

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 their similarities, there are distinct 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 enclosed 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], and 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)

Cached scripts

Interpreted languages by nature and definitional generally recompile the source code each time a program is executed, however, PIRCH takes some liberties in this area that greatly enhance the efficiency of scripts, by saving a compiled script's binary image and variable address space within a cache. This feature saves a lot of overhead and processor cycles since PIRCH will only have to compile the source code one time during a session. In addition, since the variable address space for scripts are also cached, script variables can/will retain their values even after a script has terminated, and the variables will retain their values into the next time the script is executed, and also making use of /define for saving variables between script runs obsolete. However, that being said, it also becomes important that you initial variables appropriately.

/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 that's 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 let's 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 fashion 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 may 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)

Initializing Variables

You should give all variables appropriate initializing values using the assignment statement.

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

<code>\$activeurl</code>	string	the active url in your web browser
<code>\$activeurlpage</code>	string	the page title & address the active url in your browser
<code>\$activeurltitle</code>	string	the page title of the active url in your web browser
<code>\$activewin</code>	string	Returns the name of the active window
<code>\$audience</code>	string	returns the channel/nickname of the window from which the script is called
<code>\$cliptext</code>	string	Returns text from the system clipboard
<code>\$day</code>	string	Returns the current day of the week
<code>\$date</code>	string	Returns the current system date
<code>\$ddetag</code>	string	Returns the ddetag for the connection
<code>\$time</code>	string	Returns the current system time
<code>\$getpath</code>	string	your default dcc download path
<code>\$host</code>	string	returns your current local host name
<code>\$identd</code>	string	returns the Ident daemon userid
<code>\$ip</code>	string	returns your IP address in 4 octet form
<code>\$me</code>	string	returns your current nickname
<code>\$netid</code>	string	returns the server's network id (if assigned)
<code>\$logpath</code>	string	your log file directory
<code>\$mode</code>	string	returns the channel mode

\$members	string	(only if called from a channel) returns the channel member count
\$mediafile	string	(only if called from a channel) returns the filename loaded into media player (.mid,.avi,.rmi)
\$pirchpath	string	the directory from which PIRCH is running from
\$sendpath	string	your default dcc upload path
\$server	string	Returns the name of the server to which the windows from which the script is called is associated
\$title	string	Caption form PIRCH's mail window
\$topic	string	returns the channel topic (only if called from a channel)
\$usermode	string	returns the usermode (wallops, server messages, invisibility)
\$version	string	returns the PIRCH version you are running

Global User Defined Variables

Global user defined variables are those that you create with the /DEFINE command. PIL scripts can access these global variables but you need to take one special precaution. When a PIL script is being compiled and encounters a token, PIRCH will check to see if the token is a predefined function/procedure, operator or a global variable. If the compiler does not find either of these conditions to be true, it will assume the token is a variable local in scope to the script. Therefore, you must be sure that variable names you intend to use a global variables actually exists as global variables before calling /runscript.

Procedures and Functions

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

ADDRESSMASK
BREATHE
CHAR
COMMAND
CREATGADGET
CREATWINDOW
DESTROYWINDOW
DDEPOKE
DDEREQUEST
FILEEXISTS
FILEREAD
FILESIZE
FILEWRITE
GETWINDOWTEXT
HALT
HASVOICE
INTTOSTR
IPTOLONGIP
ISOP
LOADFROMFILE
LONGIPTOIP
NICKCOUNT
NICKLIST
ORD
RANDOM
RECTANGLE
RGB
SAVETOFILE
SETBRUSH
SETPEN
SETWINDOWATTR

SETWINDOWTEXT
SNICKCOUNT
SNICKLIST
STRCOPY
STRDEL
STRINS
STRLEN
STRPOS
STRMATCH
STRUPPER
STRLOWER
STRTOKEN
STRTOINT
STRTOUNIXTIME
TEXTOUT
UNIXTIME
UNIXTIMETOSTR
USERLEVEL
WHEREIS
WRITELN

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)

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 long 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

