Perl Programmers Reference Guide

Win32, 5.6.1
24−Apr−2001

"There’s more than one way to do it."

−− Larry Wall, Author of the Perl Programming Language

blank

NAME

Win32::ChangeNotify − Monitor events related to files and directories

SYNOPSIS

```
        require Win32::ChangeNotify;

        $notify = Win32::ChangeNotify->new($Path,$WatchSubTree,$Events);
        $notify->wait or warn "Something failed: $!\n";
        # There has been a change.
```

DESCRIPTION

This module allows the user to use a Win32 change notification event object from Perl. This allows the Perl program to monitor events relating to files and directory trees.

The `wait` method and `wait_all` & `wait_any` functions are inherited from the *"Win32::IPC"* module.

Methods

`$notify` = Win32::ChangeNotify−new(`$path, $subtree, $filter`)

Constructor for a new ChangeNotification object. `$path` is the directory to monitor. If `$subtree` is true, then all directories under `$path` will be monitored. `$filter` indicates what events should trigger a notification. It should be a string containing any of the following flags (separated by whitespace and/or |).

```
  ATTRIBUTES    Any attribute change
  DIR_NAME      Any directory name change
  FILE_NAME     Any file name change (creating/deleting/renaming)
  LAST_WRITE    Any change to a file’s last write time
  SECURITY      Any security descriptor change
  SIZE          Any change in a file’s size
```

(`$filter` can also be an integer composed from the `FILE_NOTIFY_CHANGE_*` constants.)

`$notify−`close

Shut down monitoring. You could just `undef $notify` instead (but `close` works even if there are other copies of the object). This happens automatically when your program exits.

`$notify−`reset

Resets the ChangeNotification object after a change has been detected. The object will become signalled again after the next change. (It is OK to call this immediately after `new`, but it is not required.)

`$notify−`wait

See *"Win32::IPC"*. Remember to call `reset` afterwards if you want to continue monitoring.

Deprecated Functions and Methods

Win32::ChangeNotify still supports the ActiveWare syntax, but its use is deprecated.

FindFirst(`$Obj,$PathName,$WatchSubTree,$Filter`)

Use

```
  $Obj = Win32::ChangeNotify->new($PathName,$WatchSubTree,$Filter)
```

instead.

`$obj−`FindNext()

Use `$obj->reset` instead.

`$obj-`Close()

Use `$obj->close` instead.

AUTHOR

Christopher J. Madsen <***chris_madsen@geocities.com***>

Loosely based on the original module by ActiveWare Internet Corp., ***http://www.ActiveWare.com***

NAME

Win32::Clipboard − Interaction with the Windows clipboard

SYNOPSIS

```
use Win32::Clipboard;

$CLIP = Win32::Clipboard();

print "Clipboard contains: ", $CLIP->Get(), "\n";

$CLIP->Set("some text to copy into the clipboard");

$CLIP->Empty();

$CLIP->WaitForChange();
print "Clipboard has changed!\n";
```

DESCRIPTION

This module lets you interact with the Windows clipboard: you can get its content, set it, empty it, or let your script sleep until it changes. This version supports 3 formats for clipboard data:

- text (`CF_TEXT`)

  The clipboard contains some text; this is the **only** format you can use to set clipboard data; you get it as a single string.

  Example:

  ```
  $text = Win32::Clipboard::GetText();
  print $text;
  ```

- bitmap (`CF_DIB`)

  The clipboard contains an image, either a bitmap or a picture copied in the clipboard from a graphic application. The data you get is a binary buffer ready to be written to a bitmap (BMP format) file.

  Example:

  ```
  $image = Win32::Clipboard::GetBitmap();
  open    BITMAP, ">some.bmp";
  binmode BITMAP;
  print   BITMAP $image;
  close   BITMAP;
  ```

- list of files (`CF_HDROP`)

  The clipboard contains files copied or cutted from an Explorer−like application; you get a list of filenames.

  Example:

  ```
  @files = Win32::Clipboard::GetFiles();
  print join("\n", @files);
  ```

REFERENCE

All the functions can be used either with their full name (eg. **Win32::Clipboard::Get**) or as methods of a `Win32::Clipboard` object. For the syntax, refer to */SYNOPSIS* above. Note also that you can create a clipboard object and set its content at the same time with:

```
$CLIP = Win32::Clipboard("blah blah blah");
```

or with the more common form:

```
$CLIP = new Win32::Clipboard("blah blah blah");
```

If you prefer, you can even tie the Clipboard to a variable like this:

```
tie $CLIP, 'Win32::Clipboard';

print "Clipboard content: $CLIP\n";

$CLIP = "some text to copy to the clipboard...";
```

In this case, you can still access other methods using the `tied()` function:

```
tied($CLIP)->Empty;
print "got the picture" if tied($CLIP)->IsBitmap;
```

`Empty()`

Empty the clipboard.

=for html <P

`EnumFormats()`

Returns an array of identifiers describing the format for the data currently in the clipboard. Formats can be standard ones (described in the */CONSTANTS* section) or application−defined custom ones. See also `IsFormatAvailable()`.

=for html <P

`Get()`

Returns the clipboard content; note that the result depends on the nature of clipboard data; to ensure that you get only the desired format, you should use `GetText()`, `GetBitmap()` or `GetFiles()` instead. `Get()` is in fact implemented as:

```
if(    IsBitmap() ) { return GetBitmap(); }
elsif( IsFiles()  ) { return GetFiles();  }
else                { return GetText();   }
```

See also `IsBitmap()`, `IsFiles()`, `IsText()`, `EnumFormats()` and `IsFormatAvailable()` to check the clipboard format before getting data.

=for html <P

GetAs(FORMAT)

Returns the clipboard content in the desired FORMAT (can be one of the constants defined in the */CONSTANTS* section or a custom format). Note that the only meaningful identifiers are `CF_TEXT`, `CF_DIB` and `CF_HDROP`; any other format is treated as a string.

=for html <P

`GetBitmap()`

Returns the clipboard content as an image, or `undef` on errors.

=for html <P

`GetFiles()`

Returns the clipboard content as a list of filenames, or `undef` on errors.

=for html <P

GetFormatName(FORMAT)

Returns the name of the specified custom clipboard format, or `undef` on errors; note that you cannot get the name of the standard formats (described in the */CONSTANTS* section).

=for html <P

`GetText()`

Returns the clipboard content as a string, or `undef` on errors.

=for html <P

`IsBitmap()`

Returns a boolean value indicating if the clipboard contains an image. See also `GetBitmap()`.

=for html <P

`IsFiles()`

Returns a boolean value indicating if the clipboard contains a list of files. See also `GetFiles()`.

=for html <P

IsFormatAvailable(FORMAT)

Checks if the clipboard data matches the specified FORMAT (one of the constants described in the */CONSTANTS* section); returns zero if the data does not match, a nonzero value if it matches.

=for html <P

`IsText()`

Returns a boolean value indicating if the clipboard contains text. See also `GetText()`.

=for html <P

Set(VALUE)

Set the clipboard content to the specified string.

=for html <P

WaitForChange([TIMEOUT])

This function halts the script until the clipboard content changes. If you specify a `TIMEOUT` value (in milliseconds), the function will return when this timeout expires, even if the clipboard hasn‘t changed. If no value is given, it will wait indefinitely. Returns 1 if the clipboard has changed, `undef` on errors.

