Sleep Language Fundamentals

This document covers the fundamentals of the sleep language in an application neutral way. The format of this documentation is a mix between a tutorial and a reference. You should be able to read straight through this documentation to gain an initial understanding of sleep and later use it as a reference.

Sections

Scalar Variables

Variables in sleep are temporary locations in memory to store a value. Variables in sleep are called scalars. Scalars can be strings, numbers, or even a reference to an object.

Scalar variable names always begin with the $ dollar sign.

Assignment

Values are assigned to scalars using the = sign. For example

```
$x = 3;
$y = $x;
```

The above puts the integer 3 into the scalar *$x*. *$x* now contains the integer 3 until something else is assigned to it. Likewise the value of *$x* is copied into *$y*. *$y* now contains the value 3.

Operations

You can also assign the result of an operation to a scalar. An operation consists of two values and an operator. The following are examples of valid operations.

```
$x = 5 + 1;
$x = 5 - $y;
$x = $x * 2;
$x = $z / $2;
$x = $1 % 3;  # modulus
$x = $1 ** 4; # exponentation operator
```

Those operations only work on numbers though. Sleep operations for strings include:

```
$x = "Ice" . "cream";
```

The above is an example of the concatenation operator. The . period combines two strings together. *$x* now has the string value "Icecream".

```
$x = "abc" x 3;
```

The above is an example of the string multiplication operator. The resulting string is the left hand side repeated by the number on the right hand side. In the above example the value of *$x* would be "abcabcabc". Basically "abc" repeated 3 times.

Sleep also includes a set of logical operators. These include <<, >>, ^, |, &, and a function *&not()*.

Expressions

Multiple operations can be combined to form an expression. For example

```
$z = 5 + 1 * 3;
```

Is valid. The +, -, and . operators have a lower precedence than *, /, and %. So the above would be equivalent to:

```
$z = 5 + (1 * 3);
```

You can use ( ) parentheses to enclose an operation that you want to have evaluated first. Complex expressions are allowed as well:

```
$z = 3 * ($x % 3) - (ticks() / (10000 + 1));
```

Use of White Space

Sleep scripts require you to use white space in expressions. Most languages allow you to get away with very little white space. For example in Perl the following would be valid.

```
$x=1+2;
```

The above is not valid in sleep. You are always expected to use white space between operators and the stuff they are operating on. For example:

```
$x = 1 + 2;
```

Is valid. Think of it as a feature that forces some reasonable coding habits.

Scalar Types

Scalars can hold several types of data. The most common ones you will deal with are numbers and strings.

Any scalar value can be treated as a string... period. The scalar value 3 is equivalent to the string "3". Strings are covered more in their own section: String Manipulation.

Sleep supports three types of scalars for numbers. Rational numbers such as 3, 4, 5 ... 65536 are all integer values. Numbers that have a decimal in them such as 3.0, 1.1, 0.55556 are all double values. A double value is assigned to a scalar as follows:

```
$Pi = 3.1415926535;
```

Rational numbers can be specified as hex literals. Any number beginning with a 0x is turned into an integer scalar. i.e.

```
$var = 0xFF; # same as $var = 255
```

Octal literals can be used as well. Any number beginning with a 0 is interpreted to be an octal literal. i.e.

```
$oct = 077; # same as $var = 63
```

Another numerical type that sleep supports is called a long. A long is basically a higher capacity integer. An integer can be a number between -2,147,483,648 to +2,147,483,648. A long can be much bigger than that.

Longs can be explicitly declared by adding an L to the end of the number i.e.: `12345L` would be a long scalar. Adding an L to the end of the number works for coercing hex and octal literals to longs as well.

Numerical scalars are covered more in their own section: Number Crunching.

Other Scalar Types

Two other scalar types you may encounter are the null scalar and object scalars.

The null scalar is a scalar that is equivalent to nothing. It is a string with no characters, the number 0, and does not reference anything. The null scalar has a built-in name *$null*. *$null* always refers to the null scalar.

Object scalars are created by various functions. For example if you use the open() function to open a file, the open function will return an object scalar that contains a reference to the internal information needed by the I/O system. Object values have a numerical value that is equivalent to their hash code (a number java calculates for each object). Object values have a string value as well.

How Scalars are Passed

It is important to understand the concept of pass by reference and pass by value. Sleep string scalars and number scalars are always passed by value. Pass by value means that when a scalar *$x* is assigned to scalar *$y*, a copy of the value of *$x* is made and then assigned to *$y*.

```
$x = 3;
$y = $x;
$x = 4;
```

In the above scenario *$x* is initially 3. Then *$y* is assigned a copy of *$x* which is 3. *$y* now has the value 3. In the last line *$x* is assigned the value 4. This does not affect *$y*, since *$y* only has a copy of the contents of *$x*. Not the actual value.

Data structures (arrays, hashes) and object scalars are passed by reference. All data has a place in memory somewhere. A reference is a variable containing the address of some type of data in memory. If two scalars have the same reference, then any changes to the data either scalar is referring two will affect both scalars. This happens because the two scalars reference the same data. An example:

```
@foo[0] = "Ice Cream";
@bar = @foo;
@bar[0] = "Gelato";

# what is the value of @foo[0]?
```

Getting a little ahead of ourselves but any variable with an @ is an array. The first line of the above example assigns "Ice Cream" to position 0 in the array *@foo*. The second line assigns the array *@foo* to *@bar*. Arrays are big objects so it is not efficient to copy the array. Rather the reference to the array data is copied. *@bar* now references the same array data as *@foo*. The string "Gelato" is assigned to position 0 in the array *@foo*. Since *@foo* and *@bar* are the same array in memory the value of *@foo[0]* is "Gelato".

As a side note: assignment of individual array elements with string or number values works just like assigning individual scalars. In fact individual array elements are for all intents and purposes normal scalars.

Scalar Scope

Sleep has two types of scope. A global scope and a local scope. Scope is an attribute of variables that defines where they exist and can be referenced from. A variable that has global scope can be accessed any where in the script. Once created the scalar does not go away unless explicitly told to do so.

Sleep scalars have a global scope by default.

A variable that has local scope is considered to be local to a specific function. The variable is alive while the function is being executed. Once the function exits the variable goes away. Local variables have higher precedence than global variables. Local variables do not affect global variables.

```
sub verdict
{
   local('$decision');
   $decision = "not guilty";
}

$decision = "guilty";
verdict();

# what is the value of $decision?
```

In the above example the subroutine verdict is declared. Sleep has several types of functions, a subroutine is just one type of function. *$decision* is then assigned the value "guilty". The subroutine example is then called. Inside of verdict *$decision* is declared as a local variable for verdict. *$decision* is then assigned the value "not guilty". The subroutine verdict is now done. At the end of the day what is the value of *$decision*? The answer is "guilty".

When *$decision* was declared as a local variable within verdict, a new local variable *$decision* was created with no value. This local variable has nothing to do with the global variable *$decision*. The local variable *$decision* was then assigned the value "not guilty". When the subroutine verdict finished executing the local variable *$decision* went away. The global variable *$decision* was not touched.

If-Else Statements

If-Else statements let you compare different values and execute a certain part of the script based on the result of the comparison.

The basic format of an if-else statement is:

```
if (v1 operator v2) { commands }
else if (v3 operator v4) { commands }
else { commands }
```

The ( ) parentheses enclose the comparisons while the { } brackets enclose the commands to execute. If the comparison within the ( ) parentheses is true then the following commands are executed. If the initial comparison is false then the else statement is executed. Else statements can be followed by either a block of statements enclosed in { } brackets or by another if statement.

The Operators

Numerical Comparisons

Operator	Description
==	equal to
!=	not equal to

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

String Comparisons

Operator	Description
eq	equal to
ne	not equal to
lt	string v1 is less than string v2
gt	string v1 is greater than string v2
isin	substring v1 is contained in string v2
iswm	wildcard string v1 matches string v2 (see Appendix B)

Other Comparisons

Operator	Description
is	true if scalar object reference v1 equals object reference v2

Most comparisons in sleep can be negated with the ! exclamation point before the operator in the comparison.

```
if ("test" !isin "walrus")
{
   # the string "test" is not contained in the string "walrus"
}
```

Combining Comparisons

Comparisons can be combined using the logical operators && for AND and || for OR. For example:

```
sub check
{
   if (($1 > 0) && ($1 <= 10))
   {
      # $1 is greater than 0 and $1 is less than or equal to 10.
   }
}
```

The above subroutine checks if the parameter *$1* is between 1 and 10. The first comparison makes sure *$1* is greater than 0. If *$1* is less than 0 then the comparison stops immediately. For the condition in the if statement to be true *$1* has to be greater than 0 AND *$1* has to be less than or equal to 10.

