/

$arrayVal = [ “Data 0”, “Data 1”, “Data 2”, “Data 3”];
$assocVal = { (“Key 0”, “Data 0”), (“Key 1”, “Data 1”)};
$textVal = $assocVal{“Key 1”};
$arrayVal[4] = $textVal;

/*
** An assoc that’s bound to a database on disk
*/

binddb(&boundAssoc,”/usr/local/user-db”,”<>”);
$boundAssoc{“bambi”} = “David Hughes”;
$boundAssoc{“paulp”} = “Paul Pyyvaara”;
$keyVal = “bambi”;
$textVal = $boundAssoc{$keyVal};
unbinddb(&boundAssoc);
```

Sample 2 - Using ESL Arrays

As is illustrated in sample 2, usage of a bound assoc is no different to usage of an internal assoc. As was mentioned in the introductory section of this manual, an assoc may be bound to a disk resident database structure so that its contents are retained between invocations of the script. Internally, ESL uses *ndbm* to manage the disk based assoc's and a 16 way hash table for in-core assoc's.

The assoc to database bindings are maintained using the *binddb()* and *unbinddb()* functions. Each function requires that the array be passed by reference using the *&* operator. *binddb()* also requires two other arguments, the name of the database file and the access mode. If the code presented in sample 2 was executed, the familiar ndbm files *user-db.dir* and *user-db.pag* would be created in */usr/local.* The example code opens the database for read-write access as is indicated by an access mode of “<>”. ESL uses shell-like syntax for defining access modes for IO operations where they are required. In addition to read-write, the binding could be made as read-only by using an access mode of “<“ and write-only by using “>”.

There are three other features of assoc's that should be noted. Firstly, assoc's can only be defined as global variables. They cannot be declared within the local scope of a function. Similarly, assoc's cannot be passed in their entirety to function calls as an argument. Elements of an assoc may be passed as a text argument, but the entire array cannot be passed at all. Finally, an assoc binding may only be formed if the array in question has no defined value. An attempt to bind an assoc that contains in-core data will result in a runtime error.

Conditionals

Conditional execution of code is available using standard C styled *if/else* clauses. A restriction placed on the syntax of *if* clauses is that parenthesis must be used to block the code within an *if* or *else* clause regardless of the number of statements used. This is in contrast to C's syntax in which you are not required to use parenthesis if there is only one statement within the clause.

Another difference to C is that the list of possible comparisons excludes the bitwise operators, but includes regular expression tests on text objects. The syntax used for the comparisons is identical to that used by C with the inclusion of =~ for regular expression matches. A full list of condition operators is depicted in figure 3.

In keeping with the design goals, ESL attempts to provide flexibility and ease of use to the definition of conditions. Use of the available operators is not restricted to integer and character types. As an example, the equality of two text objects can be checked using == rather than the more programmatic approach of calling *strcmp()* as would be required in a C program. With the exception of both of the array types, comparisons can be made between any two values of the same type. The use of type casting extends the range of possible comparisons. For example, an integer value could be cast to *(text)* so that a regular expression test could be performed on the numeric value now represented in a textual form.

==	Equality
!=	Inequality
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal
=~	Regular expression match

Figure 3 - Comparison Operators

As with C, ESL allows the programmer to write implied conditions, such as

```
if($var)
```

The implied condition is that $var does not equal zero. ESL also supports the logical *not* found in C. As such, it is quite legal to write a conditional statements such as

```
if(!$val)

if (!defined(&val)).
```

ESL also follows the C syntax for defining complex conditional clauses using the logical AND operator, &&, and the logical OR operator, ||. There is no restriction on the number of components of a condition or the types used. For example, it is quite legal to write a condition such as