CONSTANTS

These constants are the standard clipboard formats recognized by Win32::Clipboard:

```
        CF_TEXT             1
        CF_DIB              8
        CF_HDROP            15
```

The following formats are **not recognized** by Win32::Clipboard; they are, however, exported constants and can eventually be used with the `EnumFormats()`, `IsFormatAvailable()` and `GetAs()` functions:

```
        CF_BITMAP           2
        CF_METAFILEPICT     3
        CF_SYLK             4
        CF_DIF              5
        CF_TIFF             6
        CF_OEMTEXT          7
        CF_PALETTE          9
        CF_PENDATA          10
        CF_RIFF             11
        CF_WAVE             12
        CF_UNICODETEXT      13
        CF_ENHMETAFILE      14
        CF_LOCALE           16
```

AUTHOR

Aldo Calpini <***dada@perl.it***

Original XS porting by Gurusamy Sarathy <***gsar@activestate.com***.

NAME

Win32::Console − Win32 Console and Character Mode Functions

DESCRIPTION

This module implements the Win32 console and character mode functions. They give you full control on the console input and output, including: support of off−screen console buffers (eg. multiple screen pages)

- reading and writing of characters, attributes and whole portions of the screen
- complete processing of keyboard and mouse events
- some very funny additional features :)

Those functions should also make possible a port of the Unix‘s curses library; if there is anyone interested (and/or willing to contribute) to this project, e−mail me. Thank you.

REFERENCE

Methods

Alloc

Allocates a new console for the process. Returns `undef` on errors, a nonzero value on success. A process cannot be associated with more than one console, so this method will fail if there is already an allocated console. Use Free to detach the process from the console, and then call Alloc to create a new console. See also: `Free`

Example:

```
$CONSOLE−>Alloc();
```

Attr [attr]

Gets or sets the current console attribute. This attribute is used by the Write method.

Example:

```
$attr = $CONSOLE−>Attr();
$CONSOLE−>Attr($FG_YELLOW | $BG_BLUE);
```

Close

Closes a shortcut object. Note that it is not "strictly" required to close the objects you created, since the Win32::Shortcut objects are automatically closed when the program ends (or when you elsehow destroy such an object).

Example:

```
$LINK−>Close();
```

Cls [attr]

Clear the console, with the specified *attr* if given, or using ATTR_NORMAL otherwise.

Example:

```
$CONSOLE−>Cls();
$CONSOLE−>Cls($FG_WHITE | $BG_GREEN);
```

Cursor [x, y, size, visible]

Gets or sets cursor position and appearance. Returns `undef` on errors, or a 4−element list containing: *x*, *y*, *size*, *visible*. *x* and *y* are the current cursor position; ...

Example:

```
($x, $y, $size, $visible) = $CONSOLE−>Cursor();

# Get position only
```

```
($x, $y) = $CONSOLE->Cursor();

$CONSOLE->Cursor(40, 13, 50, 1);

# Set position only
$CONSOLE->Cursor(40, 13);

# Set size and visibility without affecting position
$CONSOLE->Cursor(-1, -1, 50, 1);
```

Display

Displays the specified console on the screen. Returns `undef` on errors, a nonzero value on success.

Example:

```
$CONSOLE->Display();
```

FillAttr [attribute, number, col, row]

Fills the specified number of consecutive attributes, beginning at *col*, *row*, with the value specified in *attribute*. Returns the number of attributes filled, or `undef` on errors. See also: `FillChar`.

Example:

```
$CONSOLE->FillAttr($FG_BLACK | $BG_BLACK, 80*25, 0, 0);
```

FillChar char, number, col, row

Fills the specified number of consecutive characters, beginning at *col*, *row*, with the character specified in *char*. Returns the number of characters filled, or `undef` on errors. See also: `FillAttr`.

Example:

```
$CONSOLE->FillChar("X", 80*25, 0, 0);
```

Flush

Flushes the console input buffer. All the events in the buffer are discarded. Returns `undef` on errors, a nonzero value on success.

Example:

```
$CONSOLE->Flush();
```

Free

Detaches the process from the console. Returns `undef` on errors, a nonzero value on success. See also: `Alloc`.

Example:

```
$CONSOLE->Free();
```

GenerateCtrlEvent [type, processgroup]

Sends a break signal of the specified *type* to the specified *processgroup*. *type* can be one of the following constants:

```
CTRL_BREAK_EVENT
CTRL_C_EVENT
```

they signal, respectively, the pressing of Control + Break and of Control + C; if not specified, it defaults to CTRL_C_EVENT. *processgroup* is the pid of a process sharing the same console. If omitted, it defaults to 0 (the current process), which is also the only meaningful value that you can pass to this function. Returns `undef` on errors, a nonzero value on success.

Example:

```
# break this script now
$CONSOLE->GenerateCtrlEvent();
```

GetEvents

Returns the number of unread input events in the console's input buffer, or `undef` on errors. See also: `Input`, `InputChar`, `PeekInput`, `WriteInput`.

Example:

```
    $events = $CONSOLE->GetEvents();
```

Info Returns an array of informations about the console (or `undef` on errors), which contains:

- columns (X size) of the console buffer.
- rows (Y size) of the console buffer.
- current column (X position) of the cursor.
- current row (Y position) of the cursor.
- current attribute used for `Write`.
- left column (X of the starting point) of the current console window.
- top row (Y of the starting point) of the current console window.
- right column (X of the final point) of the current console window.
- bottom row (Y of the final point) of the current console window.
- maximum number of columns for the console window, given the current buffer size, font and the screen size.
- maximum number of rows for the console window, given the current buffer size, font and the screen size.

See also: `Attr`, `Cursor`, `Size`, `Window`, `MaxWindow`.

Example:

```
    @info = $CONSOLE->Info();
    print "Cursor at $info[3], $info[4].\n";
```

Input

Reads an event from the input buffer. Returns a list of values, which depending on the event's nature are:

keyboard event

The list will contain:

- event type: 1 for keyboard
- key down: TRUE if the key is being pressed, FALSE if the key is being released
- repeat count: the number of times the key is being held down
- virtual keycode: the virtual key code of the key
- virtual scancode: the virtual scan code of the key
- char: the ASCII code of the character (if the key is a character key, 0 otherwise)
- control key state: the state of the control keys (SHIFTs, CTRLs, ALTs, etc.)

mouse event

The list will contain:

- event type: 2 for mouse
- mouse pos. X: X coordinate (column) of the mouse location
- mouse pos. Y: Y coordinate (row) of the mouse location
- button state: the mouse button(s) which are pressed
- control key state: the state of the control keys (SHIFTs, CTRLs, ALTs, etc.)
- event flags: the type of the mouse event

This method will return `undef` on errors. Note that the events returned are depending on the input `Mode` of the console; for example, mouse events are not intercepted unless ENABLE_MOUSE_INPUT is specified. See also: `GetEvents`, `InputChar`, `Mode`, `PeekInput`, `WriteInput`.

Example:

```
@event = $CONSOLE->Input();
```

InputChar number

Reads and returns *number* characters from the console input buffer, or `undef` on errors. See also: `Input`, `Mode`.

Example:

```
$key = $CONSOLE->InputChar(1);
```

InputCP [codepage]

Gets or sets the input code page used by the console. Note that this doesn‘t apply to a console object, but to the standard input console. This attribute is used by the Write method. See also: `OutputCP`.

