Introduction to PIL

PIRCH Interpreted Language (PIL) is an internal scripting language of PIRCH Internet Relay Chat client software.

PIL's language structure is similar in syntax and structure to PASCAL, combined with the established syntax of PIRCH's alias coding structure.

Using PIL
PIL Scripts are NOT PIRCH aliases, nor are PIRCH aliases PIL scripts. While alias constructs and PIL statements have thier similarities, there are disctinct language differences, and there are key reasons for these differences which I will not cover in this document as it is slightly off topic.

PIL scripts however are installed into the PIRCH alias window/editor, and the contents of scripts are stored within the PIRCH alias file (by default aliases.paf). Its recommended that you adopt a naming convention for PIL scripts, in which the scriptname is enclose in square brackets, ie [MYSCRIPT]. This allows your PIL scripts to be sorted to the bottom of the alias editor list, keeping all script together. While this convention is recommended, it is not a requirement.

The alias window in PIRCH has a couple of new features that make installing individual PIL scripts easy. By right clicking the alias name list in the alias window, a popup menu is displayed which contains two (2) PIL related commands.

Add PIL Script	Loads a PIL script from a text file and installs it into the alias window
Extract PIL Script	Saves a PIL script to a text file

Both of the above commands use the [] bracket convention for naming files. When a PIL script is loaded, PIRCH will place brackets around the name. When a PIL script is extracted, the brackets are removed. Extracting and Adding PIL scripts is useful when you want to share your scripts with others, or you want to install you scripts written by others.

PIRCH PIL EXTENSIONS
PIRCH (version 0.82) introduces two new commands that are used specifically for executing PIL scripts. These are covered in detail below and in the PIRCH help file.

/RUNSCRIPT

PIL scripts are executed with the PIRCH command /RUNSCRIPT.
The format for the command is:

> /RUNSCRIPT <scriptname> <parameters>.

Unlike aliases, you can not simply type /scriptname. You can however make an alias to run the script. An example will better demonstrate this point. Lets assume you have a script, called [MYSCRIPT], an you wish to simply type /MYSCRIPT to execute it... add an alias as follows

> myscript:/runscript [myscript] *1

NOTE: typing /[myscript] will NOT work.

Passing information to a script
PIL scripts can access a number of system defined variables, such as $date to retrieve a number of pieces of information. However the most useful information can be the parameter string which you pass to your script when you execute a /runscript command. The parameters can be any information which your script requires or can make use of and can be accessed from within the script using the following variables

$1..$nn	Individual parameters (space delimited)
*1..*nn	Sequential set of parameters (sequential command lines parameters start with the number provided)

For example: executing `/runscript [myscript] My name is Bob`
would make $1 within your script equivalent to the word "My", $2 to the word "name" and so on. *1 would

be equivalent to “My name is Bob”, *2 would be “name is Bob” and so on. (each without the quote marks)

/CALLBACK

/CALLBACK is the probably the most advanced, and therefore, unfortunately most complicated command to understand and master. Basically /CALLBACK allows you to install a script to handle a variety of incoming server messages. This is similar to what PIRCH events do, and in most cases use of simple ON XXXXX is recommended. However, not all server messages are accessible in events and thats where use of these server callbacks can become useful.

For example, when you type /whois <nickname>, the IRC server returns up to four (4) distinct messages to PIRCH, and PIRCH will normally display the information for you in the server/status window. But lets assume you want to do something with this specific with this information; and since there is no ON XXXX event which covers the information returned by /whois, you can install a PIL script to handle/manipulate the returned information as desired.

The format for the /CALLBACK command is as follows:

```
/CALLBACK <server rpl code> <scriptname>
```

Obviously to make use of this command and implement callback features you need to have information about server replies/messages and the format of these messages, all of which can be found in RFC1459 which is the official RFC protocol specification for IRC. Albeit to some degree out of date, the RFC document is still the most reliable and informative source. This document can be obtained from all complete RFC list sources and at last check was available at ftp.undernet.org

When a callback script is installed, PIRCH passes all the information contained in the server reply/message to the script as a parameter, when PIRCH encounters the specific reply/message code. This information can then be disseminated by the script in whatever fassion you desire.