```
if (($intVar > 7 || $intVar % 2) && $textVar =~ "^.+[0-9]")
```

Looping

Loops are available within ESL scripts in the form of C styled *while* loops. Any of the conditions outlined above may be used as the loop control expression. ESL also supports the *break* and *continue* clauses defined by C for controling the flow of execution within a while loop . As with the i*f/else* clause, the block of code to be executed within the *while* loop must be enclosed in parenthesis regardless of the number of statements to be executed. This is in contrast to the flexible, and often obscuring manner in which C allows blocking of code within a while loop.

It should be noted that as of this release, ESL does not support any looping mechanisms that resemble the *for* loop. This is another area that may be addressed as development continues although it is more likely to result in a *foreach* styled loop construct rather than a simple *for* loop.

Functions

ESL supports the definition of user defined functions as was illustrated in the “Hello World” sample. Due to the stack based nature of the code execution, functions may employ either direct or indirect recursion. Naturally, the level of recursion supported during execution is limited by the memory available to the simulator’s internal stack. As with all languages that do not incorporate recursive loop optimisation (such as tail recursion), recursive implementations of algorithms will result in comparatively slow execution as stack frames are pushed during the function calls.

Data may be passed to and returned from functions in a manner that mimics C. The familiar *return()* directive is used to pass a single value out of a function call. Seeing as pointers to variables are not supported, using return() is the only mechanism for modifying non-global data that is out of the scope of the function. If multiple pieces of information need to be returned, passing textual representations of the data within an *array* is an effective solution. Because functions are not defined as returning any one particular type, any value other than an assoc may be returned. A single function may return a different type of data depending on the arguments passed to it. The runtime code will check the type of the returned data when it is used (e.g. in an assignment statement) rather than when it is returned. As long as the statement using the returned information is typed correctly, this flexibility may be employed.

If during execution of a function, the end of a function is reached (i.e. no explicit call to return has been made), a default NULL symbol with an undefined value will be returned to the caller. As with all symbols, this situation may be checked with the *defined()* function.

Predefined Variables

The ESL runtime module defines several variables prior to execution of the script. Currently, it defines the following global variables

int $stdin	Standard input file descriptor
int $stdout	Standard output file descriptor
int $stderr	Standard error file descriptor
text $errmsg	Error message - set by libraries on error

In addition to the above global variables, the following variables are defined within the local scope of the main() function :-

int $argc	Argument count
array $argv	Argument vector

As these standard paramaters are defined at runtime as locals within the scope of the main function, defining them as arguments to main() is not only redundant, but also erroneous.

Maths Operators

ESL supports the majority of the C maths operations. A full list of the available maths operators is presented in figure 4.

```
+        Addition
-        Subtraction
*        Multiplication
/        Division
%        Modulus
+=       Addition to left hand variable
-=       Subraction from left hand variable
*=       Multiplication of left hand variable
/=       Division of left hand variable
++       Increment
--       Decrement
|        Bitwise OR
&        Bitwise AND
^        Bitwise exclusive OR
>>       Bitwise shift right
<<       Bitwise shift left
```

Figure 4 - Maths Operators

It should be noted that ESL follows the convention of C and allows increment and decrement operations to be defined as a statement or as a part of an expression. When used in an expression, such as a conditional or maths operation, both pre- and post- increment and decrement are allowed. That is, the following code fragments would have the expected results :-

```
$var ++;
while($var++ < 10)
$answer = ++$var / $var2 --;
```

ESL also supports the familiar C assignment operators, such as

```
+=
-=
*=
/=
```

Another maths related feature of ESL is the count operator (written as ‘#’). The count operator may be employed on a text, array or assoc value. When used with a text variable, it returns the number of characters in the string. When used with either of the array types, it returns the number of elements contained within the array. The count of a variable, $var, would be obtained using the following syntax :-

```
#$var
```

For example :-

```
int     $intVal;
text    $textVal = “Hello World”;

$intVal = #$textVal;
```

The above code would result in $intVal containing the value 11.

Signal Handling

Signal handling is available via a mechanism virtually identical to that used by C. During installation of ESL, the systems signal header file is interrogated and an ESL header is created in <signal.eh>. This header defines the standard signal names such as SIGCHLD and SIGHUP. It also defines the ESL version of the SIG_IGN and SIG_DFL macros. Delivery of signals to the ESL program is handled by a “catch-all” C function in the runtime module. This routine uses a mapping table to decide which ESL function is to be called in receipt of a particular signal. The mappings are maintained with the familiar signal() routine which mimics the C version. The only difference between the C and ESL signal routine is due to the lack of pointers, and in particular pointers to functions, within the ESL design. To overcome this missing feature, the name of the signal handling function is passed to signal() as a text object. It is resolved at runtime when the signal is delivered. An example of defining a signal handler is given in sample 3.