Unary Operators

Many of the comparison operators have a left and a right parameter. Sleep also has unary comparison operators. Meaning the comparison operator has only one parameter. Unary operators are recognizable as they always begin with the - dash character.

Unary operators can be negated by putting an ! exclamation point before the - in the operator.

Operator	Description
-isarray	is the specified value a scalar array
-ishash	is the specified value a scalar hash
-isletter	is the specified character a letter
-isnumber	is the specified string a number
-istrue	is the specified value non-null

```
if (-isnumber "3")
{
   # true because 3 is a number
}
```

The above checks if 3 is a number. Unary operators can be combined with normal comparison operators using && AND and || OR.

It is also possible to use a scalar by itself as the comparison for an if statement. A scalar that has any value other than the null scalar will result in the comparison being true.

```
if ($scalar)
{
```

```
   # execute me
}
```

The above is the same as doing a comparison using the -istrue operator with *$scalar* as the parameter.

The Conditional Operator

Sleep has a conditional operator iff. iff takes a condition as it's first parameter and returns it's second parameter if and only if the condition is true. The third parameter is returned if and only if the condition is false.

```
iff(comparison, expression_if_true, expression_if_false)
```

An example:

```
$value = "Calculation took" . $x . " second" . iff($x > 1, "s", "");
```

$value would be "Calculation took 2 seconds" if $x is equal to 2. If *$x* was equal to 0 then *$value* would be "Calculation took 1 second". The iff example above checks if *$x* is greater than 1. If it is then "s" is returned. Otherwise "" is returned.

Loops

While Loops

While loops are a way of executing certain statements while a comparison continues to be true.

```
while (comparison) { commands }
```

An example of a while loop that counts to 100:

```
$x = 0;
while ($x < 100)
{
   $x = $x + 1;
}
```

For Loops

For loops are the same as while loops except they have a few syntax differences. The following for loop is equivalent to the code in the above example.

```
for ($x = 0; $x < 100; $x = $x + 1) { }
```

The Break Command

Sometimes you may want to end a loop before it has completed. An easy way to do this is with the break command.

An example:

```
$x = 0;
while ($x < 100)
{
   if ($x == 50)
   {
      break;
   }

   $x = $x + 1;
}
```

The loop example above is setup to count to 100. Once the count reaches 50 though the loop is broken out of. The break statement is a way of saying "this is the end of the loop, right here".

Number Crunching

The Scalars -> Expressions section introduced sleep's family of numerical scalars.

Strings used in numerical expressions are automatically converted to an integer scalar.

When performing an operation on scalar numbers of two different types a conversion occurs. If either of the scalar numbers is a double, then the other scalar is converted to a double and the operation occurs. If either of the scalar numbers is a long, then the other scalar is converted to a long and the operation occurs. Integer scalars have the lowest priority.

It is possible to force a scalar to be a certain type of scalar number. You can use the scalar casting functions. These functions are:

Function	Description
double(*$x*)	returns the value of *$x* as a double scalar
long(*$x*)	returns the value of *$x* as a long scalar
int(*$x*)	returns the value of *$x* as an int scalar

Increment and Decrement Operators

Sleep has a special operator for integer scalars. Instead of typing

```
$x = $x + 1;
```

You can instead use the increment operator on the scalar *$x*.

```
$x++;
```

The two are equivalent. `$x++` increments the value of *$x* and returns `$x + 1`. The decrement operator does the same thing except for subtraction. `$x--` decrements the value of *$x* and returns `$x - 1`.

`$x--` is equivalent to `$x = $x - 1;`

Subroutines

Subroutines in sleep are basically mini programs. You can pass arguments to them and they can return values.

```
sub add
{
   return $1 + $2;
}

$x = add(3, 4);
```

The example above is an add subroutine. *$x* will receive the result of the add subroutine. Which is 7. The 3 and 4 separated by a comma are arguments to the subroutine. Arguments are separated by commas Arguments to a subroutine in sleep are numbered beginning with *$1*. So the first argument would be *$1*, the second argument *$2*, the third argument *$3* etc. All subroutines are placed in the global scope.

The special array @_ contains all of the arguments for a subroutine as well. *@_[0]* is *$1*, *@_[1]* is *$2*, etc.

The Return Command

The return command causes the current subroutine to stop executing and return to where it started. You can specify a scalar with return that will be the value of the function call.

Arrays and Hashes as Arguments

Arrays and Hashes can be passed to subroutines as arguments.

```
sub clearArray
{
   clear($1);
}

clearArray(@data);
```

In the above example we declare the subroutine clearArray. All arguments to a subroutine are accessed as $1, $2, ... $n regardless of the type of argument. In the case of clearArray, $1 is the passed in scalar array.

If you pass an *@array* to a function where a *$scalar* is expected then the *@array* will be converted to a scalar string representing the array. All of the information in this section applies for %hashes as well.

Recursion

Sleep subroutines allow for recursion. A recursive function is a function that calls itself. An example of a recursive subroutine is factorial:

```
sub fact
{
   if ($1 == 0)
   {
      return 1;
```

```
    }

    return $1 * fact($1 - 1)
}
```

Nesting Subroutines

Everything in sleep is evaluated in a (semi) straight forward manner. When sleep executes a statement that consists of the keyword sub, followed by a name, and then a block of statements it puts the subroutine into the environment. It is possible to nest subroutines within eachother or declare a different version of the same subroutine based on a conditional.

```
if ($favorite eq "red")
{
    sub favoriteColor
    {
        return "my favorite color is red";
    }
}
else
{
    sub favoriteColor
    {
        return "I don't know what my favorite color is";
    }
}
```

If *$favorite* is equal to "red "initially then the subroutine favoriteColor will be declared as a subroutine that returns "my favorite color is red". Otherwise the subroutine favoriteColor will be declared as a subroutine that returns "I don't know what my favorite color is".

Arrays

An array is a variable that holds a bunch of variables. Arrays reference variables in numerical order. So each variable in the array has an index (or position). Arrays in sleep always begin with the @ at symbol. This goes for referencing array elements individually and referencing the array as a whole.

```
$x = 3;
@foo[0] = "Raphael";
@foo[1] = 42.5;
@foo[2] = "Donatello";
@foo[$x] = "Michelangelo";
```

The above sets index 0 .. 2 of *@foo* to various values. *$x* is evaluated in the last line and *@foo[3]* is set to "Michelangelo". *@foo[3]* is now equivalent to a normal string scalar that has the value "Michelangelo".

Arrays can be assigned to each other as well. As stated in the Scalars section assigning an array to another array just copies the reference. Both *@array*'s will point to the same data. A change in one array will affect the other array.

The function `size(@array)` takes an array as a parameter and returns the total number of elements in the array.

To remove an element from an array use the remove function i.e. `remove(@array, $scalar)`

Many arrays returned by built-in functions may be read only. As such code that changes the array will have no effect on it.

Foreach Loops

Sleep provides a construct for easily looping through all of the elements in an array. This is the foreach loop.

```
foreach $scalar (@array)
{
    commands
}
```

The foreach loop above iterates through each element in *@array*. For each element in *@array*, the element value is set to *$scalar*. Accessing *$scalar* is the same as accessing the specific element in the array.

Specifying a *%hash* in place of an *@array* will cause the foreach loop to iterate through the keys of *%hash*. Foreach also has a second syntax:

```
foreach $index => $value (source)
{
    commands
}
```

The foreach loop above will loop through the source (either an *@array* or a *%hash*). The index of the source will be assigned to the scalar on the left. The value of the source will be assigned to the scalar on the right.