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');
```

PIL Procedures and Functions

Procedures & Functions

PIL currently doesn't 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.

- ADDRESSMASK
- BREATHE
- CHAR
- COMMAND
- CREATGADGET
- CREATWINDOW
- DESTROYWINDOW
- DDEPOKE
- DDEREQUEST
- FILEEXISTS
- FILEREAD
- FILESIZE
- FILEWRITE
- GETWINDOWTEXT
- HALT
- HASVOICE
- INTTOSTR
- IPTOLONGIP
- ISOP
- LOADFROMFILE
- LONGIPTOIP
- NICKCOUNT
- NICKLIST
- ORD
- RANDOM
- RECTANGLE
- RGB
- SAVETOFILE
- SETBRUSH
- SETPEN
- SETWINDOWATTR

SETWINDOWTEXT
SNICKCOUNT
SNICKLIST
STRCOPY
STRDEL
STRINS
STRLEN
STRPOS
STRMATCH
STRUPPER
STRLOWER
STRTOKEN
STRTOINT
STRTOUNIXTIME
TEXTOUT
UNIXTIME
UNIXTIMETOSTR
USERLEVEL
WHEREIS
WRITELN

ADDRESSMASK

Declaration

procedure AddressMask(Address : string; MaskMode : value) : string;

Description

Converts an address in irc format (nick!ident@server) to a address mask according to the maskmode parameter as follows:

0 = User	nick!ident@port.server
1 = IdentPortServer	*!ident@port.server
2 = IdentServer	*!ident@*.server
3 = PortServer	*!*@port.server
4 = NickOnly	nick!*@*
5 = serverOnly	*!*@*.server

Example

```
begin
  $address := 'billg!im2rich@ppp666.microsoft.com';
  writeln('the mask is ',addressmask($address));
end;
```

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 interpreter 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;
```

See also: [ord](#)

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


CREATEWINDOW

Declaration

procedure createwindow(parent : value; left,right,width,height : value);

Description

Createwindow creates a new blank window and gives it the position and dimensions specified by left, top and width, height respectively. All newly created windows are initially invisible and must be made visible by calling SetWindowAttr(Visible,true). This will allow you to create the window and any required gadgets without screen flicker.

CREATEGADGET

Declaration

procedure creategadget(parent,gadgettype,left,right,width,height : value);

Description

Creategadget creates a gadget control within a parent window identified by *parent*. Gadgettype may be any of the values from the table below, identify the type of control to be created. Upon creation, all gadgets will have their position assigned within the parent window based on the left, right, top & bottom parameters. By default, PIL gadgets contain no text labels, these should be set with a call to SetWindowText.

Type	Value	Description
Label	1	Single line label control
Edit	2	Single line edit control
Memo	3	Multi-line edit control
Button	4	Standard pushbutton
CheckBox	5	Standard checkbox
RadioButton	6	Standard Radiobutton
ListBox	7	Multi-line list
Combo	8	Dropdown combo box
Group	9	Standard Group box
Panel	10	Bevelled panel
Card	11	Playing card
Grid	12	Grid (multi-column/row) (not implemented)
TextScroller	13	PIRCH Text Scroll Control
Image	14	Control which displays bitmaps, metafiles ,icon

DDEPOKE

Declaration

```
procedure ddepoke(service,topic,item,Data : string);
```

Description

Performs a DDE transaction of XTYP_POKE to the topic and service.

DDEREQUEST

Declaration

```
function DDERequest(service,topic,item : string) : string;
```

Description

Performs a DDE transaction of XTYP_REQUEST to the topic and service and returns the information requested from the service DDE server.

DESTROYWINDOW

Declaration

```
procedure destroywindow(windowid : value);
```

Description

Destroy window destroys the window defined by windowid. This procedure can be used to free the memory occupied by a window or gadget. However it should only be used when you need to close a window from within your code.

By default, PIRCH will automatically destroy the window for you when a user closes a window and the use of the destroywindow() procedure is not necessary in such cases.