```
#include <signal.eh>

hup_handler($sig)
     int   $sig;
{
     printf(“Hit by a SIGHUP! (Signal = $sig)\n”);
}

main()
{
     signal(SIGHUP,”hup_handler”);
     signal(SIGINT, SIG_IGN);
}
```

Sample 3 - Signal Handling Example

As is the case in C, a signal can be ignored by defining its handler to be SIG_IGN. It can be reset to its default action by using SIG_DFL. Note that SIG_IGN and SIG_DFL are preprocessor macros and need not be wrapped in quotes as a text string. Another point to note is that signal handlers are called with one argument containing the number of the signal being delivered. All signal handlers must be written to receive a single integer parameter otherwise a runtime error regarding the parameter count or type will be reported when the signal is delivered.

Indirect Function Calls

The above mentioned lack of function pointers forces limitations in areas other than just signal handling. It is conventient, and in fact an efficient programming technique, to be able to call a function based on the contents of a particular variable. Providing a large *if* clause to cater for such a situation is not an elegant solution. To overcome the lack of function pointers in general code, ESL provides the *call()* function. As with the signal handling, the name of the function to execute is passed to the function as a text string. Using this mecahnism, an indirect call to the designated function will be made although no arguments may be passed.

File IO

ESL offers a file input/output library similar in concept to the standard IO library provided by C. Read access to IO sources is achieved through per-file buffers to improve performance while writes are not buffered in any way. Although conceptually similar to *stdio*, the ESL library allows many objects other than regular files to be accessed. Depending on the access mode used during the call to *open()*, the returned file descriptor may reference one of several types of underlying data sources. Details of the library routines are outlined below.

open() - Open an IO channel

Function :	open($path,$mode)	
Arguments:	text	$path (path to file or command)
	text	$mode (access mode)
Returns:	int	(file descriptor)
Notes :	-1 returned on error with $errmsg set.	

Possible modes are :-

“<“	Input from regular file
“>”	Output to regular file
“<>”	Input/Output from regular file
“<P”	Create and input from named pipe (FIFO)
“>P”	Create and output to named pipe (FIFO)
“<|”	Input from process
“>|”	Output to process

It should be noted that both the pipe related modes, i.e. “<P” and “>P”, create the pipe prior to accessing it. If the pipe exisits in the file system prior to the call, *open()* will fail.

read() - Read data from IO channel

Function :	read($fd, $numBytes)	
Arguments:	int	$fd (file descriptor)
	int	$numBytes (number of bytes to read)
Returns:	text	(read data)
Notes :	“undefined” returned on EOF	

readln() - Read data from IO channel until end-of-line

Function :	readlen($fd)	
Arguments:	int	$fd (file descriptor)
Returns:	text	(read data)
Notes :	“undefined” returned on EOF The new line character is not included in the returned string.	

readtok() - Read data from IO channel until token in input data

Function :	readtok($fd,$tok)	
Arguments:	int	$fd (file descriptor)
	char	$tok (token)
Returns:	text	(read data)
Notes :	“undefined” returned on EOF The token character is not included in the returned string.	

printf() - Formatted output to standard outout device

Function :	printf($fmt,$arg1.........)	
Arguments:	text	$fmt (format string)
	variable number of args	
Returns:	Nothing	
Notes :	Format string may include %s and %d, justification control and field width control (ie. no precision) eg. “%-10s”. All scalar variables may be enbedded in the format string, eg. “Result = $result\n”. Array and assoc elements may not be enbedded within the format string.	

fprintf() - Formatted output to IO channel

Function :	fprintf($fd,$fmt,$arg1......)	
Arguments:	int	$fd (file descriptor)
	text	$fmt
	variable number of args	
Returns:	Nothing	
Notes :	As per printf()	

close() - Close an IO channel

Function :	close($fd)	
Arguments:	int	$fd (file descriptor)
Returns:	Nothing	
Notes :		

System Calls

The system calls available to the ESL programmer map directly onto the underlying UNIX system calls. The only change is the methods used to pass and return data when the C conventions are not available.

alarm() - Schedule an alarm signal

Function :	alarm($time)	
Arguments:	int	$time(number of seconds)
Returns:	Nothing	
Notes :	A SIGALRM is sent to the process after the specified period. If a handler has not been defined for the signal prior to its delivery, the program will be killed.	

chdir() - Change current working directory

Function :	chdir($path)	
Arguments:	text	$path (path to new directory)
Returns:	int	(success status)
Notes :	-1 returned on error with $errmsg set	

chmod() - Change mode of file

Function :	chmod($path, $mode)	
Arguments:	text	$path (path to file)
	int	$mode (new file mode)
Returns:	int	(success status)
Notes :	-1 returned on error with $errmsg set	