Example:

```
$codepage = $CONSOLE->InputCP();
$CONSOLE->InputCP(437);

# you may want to use the non-instanciated form to avoid confuzion :)
$codepage = Win32::Console::InputCP();
Win32::Console::InputCP(437);
```

MaxWindow

Returns the size of the largest possible console window, based on the current font and the size of the display. The result is `undef` on errors, otherwise a 2−element list containing col, row.

Example:

```
($maxCol, $maxRow) = $CONSOLE->MaxWindow();
```

Mode [flags]

Gets or sets the input or output mode of a console. *flags* can be a combination of the following constants:

```
ENABLE_LINE_INPUT
ENABLE_ECHO_INPUT
ENABLE_PROCESSED_INPUT
ENABLE_WINDOW_INPUT
ENABLE_MOUSE_INPUT
ENABLE_PROCESSED_OUTPUT
ENABLE_WRAP_AT_EOL_OUTPUT
```

For more informations on the meaning of those flags, please refer to the *"Microsoft‘s Documentation"*.

Example:

```
$mode = $CONSOLE->Mode();
$CONSOLE->Mode(ENABLE_MOUSE_INPUT | ENABLE_PROCESSED_INPUT);
```

MouseButtons

Returns the number of the buttons on your mouse, or `undef` on errors.

Example:

```
print "Your mouse has ", $CONSOLE->MouseButtons(), " buttons.\n";
```

new Win32::Console standard_handle
new Win32::Console [accessmode, sharemode]

Creates a new console object. The first form creates a handle to a standard channel, *standard_handle* can be one of the following:

```
STD_OUTPUT_HANDLE
STD_ERROR_HANDLE
STD_INPUT_HANDLE
```

The second form, instead, creates a console screen buffer in memory, which you can access for reading and writing as a normal console, and then redirect on the standard output (the screen) with `Display`. In this case, you can specify one or both of the following values for *accessmode*:

```
GENERIC_READ
GENERIC_WRITE
```

which are the permissions you will have on the created buffer, and one or both of the following values for *sharemode*:

```
FILE_SHARE_READ
FILE_SHARE_WRITE
```

which affect the way the console can be shared. If you don‘t specify any of those parameters, all 4 flags will be used.

Example:

```
$STDOUT = new Win32::Console(STD_OUTPUT_HANDLE);
$STDERR = new Win32::Console(STD_ERROR_HANDLE);
$STDIN  = new Win32::Console(STD_INPUT_HANDLE);

$BUFFER = new Win32::Console();
$BUFFER = new Win32::Console(GENERIC_READ | GENERIC_WRITE);
```

OutputCP [codepage]

Gets or sets the output code page used by the console. Note that this doesn‘t apply to a console object, but to the standard output console. See also: `InputCP`.

Example:

```
$codepage = $CONSOLE->OutputCP();
$CONSOLE->OutputCP(437);

# you may want to use the non-instanciated form to avoid confuzion :)
$codepage = Win32::Console::OutputCP();
Win32::Console::OutputCP(437);
```

PeekInput

Does exactly the same as `Input`, except that the event read is not removed from the input buffer. See also: `GetEvents`, `Input`, `InputChar`, `Mode`, `WriteInput`.

Example:

```
@event = $CONSOLE->PeekInput();
```

ReadAttr [number, col, row]

Reads the specified *number* of consecutive attributes, beginning at *col*, *row*, from the console. Returns the attributes read (a variable containing one character for each attribute), or `undef` on errors. You can then pass the returned variable to `WriteAttr` to restore the saved attributes on screen. See also: `ReadChar`, `ReadRect`.

Example:

```
$colors = $CONSOLE->ReadAttr(80*25, 0, 0);
```

ReadChar [number, col, row]

Reads the specified *number* of consecutive characters, beginning at *col*, *row*, from the console. Returns a string containing the characters read, or `undef` on errors. You can then pass the returned variable to `WriteChar` to restore the saved characters on screen. See also: `ReadAttr`, `ReadRect`.

Example:

```
$chars = $CONSOLE->ReadChar(80*25, 0, 0);
```

ReadRect left, top, right, bottom

Reads the content (characters and attributes) of the rectangle specified by *left*, *top*, *right*, *bottom* from the console. Returns a string containing the rectangle read, or `undef` on errors. You can then pass the returned variable to `WriteRect` to restore the saved rectangle on screen (or on another console). See also: `ReadAttr`, `ReadChar`.

Example:

```
 $rect = $CONSOLE->ReadRect(0, 0, 80, 25);
```

Scroll left, top, right, bottom, col, row, char, attr,

```
          [cleft, ctop, cright, cbottom]
```

Moves a block of data in a console buffer; the block is identified by *left*, *top*, *right*, *bottom*, while *row*, *col* identify the new location of the block. The cells left empty as a result of the move are filled with the character *char* and attribute *attr*. Optionally you can specify a clipping region with *cleft*, *ctop*, *cright*, *cbottom*, so that the content of the console outside this rectangle are unchanged. Returns `undef` on errors, a nonzero value on success.

Example:

```
# scrolls the screen 10 lines down, filling with black spaces
$CONSOLE->Scroll(0, 0, 80, 25, 0, 10, " ", $FG_BLACK | $BG_BLACK);
```

Select standard_handle

Redirects a standard handle to the specified console. *standard_handle* can have one of the following values:

```
STD_INPUT_HANDLE
STD_OUTPUT_HANDLE
STD_ERROR_HANDLE
```

Returns `undef` on errors, a nonzero value on success.

Example:

```
$CONSOLE->Select(STD_OUTPUT_HANDLE);
```

Size [col, row]

Gets or sets the console buffer size.

Example:

```
($x, $y) = $CONSOLE->Size();
$CONSOLE->Size(80, 25);
```

Title [title]

Gets or sets the title bar the string of the current console window.

Example:

```
$title = $CONSOLE->Title();
$CONSOLE->Title("This is a title");
```

Window [flag, left, top, right, bottom]

Gets or sets the current console window size. If called without arguments, returns a 4−element list containing the current window coordinates in the form of *left*, *top*, *right*, *bottom*. To set the window size, you have to specify an additional *flag* parameter: if it is 0 (zero), coordinates are considered relative to the current coordinates; if it is non−zero, coordinates are absolute.

Example:

```
($left, $top, $right, $bottom) = $CONSOLE->Window();
$CONSOLE->Window(1, 0, 0, 80, 50);
```

Write string

Writes *string* on the console, using the current attribute, that you can set with `Attr`, and advancing the cursor as needed. This isn‘t so different from Perl‘s "print" statement. Returns the number of characters written or `undef` on errors. See also: `WriteAttr`, `WriteChar`, `WriteRect`.

Example:

```
$CONSOLE->Write("Hello, world!");
```

WriteAttr attrs, col, row

Writes the attributes in the string *attrs*, beginning at *col*, *row*, without affecting the characters that are on screen. The string attrs can be the result of a `ReadAttr` function, or you can build your own attribute string; in this case, keep in mind that every attribute is treated as a character, not a number (see example). Returns the number of attributes written or `undef` on errors. See also: `Write`, `WriteChar`, `WriteRect`.

Example:

```
$CONSOLE->WriteAttr($attrs, 0, 0);

# note the use of chr()...
$attrs = chr($FG_BLACK | $BG_WHITE) x 80;
$CONSOLE->WriteAttr($attrs, 0, 0);
```

WriteChar chars, col, row

Writes the characters in the string *attr*, beginning at *col*, *row*, without affecting the attributes that are on screen. The string *chars* can be the result of a `ReadChar` function, or a normal string. Returns the number of characters written or `undef` on errors. See also: `Write`, `WriteAttr`, `WriteRect`.