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 succeeded. (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
    $filename := 'c:\purch\logs\#purch.log';
    if fileexists($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;
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 succeeded. (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;
```

GETWINDOWATTR

Declaration

function getwindowattr(gadgetid,attribute : value) : value;

Description

GetWindowAttr retrieves the attribute value specified by *attribute* for the indicated *windowid*. *Windowid* may be either a window or gadget control.

Ordinal	Gets attribute	value range
1	background color	0..16777215
2	font/text color	0..16777215
3	alignment of the control	0..5
4	alignment of text in the control	0..2
5	border	0/1
6	enabled/disabled	0/1
7	visible/hidden	0/1

The table below demonstrates which attribute apply to which controls.

Label	1	2	3	4	5	6	7	
Edit	X	X		X		X	X	
Memo	X	X	X			X	X	
Button		X				X	X	
Checkbox	X	X				X	X	
Radio Button	X	X				X	X	
List Box	X	X		X	X	X	X	
Combo Box		X				X	X	
Group Box		X	X			X	X	
Panel	X	X	X	X	X	X	X	
Card						X	X	
Grid			(not implemented yet)					
TextScroller					X		X	

The following two attributes apply to only the card control

AttrCardValue	20	card index of card control	card
AttrCardState	21	card is selected/unselected	card

Using MNUEMONICS for more readable code

The following block of code can be copied into your script allowing you to use the following mnuemonics in place of numeric values for gadgettypes, common attributes, and attribute values.

```
{ Attribute mnuemonics for SetWindowAttr/GetWindowAttr }
```

```
AttrColor      := 1;  
AttrTextColor  := 2;  
AttrAlign      := 3;  
AttrAlignment  := 4;  
AttrBorder     := 5;  
AttrEnabled    := 6;  
AttrVisible    := 7;  
AttrCard       := 20;  
AttrCardState  := 21;
```

```
{ Value mnuemonics for AttrAlign }
```

```
alNone         := 0;  
alTop          := 1;  
alBottom       := 2;  
alLeft         := 3;  
alRight        := 4;
```

```
alClient      := 5;

{ Value mnemonics for AttrAlignment}
taLeftJustify := 0;
taRightJustify := 1;
taCenter      := 2;

{ Value mnemonics for CreateGadget (gadgettype) }
PIL_Label     := 1;
PIL_Edit      := 2;
PIL_Memo      := 3;
PIL_Button    := 4;
PIL_ChkBox    := 5;
PIL_Radio     := 6;
PIL_ListBox   := 7;
PIL_Combo     := 8;
PIL_Group     := 9;
PIL_Panel     := 10;
PIL_Card      := 11;
PIL_Grid      := 12;
PIL_TextScroller := 13;
```

GETWINDOWTEXT

Declaration

```
function getwindowtext(windowid : value) : string;
```

Description

Retrieves the text/caption of a the window or gadget defined by windowid. *Windowid* may be either a window or gadget control.

Applies to windows and the following gadgets

- Edit
- Memo
- Button
- Checkbox
- Radio Button
- Combo Box
- Group Box
- Panel

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 strictly 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;
```

INIREAD

Declaration

```
function iniread($filename, $section, $key, $default : string) : string;
```

Description

IniRead attempts to read a string value from a .ini file indicated by \$section and \$key. If the operation fails to find either the section or the key within the section, iniread will return the string indicated by \$default. If \$filename contains no path, iniread will look for the file in the \windows directory.

Example

```
begin
    $filename := 'c:\pirch32\myini';
    $value := iniread($filename, 'mybot', 'botnick', 'notfound');
    writeln('The value of key is', $value);
end;
```

INIWRITE

Declaration

```
procedure iniwrite($filename, $section, $key, $value : string);
```

Description

IniWrite attempts to write a string value to the .ini file indicated by \$filename. If \$filename contains no path, iniwrite will store the ini file in the \windows directory.

Example

```
begin
    $filename := 'c:\pirch32\myini';
    $value := 'coffeebot';
    iniwrite($filename, 'mybot', 'botnick', $value);
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;
```

IPTOLONGIP

Declaration

```
function IpToLongIp(IpAddress : string) : string;
```

Description

Converts a dotted decimal IP (internet protocol) address to double word form in network byte order. ie 206.9.80.39 to 3456716839. Return value is in string form.