Intended Audience

This document is intended for as a reference guide for persons requiring detailed descriptions of the language syntax, structure and components. Much of the document is intended for persons who have at least a fundamental understanding of structured computer languages, and the workings of procedure, functions, conditionals and loops.

This document is structured as a reference guide, and is NOT a step by step guide to writing PIL scripts.

PIL Language Definition and Structure

PIL Language Definition and Structure

Embedding Comments in Scripts

Before proceeding with the technical information, a word about commenting your script code. PIL uses the pascal method for placing comments directly within script code, by embedding comments within curly braces { }. The PIL compiler ignores anything it encounters following an open curly brace { until it encounters a closing curly brace } (with the exception of string literals that contain a curly brace character.)

Although for the most part, PIL scripts tend to be readable, and understandable on their own, commenting of script code is a fundamental part of any programming language and should not be ignored.

Commenting can serve multiple purposes, providing the programmer with ability to internally document complicated sections of script code for future reference, possibly indicating why a possible method was chosen over another. If you distribute your scripts for others to use it maybe important for your intended audience to understand what a script does and/or how it does it.

Commenting can also aid in debugging: by commenting out statements, you can see how it changes the result of your script.

Keywords and Identifiers

PIRCH reserves a number of *key words* for special purposes, generally representing built-in commands and language symbols. Identifiers are words/symbols which you can create to identify variables.

BEGIN	END	FOR	TO
WHILE	DO	IF	THEN
ELSE	VAR		

Each of the above keywords are discussed in detail later in this document.

Variables

In PIL as in most other programming/scripting languages, variables hold the data on which the script operates. PIL currently supports two primary variable types:

Strings	Alpha-numeric character arrays
Numbers	Numeric values

(internally stored as a 32 bit signed integer with a value range of -2147483648..2147483647

Declaring Variables

PIL does not currently use an explicit forced variable declaration system, instead PIL will dynamically create and allocate variables as required during the script compilation phase. The disadvantage is that PIL must provide a method of type identification for variables according to the identifier used.

The variable identifier used will explicitly declare it type by the following rule: all string variables will be prefixed with a single $ symbol and numeric variable types my not contain the $ symbol.

$workstr	is explicitly typed as string
numvar	is explicitly typed a numeric variable

Attempting to assign a string expression to a numeric variable will result in a compilation error of “type mismatch”, as will assignment of a numeric expression to a string variable. (see STRTOINT and INTTOSTR function later in this document for type conversion)

Literal Values

A literal value is simply a declared number or string value. String literals must be enclosed within single quotation marks (‘).

50	Declares a numeric literal value
'hello'	Declares a literal string value

Constants

At its current stage of development, PIL does not support strict constants. However, programmers can achieve almost identical functionality by using variable declarations (with immediate value assignment).

Example: five := 5; would be functionally equivalent to a strict constant declaration. The only difference is that should an assignment attempt be made upon the variable, the assignment would be permitted and the script would continue to execute, whereas attempted assignment to a strict constant would result in a fatal error at compilation time.

Assignment of Values to Variables

The values of variable can be changed through one of two methods:

A. Assignment of an expression
Assignment is achieved by the use of the := symbol, with a variable on the left side and a like type expression on the right.

numvar := 5 * 20; would assign the value 100 to numvar
$s := 'hello world'; would assign 'hello world' to the $s
$s := 5 * 20; would result in a 'type mismatch' error

Once an assignment is made, any usage of the variable in an expression would actually be using the value associated with the variable. The value of variable may be changed during the execution of a script as often as you like.