exec() - Execute a file

Function :	exec($path, $args)	
Arguments:	text	$path (path of program)
	array	$args (argument vector)
Returns:	Only returns on error	
Notes :	-1 returned on error with $errmsg set	

exit() - Terminate a process

Function :	exit($status)	
Arguments:	int	$status (exit status)
Returns:	Never returns	
Notes :		

fork() - Create a new process

Function :	fork()	
Arguments:	None	
Returns:	int	(process ID)
Notes :	A PID of 0 is returned to the child process and the PID of the child is returned to the parent. -1 returned on error with $errmsg set.	

getenv() - Retrieve the value of an environment variable

Function :	getenv($variable)	
Arguments:	text	$variable (name of environment variable)
Returns:	int	(success status)
Notes :	-1 returned on error with $errmsg set.	

getpid() - Get process identification

Function :	getpid()	
Arguments:	None	
Returns:	int	(process ID)
Notes :		

getppid() - Get parent’s process identification

Function :	getppid()	
Arguments:	None	
Returns:	int	(parent’s process ID)
Notes :		

kill() - Send a signal to a process

Function :	kill($pid, $signal)	
Arguments:	int	$pid
	int	$signal (or a define from <signal.h>)
Returns:	int	(success status)
Notes :	-1 returned on error with $errmsg set.	

link() - Create a new link to a file

Function :	link($path, $linkName)	
Arguments:	text	$path (original file name)
	text	$linkName (name of new link)
Returns:	int	(success status)
Notes :	-1 returned on error with $errmsg set.	

mkdir() - Make a new directory

Function :	mkdir($path)	
Arguments:	text	$path
Returns:	int	(success status)
Notes :	-1 returned on error with $errmsg set. The file mode is set to 0777 but modified by the current umask()	

rename() - Rename a file

Function :	rename($oldName, $newName)	
Arguments:	text	$oldName (original file name)
	text	$newName (new file name)
Returns:	int	(success status)
Notes :	-1 returned on error with $errmsg set.	

rmdir() - Remove a directory

Function :	rmdir($path)	
Arguments:	text	$path (path to directory)
Returns:	int	(success status)
Notes :	-1 returned on error with $errmsg set.	

symlink() - Create a symbolic link to a file

Function :	symlink($path, $linkName)	
Arguments:	text	$path (original file name)
	text	$linkName (name of symbolic link)
Returns:	int	(success status)
Notes :	-1 returned on error with $errmsg set. If the symlink() system call is not available, -1 will be returned with an aoppropriate message in $errmsg.	

sleep() - Suspend process execution for interval

Function :	sleep($time)	
Arguments:	int	$time (number of seconds)
Returns:	Nothing	
Notes :	I know it's not a system call as such, but what the, hell :-)	

stat() - Gather information about a file

Function :	stat($path)	
Arguments:	text	$path
Returns:	array	(stat information)
Notes :	The following data is returned in the array :-	
	[0]	inode number
	[1]	number of links
	[2]	UID
	[3]	GID
	[4]	Size in bytes
	[5]	Access Time
	[6]	Modification Time
	[7]	Creation Time
	File mode details can be tested using the *test()* function	

system() - Execute a command in a sub shell

Function :	system($command)	
Arguments:	text	$command (command to pass to the shell)
Returns:	int	(exit code of the program)
Notes :		

truncate() - Truncate a file at a given length

Function :	truncate($path, $length)	
Arguments:	text	$path (path to desired file)
	int	$length (length in bytes at which to truncate)
Returns:	int	(success status)
Notes :	-1 returned on error with $errmsg set.	

umask() - Set file creation mode mask

Function :	umask($mask)	
Arguments:	int	$mask(file mode mask)
Returns:	int	(old mask)
Notes :		

unlink() - Remove a directory entry

Function :	unlink($path)	
Arguments:	text	$path (pathname to remove)
Returns:	int	(success status)
Notes :	-1 returned on error with $errmsg set	

wait() - wait for process to terminate

Function :	wait($pid)	
Arguments:	int	$pid (process ID of child process)
Returns:	int	(PID of dead child)
Notes :	If a process ID of -1 is used, then wait() will return after the death of any child. Otherwise it waits for the specified child. Note that the exit status is not available to the caller.	

SNMP Support

The interface to the SNMP library is similar in design to the file IO library. An SNMP session is created by calling snmpopen() with a nodename (or IP address) and a community string. A numeric value known as a session descriptor is returned. This descriptor is then used to identify a particular session. Many sessions to different nodes, or the same node with different community strings, can be active at the same time. snmpclose() is used to destroy the session when it is no longer required.