Example:

```
$CONSOLE->WriteChar("Hello, worlds!", 0, 0);
```

WriteInput (event)

Pushes data in the console input buffer. *(event)* is a list of values, for more information see `Input`. The string chars can be the result of a `ReadChar` function, or a normal string. Returns the number of characters written or `undef` on errors. See also: `Write`, `WriteAttr`, `WriteRect`.

Example:

```
$CONSOLE->WriteInput(@event);
```

WriteRect rect, left, top, right, bottom

Writes a rectangle of characters and attributes (contained in *rect*) on the console at the coordinates specified by *left*, *top*, *right*, *bottom*. *rect* can be the result of a `ReadRect` function. Returns `undef` on errors, otherwise a 4−element list containing the coordinates of the affected rectangle, in the format *left*, *top*, *right*, *bottom*. See also: `Write`, `WriteAttr`, `WriteChar`.

Example:

```
$CONSOLE->WriteRect($rect, 0, 0, 80, 25);
```

Constants

The following constants are exported in the main namespace of your script using Win32::Console:

```
BACKGROUND_BLUE
BACKGROUND_GREEN
BACKGROUND_INTENSITY
BACKGROUND_RED
CAPSLOCK_ON
CONSOLE_TEXTMODE_BUFFER
ENABLE_ECHO_INPUT
ENABLE_LINE_INPUT
ENABLE_MOUSE_INPUT
ENABLE_PROCESSED_INPUT
ENABLE_PROCESSED_OUTPUT
ENABLE_WINDOW_INPUT
ENABLE_WRAP_AT_EOL_OUTPUT
ENHANCED_KEY
FILE_SHARE_READ
FILE_SHARE_WRITE
FOREGROUND_BLUE
FOREGROUND_GREEN
FOREGROUND_INTENSITY
FOREGROUND_RED
LEFT_ALT_PRESSED
LEFT_CTRL_PRESSED
NUMLOCK_ON
GENERIC_READ
GENERIC_WRITE
RIGHT_ALT_PRESSED
RIGHT_CTRL_PRESSED
SCROLLLOCK_ON
SHIFT_PRESSED
STD_INPUT_HANDLE
STD_OUTPUT_HANDLE
STD_ERROR_HANDLE
```

Additionally, the following variables can be used:

```
$FG_BLACK
$FG_BLUE
$FG_LIGHTBLUE
$FG_RED
$FG_LIGHTRED
$FG_GREEN
$FG_LIGHTGREEN
$FG_MAGENTA
$FG_LIGHTMAGENTA
$FG_CYAN
$FG_LIGHTCYAN
$FG_BROWN
$FG_YELLOW
$FG_GRAY
$FG_WHITE
```

```
$BG_BLACK
$BG_BLUE
$BG_LIGHTBLUE
$BG_RED
$BG_LIGHTRED
$BG_GREEN
$BG_LIGHTGREEN
$BG_MAGENTA
$BG_LIGHTMAGENTA
$BG_CYAN
$BG_LIGHTCYAN
$BG_BROWN
$BG_YELLOW
$BG_GRAY
$BG_WHITE
```

```
$ATTR_NORMAL
$ATTR_INVERSE
```

ATTR_NORMAL is set to gray foreground on black background (DOS‘s standard colors).

Microsoft‘s Documentation

Documentation for the Win32 Console and Character mode Functions can be found on Microsoft‘s site at this URL:

http://www.microsoft.com/msdn/sdk/platforms/doc/sdk/win32/sys/src/conchar.htm

A reference of the available functions is at:

http://www.microsoft.com/msdn/sdk/platforms/doc/sdk/win32/sys/src/conchar_34.htm

VERSION HISTORY

- 0.031 (24 Sep 1999)
  - Fixed typo in `GenerateCtrlEvent()`.
  - Converted and added pod documentation (from Jan Dubois <jand@activestate.com).
- 0.03 (07 Apr 1997)
  - Added "GenerateCtrlEvent" method.
  - The PLL file now comes in 2 versions, one for Perl version 5.001 (build 110) and one for Perl version 5.003 (build 300 and higher, EXCEPT 304).

- added an installation program that will automatically copy the right version in the right place.

- 0.01 (09 Feb 1997)
  - First public release.

AUTHOR

Aldo Calpini <a.calpini@romagiubileo.it

CREDITS

Thanks to: Jesse Dougherty, Dave Roth, ActiveWare, and the Perl−Win32−Users community.

DISCLAIMER

NAME

Win32::Event − Use Win32 event objects from Perl

SYNOPSIS

```
use Win32::Event;

$event = Win32::Event->new($manual,$initial,$name);
$event->wait();
```

DESCRIPTION

This module allows access to the Win32 event objects. The `wait` method and `wait_all` & `wait_any` functions are inherited from the *"Win32::IPC"* module.

Methods

`$event` = Win32::Event−new([`$manual`, [`$initial`, [`$name`]]])

Constructor for a new event object. If `$manual` is true, you must manually reset the event after it is signalled (the default is false). If `$initial` is true, the initial state of the object is signalled (default false). If `$name` is omitted, creates an unnamed event object.

If `$name` signifies an existing event object, then `$manual` and `$initial` are ignored and the object is opened.

`$event` = Win32::Event−open(`$name`)

Constructor for opening an existing event object.

`$event-`pulse

Signal the `$event` and then immediately reset it. If `$event` is a manual−reset event, releases all threads currently blocking on it. If it‘s an auto−reset event, releases just one thread.

If no threads are waiting, just resets the event.

`$event-`reset

Reset the `$event` to nonsignalled.

`$event-`set

Set the `$event` to signalled.

`$event-`wait([$timeout])

Wait for `$event` to be signalled. See *"Win32::IPC"*.

AUTHOR

Christopher J. Madsen <***chris_madsen@geocities.com***>

NAME

Win32::EventLog − Process Win32 Event Logs from Perl

SYNOPSIS

```
use Win32::EventLog
$handle=Win32::EventLog->new("Application");
```

DESCRIPTION

This module implements most of the functionality available from the Win32 API for accessing and manipulating Win32 Event Logs. The access to the EventLog routines is divided into those that relate to an EventLog object and its associated methods and those that relate other EventLog tasks (like adding an EventLog record).

The EventLog Object and its Methods

The following methods are available to open, read, close and backup EventLogs.

Win32::EventLog−new(SOURCENAME [,SERVERNAME]);

The `new()` method creates a new EventLog object and returns a handle to it. This hande is then used to call the methods below.

The method is overloaded in that if the supplied SOURCENAME argument contains one or more literal '\' characters (an illegal character in a SOURCENAME), it assumes that you are trying to open a backup eventlog and uses SOURCENAME as the backup eventlog to open. Note that when opening a backup eventlog, the SERVERNAME argument is ignored (as it is in the underlying Win32 API). For EventLogs on remote machines, the SOURCENAME parameter must therefore be specified as a UNC path.

`$handle−`Backup(FILENAME);

The `Backup()` method backs up the EventLog represented by `$handle.` It takes a single arguemt, FILENAME. When `$handle` represents an EventLog on a remote machine, FILENAME is filename on the remote machine and cannot be a UNC path (i.e you must use ***C:\TEMP\App.EVT***). The method will fail if the log file already exists.