Within a foreach loop functions that modify the structure of *@array* are not allowed. This includes add(), remove(), removeAt(), push(), pop() etc. An attempt to modify the structure of @array during a foreach loops execution will result in a script warning.

Arrays as Stacks

A stack is a data structure that has a push and pop operation. Push puts data on top of the stack. Pop takes data off the top of the stack. Sleep arrays can be used as stacks.

```
push(@array, "some data");

$value = pop(@array);
```

In the first line the string "some data" is pushed on top of the array *@array*. In the second line "some data" is popped off of the stack and returned by the function pop(). *$value* now contains the value "some data".

Tuple Assignment

To quickly assign elements of an array to scalars specified in a tuple use:

```
($x, $y, $z) = @array;
```

In the above example, *$x* will be set to the first element in *@array*, *$y* will be set to the second element, and *$z* will be set to the third element.

If there are not enough values in *@array* for all of the specified scalars then the remaining scalars will be set to *$null*.

If the value on the right hand side of the = assignment is not an array, then the above will act exactly as if *$x*, *$y*, and *$z* had been assigned to the specified value individually.

Sorting

Sleep includes functions for sorting arrays. You can sort arrays numerically (double or int variety), or alphabetically. To sort an array in sleep:

```
@sorted = sorta(@array);
```

The above sorts the array *@array* alphabetically. To sort an array of integer scalars use `sortn(@array)`, to sort an array of double scalars use `sortd(@array)`.

You can also define your own custom sorting criteria using the sort() function.

```
sub mysort
{
   if ($1 lt $2) { return -1; }
   if ($1 eq $2) { return 0; }
   if ($1 gt $2) { return 1; }
}

@sorted = sort(&mysort, @array);
```

The above is equivalent to the first sorting example. The code snippet sorts @array alphabetically. The subroutine mysort is the custom sorting criteria. All the sorting decisions are made by the subrouting mysort. To define your own mysort simply make it return a value less than 0 if $1 is less than $2. If $1 is equal to $2 make your mysort return 0. If $1 is greater than $2 then make your mysort return a value greater than 0.

The function sort takes two arguments. A function handle and an array. A function handle is a way of passing a subroutine as an argument in sleep. To specify a function handle simply append the subroutine name to the & ampersand symbol.

Sorting Operators

Sleep defines two operators to assist with sorting.

Operator	Description
$a cmp $b	returns a value > 0 if $a is greater than $b, a value < 0 if $a is less than $b, and 0 if $a is equal to $b. The cmp op performs a string comparison.
$a <=> $b	returns a value > 0 if $a > $b, a value < 0 if $a < $b, and 0 if $a == $b. The <=> op performs a numerical comparison.

These functions are useful if you would like to easily implement your own sorting function. For example to implement a case insensitive sort:

```
@array = array("Jill", "Jack", "BoB", "iReNE", "aDaWG");

sub my_sort
{
   return lc($1) cmp lc($2);
```

```
}

sort(&my_sort, @array);

printAll(@array);
```

Multidimensional Arrays

Sleep arrays are just arrays of scalars. It is possible for one of the elements of a scalar array to contain another scalar array (or hash for that matter). The following sets up a multidimensional array:

```
@data = array(
          array("a", "b", "c"),
          array(1, 2, 3, 4),
          array('.', '!', '#', '*')
       );
```

In the example above @data is an array of arrays. To access an individual element of @data:

```
$temp = @data[2][3]; # $temp is now '*'
```

The array from the first example could have also been setup with the following code:

```
@data[0][0] = "a";
@data[0][1] = "b";
@data[0][2] = "c";
@data[1][0] = 1;
@data[1][1] = 2;
@data[1][2] = 3;
@data[1][3] = 4;
@data[2][0] = '.';
@data[2][1] = '!';
@data[2][2] = '#';
@data[2][3] = '*';
```

There is no limit to how many levels deep your arrays/data structures can go. Levels do not need to be the same size.

Hashes

Hashes are a special scalar type that can hold a bunch of variables as well. Unlike arrays hashes reference variables with a key. Each variable in the hash has a unique key associated with it. Hashes in sleep always begin with the % at symbol. This goes for referencing hash elements individually and referencing the hash as a whole.

```
$x = 3;
%foo["name"] = "Raphael";
%foo["job"]  = "wasting time";
%foo[$x]     = "Michelangelo";
```

The above associates the key "name" with "Raphael" inside of %foo. The key "job" is associated with "wasting time", and the key 3 is associated with "Michelangelo". It is worth noting that the keys "3" and 3 are equivalent. Keys to sleep hashes are always treated as strings.

Hashes can be assigned to each other as well. Similar to assigning an array to another array, assigning a hash to another hash just copies the reference. Both %hashe's will point to the same data. A change in one hash will affect the other hash.

Many hashes returned by built-in functions may be read only. As such code that changes the hash will have no effect on it.

To obtain an @array of all of the keys in a hash you can use the keys() function with the %hash as the parameter. To remove a key from a hash you can use remove(%hash, "key").

The Key/Value Operator

Hashes can quickly be created using the `hash()` function. The hash() function takes a special kind of parameter known as a key/value pair:

```
%hash = hash(key1 => "this is a value", key2 => 3 * (9 % 7));
```

The => is the key/value pair operator. Any time a key/value pair is required as a parameter it can be specified with => or as a string separating the key/value by an equal sign i.e. 'key=value'. The => operator is special because the left hand side (key) is taken as-is with no evaluation by the sleep language.

Multidimensional Hashes/Arrays

Multidimensional hashes work exactly the same as Sleep arrays. It is also possible to have an array of hashes, or a hash of arrays. Sleep even allows these two data structures to be mixed and matched as you please.

```
%hash = hash(letters => array("a", "b", "c", "d"),
             names   => hash(
                          rsm => "Raphael Mudge",
                          fvm => "Frances Mudge")
            );
```

When a script tries to index to a level that is deeper than has been indexed prior Sleep will create a new hash/array as appropriate. Sleep knows which new structure to create based on the variable at the beginning of the index chain.

The indexing operator can be applied to function calls, expressions wrapped in parentheses, and $scalars. When the indexing operator is called on these items, Sleep will assume that there is either a hash or array data structure being referenced. If a script attempts to index to a new level from this context then no new structure will be created and $null will be returned.

```
$temp = split(' ', "A B C")[1]; # $temp is now "B"
```

String manipulation

Sleep strings come in two varieties. Literals and parsed literals. A literal string is a string where what you type is exactly what you get.

```
$name = 'The Simple Language for Environment Extension Purposes';
```

A literal string is always enclosed in ' single quotes. A parsed literal string is always enclosed in " double quotes.

```
$name = "The Simple Language for Environment Extension Purposes";
```

Parsed Literals

Parsed literals allow you to embed $scalar variables directly inside of them. Variables inside of parsed literals are evaluated to their current value.

```
$parsed = "The language is $name";
$normal = 'The language is $name';
```

The scalar $parsed would now contain the value "The language is The Simple Language for Environment Extension Purposes". Where as the scalar $normal would now contain the value 'The language is $name'.

Parsed literals also have a special scalar $+. The scalar $+ is a concatenation operator within parsed literals. In case you want to combine two scalars within a parsed literal.

```
$a = "Super";
$b = "man";
$value = "$a $+ $b";
```

The scalar $value would have the value "Superman".

Wait, theres more. Scalars inside of parsed literals also support some built in formatting. You can format a scalar inside of a parsed literal as follows.

```
$first = "John";
$last  =  "Doe";
$value = "$[10]first $[10]last";
```

The scalar $value would have the value "John Doe ". The number inside of the [ ] square brackets specifies the number of characters wide the scalar should be. The scalar is typically padded with spaces to the right of the value. If you specify a negative number the scalar will be padded with spaces to the left.

It is also possible to use an expression inside of the alignment [ ] square brackets. Expressions inside of the alignment [] square brackets may not include a reference to an array or hash index.

Parsed literals only evaluate $scalar values. Parsed literals do not evaluate @array and %hash references or indices.