B. Use of a variable as a VAR type parameter to a procedure/function call

Attempting to assign a string expression to a numeric variable will result in a compilation error of "type mismatch", as will assignment of a numeric expression to a string variable. (see STRTOINT and INTTOSTR function later in this document for type conversion)

Global System Variables

Global systems variables contain information that may change according to system state, however are not true PIL variables, meaning that you may not directly values to them, or use these variables as VAR parameters in procedure and/or function calls. In all cases these identifiers begin with the $ symbol

$day	string	Returns the current day of the week
$date	string	Returns the current system sate
$time	string	Returns the current system time
$server	string	Returns the name of the server to which the windows from which the script is called is associated
$host	string	returns your current local host name
$ip	string	returns your IP address in 4 octet form
$me	string	returns your current nickname
$netid	string	returns the server's network id (if assigned)
$audience	string	returns the channel/nickname of the window from which the script is called
$topic	string	returns the channel topic (only if called from a channel)
$mode	string	returns the channel mode (only if called from a channel)
$members	string	returns the channel member count (only if called from a channel)
$version	string	returns the PIRCH version you are running

Procedures and Functions

The following is a list of internally support procedures and functions, each of these is discussed later in this document.

BREATHE	CHAR	HALT	RANDOM
STRCOPY	STRDEL	STRINS	STRLEN

STRPOS	STRMATCH	STRUPPER	STRLOWER
STRTOKEN	INTTOSTR	STRTOINT	WRITELN
COMMAND			

Simple Expressions & Statements

Definitions

An expression combines constants, variables and function results into a single result. There are 3 primary expression types recognized by PIL: numeric, string and boolean.

Numeric Expressions & Operators

Numeric expressions may be either simple numeric variables, numeric constants numeric literals, or a mathematical manipulation of numeric identifiers using the following operators.

Mathematical Operators

+	addition
-	subtraction
*	multiplication
/	division
^	exponential
mod	modulus

Bitwise Operators

shl	bitwise shift left
shr	bitwise shift right
bitand	bitwise AND
bitor	bitwise OR
bitxor	bitwise XOR
bitnot	bitwise NOT

Boolean (Logical) Expressions

Boolean expressions result in a return type of either true or false and generally require the use of comparison symbols and an expression on both the left and right side. While there is no explicit boolean type declared within the PIL language structure, PIL generates ordinal values for boolean expressions, where 0 = false and 1 = true.

Symbol	Comparison	Example
=	equal	(a = b)
>	greater than	(a > b)
>=	greater than or equal	(a >= b)
<	less than	(a < b)
<=	less than or equal	(a <= b)

Logical expressions may also include logic operators. Logic operators can be used to evaluate two boolean expressions located on the left and right sides of the expression. Currently PIL supports the following logical operators.

Operator	Comparison	Example
AND	Logical And	(a and b)
OR	Logical Or	(a or b)
NOT	Logical negation (not b)	

The following boolean truth table displays how the above boolean operators affect and expression

A	AND	B	=	C
false	|	false	|	false
false	|	true	|	false
true	|	false	|	false
true	|	true	|	true

A	OR	B	=	C
false	|	false	|	false
false	|	true	|	true
true	|	false	|	true
true	|	true	|	true

The NOT operator is used to negate a boolean ordinal.
NOT true is equivalent to false
NOT false is equivalent to true

Operator Precedence

PIL interprets all expression from left to right, but applies precedence to operators according to the following table, ranging from highest precedence to lowest:

Parenthetical		()
NOT, unary minus		not, -Expression
exponents		^
multiplication,division,modulus	*, /, mod	
addition,subtraction		+.-
AND,OR,XOR		and,or,xor

Examples

2 + 3 * 4 result = 14
multiplication has a higher precedence than addition

4 * (3 + 2) result = 20
parenthesized expression (2+3) is evaluated prior to multiplying by 4.

Single Statements

A statement is either a procedure call or an assignment of an expression or function result to a variable. Each statement must be terminated with a semicolon (;).

Compound Statements

A compound statement is a group of individual statements, each separated with a semicolon (;). You can use compound statements anywhere an ordinary statement is allowed. Because compound and single statements are interchangeable, the term *statement* will be used to mean either a compound or singular statement from now on throughout this document.