Example

```
begin
    $mylongip := iptolongip($ip);
    writeln('my ip ', $ip, ' in long form is ', $mylongip);
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;
```

LINE

Declaration

procedure line(image : value; X1,Y1,X2,Y2 : value);

Description

Draws a line from the coordinates x1,y1 to x2,y2 using the current pen color, style & mode. Image is created with creategadget using 14 as the class type.

Applies to

Image

See Also: lineto, setpen, setbrush

LINETO

Declaration

procedure lineto(image : value; X,Y : value);

Description

Draws a line from the logical pen position (set by [moveto](#)) to the coordinates at x,y using the current pen color, style & mode. Image is created with [creategadget](#) using 14 as the class type.

Applies to

Image

See Also: [moveto](#), [line](#), [setpen](#), [setbrush](#)

LISTAPPEND

Declaration

```
function listappend(listid : value; item : string) : value;
```

Description

List append adds item to the end of the list control identified by *listid*. It returns the index at which the item was inserted.

Applies to

ListBoxes, ComboBoxes and TextScrollers

Example

LISTCLEAR

Declaration

procedure listclear(listid : value);

Description

ListClear removes all items from a list control identified by *listid*.

Applies to

ListBoxes, ComboBoxes and TextScrollers

Example

LISTCOUNT

Declaration

function listcount(listid : value) : value;

Description

ListCount returns the number of items contain in a list control identified by *listid*.

Applies to

ListBoxes, Comboboxes

Example

LISTDELETE

Declaration

function listdelete(listid : value; index : value) : value;

Description

ListDelete removes the item at position index from a list control identified by *listid*. Returns 1 if succesful, 0 if not. The first item in the list is at index 1.

Applies to

ListBoxes, Comboboxes

LISTGET

Declaration

function listget(listid : value; index : value) : string;

Description

ListGet returns the item at position *index* from a list control identified by *listid*. The first item in the list is at index 1. If index is not valid, listget will return an empty string.

Applies to

ListBoxes, Comboboxes

LISTINSERT

Declaration

function listinsert(listid : value; index : value; item : string) : value;

Description

ListInsert inserts *item* at position *index* in a list control identified by *listid*. The first item in the list is at index 1. If listinsert is succesful it returns 1, otherwise it returns 0.

Applies to

ListBoxes, Comboboxes

LISTSEARCH

Declaration

function listsearch(listid : value; item : string) : value;

Description

ListSearch search the contents of a list control identified by *listid* looking for *item*. Listsearch will return the index at which the item was found, or 0 if it was not found.

Applies to

ListBoxes, Comboboxes

LISTSELECTION

Declaration

function listselection(listid : value) : value;

Description

ListSelection returns the highlighted item *index* from a list control identified by *listid*. The first item in the list is at index 1. If no item is highlighted the function returns 0.

Applies to

ListBoxes

LOADFROMFILE

Declaration

function loadfrom(control : value; filename : string) : value;

Description

Loadfromfile loads a disk file into memory for the particular gadget type.

This function is only applicable to the following pil gui gadget types:

Memos

Loads a text file into an editable memo gadget

Listboxes

Loads a text file, each line becomes an item in the list

Images

Loads a image file which may be one of the following types

bitmaps	.bmp files
icons	.ico files
metafiles	.wmf files

Applies to

Memos, ListBoxes, and Images

See also: [SaveToFile](#)

LONGIPTOIP

Declaration

```
function longiptoip(longip : string) : string;
```

Description

LongIpToIP converts a long IP from double word network byte order to dotted decimal form.

MOVETO

Declaration

procedure moveto(image : value; X,Y : value);

Description

Set the logical pen positions to the coordinates at x,y. Image is created with [creategadget](#) using 14 as the class type.