`$handle−`Read(FLAGS, OFFSET, HASHREF);

The `Read()` method read an EventLog entry from the EventLog represented by `$handle.`

`$handle−`Close();

The `Close()` method closes the EventLog represented by `$handle.` After `Close()` has been called, any further attempt to use the EventLog represented by `$handle` will fail.

`$handle−`GetOldest(SCALARREF);

The `GetOldest()` method number of the the oldest EventLog record in the EventLog represented by `$handle.` This is required to correctly compute the OFFSET required by the `Read()` method.

`$handle−`GetNumber(SCALARREF);

The `GetNumber()` method returns the number of EventLog records in the EventLog represented by `$handle.` The number of the most recent record in the EventLog is therefore computed by

```
$handle->GetOldest($oldest);
$handle->GetNumber($lastRec);
$lastRecOffset=$oldest+$lastRec;
```

`$handle−`Clear(FILENAME);

The `Clear()` method clears the EventLog represented by `$handle.` If you provide a non−null FILENAME, the EventLog will be backed up into FILENAME before the EventLog is cleared. The method will fail if FILENAME is specified and the file refered to exists. Note also that FILENAME specifies a file local to the machine on which the EventLog resides and cannot be specified as a UNC name.

`$handle-`Report(HASHREF);

The `Report()` method generates an EventLog entry. The HASHREF should contain the following keys:

`Computer`

The `Computer` field specfies which computer you want the EventLog entry recorded. If this key doesn‘t exist, the server name used to create the `$handle` is used.

`Source`

The `Source` field specifies the source that generated the EventLog entry. If this key doesn‘t exist, the source name used to create the `$handle` is used.

`EventType`

The `EventType` field should be one of the constants

`EVENTLOG_ERROR_TYPE`

An Error event is being logged.

`EVENTLOG_WARNING_TYPE`

A Warning event is being logged.

`EVENTLOG_INFORMATION_TYPE`

An Information event is being logged.

`EVENTLOG_AUDIT_SUCCESS`

A Success Audit event is being logged (typically in the Security EventLog).

`EVENTLOG_AUDIT_FAILURE`

A Failure Audit event is being logged (typically in the Security EventLog).

These constants are exported into the main namespace by default.

`Category`

The `Category` field can have any value you want. It is specific to the particular Source.

`EventID`

The `EventID` field should contain the ID of the message that this event pertains too. This assumes that you have an associated message file (indirectly referenced by the field `Source`).

`Data`

The `Data` field contains raw data associated with this event.

`Strings`

The `Strings` field contains the single string that itself contains NUL terminated sub−strings. This are used with the EventID to generate the message as seen from (for example) the Event Viewer application.

Other Win32::EventLog functions.

The following functions are part of the Win32::EventLog package but are not callable from an EventLog object.

GetMessageText(HASHREF);

The `GetMessageText()` function assumes that HASHREF was obtained by a call to `$handle->Read()`. It returns the formatted string that represents the fully resolved text of the EventLog message (such as would be seen in the Windows NT Event Viewer). For convenience, the key ‘Message’ in the supplied HASHREF is also set to the return value of this function.

If you set the variable `$Win32::EventLog::GetMessageText` to 1 then each call to `$handle->Read()` will call this function automatically.

Example 1

The following example illustrates the way in which the EventLog module can be used. It opens the System EventLog and reads through it from oldest to newest records. For each record from the **Source** EventLog it extracts the full text of the Entry and prints the EventLog message text out.

```
use Win32::EventLog;

$handle=Win32::EventLog->new("System", $ENV{ComputerName})
       or die "Can't open Application EventLog\n";
$handle->GetNumber($recs)
       or die "Can't get number of EventLog records\n";
$handle->GetOldest($base)
       or die "Can't get number of oldest EventLog record\n";

while ($x < $recs) {
       $handle->Read(EVENTLOG_FORWARDS_READ|EVENTLOG_SEEK_READ,
                                 $base+$x,
                                 $hashRef)
               or die "Can't read EventLog entry #$x\n";
       if ($hashRef->{Source} eq "EventLog") {
               Win32::EventLog::GetMessageText($hashRef);
               print "Entry $x: $hashRef->{Message}\n";
       }
       $x++;
}
```

Example 2

To backup and clear the EventLogs on a remote machine, do the following :−

```
use Win32::EventLog;

$myServer="\\\\my-server";     # your servername here.
my($date)=join("-", ((split(/\s+/, scalar(localtime)))[0,1,2,4]));
my($dest);

for my $eventLog ("Application", "System", "Security") {
       $handle=Win32::EventLog->new($eventLog, $myServer)
               or die "Can't open Application EventLog on $myServer\n";

       $dest="C:\\BackupEventLogs\\$eventLog\\$date.evt";
       $handle->Backup($dest)
               or warn "Could not backup and clear the $eventLog EventLog on $myServ

       $handle->Close;
}
```

Note that only the Clear method is required. Note also that if the file `$dest` exists, the function will fail.

BUGS

None currently known.

The test script for 'make test' should be re−written to use the EventLog object.

AUTHOR

Original code by Jesse Dougherty for HiP Communications. Additional fixes and updates attributed to Martin Pauley <martin.pauley@ulsterbank.ltd.uk) and Bret Giddings (bret@essex.ac.uk).

NAME

Win32::File − manage file attributes in perl

SYNOPSIS

```
        use Win32::File;
```

DESCRIPTION

This module offers the retrieval and setting of file attributes.

Functions

NOTE

All of the functions return FALSE (0) if they fail, unless otherwise noted. The function names are exported into the caller‘s namespace by request.

GetAttributes(filename, returnedAttributes)

Gets the attributes of a file or directory. returnedAttributes will be set to the OR−ed combination of the filename attributes.

SetAttributes(filename, newAttributes)

Sets the attributes of a file or directory. newAttributes must be an OR−ed combination of the attributes.

Constants

The following constants are exported by default.

ARCHIVE
COMPRESSED
DIRECTORY
HIDDEN
NORMAL
OFFLINE
READONLY
SYSTEM
TEMPORARY

NAME

Win32::FileSecurity − manage FileSecurity Discretionary Access Control Lists in perl

SYNOPSIS

```
        use Win32::FileSecurity;
```

DESCRIPTION

This module offers control over the administration of system FileSecurity DACLs. You may want to use Get and EnumerateRights to get an idea of what mask values correspond to what rights as viewed from File Manager.

CONSTANTS

```
  DELETE, READ_CONTROL, WRITE_DAC, WRITE_OWNER,
  SYNCHRONIZE, STANDARD_RIGHTS_REQUIRED,
  STANDARD_RIGHTS_READ, STANDARD_RIGHTS_WRITE,
  STANDARD_RIGHTS_EXECUTE, STANDARD_RIGHTS_ALL,
  SPECIFIC_RIGHTS_ALL, ACCESS_SYSTEM_SECURITY,
  MAXIMUM_ALLOWED, GENERIC_READ, GENERIC_WRITE,
  GENERIC_EXECUTE, GENERIC_ALL, F, FULL, R, READ,
  C, CHANGE
```

FUNCTIONS

NOTE:

All of the functions return FALSE (0) if they fail, unless otherwise noted. Errors returned via `$!` containing both Win32 `GetLastError()` and a text message indicating Win32 function that failed.

constant(`$name`, `$set`)

Stores the value of named constant `$name` into `$set`. Same as `$set = Win32::FileSecurity::NAME_OF_CONSTANT();`.