Prior to the first session being created, a call must be made to snmpinit(). This causes the MIB to be parsed and a couple of other internal functions to be executed. Once a session has been created, the snmpget() and snmpgetnext() routines can be used to retrieve data from the remote node. Both of these functions accept an array of object identifiers in either numeric or textual format as an argument. This enables multiple requests to be formatted into a single packet. Both routines return an array containing the retrieved data. Each element in the returned array contains data from the object identifier in the corresponding element of the source array. Each element of data returned contains the textual name of the object, followed by a colon, followed by a textual representation of the data retrieved. This allows multiple calls to snmpgetnext() to traverse a table while checking the name of the identifier returned to determine when the end of the table has been reached.

snmpinit() - Initialise the SNMP library

Function :	snmpinit()	
Arguments:	None)	
Returns:	Nothing	
Notes :	Forces a parse of the MIB and other setup functions	

snmpopen() - Create SNMP session

Function :	snmpopen($node, $community)	
Arguments:	text	$node (remote nodename or IP address)
	text	$community (SNMP community string)
Returns:	int	(session descriptor)
Notes :	-1 returned on error with $errmsg set.	

snmpclose() - Close SNMP session

Function :	snmpclose($sd)	
Arguments:	int	$sd (session descriptor)
Returns:	Nothing	
Notes :		

snmpget() - Perform SNMP Get operation

Function :	snmpget($sd, $vars)	
Arguments:	int	$sd (session descriptor)
	array	$vars (SNMP variables / object ID’s)
Returns:	array	(retrieved data)
Notes :	“undefined” returned on error with $errmsg set	

snmpgetnext() - Perform SNMP GetNext operation

Function :	snmpgetnext($sd, $vars)	
Arguments:	int	$sd (session descriptor)
	array	$vars (SNMP variables / object ID’s)
Returns:	array	(retrieved data)
Notes :	“undefined” returned on error with $errmsg set	

snmptrapopen()- Create a Trap session descriptor

Function :	snmptrapopen()	
Arguments:	None	
Returns:	int	(session descriptor)
Notes :	-1 returned on error. Any SNMP trap received will be available for reading from this descriptor.	

snmptrapread() - Read an trap packet

Function :	snmptrapread($sd)	
Arguments:	int	$sd (session descriptor)
Returns:	array	(retrieved data) [0] = IP Address of originator [1] = Trap identifier [2] onwards = trap variables
Notes :	“undefined” returned on error with $errmsg set	

Misc. ESL Functions

binddb() - Bind an associative array to a database

Function :	binddb(&assoc, $path, $mode)	
Arguments:	assoc	&assoc (reference to assoc array)
	text	$path (path for basename of database files)
	text	$mode (access mode)
Returns:	int	(success status)
Notes :	-1 returned on error with $errmsg set. Binding will fail if the assoc contains in-core data. Database files will be created, if required, with a file mode of 0777 modified by the umask.	

defined() - Check status of a variable

Function :	defined(&var)	
Arguments:	any	&var (reference to any type variable)
Returns:	int	(defined status)
Notes :	1 returned if variable has a defined value. Elements of both arrays and assoc's can be tested by passing a reference to the indexed array (e.g. &assoc{"foo"} or &array[1])	

existsdb() - Determine if a database exists

Function :	existsdb($path)	
Arguments:	text	$path (path to basename of database files)
Returns:	int	(success status)
Notes :	0 returned if both *.dir* and *.pag* files exist	

keys() - Build list of associative array keys

Function :	keys(&assoc)	
Arguments:	assoc	&assoc (reference to assoc)
Returns:	array	(keys currently in use by assoc)
Notes :	The order in which keys are found is apparently random. This is due to the hashing used by both the in-code and bound assoc's	

signal() - Define signal handler

Function :	signal($sig, $handler)	
Arguments:	int	$sig (signal number)
	text	$handler (name of handler function)
Returns:	text	(name of old handler)
Notes :		

split() - Split text string into sub-strings

Function :	split($str, $tok)	
Arguments:	text	$str (original text string)
	char	$tok (delimiting token)
Returns:	array	(each element is a sub-string of $str)
Notes :	The token is not included in the sub-string	

substr() - egrep styled string splitter

Function :	substr($str, $regexp, $extPat)	
Arguments:	text	$str (original text string)
	text	$regexp (Regular expression to match)
	text	$extPat (Extraction Pattern)
Returns:	text	(extracted sub-sting)
Notes :	substr allows you to extract parts of a string using an egrep styled regexp. Parts of the regexp enclosed in brackets can be extracted using an index in the extPat. eg. substr(“99 green bottles”, ”([0-9]*) [a-z]* ([a-z]*)”, “$1 $2”); would return “99 bottles”.	

test() - Shell-like test function

Function :	text($test, $path)	
Arguments:	char	$test (test to be conducted)
	text	$path (path of file to test)
Returns:	int	(success status)
Notes :	1 returned on success.	
	Available tests are :-	

‘b’	: File is block mode device
‘c’	: File is character mode device
‘d’	: File is a directory
‘p’	: File is a named pipe (FIFO)
‘s’	: File is a non-empty regular file
‘f’	: File is a regular file
‘u’	: File has setuid bit set
‘g’	: File has setgid bit set

unbinddb() - Remove a database binding

Function :	unbinddb(&assoc)	
Arguments:	assoc	&assoc (reference to assoc)
Returns:	Nothing	
Notes :		

undef() - Destroy the value of a variable

Function :	undef(&var)	
Arguments:	any	&var (reference to any variable)