Characters within parsed literals can be escaped using the \ back slash character. The character immediately following a \ back slash is ignored during processing (and the initial back slash is removed). Sometimes characters following backslashes have special meanings. Initial special meanings include:

Escape	Description
\n	newline character
\r	return character
\t	tab character
\\	back slash \ character

Regular Expressions

Sleep includes built-in regular expressions. Regular expressions are a powerful way to parse strings without having to write much code. Regular expressions consist of defining a pattern. Patterns can then be applied to a string to see if the string matches the pattern or to extract information from the string. Unfortunately regular expression pattern syntax is like a whole other language in of itself. Have fun.

In this documentation a correctly formatted regular expression pattern is referred to as a '**pattern**'.

To test if a string matches a pattern in sleep you can use the ismatch comparison operator.

```
if ("my string" ismatch 'pattern') { commands }
```

To extract the "remembered" matches directly after an ismatch comparison (see Appendix C -> Pattern Grouping) you can use the matched function.

```
@array = matched();
```

To simply extract "remembered" matches from a pattern (see Appendix C -> Pattern Grouping) you can use the matches function.

```
@array = matches("some string", 'pattern');
```

Sleep also lets you split a string up using a regular expression pattern as the token delimeter.

```
@array = split('pattern', "some string");
```

Consequently the join function takes a string and an array as a paramter and joins the array together with the specified string.

```
$value = join("with", @array);
```

You can also use the replace function to replace text in a string that matches a certain pattern with some new text.

```
$value = replace("the entire string", 'pattern', "replacement value");
```

The above is equivalent to using:

```
$value = join("replacement value", split('pattern', "the entire string"));
```

Input/Output Capabilities

The sleep IO capabilities allow you to read and write data from/to files, tcp sockets, and other processes.

To read a file into an array:

```
$handle = openf("/etc/passwd");
@data = readAll($handle);
closef($handle);
```

The first line of the above program opens the file /etc/passwd for reading. The value returned by the openf function is an object scalar with information about the open file. The function readAll() takes an object scalar returned by a sleep IO function. All of the data is read from the file and then return as an array. Finally the function closef closes the handle.

To read a line of text from an open file you would simply use the readln function.

```
if (!-eof $handle) { $data = readln($handle); }
```

The above would read one single line of text from the file. The unary comparison operator eof checks whether or not the handle is at the end of the file. If the handle is at the end of the file then there is no more data to be read.

Most functions can do without a $handle argument and just assume stdin/stdout. In cases where a handle for the stdin/stdout is required use the getConsole() function. The predicate operator -eof is one such case where a handle must always be specified.

To write data to a file you have to first open the file for writing.

```
$handle = openf(">>/etc/passwd");
println($handle, "raffi:x:0:0::/:/bin/sh");
closef($handle);
```

The open function works much like redirecting files on the command line. If you specify a filename by itself, the file will be open for reading. If you specify >> two angle brackets then the file will be opened for writing in append mode. If you specify a > single angle bracket then the file will be opened for writing in overwrite mode. The function println takes a scalar object with a handle and a string to write as a parameter. The newline character is automatically appended when using println.

The function print works much like the function println except it does not append a newline character to the written data.

Remember that sleep is a cross platform scripting language. As a convention use the / forward slash character as a path separator so that your scripts will work in a platform neutral way.

Callback Reading

Another way of reading data from a scalar object with a handle is to use callback reading. Callback reading is done with the read() function.

For example:

```
sub myread { printf("Read $1"); }

$handle = openf("/etc/shadow");
read($handle, &myread);
```

The read function takes two parameters. A scalar object containing a io handle and a function handle. In the example above the function handle is for the subroutine myread. Each time a line of text is available to be read myread will be called with $1 pointing to the io handle and $2 being the text read from the file.

This isn't very useful for reading from files but its necessary for reading from other data sources that might have a lot of time in between data being available.

Sockets

Sleep IO can be used to connect to hosts on the internet or your local network with its sockets functionality. Using sockets is similar to working with files. Except when a socket is opened it is opened for both reading and writing. Writing to sockets and reading from sockets is exactly the same as files.

```
$socket = connect("www.yahoo.com", 80);
println($socket, "GET /");
@data = readAll($socket);
closef($socket);
```

The above example is a simple web client. A socket is opened to www.yahoo.com on port 80. A request is then made for the index page. All of the data from the socket is then read into @data. The socket is then immediately closed

Its usually a good idea to use callback reading with sockets. The http protocol is a simpler protocol where a request is made, the data is returned, and then the socket is closed. So using readAll is safe with http. Other protocols are a little bit more interactive than http.

Listening Sockets

Sleep IO can also listen for a connection from a host. Using a server socket is much like working with a normal socket.

```
$socket = listen(31337, 60 * 1000, $host);
println("Received a connection from: $host");
@data = readAll($socket);
closef($socket);
```

The above example listens for a connection on port number 31337. If no one connects to port 31337 after 60,000ms or 60 seconds then the socket will give up. Once a connection is received the ip address of the connecting machine will be in the scalar $host.

If you don't want the socket to give up listening you can specify 0 as the timeout value.

Processes

External programs can be executed and communicated with via sleeps IO as well. Reading from and writing to an executing process is the same as working with files or sockets.

```
$process = exec("ls -al");
@data = readAll($process);
closef($process);
```

Calling closef on a process will kill the process. The above program executes the UNIX command ls -al which lists all of the files in the current directory with some details.

Pipes

Sometimes it is helpful for a script to execute multiple tasks concurrently. Each task being executed at the same time by a given program is usually referred to as a thread. Sleep can execute multiple threads with the function `&fork()`.

To launch a new thread using fork:

```
sub my_taskA
{
   for ($x = 0; $x < 100; $x++)
   {
      println("Task A: $x");
   }
}

sub my_taskB
{
   for ($x = 0; $x < 100; $x++)
   {
      println("Task B: $x");
```

```
    }
}

fork(&my_taskA);
fork(&my_taskB);
```

When the above script executes the subroutines `&my_taskA` and `&my_taskB` will both execute at the same time. The output will be a seemingly random mix of Task A counting to 100 and Task B counting to 100. This is because Java lets `&my_taskA` run for a little while and then it lets `&my_taskB` run for a little while.

When a Sleep script forks, a completely new script environment is created. This new script environment is mostly isolated from its parent script environment. No variables that were visible or available in the parent environment are available in the fork. All forked scripts are isolated from eachother as well. All subroutines, operators, and predicates available in the parent are available to the child.

You may be asking, if no variables are visible, then how does the parent script communicate with its forked children?

The function `&fork()` returns a *$handle* similiar to any of the Input/Output functions for opening files, executing processes, and connecting to servers. The parent script can read from and write to the *$handle* returned by `&fork()` using any of Sleep's built-in IO functions.

Inside each of the children there is a variable called ***$source***. The variable *$source* is an input/output handle as well. Anything written to the *$handle* returned by `&fork` can be read from *$source*. Conversely anything written to *$source* within the child forked script can be read from the *$handle* returned by `&fork`. This is called a pipe.

Working with Binary Data

Sleep also has the ability to work with binary data. A byte of data is generally represented as a string of one character in Sleep. An array of bytes is a Sleep string.

The following is an example of a file copy script:

```
# copy.sl [original file] [new file]

$in = openf(@ARGV[0]);
$data = readb($in, lof(@ARGV[0]));

$out = openf(">" . @ARGV[1]);
writeb($out, $data);

closef($in);
closef($out);
```

The above example uses the readb and writeb functions to read and write binary data. Sleep also has functions for creating byte strings and extracting data from them:

```
For example to write several different data types to a binary file:

$out = openf(">mydatafile");
bwrite($out, 'CidSl', "A", 42, 3.5, "hehe this is a string", 1234567890);
```

To read this data back in:

```
$in = openf("mydatafile");
($char, $int, $double, $string, $long) = bread($in, 'CidSl');

println("char   data is: $char");
println("int    data is: $int");
println("double data is: $double");
println("string data is: $string");
println("long   data is: $long");
```

You may notice the 'CidSl' parameter for &bread (mmm bread) and &bwrite. This parameter is a string specifying the format of the binary data. This parameter is used in the &pack and &unpack functions as well. Information about this format can be found in Appendix E - Binary Data Format Strings.