Simple compound statement construct

```
begin
    a := b * c;
    $s := 'answer';
    writeln($s,' = ',a);
end;
```

Repetitive Statements (loops)

PIL normally executes code in a sequential order, top to bottom, left to right, one statement after the other. However, this default behavior can be modified by use of *repetitive* statements which makes loops in a script, repeating operations. and *conditional* statements which make decisions, selecting one statement over others and changing the flow of the script.

Repetitive statements cause one or more statements to repeat. There are two (2) repetitive statement constructs used in PIL, WHILE and FOR loops.

WHILE/DO Loops

WHILE loops cause a statement or statement group to repeat as longs as a given boolean expression is evaluated to be true.

```
SYNTAX: while <boolean expression> do <statement>
```

Simple WHILE loop construct

```
a := 0;
while a <= 10 do
begin
    a := a + 1;
    writeln(a);
end;
```

The above example counts from 1 to 10, displaying each number on your screen.

Its quite possible and legal that the boolean expression within a while statement to be initially evaluated as false, causing the statements controlled by the loop never to be executed. In the above example, assume we initially set the value of *a* to 20, then the *a := a + 1;* statement would never have executed.

FOR/TO/DO Loops

FOR loops cause a statement block to execute a specific number of times.

```
SYNTAX: for <control variable> := <source expression> to <target expression> do
<statement>
```

Simple FOR loop construct

```
a := 1;
for a := 1 to 10 do
begin
    writeln(a);
end;
```

The above example counts from 1 to 10, displaying each number on your screen. This is identical to what the example for the WHILE loops does, but the for loop is much more efficient in performing the task.

FOR loops require a control variable, which must be numeric. The control variable is initially assigned the value of the *source expression*, and subsequently modified by 1 during each iteration of the loop, executing the statement block until it reaches the *target expression* value.

As with the WHILE loop, a FOR loop may never execute its statements, if the *source expression* value is initially greater than the *target expression* value.

Conditional Statements

IF/THEN/ELSE Statement

IF statements allow your script to make decisions about what statements to execute based on a logical expression that evaluates to either true or false.

Syntax: IF <expression> THEN <statement> [ELSE <Statement>]

Simple IF/THEN/ELSE statement construct

```
if a <> 0 then
    writeln('a does not equal 0')
else
    writeln('a does equal 0');
```

In it is not required than an ELSE clause be provided for each if statement, rather this is optional and may be omitted entirely.

Simple IF/THEN statement construct (ELSE clause omitted)

```
if a <> 0 then
    writeln('a does not equal 0');
```

You can chain multiple IF/THEN/ELSE statements together, allowing your script to make a series of decisions, by simply using another IF statement following an ELSE clause.

```
if a = 0 then
    writeln('a is 0')
else if a < 10 then
    writeln('a is greater than 0 but less than 10')
else if a < 50 then
    writeln('a is greater than 10 but less than 50')
else
    writeln('a is greater than 50');
```

Procedures & Functions

PIL currently doesn't not allow user defined procedures and functions to be created within a script, however, this will be implemented in future versions.
Procedures and functions are still an integral part of the PIL definition and the concepts and definitions need to be understood by the programmer.

Since current implementations of PIL support only two primary variable types, this document will address types as either being string or value.

Procedures

PIL defines a procedure as a subroutine or that acts as an individual statement. Procedures may not be used as a component in an expression, as procedures themselves generate not assignable return value.

Functions

Functions, like procedures, are defined as subroutines, however, functions in and of themselves do not constitute a complete statement. Functions may instead be used as a component in an expression, as all functions by definition must return an assignable value.

This document will use the pascal standards for define of procedures and functions.

```
procedure procname(param[,param] : type[; [...]]);
function funcname(param[,param] : type[; [...]]) : type;
```

PIL Procedures and Function Reference

The following is a list of internally support procedures and functions, each of these is discussed later in this document.