Returns:	Nothing	
Notes :	This can be used to remove (undefine) entire entries from assoc’s, including bound assoc’s	

Mini SQL

Like ESL, Mini SQL (or mSQL) is another development that has come from the Minerva Network Management Environment. Because they are so closely related, support has been included within ESL for communicating with the mSQL database engine. Seeing that mSQL is a client server based database engine supporting a subset of ANSI SQL, allowing ESL scripts to utilise the database provides great flexibility and power in the handling of data. In simple cases, a script can utilise a bound assoc to provide keyed data access. As the complexity of scripts and their data increases, the mSQL interface can be used to provide a more powerful mechanism. More information on mSQL is available in “Mini SQL : A Lightweight Database Engine”.

msqlConnect() - Connect to the mSQL server

Function :	msqlConnect()
Arguments:	None
Returns:	-1 on error with $errmsg set
Notes :	If the database resides on a host other than the localhost, set the MSQL_HOST environment variable to the hostname or IP address of the machine running the server.

msqlClose() - Close the connection to the mSQL server

Function :	msqlClose()
Arguments:	None)
Returns:	Nothing
Notes :	

msqlInitDB() - Select a database to access

Function :	msqlInitDB($dbName)	
Arguments:	text	$dbName (name of desired database)
Returns:	-1 on error with $errmsg set	
Notes :		

msqlQuery() - Submit a query to the server

Function :	msqlQuery($query)	
Arguments:	text	$query (mSQL query)
Returns:	-1 on error with $errmsg set	
Notes :		

msqlFetchData() - Retrieve data generated by query

Function :	msqlFetchData()
Arguments:	None
Returns:	Array of data if available, otherwise the undefined value is returned.
Notes :	The elements of the array match the order of the fields requested in the mSQL select statement.

msqlListDBs() - List the available databases

Function :	msqlListDBs()
Arguments:	None
Returns:	Array of database names, undefined if no databases found.
Notes :	

msqlListTables() - List the tables of a database

Function :	msqlListTablesd()
Arguments:	None
Returns:	Array of table names, undefined if no tables found.
Notes :	This function uses the currently selected database as specified in a call to msqlInitDB()

msqlInitFieldList() - Build the list of fields for the requested table

Function :	msqlInitFieldList($table))
Arguments:	text $table (desired table in current database)
Returns:	Nothing
Notes :	The list is retrieved using the msqlListFields function. This is similar in manner to submitting a query with msqlQuery() and then accessing the data with msqlFetchData().

msqlListFields() - Access the list of field definitions

Function :	msqlListFields()
Arguments:	None
Returns:	Array of field information, undefined if no data is available.
Notes :	The details for a single field are returned within the array during each invocation of this function. The data is returned using :- [0] Field Name [1] Field Type [2] Field Length msql.h defines the following field types #define INT_TYPE1 #define CHAR_TYPE2 #define REAL_TYPE3

UNIX Data Files

The standard ESL library provides an interface to two of the most commonly accessed data-files provided by UNIX, being that passwd file and the hosts table. The interface provided mimics that used by the C library:

gethostbyaddr() - Retrieve a host's detail by address

Function :	gethostbyaddr($IPAddr)	
Arguments:	text	$IPAddr (IP Address of host)
Returns:	array of host details, undefined if host not found	
Notes :	The data is presented in the array as follows:-	

[0] Official host name
[1] Primary IP Address

gethostbyname() - Retrieve a host's detail by name

Function :	gethostbyname($hostname)	
Arguments:	text	$hostname (name of the host)
Returns:	array of host details, undefined if host not found	
Notes :	The data is presented in the array as follows:-	

[0] Official host name
[1] Primary IP Address

getpwnam() - Retrieve a user's passwd file details

Function :	getpwnam($uname)	
Arguments:	text	$uname (username)
Returns:	array of user's details, undefined if host not found	
Notes :	The data is presented in the array as follows:-	

[0] Username
[1] Encrypted Password
[2] UID
[3] GID
[4] Full name or GECOS field
[5] Home directory
[6] Shell

getpwuid() - Retrieve a user's passwd file details by UID

Function :	getpwuid($uid)
Arguments:	int $uid (user ID)
Returns:	array of user's details, undefined if host not found
Notes :	The data is presented in the array as follows:-

[0] Username
[1] Encrypted Password
[2] UID
[3] GID
[4] Full name or GECOS field
[5] Home directory
[6] Shell

Extending ESL

One of the design criteria for ESL was that it offered a flexible mechanism for extending the capabilities of the language. This has been provided using a simple table to map from ESL function names to external C functions and by providing mechanisms for passing data from and returning data to ESL. External function definitions are contained in the file extern.h. An example of the data provided in that file is given below in figure 5.