Get(`$filename`, \%permisshash)

Gets the DACLs of a file or directory.

Set(`$filename`, \%permisshash)

Sets the DACL for a file or directory.

EnumerateRights(`$mask`, \@rightslist)

Turns the bitmask in `$mask` into a list of strings in @rightslist.

MakeMask(qw(DELETE READ_CONTROL))

Takes a list of strings representing constants and returns a bitmasked integer value.

%permisshash

Entries take the form `$permisshash{USERNAME}` = `$mask`;

EXAMPLE1

```
    # Gets the rights for all files listed on the command line.
    use Win32::FileSecurity qw(Get EnumerateRights);

    foreach( @ARGV ) {
        next unless -e $_ ;
        if ( Get( $_, \%hash ) ) {
            while( ($name, $mask) = each %hash ) {
                print "$name:\n\t";
                EnumerateRights( $mask, \@happy ) ;
                print join( "\n\t", @happy ), "\n";
            }
        }
        else {
            print( "Error #", int( $! ), ": $!" ) ;
        }
    }
```

EXAMPLE2

```
        # Gets existing DACL and modifies Administrator rights
        use Win32::FileSecurity qw(MakeMask Get Set);

        # These masks show up as Full Control in File Manager
        $file = MakeMask( qw( FULL ) );

        $dir = MakeMask( qw(
                FULL
            GENERIC_ALL
        ) );

        foreach( @ARGV ) {
            s/\\$//;
            next unless -e;
            Get( $_, \%hash ) ;
            $hash{Administrator} = ( -d ) ? $dir : $file ;
            Set( $_, \%hash ) ;
        }
```

COMMON MASKS FROM CACLS AND WINFILE

READ

```
            MakeMask( qw( FULL ) ); # for files
            MakeMask( qw( READ GENERIC_READ GENERIC_EXECUTE ) ); # for directories
```

CHANGE

```
            MakeMask( qw( CHANGE ) ); # for files
            MakeMask( qw( CHANGE GENERIC_WRITE GENERIC_READ GENERIC_EXECUTE ) ); # for di
```

ADD & READ

```
            MakeMask( qw( ADD GENERIC_READ GENERIC_EXECUTE ) ); # for directories only!
```

FULL

```
            MakeMask( qw( FULL ) ); # for files
            MakeMask( qw( FULL  GENERIC_ALL ) ); # for directories
```

RESOURCES

From Microsoft: check_sd
http://premium.microsoft.com/download/msdn/samples/2760.exe

(thanks to Guert Schimmel at Sybase for turning me on to this one)

VERSION

1.03 ALPHA 97−12−14

REVISION NOTES

1.03 ALPHA 1998.01.11

Imported diffs from 0.67 (parent) version

1.02 ALPHA 1997.12.14

Pod fixes, @EXPORT list additions <gsar@activestate.com

Fix unitialized vars on unknown ACLs <jmk@exc.bybyte.de

1.01 ALPHA 1997.04.25

CORE Win32 version imported from 0.66 <gsar@activestate.com

0.67 ALPHA 1997.07.07

Kludged bug in mapping bits to separate ACE‘s. Notably, this screwed up CHANGE access by leaving out a delete bit in the `INHERIT_ONLY_ACE | OBJECT_INHERIT_ACE` Access Control Entry.

May need to rethink...

0.66 ALPHA 1997.03.13

Fixed bug in memory allocation check

0.65 ALPHA 1997.02.25

Tested with 5.003 build 303

Added ISA exporter, and @EXPORT_OK

Added F, FULL, R, READ, C, CHANGE as composite pre−built mask names.

Added server\ to keys returned in hash from Get

Made constants and MakeMask case insensitive (I don‘t know why I did that)

Fixed mask comparison in ListDacl and Enumerate Rights from simple `&` mask to exact bit match ! ((x `&` y) ^ x) makes sure all bits in x are set in y

Fixed some "wild" pointers

0.60 ALPHA 1996.07.31

Now suitable for file and directory permissions

Included ListDacl.exe in bundle for debugging

Added "intuitive" inheritance for directories, basically functions like FM triggered by presence of GENERIC_ rights this may need to change

see EXAMPLE2

Changed from AddAccessAllowedAce to AddAce for control over inheritance

0.51 ALPHA 1996.07.20

Fixed memory allocation bug

0.50 ALPHA 1996.07.29

Base functionality

Using AddAccessAllowedAce

Suitable for file permissions

KNOWN ISSUES / BUGS

1 May not work on remote drives.

2 Errors croak, don‘t return via `$!` as documented.

NAME

Win32::Internet − Access to WININET.DLL functions

INTRODUCTION

This extension to Perl implements the Win32 Internet APIs (found in ***WININET.DLL***). They give a complete support for HTTP, FTP and GOPHER connections.

See the *"Version History"* and the *"Functions Table"* for a list of the currently supported features. You should also get a copy of the *"Microsoft Win32 Internet Functions"* documentation.

REFERENCE

To use this module, first add the following line at the beginning of your script:

```
    use Win32::Internet;
```

Then you have to open an Internet connection with this command:

```
    $Connection = new Win32::Internet();
```

This is required to use any of the function of this module. It will create an Internet object in Perl on which you can act upon with the *"General Internet Functions"* explained later.

The objects available are:

- Internet connections (the main object, see `new`)
- URLs (see `OpenURL`)
- FTP sessions (see `FTP`)
- HTTP sessions (see `HTTP`)
- HTTP requests (see `OpenRequest`)

As in the good Perl tradition, there are in this extension different ways to do the same thing; there are, in fact, different levels of implementation of the Win32 Internet Functions. Some routines use several Win32 API functions to perform a complex task in a single call; they are simpler to use, but of course less powerful.

There are then other functions that implement nothing more and nothing less than the corresponding API function, so you can use all of their power, but with some additional programming steps.

To make an example, there is a function called `FetchURL` that you can use to fetch the content of any HTTP, FTP or GOPHER URL with this simple commands:

```
    $INET = new Win32::Internet();
    $file = $INET->FetchURL("http://www.yahoo.com");
```

You can have the same result (and this is actually what is done by `FetchURL`) this way:

```
    $INET = new Win32::Internet();
    $URL = $INET->OpenURL("http://www.yahoo.com");
    $file = $URL->ReadFile();
    $URL->Close();
```

Or, you can open a complete HTTP session:

```
    $INET = new Win32::Internet();
    $HTTP = $INET->HTTP("www.yahoo.com", "anonymous", "dada@divinf.it");
    ($statuscode, $headers, $file) = $HTTP->Request("/");
    $HTTP->Close();
```

Finally, you can choose to manage even the HTTP request:

```
    $INET = new Win32::Internet();
    $HTTP = $INET->HTTP("www.yahoo.com", "anonymous", "dada@divinf.it");
```

```
$HTTP->OpenRequest($REQ, "/");
$REQ->AddHeader("If-Modified-Since: Saturday, 16-Nov-96 15:58:50 GMT");
$REQ->SendRequest();
$statuscode = $REQ->QueryInfo("",HTTP_QUERY_STATUS_CODE);
$lastmodified = $REQ->QueryInfo("Last-Modified");
$file = $REQ->ReadEntireFile();
$REQ->Close();
$HTTP->Close();
```

To open and control a complete FTP session, type:

```
$Connection->FTP($Session, "ftp://ftp.activeware.com", "anonymous", "dada\@divinf
```

This will create an FTP object in Perl to which you can apply the *"FTP functions"* provided by the package:

```
$Session->Cd("/ntperl/perl5.001m/CurrentBuild");
$Session->Ascii();
$Session->Get("110-i86.zip");
$Session->Close();
```

For a more complete example, see the TEST.PL file that comes with the package.