Catching I/O Errors

All I/O open functions have potential to fail. A file might not exist, the operating system might not be able to execute a command, or a hostname might not be found when trying to open a connection. To check if an error occurred when opening an IO use the checkError() function.

```
$handle = openf("this_file_does_not_exist");

if (checkError($error))
{
   println("Could not open file: $error");
}
```

checkError() returns the latest error message since it was last called. When checkError() is called the 'current' error message is cleared. A scalar passed to checkError() as an argument will have the value of the 'current' error message placed into it. In the above example checkError sets the value of $error to be a message stating why the file could not be opened.

Working with Objects

A new feature in Sleep is the haphazard object extensions for Sleep (HOES). HOES adds some new syntax for creating, accessing, and working with Java objects. This interface is brand new and is geared towards experienced programmers. If you are comfortable with the Java API you may really enjoy playing with HOES.

Object Creation

Objects are created using HOES expressions. The following example creates an instance of the StringTokenizer class:

```
$scalar = [new java.util.StringTokenizer:"this is a test"];
```

You may notice that the package name is specified. You can avoid specifying full package names by importing packages. The syntax for importing packages is exactly the same as in Java. For example:

```
import java.util.*;

$scalar = [new StringTokenizer:"this is a test"];
```

You can also import specific packages i.e. `import java.util.StringTokenizer;`. Objects can be assigned to $scalar variables. Once instantiated they can be operated on as Sleep scalars using Sleep constructs or as Java objects using object expressions.

Object Expressions

```
[object message: parameter, parameter, ...];
```

HOES object expressions are contained inside of square brackets '[]'. Typically the first parameter in an expression is the object being operated on. The second parameter is the message being passed to the object. A message in HOES speak is equivalent to a Java method/function name. Parameters specified after the message followed by a colon ':'. Each parameter is separated by a comma ','.

Object Expression Examples

```
[System gc];
```

The above calls the method gc in the class java.lang.System. Classes inside of the package java.lang are imported for you. A full path to a class name can be specified in object expressions with the period '.' separator i.e:

```
$ctime = [java.lang.System currentTimeMillis] / 1000;
```

In the above example the message *currentTimeMillis* is retrieved from *java.lang.System*. This example also illustrates an object expression being used within a normal Sleep expression.

You can not use a period '.' to access a message/field in an object i.e.:

```
[System.out println: "test" ]; # wrong!
```

To correctly access a message/field that is multiple levels deep:

```
[[System out] println: "test" ]; # correct
```

The above is equivalent to the Java call: `System.out.println("test");`. The above example also illustrates that object expressions can be nested. Another thing to note is that `[System out]` accesses the field *out* within the *System* class. In HOES methods and fields are both accessed the same way. The only difference is fields cannot accept parameters.

Scalars and expressions can be operated on within object expressions:

```
$value = ["this is a String" lastIndexOf: "i" ];
```

The above is equivalent to `"this is a string".lastIndexOf("i")` in Java. Numbers are treated as there wrapper java objects. For example a double scalar is converted into a Double object for use in object expressions:

```
$value = [3.45 isNaN];
```

The above calls the method/message isNaN on the Double value 3.45. It is worth noting that fields/methods that return a Java boolean type will end up being turned into a string scalar in Sleep. Basically a boolean value from a Java method will be a string value of "true" or "false".

The following is a more in-depth example of object expressions in action:

```
import java.util.*;
```

```
$scalar = [new StringTokenizer: "this is a test", " "];

while ([$scalar hasMoreTokens] eq "true")
{
   println("Token: " . [$scalar nextToken]);
}
```

The above example breaks the string "this is a test" down by the delimeter " " using the java.util.StringTokenizer class.

Sleep Interfaces

A neat feature of HOES is how interfaces can be quickly and easily created and passed to Java objects. A Java interface defines methods that a Java class should implement to facilitate interacting with certain objects.

```
import javax.swing.*;
import java.awt.*;

$frame = [new JFrame:"Test"];
[$frame setSize:240, 120];
[$frame setDefaultCloseOperation: [JFrame EXIT_ON_CLOSE]];

$button = [new JButton:"Click me"];

[[$frame getContentPane] add:$button];

[$frame show];

$clicked = 0;

sub button_pressed
{
   # $0      = the message passed i.e. "actionPerformed"
   # $1 .. n = the message parameters i.e. a java.awt.event.ActionEvent object

   $clicked++;

   [[$1 getSource] setText:"Clicked $clicked time(s)"];
}

[$button addActionListener:&button_pressed];
```

This script example creates a window with a button on it. The button is updated to show a click count each time it is clicked. Closures can be specified where an object implementing a certain interface is asked for. When an object tries to call a "method" on an interface subroutine, the closure is called with the method name specified as a 0th parameter. The parameters 1..n are the actual parameters passed with the message.

Above, the *&button_pressed* subroutine is acting as a java.awt.event.ActionListener interface. The only message passed to ActionListener interfaces is the actionPerformed message. The button_pressed subroutine is passed as a closure parameter in this line: `[$button addActionListener:&button_pressed];`.

Catching Exceptions

Often times Java API's acessed via HOES can throw an "exception". Exceptions are a mechanism used to notify programs that some type of error has occured. You can check for an exception using the `checkError($scalar)` function.

Sleep Closures

Closely coupled with HOES is the concept of Sleep closures. A closure in sleep is a block of code that contains its own lexical scope. i.e. there are variables specific to each closure instance.

Subroutines in Sleep are considered named closures. They are accessed and passed as arguments using their names. i.e *&my_subroutine*.

To create a simple closure:

```
$closure = {
   println("My name is: $1");
};

[$closure: "Raphael"];
```

The output of the above is:

```
My name is: Raphael
```

Closures take arguments just like subroutines. The first parameter is $1, the second parameter $2, so on and so forth. Subroutines can also be accessed using HOES syntax:

```
sub my_sub {
   println("My name is: $1");
}

[&my_sub: "Raphael"];
```

The $0 parameter of a closure call is the message/method name passed to the closure.

Closure Scope

Closure instances have their own variable scope. To declare a new closure instance and some of its initial values you can use the lambda function:

```
$myfunc = lambda({
   println("My initial name is $name");
},  $name => "Raffi");

[$myfunc];
```

The above creates a new closure assigned to $myfunc. The initial value of $name inside of the closure scope is "Raffi".

To explicitly declare a value or set of values to be within a closure scope use the function `this()`>. i.e.

```
$myfunc = {
   this('$a $b $c @array %hash');
   # do stuff...
};
```

In the above closure the variables *$a, $b, $c, @array,* and *%hash* are all available when *$myfunc* is executed. They are not available outside of the *$myfunc* scope.

Another special variable available within closures is the *$this* variable. *$this* refers to "this" closure.

Accumulator Example

This first example is a Sleep Accumulator Generator. An Accumulator Generator is a function that takes a parameter returns a function that increments the accumulator generator parameter by a parameter specified to the generated function.

```
sub accum
{
   return lambda(
          {
             $i = $i + $1;
             return $i;
          },  $i => $1);
}

$accum_a = accum(3);
println("a: " . [$accum_a: 1]);
println("a: " . [$accum_a: 1]);

$accum_b = accum(30);
println("b: " . [$accum_b: 2]);
println("b: " . [$accum_b: 2]);
```

The output of the above is:

```
a: 4
a: 5
b: 32
b: 34
```

Notice that two accumulators were generated and assigned to *$accum_a* and *$accum_b*. The *$accum_a* accumulator starts out with an initial value of 3. The first call against it increments it by 1 making the value within the accumulator 4. The next call increments it by 1 making for an internal value of 5.

Closures as Pseudo Objects

Closures can also accept a "message" parameter just like HOES objects. i.e.:

```
[$closure message: arg1, arg2, ... ];
```

This message parameter is available as the variable *$0*. Using $0 it is possible to make closures that behave much like objects. The following example

generates a Stack data structure closure:

```
sub BuildStack
{
   return {
              this('@stack');

              if ($0 eq "push") { push(@stack, $1); }
              if ($0 eq "pop")  { return pop(@stack); }
           };
}

$mystack = BuildStack();  # construct a new stack closure...
[$mystack push: "test"];  # push the string "test" onto the stack

println("Top value is: " . [$mystack pop]); # pop the top value off of the stack and print it
```

The above is an example of a subroutine that generates a closure that emulates a Stack object. Notice that the push and pop calls on this closure are accessed just like a normal Object would be via HOES.