```
{ "unlink", eslUnlink, 1, {TEXT,0} },
{ "mkdir", eslMkdir, 1, {TEXT,0} },
{ "chdir", eslChdir, 1, {TEXT,0} },
{ "exit", eslExit, 1, {INT, 0}},
{ "printf", eslPrintf, -1, { TEXT, 0} },
{ "umask", eslUmask, 1, {INT, 0}},
{ "sleep", eslSleep, 1, {INT, 0}},
{ "alarm", eslAlarm, 1, {INT, 0}},
{ "fork", eslFork, 0, {0}},
{ "call", eslCall, 1, {TEXT,0}},
{ "exec", eslExec, 2, {TEXT, ARRAY,0}},
{ "system", eslSystem, 1, {TEXT,0}},
```

Figure 5 - C Function Mappings

As you can see, the mappings are proded by a C table that provides the name of the ESL function, a pointer to the C function, the number of function parameters and the types of the parameters. This allows the strongly typed nature of ESL to encompass external functions. Two things about the table should be noted. Firstly, the list of parameter types is terminated with a zero. Secondly, if the function has a variable number of paramters, such as the definition for printf(), the number of parameters is given as -1. In this case, the types of the parameters are checked where possible and the rest are passed unchecked to the function. In the case of printf(), the only type defined is that of the first parameter. In this case, the printf() format string must be a TEXT parameter ,while any other parameters provided can be of any type.

The external C function is passed the ESL parameters as a linked list of plist_t struct's as defined in esl.h. The structure, as depiced in figure 6, contains a pointer to the data, the type of the data. and a pointer to the next parameter. As would be expected with a linked list, the last parameters is indicated by a plist_t struct with a NULL next pointer. Although the type is provided, in most cases it isn't used by the external code. If the full number of parameters is defined in the mapping table, ESL will ensure that the correct number and type of parameters are used. The type field is provided for functions that offer a variable number of parameters, such as printf(), that can't be checked by ESL at runtime.

To allow an external function to return data to the calling ESL code, the runtime module provides a function called setReturnValue(). This function places the return data in a known location so that it will be pushed onto the runtime stack of ESL's simulator on return prior to returning from the external call. setRetunValue() requires two parameters : a pointer to an allocated data item, and the type of the data item. The data must be contained ina malloc()'ed space as it is free()'ed by the simulator as it comes off the stack. The undefined value is available by returning NULL as the data element.