Applies to

Image

See Also [lineto](#)

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;
```

ORD

Declaration

```
function ord(character : string) : value;
```

Description

Ord returns the ordinal value of an ascii character.

Example

```
begin
    $char := '#';
    writeln('Ordinal value of ', $char, ' character is ', ord($char));
end;
```

See also: [char](#)

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;
```

RECTANGLE

Declaration

procedure rectangle(image : value; X1,Y1,X2,Y2 : value);

Description

Draws a rectangle using the coordinates provided by X1..Y2 using the current pen color, style & mode, and filling with the current brush color & style. Image is created with [creategadget](#) using 14 as the class type.

Applies to

Image

See Also: [setpen](#), [setbrush](#)

RGB

Declaration

```
function rgb(redval,greenval,blueval : value) : value;
```

Description

RGB returns a 32 bit color value based on the intensities of redval,greenval and blueval which must be in the range of 0..255

Example

```
begin
    writeln('Bright red has an RGB value of ',rgb(255,0,0));
end;
```

SAVETOFILE

Declaration

function savetofile(control : value; filename : string) : value;

Description

Saves a disk file for the particular gadget's contents.

This function is only applicable to the following pil gui gadget types:

Memos

Saves a text file of a memo gadget's contents

Listboxes

Saves a text file, each line of the list gadget becomes a new line in the text file

Images

Saves a image file which may be one of the following types

bitmaps .bmp files

icons .ico files

metafiles .wmf files

Applies to

Memos, ListBoxes, and Images

See also: [LoadFromFile](#)

SETBRUSH

Declaration

procedure SetBrush(image : value; style, color : value);

Description

Sets the currently active brush attributes for an image object. Image is created with with creategadget using 14 as the class type.

Style may be one of the following values

0	Solid
1	Clear
2	Horizontal
3	Vertical
4	FDiagonal
5	BDiagonal
6	Cross
7	DiagCross

SETPEN

Declaration

procedure SetPen(image : value; style, color, mode : value);

Description

Sets the currently active pen attributes for an image object. Image is created with [creategadget](#) using 14 as the class type.

Style may be one of the following

0	Solid
1	Dash
2	Dot
3	DashDot
4	DashDotDot
5	Clear
6	InsideFrame

Color may a 32 bit RGB value. See function [RGB](#).

Mode may be one of the following - consult the Microsoft Windows Programmers Reference for meaning of raster mode

0	Blackness
1	Whiteness
2	Nop
3	Not
4	Copy
5	NotCopy
6	MergePenNot
7	MaskPenNot
8	MergeNotPen
9	MaskNotPen
10	Merge
11	NotMerge
12	Mask
13	NotMask
14	Xor
15	NotXor

SETEVENT

Declaration

procedure setevent(windowid : value; eventid: value; eventhandler : string);

Description

SetEvent installs an event handler for a particular window or control gadget identified by *windowid*. Currently only 2 events are implemented. *EventHandler* is a string representing an PIRCH or IRC /command to be executed when the action occurs. The term event in this context is independant of IRC event triggers like ON TEXT, rather these events relate to interaction with windows/controls created in PIL scripts.

Applies to windows and the following gadgets according to the following table

For Windows

EventId	Occurs When
1	The window is closed

For Buttons

EventId	Occurs When
1	The button is clicked

See the `uwho.pil` script for an example.

SETWINDOWATTR

Declaration

procedure setwindowattr(windowid,attribute,value : value);

Description

Set the specified *attribute* for *windowid*. *Windowid* may be either a window or gadget control.

Ordinal	Sets attribute	value range
1	background color	0..16777215
2	font/text color	0..16777215
3	alignment of the control	0..5
4	alignment of text in the control	0..2
5	border	0/1
6	enabled/disabled	0/1
7	visible/hidden	0/1
8	checked/unchecked	0/1
9	default button (traps enter key)	0/1
10	cancel button (traps esc key)	0/1

The table below demonstrates which attribute apply to which controls.