BREATHE
CHAR
COMMAND
FILEEXISTS
FILEREAD
FILESIZE
FILEWRITE
HALT
HASVOICE
INTTOSTR
ISOP
NICKCOUNT
NICKLIST
RANDOM
SNICKCOUNT
SNICKLIST
STRCOPY
STRDEL
STRINS
STRLEN
STRPOS
STRMATCH
STRUPPER
STRLOWER
STRTOKEN
STRTOINT
WRITELN

BREATHE

Declaration
procedure breathe;

Description
Breathe is used to allow the Microsoft Windows system to process messages during lengthy or time consuming script operations. The use of this procedure is never a requirement. When breathe is called, the PIL interpretor returns processor control back to the system and PIRCH will process any pending Windows messages, including processing of incoming data from the server. NOTE: Because a PIRCH event may be triggered during a breathe operation, you must take precautions against the same script being activated via and event and causing recursion.

Example

```
begin
	answer := 0;
	for x := 1 to 5000 do
	begin
			breathe;
			answer := answer + 2;
	end;
	writeln('the answer is ',answer);
end;
```

CHAR

Declaration
function char(index ; value) : string;

Description
Char is returns a 1 character length string holding the character at position index in the system's character table. For ASCII character sets index should be a value ranging from 0 to 255.

Example

```
begin
       $chan := char(35)+'pirch'; { prefix # symbol }
       command('/msg',' ',$chan,' ','hello all');
end;
```

COMMAND

Declaration
procedure command(v1,v2,...,vn);

Description
Use Command to issue IRC or PIRCH specific commands from within a PIL script. Separate multiple items with commas within Command's parenthesis. This items may be of different types, i.e. strings, values or expressions. The resulting syntax of the parameters used for Command must result in a valid command line string which PIRCH can interpret.

Example

```
begin
       $chan := '#pirch';
       command('/msg',' ',$chan,' ','hello all');
end;
```

FILEEXISTS

Declaration
function fileexists($filename : string) : value;

Description
Fileexists returns a boolean ordinal (0 = false, 1 = true), if *filename* exists on the system. Fileexists makes no assumptions about the path for the file, therefore, you should use fully qualified filenames with this function.

Example

```
begin
       $filename := 'c:\pirch\logs\#pirch.log';
       if Fileexists($filename) then
           writeln('#pirch log file was found')
       else
           writeln('#pirch log file was NOT found');
end;
```

FILEREAD

Declaration
function fileread($filename : string; linenumber : value; VAR $s : string) : value;

Description
Fileread reads a string value from $filename. Linenumber may be 0, in which case fileread will read a random line from the file, otherwise linenumber indicates the specific line to be read, where the first line in the file is line number 1. $s must be a variable string type, which is filled with the information read from the file. The return value is a boolean ordinal indicating whether or not the operation suceeded. (1 means the operation was successful, 0 means it failed) Fileread makes no assumptions about the path for the file, therefore, you should use fully qualified filenames with this function.

Example

```
begin
       $filename := 'c:\pirch\logs\#pirch.log';
       if fileexists($filename) then
       begin
           if fileread($filename,-1,$s) then
              writeln($s)
           else
              writeln('unable to read a line');
       end
       else
           writeln('unable to find file: ',$filename);
end;
```

FILESIZE

Declaration
function filesize($filename : string) : value;

Description
Filesize returns a value indicating the size in bytes of a file indicated by $filename. Filesize makes no assumptions about the path for the file, therefore, you should use fully qualified filenames with this function. If the file is not found, or other error occurs during an attempt to retrieve the file size, this function will return -1.