```
/*
** External parameter list
*/

typedef struct plist_s {
        char    *data;
        int     type;
        struct  plist_s *next;
} plist_t;
```

Figure 6 - External Parameter Structure

Some external functions may be required to return an array to the calling ESL code. This is catered for by providing a set of functions that creates and manipulates ESL arrays to the C code. The two functions both operate on a symbol table element, a typedef'ed struct called symtab, that is defined in esl.h. The first step is for the external function to create an empty array. This is done using createArray(). The function takes no parameters and returns a pointer to a newly created symtab entry. From then on, the external function can add data to the array by using addArrayElement(array, data, index), where array is the *symtab entry, data is a cahr pointer to a text string, and index is an integer identifying which element of the array is being defined. The text data need not be in dynamic memory as it is duplicated by addArrayEntry() prior to adding it to the array. An example of using these functions is provided in figure 7.

NOTE : The mechanism for providing the ESL to C mappings will change in version 0.3 to enable the mappings to be updated at runtime. This, coupled with a couple of other cosmetic changes, will allow ESL to be embedded into any other application that may require scripting facilities. The application will be able to define the functions for accessing its features at runtime before passing control to ESL. A similar table structure for the mappings will be used, although it will not be a static object.

```
/*
** An external function implementing stat() for ESL scripts
*/

void eslStat(params)
     plist_t *params;
{
     struct stat buf;
     char  *path,
           txtbuf[80];
     symtab*array;
     int   index;

     /* Grab the path as passed to us from ESL */
     path = params->data;

     /* Stat the file and return "undefined" on error */
     if (stat(path,&buf) < 0)
     {
           setReturnValue(NULL,ARRAY);
           return;
     }

     /* Create and fill the array */
     index = 0;
     array = createArray();
     sprintf(txtbuf,"%d",buf.st_ino);
     addArrayElement(array,txtbuf,index++);
     sprintf(txtbuf,"%d",buf.st_nlink);
     addArrayElement(array,txtbuf,index++);
     sprintf(txtbuf,"%d",buf.st_uid);
     addArrayElement(array,txtbuf,index++);
     sprintf(txtbuf,"%d",buf.st_gid);
     addArrayElement(array,txtbuf,index++);
     sprintf(txtbuf,"%d",buf.st_size);
     addArrayElement(array,txtbuf,index++);
     sprintf(txtbuf,"%d",buf.st_atime);
     addArrayElement(array,txtbuf,index++);
     sprintf(txtbuf,"%d",buf.st_mtime);
     addArrayElement(array,txtbuf,index++);
     sprintf(txtbuf,"%d",buf.st_ctime);
     addArrayElement(array,txtbuf,index++);
     setReturnValue(array,ARRAY);
}
```

Figure 7 - An External C function Implementation of stat()

List of Figures

Index

T

U

V

W