Function Library

This appendix is a list of the application neutral built-in functions for sleep. I expect this list to grow as time goes on. Much of the functionality for sleep should in theory come from the bridges provided by the application author.

Conventions

$ add(@array, $scalar, [index])

The $ to the left of the function name represents the return type of the function. Some functions will return an array (@), some a hash (%), and others a scalar ($). Functions that have no return value will have nothing to the left.

Optional parameters will be enclosed in [ ] square brackets. The ellipse ... as a parameter indicates the function can handle as many arguments as you are willing to specify.

Note, the $ is not part of the function name. It is solely shorthand for specifying the return type of the function.

Array Functions

	Function	Description
$	add(@array, $scalar, [index])	inserts $scalar into @array at the specified index
@	array("string 1", 2, $var3, ...)	creates an array with all of the function arguments
	clear(@array)	clears @array
@	copy(@array)	returns a copy of @array with copies of its elements
@	map(&closure, @array)	Evaluates the specified closure against each element of @array. Each value returned by &closure is collected into an array that is returned by the map() call.
$	pop(@array)	removes the last element from @array and returns it
$	push(@array, $scalar)	pushes $scalar on to the top of @array
	remove(@array, $scalar, ...)	removes each specified $scalar from @array..
$	removeAt(@arrray, index)	removes the element at index from @array.
@	reverse(@array)	returns a copy of @array in reverse order
$	shift(@array)	removes the first element from @array and returns it.
$	size(@array)	returns the size of the specified @array
@	sort(&closure, @array)	returns a copy of @array sorted using &closure
@	sorta(@array)	returns a copy of @array sorted alphabetically
@	sortd(@array)	returns a copy of @array sorted numerically. For use with an array of scalar doubles.
@	sortn(@array)	returns a copy of @array sorted numerically. For use with an array of scalar integers.
@	subarray(@array, n, [m])	extracts a subarray of range n-m from @array

Date/Time Functions

In general sleep dates are represented using a scalar long that holds the number of milliseconds since the epoch. The epoch is January 1st, 1970. Many external sources will return a numerical time value in terms of the number of *seconds* since the epoch.

The 'format' parameter is used to specify the format for parsing or returning a date string. For more information on how to specify the format parameter see

Appendix D - Date/Time Formatting

Sleep provides the following functions for date/time manipulation:

	Function	Description
$	formatDate([date], 'format')	returns the specified data parameter formatted as a string using the specified format. The specified date should be a scalar long representing the number of milliseconds since the epoch.
$	parseDate('format', "date string")	parses the specified "date string" into a scalar long following the specified 'format' to figure out exactly how the "date string" is formatted.
$	ticks()	returns the number of milliseconds since the epoch

File System Information

Sleep supports the following comparison operators for querying a files information:

Operator	Description
-canread "file"	true if the specified file can be read
-canwrite "file"	true if the specified file can be written to
-exists "file"	true if the specified file exists
-isDir "file"	true if the specified file is a directory
-isFile "file"	true if the specified file is a file
-isHidden "file"	true if the specified file is a hidden file

Sleep also supports the following functions for getting information about files and directories:

	Function	Description
	createNewFile("file")	creates a new file
	deleteFile("file)	deletes a file
$	getCurrentDirectory()	returns the path for the current working directory
$	getFileName("/path/file")	returns the filename extracted from the full path
$	getFileParent("/path/file")	returns the files parent extracted from the full path
$	getFilePath("/path/file")	returns the file path extracted from the full path
$	getFileProper("path", "file", ...)	combines /path and files into an appropriate filename and path for the current platform.
$	lastModified("file")	returns a scalar long with the time the file was last modified
@	listRoots()	returns all of the "root" points for the file system. In UNIX this is just /, in Windows it would be C:/, D:/, A:/
$	lof("file")	returns the size of the specified file
@	ls("directory")	gets a list of all the files in the specified directory
	mkdir("directory")	creates a new directory
	rename("old", "new")	renames a file
	setLastModified("file", n)	sets the last modified time of a file
	setReadOnly("file")	sets a files mode to read only.

Hash Functions

	Function	Description
	clear(%hash)	clears %hash
%	hash("key=value",blah => "value", ...)	instantiates a hash with the specified values
@	keys(%hash)	returns a sleep string array of all the keys in %hash
	remove(%hash, $key, ...)	removes all specified keys from %hash
@	values(%hash, $key)	returns a flat array of all the values in %hash

IO Related

	Function	Description
@	bread([$handle], 'format')	reads data from $handle. Returned as a scalar array with types specified by the format string
	bwrite([$handle], 'format', ...)	writes data to $handle. each parameter represents a piece of data as specified in format. parameters can either be 1 parameter per item in the format or a single array with an element

		for each item described by the format.
	closef($handle)	closes the IO for $handle
$	connect("host", port, [&closure])	connects to the specified host:port and returns a $handle. Check for issues connecting to a host with checkError(). If &closure is specified, this call will not block. &closure will be called when a connection is established.
$	exec("command")	executes the specified command and returns a $handle. Check for issues executing a process with checkError()
$	fork(&closure)	executes the specified &closure as a new thread in an isolated script environment. the return value is a $handle that can be used to read/write data from/to the isolated environment. Within the isolated environment data can be written to/read from the returned $handle using the $source variable.
$	getConsole()	returns the $handle for stdin/stdout.
$	listen(port, [timeout], [$host], [&closure])	listens for a connection on the specified port. Check for listening issues with the checkError() function. If &closure is specified, this call will not block. &closure will be called when a connection is established. The specified scalar $host will be set to the name of the host that connects (if successful).
	mark([$handle], n)	marks the current point in this IO stream. this mark can be reset back too up until n bytes has been reached.
$	openf("[>>|>]file")	opens the specified file for read or write and returns a $handle. >>file = append mode, >file = overwrite mode. Check for issues opening a file with checkError()
$	pack('format', ...)	packs data into a sleep string. each parameter represents a piece of data as specified in 'format'. data can be specified as either 1 parameter per format item or a single array with an element for each item described by the format.
	print([$handle], "text")	prints "text" to the specified handle
	printAll([$handle], @array)	prints entire contents of @array to $handle
	printEOF([$handle])	signals EOF by shutting down output for $handle
	println([$handle], "text")	prints "text" with an appended newline character to the specified handle
	read([$handle], &closure, [n])	each time text is read from $handle, &closure will be called with the $1 = to the handle and $2 = to the text. specifying a n value will cause $2 to be a string of n bytes. if n is specified sleep will try to read n bytes before calling the closure. the closure will be called for the last read even if < n bytes are in the buffer.
@	readAll([$handle])	reads all of the text from the specified handle
$	readb([$handle], n)	reads n bytes from $handle
$	readln([$handle])	reads a single line of text from the specified handle
	reset([$handle])	resets this IO stream back to the last mark
$	skip([$handle], n)	tells the handle to skip the next n bytes
@	unpack('format', "string")	unpacks data from the specified sleep string. data is returned as a sleep array with each scalar set to a type as specified in the format string
	writeb([$handle], "string")	writes the byte data of "string" to $handle

Note: STDIN/STDOUT will be used by default for any of the IO read/write functions if a $handle is not specified.