Label	1	2	3	4	5	6	7	8	9	10
Edit		X	X		X		X	X		
Memo	X	X	X			X	X			
Button		X				X	X		X	X
Checkbox		X	X				X	X	X	
Radio Button	X	X				X	X			
List Box	X	X		X	X	X	X			
Combo Box		X				X	X			
Group Box		X	X			X	X			
Panel	X	X	X	X	X	X	X			
Card						X	X			
Grid		(not implemented yet)								
TextScroller					X		X			

The following attributes apply only to parent windows (created with CreateWindow)

* denotes default value

Name	Ordinal	Allowed Values
AttrWinStyle	11	Stay on Top (0) * MDI child (1)
AttrWinBorder	12	None (0) Single (1) Sizeable (2) Dialog (3) ToolWindow (4) * SizeToolWin (5)

The following attributes apply only to image controls

AttrImageCenter	13	forces image centering*
AttrAutoSize	14	forces image control to resize to contents*

The following two attributes apply to only the card control

AttrCardValue	20	card index of card control	card
AttrCardState	21	card is selected/unselected	card

Using MNUEMONICS for more readable code

The following block of code can be copied into your script allowing you to use the following mnuemonics in

place of numeric values for gadgettypes, common attributes, and attribute values.

```
{ Attribute mnemonics for SetWindowAttr/GetWindowAttr }
```

```
AttrColor      := 1;  
AttrTextColor  := 2;  
AttrAlign      := 3;  
AttrAlignment  := 4;  
AttrBorder     := 5;  
AttrEnabled    := 6;  
AttrVisible    := 7;  
AttrChecked    := 8;  
AttrDefButton  := 9;  
AttrCancelButton := 10;  
AttrWinStyle   := 11;  
AttrWinBorder  := 12;  
AttrImageCenter := 13;  
AttrAutosize   := 14;  
AttrCard       := 20;  
AttrCardState  := 21;
```

```
{ Value mnemonics for AttrAlign }
```

```
alNone      := 0;  
alTop       := 1;  
alBottom    := 2;  
alLeft      := 3;  
alRight     := 4;  
alClient    := 5;
```

```
{ Value mnemonics for AttrAlignment }
```

```
taLeftJustify := 0;  
taRightJustify := 1;  
taCenter      := 2;
```

```
{ Value mnemonics for CreateGadget (gadgettype) }
```

```
PIL_Label      := 1;  
PIL_Edit       := 2;  
PIL_Memo       := 3;  
PIL_Button     := 4;  
PIL_ChkBox     := 5;  
PIL_Radio      := 6;  
PIL_ListBox    := 7;  
PIL_Combo      := 8;  
PIL_Group      := 9;  
PIL_Panel      := 10;  
PIL_Card       := 11;  
PIL_Grid       := 12;  
PIL_TextScroller := 13;  
PIL_Image      := 14;
```

SETWINDOWTEXT

Declaration

procedure setwindowtext(windowid : value; text : string);

Description

Set the text/caption of a the window or gadget defined by windowid.

Applies to windows and the following gadgets

Edit
Memo
Button
Checkbox
Radio Button
Combo Box
Group Box
Panel

SNICKCOUNT

Declaration

```
function snickcount(channelname : string) : value;
```

Description

Snickcount 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
    $chan := '#purch';
    for i := 1 to snickcount($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
    $chan := '#purch';
    for i := 1 to snickcount($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;
```


STRTOUNIXTIME

Declaration

```
function StrToUnixTime(Format: string; UnixTime: Value): value;
```

Description

StrToUnixTime converts a string in the format of MM/DD/YY HH:MM:SS to a UnixTime value. Specifying AM or PM as part of the time is optional, as are the seconds. You should use 24-hour time (7:45 PM is entered as 19:45, for example) if you don't specify AM or PM.

```
begin
    $christmas := '12/25/96 12:00 PM';
    val := strtounixtime($christmas);
    writeln($christmas, ' in unixtime is ',val);
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;
```

TEXTOUT

Declaration

procedure TextOut(image : value; X, Y, Alignment : value; Text : string);

Description

Draws text to an image object using coordinates X & Y.

Alignment specifies the text alignment by using a mask of the values in the following list. Only one flag can be chosen from those that affect horizontal and vertical alignment. In addition, only one of the two flags that alter the current position can be chosen. Flags should be added together or or'd bitwise using bitor.

0	NOUPDATECP
1	UPDATECP
0	LEFT
2	RIGHT
6	CENTER
0	TOP
8	BOTTOM
24	BASELINE
256	RTLREADING

Applies to
Image

UNIXTIME

Declaration

function unixtime : value;

Description

Unixtime returns a numeric value that represents the current date & time as a unix timestamp.

Many IRC functions use unix timestamps to represent dates/times.

UNIXTIMETOSTR

Declaration