General Internet Functions

General Note

All methods assume that you have the line:

```
use Win32::Internet;
```

somewhere before the method calls, and that you have an Internet object called `$INET` which was created using this call:

```
$INET = new Win32::Internet();
```

See `new` for more information.

Methods

CanonicalizeURL URL, [flags]

Converts a URL to a canonical format, which includes converting unsafe characters to escape sequences. Returns the canonicalized URL or `undef` on errors. For the possible values of *flags*, refer to the *"Microsoft Win32 Internet Functions"* document. See also `CombineURL` and `OpenURL`.

Example:

```
$cURL = $INET->CanonicalizeURL($URL);
$URL = $INET->CanonicalizeURL($cURL, ICU_DECODE);
```

Close
Close object

Closes an Internet connection. This can be applied to any Win32::Internet object (Internet connections, URLs, FTP sessions, etc.). Note that it is not "strictly" required to close the connections you create, since the Win32::Internet objects are automatically closed when the program ends (or when you elsehow destroy such an object).

Example:

```
$INET->Close();
$FTP->Close();
$INET->Close($FTP); # same as above...
```

CombineURL baseURL, relativeURL, [flags]

Combines a base and relative URL into a single URL. Returns the (canonicalized) combined URL or `undef` on errors. For the possible values of *flags*, refer to the *"Microsoft Win32 Internet Functions"* document. See also `CombineURL` and `OpenURL`.

Example:

```
    $URL = $INET->CombineURL("http://www.divinf.it/dada/perl/internet", "..");
```

ConnectBackoff [value]

Reads or sets the delay value, in milliseconds, to wait between connection retries. If no *value* parameter is specified, the current value is returned; otherwise, the delay between retries is set to *value*. See also `ConnectTimeout`, `ConnectRetries`, `QueryOption` and `SetOption`.

Example:

```
    $HTTP->ConnectBackoff(2000);
    $backoff = $HTTP->ConnectBackoff();
```

ConnectRetries [value]

Reads or sets the number of times a connection is retried before considering it failed. If no *value* parameter is specified, the current value is returned; otherwise, the number of retries is set to *value*. The default value for `ConnectRetries` is 5. See also `ConnectBackoff`, `ConnectTimeout`, `QueryOption` and `SetOption`.

Example:

```
    $HTTP->ConnectRetries(20);
    $retries = $HTTP->ConnectRetries();
```

ConnectTimeout [value]

Reads or sets the timeout value (in milliseconds) before a connection is considered failed. If no *value* parameter is specified, the current value is returned; otherwise, the timeout is set to *value*. The default value for `ConnectTimeout` is infinite. See also `ConnectBackoff`, `ConnectRetries`, `QueryOption` and `SetOption`.

Example:

```
    $HTTP->ConnectTimeout(10000);
    $timeout = $HTTP->ConnectTimeout();
```

ControlReceiveTimeout [value]

Reads or sets the timeout value (in milliseconds) to use for non−data (control) receive requests before they are canceled. Currently, this value has meaning only for `FTP` sessions. If no *value* parameter is specified, the current value is returned; otherwise, the timeout is set to *value*. The default value for `ControlReceiveTimeout` is infinite. See also `ControlSendTimeout`, `QueryOption` and `SetOption`.

Example:

```
    $HTTP->ControlReceiveTimeout(10000);
    $timeout = $HTTP->ControlReceiveTimeout();
```

ControlSendTimeout [value]

Reads or sets the timeout value (in milliseconds) to use for non−data (control) send requests before they are canceled. Currently, this value has meaning only for `FTP` sessions. If no *value* parameter is specified, the current value is returned; otherwise, the timeout is set to *value*. The default value for `ControlSendTimeout` is infinite. See also `ControlReceiveTimeout`, `QueryOption` and `SetOption`.

Example:

```
$HTTP->ControlSendTimeout(10000);
$timeout = $HTTP->ControlSendTimeout();
```

CrackURL URL, [flags]

Splits an URL into its component parts and returns them in an array. Returns `undef` on errors, otherwise the array will contain the following values: *scheme, host, port, username, password, path, extrainfo*.

For example, the URL "http://www.divinf.it/index.html#top" can be splitted in:

```
http, www.divinf.it, 80, anonymous, dada@divinf.it, /index.html, #top
```

If you don‘t specify a *flags* parameter, ICU_ESCAPE will be used by default; for the possible values of *flags* refer to the *"Microsoft Win32 Internet Functions"* documentation. See also `CreateURL`.

Example:

```
@parts=$INET->CrackURL("http://www.activeware.com");
($scheme, $host, $port, $user, $pass, $path, $extra) =
     $INET->CrackURL("http://www.divinf.it:80/perl-win32/index.sht#feedback")
```

CreateURL scheme, hostname, port, username, password, path, extrainfo, [flags]
CreateURL hashref, [flags]

Creates a URL from its component parts. Returns `undef` on errors, otherwise the created URL.

If you pass *hashref* (a reference to an hash array), the following values are taken from the array:

```
%hash=(
  "scheme"    => "scheme",
  "hostname"  => "hostname",
  "port"      => port,
  "username"  => "username",
  "password"  => "password",
  "path"      => "path",
  "extrainfo" => "extrainfo",
);
```

If you don‘t specify a *flags* parameter, ICU_ESCAPE will be used by default; for the other possible values of *flags* refer to the *"Microsoft Win32 Internet Functions"* documentation. See also `CrackURL`.

Example:

```
$URL=$I->CreateURL("http", "www.divinf.it", 80, "", "", "/perl-win32/index.sh
$URL=$I->CreateURL(\%params);
```

DataReceiveTimeout [value]

Reads or sets the timeout value (in milliseconds) to use for data receive requests before they are canceled. If no *value* parameter is specified, the current value is returned; otherwise, the timeout is set to *value*. The default value for DataReceiveTimeout is infinite. See also `DataSendTimeout`, `QueryOption` and `SetOption`.

Example:

```
$HTTP->DataReceiveTimeout(10000);
$timeout = $HTTP->DataReceiveTimeout();
```

DataSendTimeout [value]

Reads or sets the timeout value (in milliseconds) to use for data send requests before they are canceled. If no *value* parameter is specified, the current value is returned; otherwise, the timeout is set to *value*.

The default value for DataSendTimeout is infinite. See also `DataReceiveTimeout`, `QueryOption` and `SetOption`.

Example:

```
$HTTP->DataSendTimeout(10000);
$timeout = $HTTP->DataSendTimeout();
```

Error

Returns the last recorded error in the form of an array or string (depending upon the context) containing the error number and an error description. Can be applied on any Win32::Internet object (FTP sessions, etc.). There are 3 types of error you can encounter; they are recognizable by the error number returned:

- `-1` A "trivial" error has occurred in the package. For example, you tried to use a method on the wrong type of object.
- 1 .. 11999

  A generic error has occurred and the Win32::GetLastError error message is returned.
- 12000 and higher

  An Internet error has occurred; the extended Win32 Internet API error message is returned.

See also `GetResponse`.