Example

```
begin
       $filename := 'c:\pirch\logs\#pirch.log';
       if Fileexists($filename) then
       begin
          size := filesize($filename);
          writeln('#pirch log file size is ',size,' bytes');
       end
       else
          writeln('#pirch log file was NOT found');
end;
```

FILEWRITE

Declaration
function filewrite($filename : string; linenumber : value; $s : string) : value;

Description
Filewrite reads a string value to $filename. Linenumber may 0, in which case filewrite will append $s to the end of the file, or linenumber maybe any other value indicating the line number: If the value is negative, the existing line at position *linenumber* will be replaced; if the value is positive, the line will be inserted at position *linenumber*. For example, indicating linenumber as -1 would cause the first line to be replaced with the information in *$s*. If linenumber is set to 1, then *$*s would be inserted as the first line in the file and any other lines will be pushed downward. The return value is a boolean ordinal indicating whether or not the operation suceeded. (1 means the operation was successful, 0 means it failed) Filewrite makes no assumptions about the path for the file, therefore, you should use fully qualified filenames with this function.

Example

```
begin
       $filename := 'c:\pirch\logs\#pirch.log';
       if fileexists($filename) then
       begin
           if filewrite($filename,0,$s) then
              writeln('operation complete')
           else
              writeln('operation failed');
       end
       else
           writeln('unable to find file: ',$filename);
end;
```

HALT

Declaration
procedure Halt;

Description
Halt causes a script to terminate, and is useful when you want to stop processing depending on a given condition.

HASVOICE

Declaration
function hasvoice(channel : string; nickname : string) : boolean;

Description
Hasvoice returns true (1) if the *nickname* has a voice in a moderated in *channel*. Otherwise it returns false (0). While all channel operators, effectively have a voice in moderated channels, this function looks only at the +v mode state. If the operator has +v set then the function will return true, otherwise false. If testing sctrictly on who can and can not talk in a channel, use the Isop function in addition to the Hasvoice function.

Example

```
begin
       $nick := 'billy';
       $chan := '#mychannel';
       if hasvoice($chan,$nick) then
            writeln($nick,' can talk on ',$chan)
       else
            writeln($nick,' can not talk on ',$chan);
end;
```

INTTOSTR

Declaration
function inttostr(n : value) : string;

Description
Inttostr converts a value a string representation of the value.

Example

```
begin
	value := 12345;
	$s := inttostr(value);
end;
```

ISOP

Declaration
function isop(channel : string; nickname : string) : boolean;

Description
Isop returns true (1) if the *nickname* is a channel operator on *channel*. Otherwise it returns false (0).

Example

```
begin
       $nick := 'billy';
       $chan := '#mychannel';
       if isop($chan,$nick) then
            writeln($nick,' is an op on ',$chan)
       else
            writeln($nick,' is not an op on ',$chan);
end;
```

NICKCOUNT

Declaration
function nickcount(channelname : string) : value;

Description
Nickcount returns the number of names in the channel names list.

Example

```
begin
       $chan := '#pirch';
       for i := 1 to nickcount($chan)do
            writeln(nicklist($chan,i));
end;
```

NICKLIST

Declaration
function nicklist(channelname : string; index : integer) : string;

Description
Nicklist returns the nickname at position indicated by *index.* If index is greater than the number of total persons in the channel, nicklist returns an empty string. The first name is at index 1.

Example

```
begin
       $chan := '#pirch';
       for i := 1 to nickcount($chan)do
            writeln(nicklist($chan,i));
end;
```

RANDOM

Declaration
function random(range : value) : value

Description
Random returns a random number ranging from 0 to *range*.

Example

```
begin
       DiceA := random(5)+1;
       DiceB := random(5)+1;
       writeln('You rolled a ',DiceA,' and a ',DiceB);
end;
```

SNICKCOUNT

Declaration
function snickcount(channelname : string) : value;

Description
Scount returns the number of entries in the channel names list that are highlighted. If no names are highlighted then the function returns 0.

Example

```
begin
       $chan := '#pirch';
       for i := 1 to snickcount($chan)do
            writeln(snick($chan,i));
end;
```

SNICKLIST

Declaration
function snicklist(channelname : string; index : integer) : string;

Description
Snicklist returns the selected nickname at the position indicated by *index.* If index is greater than the number of total persons who's names are highlighted in the channel, snicklist returns an empty string. The first selected name is at index 1.