Number Functions

	Function	Description
$	double($scalar)	returns a copy of $scalar as a double scalar
$	formatNumber(number, [from], to)	formats specified number from specified base to specified base.
$	int($scalar)	returns a copy of $scalar as an integer scalar
$	long($scalar)	returns a copy of $scalar as a long scalar
$	not($scalar)	returns the logical not of $scalar
$	parseNumber(number, [base])	parses number of specified base value (base as in binary (2), decimal (10), hex (16) etc.)
$	rand([number])	generates a random integer between 0 and number. If number is ommited the function generates a random double between 0 and 1

String Functions

	Function	Description
$	asc("x")	returns a scalar integer of the ascii value of the specified character
$	chr(number)	returns the ascii character associated with the specified ascii value
$	charAt("string", n)	returns the character at the n'th position in the string

$	indexOf("string", "substr")	returns the index of "substr" inside of "string"
$	join("string", @array)	joins the elements of @array with "string"
$	lc("STRING")	returns a lowercase version of the specified string
$	left("string", n)	returns the left n characters of "string"
@	matched()	returns the matches from a "string" applied to a regex 'pattern' during an ismatch comparison
@	matches("string", 'pattern')	returns the matches from "string" applied to the regex 'pattern'
$	replace("string", 'pattern', "new")	splits "string" by regex 'pattern' in "string" and joins it with specified "new" value
$	right("string", n)	returns the right most n characters of "string'
@	split('pattern', "string")	splits the specified string by the specified pattern
$	strlen("string")	returns the length of the specified string
$	strrep("string", "old", "new", ...)	replaces occurrences of old with new in string. accepts multiple old, new parameters.
$	substr("string", start, end)	returns a substring of the specified "string"
$	uc("string")	returns an uppercase version of "string"

Utility Functions

	Function	Description
$	checkError([$scalar])	returns the last error message to occur. if a $scalar is specified : $scalar is set to the error string. Once chekError() is called the error message is cleared. Functions that might flag an error will be documented.
$	compile_closure("code", ...)	compiles the specified code into a sleep closure. key/value pairs can be specified as initial values similar to &lambda. syntax errors can be obtained with &checkError()
$	eval("code")	parses and evaluates the specified code returning the return value of the code. syntax errors can be obtained with &checkError()
$	expr("expr")	parses and evaluates the specified sleep expression code returning the value of the expression. syntax errors can be obtained with &checkError()
$	function('&func_name')	obtains the function handle for the function bound to the specified string
&	lambda(&closure, $key => "value", ...)	copies &closure into a new closure. The closure environment is initialized with all of the key/value pair arguments.
	local('$x $y')	parses the specified string and declares all variables in the string as local variables.
	sleep(n)	force the current executing thread to sleep for n milliseconds
%	systemProperties()	returns a hash of the available system properties. See Appendix A for a list of some of the system properties
	this('$x $y')	parses the specified string and declares all variables in the string as variables specific to the current closure.
	use("Loadable class")	loads the specified class into the current script environment. Classes specified as package.Name are loaded from the Java classpath. Classes can also be specified as filenames without the .class extension. The specified class must be a Loadable bridge (i.e. it implements sleep.interfaces.Loadable).
	use("/path/to/file.jar", "Loadable")	loads the specified class into the current script environment. The sleep.interfaces.Loadable class is loaded from the specified jar file. Use to import sleep modules and the like.

Appendix A - System Properties

This appendix contains a list of system properties available in the hash returned by the systemProperties() function. This list is not comprehensive:

Key	Example Value
java.io.tmpdir	/tmp
java.runtime.name	Java(TM) 2 Runtime Environment, Standard Edition
java.vm.version	1.4.1_01-24
path.separator	: *the file path separator for the current platform*
os.arch	ppc
os.name	Mac OS X
os.version	10.2.8
user.country	US
user.dir	/home/raffi/sleep/bin *the current active directory*
user.home	/home/raffi
user.timezone	EST

Appendix B - Wildcard Strings

Sleep has the ability to determine if a string matches a specified wildcard string. This appendix is just a quick refresher.

A wildcard string can be a normal string. For example you can check if the wildcard string "Jack" matches "John". The result will be false. This is because Jack does not match John. They are not the same thing.

Wildcard strings have four special sequences *, **, ?, and \.

The ? question mark character allows for any one character to be in that position. For example the wildcard string "J?ck" matches the string "Jack". "J?ck" will also matche "Jick", "Jock", and "Jeck". Where the ? question mark is any thing can go. Something must go there though. For example "J?ck" is not a wildcard match for "Jck".

The * asterisk special character allows for zero or more characters to be in that position. For example the wildcard string "J?c*" matches "Jack". The wildcard string also matches "Jock Strap", "Jeck is not a word", and other fun phrases. When matching is done with * it is done in a non-greedy fashion. This means as soon as the the matcher encounters a sequence that allows it to continue to the next part of the wildcard string it will.

```
if ("*@aol.com" iswm $email)
{
   # address belongs to an AOL wARRIOR
}
```

The above code snippet checks if the wildcard string *@aol.com is a a match for $email. If it is then we know that the email address is an AOL email address.

The ** asterisk sequence also allows for zero or more characters to be in that position. The only difference is matching is handled in a greedy fashion. This means the matcher will try to consume as many characters as possible before continuing to the next part of the wildcard string.

The '\' backslash character is used to escape a character. If you want to specify a literal question mark '?' or asterisk '*' in your pattern prefix it with a '\' backslash.

Greedy versus Non-Greedy

```
if ('this*test' iswm 'this is me testing with this test')
{
   # Non-greedy example: no match
}

if ('this**test' iswm 'this is me testing with this test')
{
   # Greedy example: we have a match
}
```

The two examples above illustrate greedy versus non-greedy matching. In the non-greedy example the matcher will find "this" and then it will use the * for "is me" before encountering the next part of the pattern which is "test" in the string "testing with this test".

In the greedy example the matcher will find "this" and it will then use * for as much stuff as possible. Taking it all the way to the end of the string where "test" is.

Appendix C - Regular Expression Pattern Syntax

Sleeps regex functions are all driven off java's regex API's. For a more complete list of java regex pattern syntax visit:

http://java.sun.com/j2se/1.4.1/docs/api/java/util/regex/Pattern.html

Think of a pattern as a description of a string. A pattern describes what an "acceptable" string looks like. When a string is applied to a pattern, the regex engine goes through the string character by character advancing it through the pattern. If the entire string makes it through the entire pattern it matches. If it doesn't then the string does not match the pattern i.e. it is not an "acceptable" string.

Regular expression patterns are composed of pattern elements. A pattern element is just a way of specifying what part of a string should look like. This appendix describes the literal character and character group pattern elements. It also describes how to group pattern elements and how to attach quantifiers to the pattern elements.

This appendix is only meant to serve as an introduction to the topic. It is by no means a complete reference.

Literal Characters

Literal characters match exactly. For example letters (a-z, A-Z), digits (0-9), and some special characters (%, @) all mean themselves. When the regex engine encounters a literal character it is looking for the next character in the applied string to be the literal character.

To make special characters into literal characters escape them using a \ backslash. For example ? question mark has a special meaning. You can escape it with

\?.

Character Groups

Character groups are a way of grouping characters together. When the regex engine encounters a character group it is looking to see if the next character in the applied string matches anything specified in the character group. Character groups are enclosed in [ ] square brackets.

```
if ("i" ismatch '[aeiou]') { }
```

In the above example we are looking to see if y matches the pattern of vowels. The character group within the pattern will accept 'a' or 'e' or 'i' or 'o' or 'u'. Nothing else will be accepted for that first and only letter.

It is also possible to negate a character group by making the first character a ^ hat symbol.

```
if ("y" ismatch '[^aeiou]') { }
```

In the above example any consonant will match. Any vowels will be rejected because the ^ hat in the beginning of the character says 'do not accept any of these characters'. The value "y" will be accepted as a match since y is not a, e, i, o, or u.

Within the square brackets it is also possible to specify a range of characters.

```
if ("C" ismatch '[a-zA-Z]') { }
```

In the above example we are looking to match any character a through z or A through Z. You could just as easily write '[a-cD-Z]' which would match any character a through c or D through Z. The value "C" is a match since "C" is in one of the ranges a-z or A-Z.

The regex engine also includes quick shortcuts for writing common character groups. The escape "\d" matches any digit. Using "\d" is equivalent to using "[0-9]". "\w" will match any word character a-z, A-Z, and _ underscore. "\s" will match any white space character such as tabs, spaces, and new lines. Uppercase versions of these shortcuts are the same as using ^ hat. "\W" will match any character that is not a word character.

The meta character . period is a character class that will match any character.

Pattern Grouping

Pattern elements enclosed in ( ) parentheses are considered to be grouped together. This grouping makes the regex engine consider the elements in the group to be one element. This way count quantifiers can be applied to the group as a whole.