```
function unixtimetostr(unixtime : value) : string;
```

Description

UnixTimeToStr formats the date and time value given by UnixTime using the format given by Format. Format specifiers are case insensitive. If the string given by the Format parameter is empty, the date and time value is formatted as if a 'c' format specifier had been given. The following formatting specifiers are supported:

Code Displays

c	Displays the date using the format given by the ShortDateFormat global variable, followed by the time using the format given by the LongTimeFormat global variable. The time is not displayed if the fractional part of the DateTime value is zero.
d	Displays the day as a number without a leading zero (1-31).
dd	Displays the day as a number with a leading zero (01-31).
ddd	Displays the day as an abbreviation (Sun-Sat)
dddd	Displays the day as a full name (Sunday-Saturday)
dddddd	Displays the date using standard Short Date Format
dddddd	Displays the date using standard Long Date Format
m	Displays the month as a number without a leading zero (1-12). If the m specifier immediately follows an h or hh specifier, the minute rather than the month is displayed.
mm	Displays the month as a number with a leading zero (01-12). If the mm specifier immediately follows an h or hh specifier, the minute rather than the month is displayed.
mmm	Displays the month as an abbreviation (Jan-Dec)
mmmm	Displays the month as a full name (January-December)
yy	Displays the year as a two-digit number (00-99).
yyyy	Displays the year as a four-digit number (0000-9999).
h	Displays the hour without a leading zero (0-23).
hh	Displays the hour with a leading zero (00-23).
n	Displays the minute without a leading zero (0-59).
nn	Displays the minute with a leading zero (00-59).
s	Displays the second without a leading zero (0-59).
ss	Displays the second with a leading zero (00-59).
t	Displays the time using standard Short Time Format
tt	Displays the time using standard Long Time Format
am/pm	Uses the 12-hour clock for the preceding h or hh specifier, and displays 'am' for any hour before noon, and 'pm' for any hour after noon. The am/pm specifier can use lower, upper, or mixed case, and the result is displayed accordingly.
a/p	Uses the 12-hour clock for the preceding h or hh specifier, and displays 'a' for any hour before noon, and 'p' for any hour after noon. The a/p specifier can use lower, upper, or mixed case, and the result is displayed accordingly.
ampm	Uses the 12-hour clock for the preceding h or hh specifier, and displays the contents of the TimeAMString global variable for any hour before noon, and the contents of the TimePMString global variable for any hour after noon.
/	Displays the date separator character given by the DateSeparator global variable.
:	Displays the time separator character given by the TimeSeparator global variable.
'xx'/'"xx"	Characters enclosed in single or double quotes are displayed as-is, and do not affect formatting.

USERLEVEL

Declaration

```
function UserLevel(Userid : string; Channel : string) : string;
```

Description

Finds the events user level of a user for a channel UserId should be in the form of nickname!ident@host

Example

```
begin
    $address := 'billg!im2rich@ppp666.microsoft.com';
    $channel := '#pirch'
    $level := userlevel($address,$channel);
    writeln($address,' matched ', $level);
end;
```

WHEREIS

Declaration

```
function Whereis(filename,RootPath : string) : string;
```

Description

Whereis searches rootpath and any subdirectories below rootpath for a specified file and returns the first matching file it finds. If no matching file is found then the return string will be empty.

Example

```
begin
  $searchfile := 'pirch32.exe';
  $found := whereis($searchfile,'C:\');
  if strlen($found) = 0 then
    writeln($searchfile,' was not found')
  else
    writeln($searchfile,' was found at ', $found);
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 be changed by setting the PIRCH system variable DEBUGWIN with the /set command. The following demonstrates one 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. These items may be of different types, i.e. strings, values or expressions.

Example

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