Example

```
begin
       $chan := '#pirch';
       for i := 1 to snickcount($chan)do
            writeln(snicklist($chan,i));
end;
```

STRCOPY

Declaration
function strcopy($s : string; index, len : value) : value;

Description
Strcopy is used to return a sub-string from within *$s*, starting at *index* and is at most *len* characters long. If there are fewer characters in *$s* from *index* to the end of *$s* than *len*, strcopy returns only as many characters as it can.

Example

```
begin
       $msg := 'this is a test';
       $newmsg := strcopy($msg,1,4);
       writeln($newmsg);
end;
```

STRDEL

Declaration
procedure strdel(var $s : string; index, len : value) : value;

Description
Strdel removes *len* characters from *$s* starting at index. *$s* must be a string variable. If there are fewer characters in *$s* from *index* to the end of *$s* than *len*, strdel removes only as many characters as it can.

Example

```
begin
       $msg := 'this is a test';
       strdel($msg,1,4);  {remove the word ‘this’}
       writeln($msg);
end;
```

STRINS

Declaration
procedure strdel($source : string; VAR $target : string; index : value) : value;

Description
Strins inserts *$source* into *$target* at position *index*.

Example

```
begin
       $msg := 'this is a test';
       strins('new ',$msg,11);
       writeln($msg);
end;
```

STRLEN

Declaration
function strlen($s : string) : value;

Description
Strlen returns the length of a string.

Example

```
begin
	$msg := 'this is a test';
	len := strlen($msg);
	writeln('The length of the message is ',len);
end;
```

STRLOWER

Declaration
function strlower($s : string) : string;

Description
Strlower returns the a copy of the string parameter with all letters converted to lowercase.

Example

```
begin
	$msg := 'This is a test';
	$msg := strlower($msg);
	writeln($msg);
end;
```

STRPOS

Declaration
function strpos($searchstr, $s : string) : value;

Description
Strpos returns the position at which *$searchstr* is found within *$s*. If *$searchstr* is not found, strpos returns 0.

Example

```
begin
	$msg := 'This is a test';
	index := strpos('is',$msg);
	writeln('“is” was found at position ',index);
end;
```

STRMATCH

Declaration
function strmatch($pattern, $s : string) : value;

Description
Strmatch returns a boolean ordinal indicating whether or not $s matches a pattern specified in $pattern. $pattern may use question marks (?) and/or asterisks (*) as wildcards.

Example

```
begin
       $msg := 'This is a test';
       $pattern := '*is*';
       if strmatch($pattern,$msg) then
           writeln('it matches')
       else
           writeln('no match');
end;
```

STRTOINT

Declaration
function strtoint($s : string) : value;

Description
Strtoint converts a string to a value. The string *$*s must contain only the character ’0’..’9’.

Example

```
begin
       $s := '12345';
       value := strtoint($s);
end;
```

STRTOKEN

Declaration
function strtoken(var $s : string) : string;

Description
Strtoken removes and returns the first word from within a *$s*.

Example

```
begin
	$msg := 'This is a test';
	$newmsg := strtoken($msg);
	writeln(newmsg,' : ',$msg);
end;
```

STRUPPER

Declaration
function strupper($s : string) : string;

Description
Strupper returns the a copy of the string parameter with all letters converted to uppercase.

Example

```
begin
	$msg := 'This is a test';
	$msg := strupper($msg);
	writeln($msg);
end;
```

WRITELN

Declaration
procedure writeln(v1,v2,...,vn);

Description
Use writeln to write information to a PIRCH window. By default, writeln will use the server window, from which the PIL script is associated, however this may changed by setting the PIRCH system variable DEBUGWIN with the /set command. The following demonstrates on way of redefining the target output window from the PIRCH command line.

```
/newwindow DEBUG
/set debugwin DEBUG
```

Separate multiple items with commas within writeln's parenthesis. This items may be of different types, i.e. strings, values or expressions.

Example

```
begin
	answer := 25 * 4;
	writeln('The result is ',answer);
end;
```