Enclosing pattern elements in ( ) parentheses also has the effect of making the regex engine "remember" the text in the applied string that matched the enclosed pattern. The matched text can be retrieved with the sleep function matches().

```
if ('You ZEBRA' ismatch '(Y[aeiou][aeiou])\sZEBRA') { }
```

In the above example three pattern elements are grouped together. The literal character Y followed by our character group for vowels. The grouping now makes the regex engine consider these three pattern elements to be one pattern element. It doesn't make much of a difference to the matching unless a quantifier is attached to the grouped pattern elements (see below). The regex engine will remember the applied string text that matched the pattern elements enclosed in the ( ) parentheses.

```
@data = matches("You ZEBRA", '(Y[aeiou][aeiou])\sZEBRA') { }
```

In the above example the array @data would now have one element and that would be the string "You".

Count Quantifiers

Count quantifiers affect the previous element. Whether this element is a literal character, a character group, or a pattern grouping. Count quantifiers are a way of specifying how many times you want to see the previous element occur at this point in the pattern.

The count quantifier + means match the previous element 1 or more times.

```
if ('uaiaaeeueeiiioiauZEBRA' ismatch '[aeiou]+ZEBRA') { }
```

In the above example the pattern is looking for the character class of 'a' or 'e' or 'i' or 'u' to match 1 or more times. So the first character in the applied string can be a, e, i, o, or u. The second character of the applied string can be a, e, i, o, or u. The regex engine keeps looking through the applied string until the current character no longer matches the character class. The regex engine then tries to apply the next part of the pattern. Which in this case is the literal characters ZEBRA.

The count quanitifer * asterisk means match the previous element 0 or more times. The count quantifier ? question mark means look for the previous element to occur once or not a all.

You can also specify your own count quantifier using { } curly braces. You can use {5} to mean match the previous element 5 times exactly. Or you can specify a range {4, 10} meaning match the previous element 4 to 10 times.

Attaching the ? question mark onto any count quantifier results in the regular expression being processed in a reluctant manner rather than a greedy manner. Reluctant means the regex engine will look to see if it can move ahead to the next part of the pattern. Greedy means the regex engine will always try to stay at the current part of the pattern when processing a string.

The OR Quantifier

Another quantifier worth writing about is the OR quantifier. The OR quantifier is represented with a | pipe symbol. The OR quantifier uses the previous and next elements. It says the next character can match the pattern element before the | pipe symbol or the next character can match the pattern element after the | pipe symbol.

```
if ('uaiaaeeueeiiioiauZEBRA' ismatch '[aeiou]+|XZEBRA') { }
if ('XZEBRA' ismatch '[aeiou]+|XZEBRA') { }
```

In the above example we have modified our 'look for a string of vowels 1 or more characters long followed by the word ZEBRA' pattern. We have attached the | quantifier after the character group followed by a count quantifier. Remember the character group with attached count quantifier is now thought of as one single element. The | quantifier says this part of an acceptable string can be either the previous element ([aeiou]+) OR it can be the next element (the literal character X). It can't be both it can only be one or the other.

Regex Options

It is possible to set/unset a few switches for your patterns. You can use (?options) to enable or (?-options) to disable certain options each specified with a single character. The *i* option when set makes the pattern matching case insensitive. The *d* option when set enables UNIX lines mode. The *m* enables Multiline mode which makes the ^ (beginning of input character) and $ (end of input character) match at the beginning/end of each line. The *s* flag makes the . character match any character including line feeds. Finally, the *x* makes the matcher ignore white space in the pattern and enables pattern comments (beginning with a #, ending with a newline).

Appendix D - Date/Time Formats

Note: The following is modified from the documentation provided by Sun for the Java java.util.SimpleDateFormat class. FYI for Sun, nice job on this API - I like it.

Date and time formats are specified by date and time pattern strings. Within date and time pattern strings, unquoted letters from 'A' to 'Z' and from 'a' to 'z' are interpreted as pattern letters representing the components of a date or time string. Text can be quoted using ' single quotes to avoid interpretation. 'at' represents the word at.

The following pattern letters are defined (all other characters from 'A' to 'Z' and from 'a' to 'z' are reserved):

Letter	Date or Time Component	Examples
G	Era designator	AD
y	Year	1996 ;96
M	Month in year	July ;Jul ;07
w	Week in year	27
W	Week in month	2
D	Day in year	189
d	Day in month	10
F	Day of week in month	2
E	Day in week	Tuesday ;Tue
a	Am/pm marker	PM
H	Hour in day (0-23)	0
k	Hour in day (1-24)	24
K	Hour in am/pm (0-11)	0
h	Hour in am/pm (1-12)	12
m	Minute in hour	30
s	Second in minute	55
S	Millisecond	978
z	Time zone	Pacific Standard Time ;PST ;GMT-08:00
Z	Time zone	-0800

Pattern letters are usually repeated, as their number determines the exact presentation:

For an item that is usually text i.e. the day of the week repeating the pattern letter 4x or more times results in the full form of the item being used. Otherwise using the pattern letter less than 4x results in a shorter version of the text being used.

For formatting a year using yy will cause the year to be truncuated to 2 digits.

Examples

The following examples show how date and time patterns are interpreted in the U.S. locale. The given date and time are 2001-07-04 12:08:56 local time in the U.S. Pacific Time time zone.

Date and Time Pattern	Result
"yyyy.MM.dd G 'at' HH:mm:ss z"	2001.07.04 AD at 12:08:56 PDT
"EEE, MMM d, "yy"	Wed, Jul 4, '01
"h:mm a"	12:08 PM
"hh 'o"clock' a, zzzz"	12 o'clock PM, Pacific Daylight Time
"K:mm a, z"	0:08 PM, PDT
"yyyyy.MMMMM.dd GGG hh:mm aaa"	02001.July.04 AD 12:08 PM
"EEE, d MMM yyyy HH:mm:ss Z"	Wed, 4 Jul 2001 12:08:56 -0700
"yyMMddHHmmssZ"	010704120856-0700

Appendix E - Binary Data Format Strings

Binary data format strings are used in the &bread, &bwrite, &pack, and &unpack functions. An array of byte data is stored as a sleep string scalar. Specifying a data format string with one of the above functions allows you to retrieve/store binary data into a string scalar or a file.

The data format string consists of the following characters:

Letter	Bytes	Description
b	1	byte (-128 to 127) (converted to/from a sleep int)
B	1	unsigned byte (0 to 255) (converted to/from a sleep int)
c	2	UTF-16 Unicode character
C	1	normal character
d	8	double (uses IEEE 754 floating-point "double format" bit layout)
f	4	float (uses IEEE 754 floating-point "single format" bit layout)
i	4	integer
I	4	unsigned integer (converted to/from a sleep long)
l	8	long
M	0	mark this point in the IO stream (for reads only)
R	0	reset this stream to the last mark point (reads only)
s	2	short (converted to/from a sleep int)
S	2	unsigned short (converted to/from a sleep int)
u	variable	UTF-8 format string ([2 bytes = length][variable = data])
x	1	skips a byte/writes a nully byte in/to this stream (no data returned)
z	variable	read/write character data until terminated with a null byte. (see note below)
Z	variable	read/write the specified number of characters (consumes the whole field)

Any of the above data descriptors can be followed by an integer that says to repeat this descriptor a number of times. Use of the '*' character following a data descriptor indicates you want the rest of the data to be read as the specified type. Whitespace is ignored inside of data format strings.

'z/Z' Note: specifying an integer following a 'z/Z' read has a different effect. Using 'Z/z' is the same as using 'C' except the entire string of characters is returned as one scalar. 'z' will loop until a null byte or until it has read the number of characters specified by the optional follow-on integer.

Appendix F - Sleep 1.0 -> Sleep 2.0 Gotchas!

Sleep 1.0 compatability is a big goal for the Sleep 2.0 development cycle. However sometimes things can not be helped. This appendix lists some of the gotchas scripters transitioning from Sleep 1.0 to Sleep 2.0 may experience

1. ### Subroutine arguments are now treated as closures

   In Sleep 2.0 subroutine arguments (i.e. passing a &function to a function) are now treated as closures. Sleep used to wait until the function was executed to evaluate the passed &function argument. This is no longer the case in Sleep 2.0. Referencing &function will immediately cause &function to be resolved to the declared code it is attached to. The implication of this is that before a &function can be passed as an argument, the &function must be declared.