Example:

```
die $INET->Error(), qq(\n);
($ErrNum, $ErrText) = $INET->Error();
```

FetchURL URL

Fetch the content of an HTTP, FTP or GOPHER URL. Returns the content of the file read (or `undef` if there was an error and nothing was read). See also `OpenURL` and `ReadFile`.

Example:

```
$file = $INET->FetchURL("http://www.yahoo.com/");
$file = $INET->FetchURL("ftp://www.activeware.com/contrib/internet.zip");
```

FTP ftpobject, server, username, password, [port, pasv, context]
FTP ftpobject, hashref

Opens an FTP connection to server logging in with the given *username* and *password*.

The parameters and their values are:

- server

  The server to connect to. Default: *none*.
- username

  The username used to login to the server. Default: anonymous.
- password

  The password used to login to the server. Default: *none*.
- port

  The port of the FTP service on the server. Default: 21.
- pasv

  If it is a value other than 0, use passive transfer mode. Default is taken from the parent Internet connection object; you can set this value with the `Pasv` method.

- context

  A number to identify this operation if it is asynchronous. See `SetStatusCallback` and `GetStatusCallback` for more info on asynchronous operations. Default: *none*.

If you pass *hashref* (a reference to an hash array), the following values are taken from the array:

```
%hash=(
  "server"   => "server",
  "username" => "username",
  "password" => "password",
  "port"     => port,
  "pasv"     => pasv,
  "context"  => context,
);
```

This method returns `undef` if the connection failed, a number otherwise. You can then call any of the *"FTP functions"* as methods of the newly created *ftpobject*.

Example:

```
$result = $INET->FTP($FTP, "ftp.activeware.com", "anonymous", "dada\@divinf.i
# and then for example...
$FTP->Cd("/ntperl/perl5.001m/CurrentBuild");

$params{"server"} = "ftp.activeware.com";
$params{"password"} = "dada\@divinf.it";
$params{"pasv"} = 0;
$result = $INET->FTP($FTP,\%params);
```

GetResponse

Returns the text sent by a remote server in response to the last function executed. It applies on any Win32::Internet object, particularly of course on *FTP sessions|"FTP functions"*. See also the `Error` function.

Example:

```
print $INET->GetResponse();
$INET->FTP($FTP, "ftp.activeware.com", "anonymous", "dada\@divinf.it");
print $FTP->GetResponse();
```

GetStatusCallback context

Returns information about the progress of the asynchronous operation identified by *context*; those informations consist of two values: a status code (one of the INTERNET_STATUS_* *"Constants"*) and an additional value depending on the status code; for example, if the status code returned is INTERNET_STATUS_HANDLE_CREATED, the second value will hold the handle just created. For more informations on those values, please refer to the *"Microsoft Win32 Internet Functions"* documentation. See also `SetStatusCallback`.

Example:

```
($status, $info) = $INET->GetStatusCallback(1);
```

HTTP httpobject, server, username, password, [port, flags, context]
HTTP httpobject, hashref

Opens an HTTP connection to *server* logging in with the given *username* and *password*.

The parameters and their values are:

- server

  The server to connect to. Default: *none*.

- username

  The username used to login to the server. Default: anonymous.

- password

  The password used to login to the server. Default: *none*.

- port

  The port of the HTTP service on the server. Default: 80.

- flags

  Additional flags affecting the behavior of the function. Default: *none*.

- context

  A number to identify this operation if it is asynchronous. See `SetStatusCallback` and `GetStatusCallback` for more info on asynchronous operations. Default: *none*.

Refer to the *"Microsoft Win32 Internet Functions"* documentation for more details on those parameters.

If you pass *hashref* (a reference to an hash array), the following values are taken from the array:

```
%hash=(
  "server"   => "server",
  "username" => "username",
  "password" => "password",
  "port"     => port,
  "flags"    => flags,
  "context"  => context,
);
```

This method returns `undef` if the connection failed, a number otherwise. You can then call any of the *"HTTP functions"* as methods of the newly created *httpobject*.

Example:

```
$result = $INET->HTTP($HTTP,"www.activeware.com","anonymous","dada\@divinf.it
# and then for example...
($statuscode, $headers, $file) = $HTTP->Request("/gifs/camel.gif");

$params{"server"} = "www.activeware.com";
$params{"password"} = "dada\@divinf.it";
$params{"flags"} = INTERNET_FLAG_RELOAD;
$result = $INET->HTTP($HTTP,\%params);
```

new Win32::Internet [useragent, opentype, proxy, proxybypass, flags]
new Win32::Internet [hashref]

Creates a new Internet object and initializes the use of the Internet functions; this is required before any of the functions of this package can be used. Returns `undef` if the connection fails, a number otherwise. The parameters and their values are:

- useragent

  The user agent passed to HTTP requests. See `OpenRequest`. Default: Perl−Win32::Internet/*version*.

- opentype

  How to access to the Internet (eg. directly or using a proxy). Default: INTERNET_OPEN_TYPE_DIRECT.

- proxy

  Name of the proxy server (or servers) to use. Default: *none*.

- proxybypass

  Optional list of host names or IP addresses, or both, that are known locally. Default: *none*.

- flags

  Additional flags affecting the behavior of the function. Default: *none*.

Refer to the *"Microsoft Win32 Internet Functions"* documentation for more details on those parameters. If you pass *hashref* (a reference to an hash array), the following values are taken from the array:

```
%hash=(
  "useragent"   => "useragent",
  "opentype"    => "opentype",
  "proxy"       => "proxy",
  "proxybypass" => "proxybypass",
  "flags"       => flags,
);
```

Example:

```
$INET = new Win32::Internet();
die qq(Cannot connect to Internet...\n) if ! $INET;

$INET = new Win32::Internet("Mozilla/3.0", INTERNET_OPEN_TYPE_PROXY, "www.mic

$params{"flags"} = INTERNET_FLAG_ASYNC;
$INET = new Win32::Internet(\%params);
```

OpenURL urlobject, URL

Opens a connection to an HTTP, FTP or GOPHER Uniform Resource Location (URL). Returns `undef` on errors or a number if the connection was succesful. You can then retrieve the URL content by applying the methods `QueryDataAvailable` and `ReadFile` on the newly created *urlobject*. See also `FetchURL`.

Example:

```
$INET->OpenURL($URL, "http://www.yahoo.com/");
$bytes = $URL->QueryDataAvailable();
$file = $URL->ReadEntireFile();
$URL->Close();
```

Password [password]

Reads or sets the password used for an `FTP` or `HTTP` connection. If no *password* parameter is specified, the current value is returned; otherwise, the password is set to *password*. See also `Username`, `QueryOption` and `SetOption`.

Example:

```
$HTTP->Password("splurfgnagbxam");
$password = $HTTP->Password();
```

QueryDataAvailable

Returns the number of bytes of data that are available to be read immediately by a subsequent call to `ReadFile` (or `undef` on errors). Can be applied to URL or HTTP request objects. See `OpenURL` or `OpenRequest`.

Example:

```
$INET->OpenURL($URL, "http://www.yahoo.com/");
$bytes = $URL->QueryDataAvailable();
```

QueryOption option

Queries an Internet option. For the possible values of *option*, refer to the *"Microsoft Win32 Internet Functions"* document. See also `SetOption`.

Example:

```
$value = $INET->QueryOption(INTERNET_OPTION_CONNECT_TIMEOUT);
$value = $HTTP->QueryOption(INTERNET_OPTION_USERNAME);
```

ReadEntireFile

Reads all the data available from an opened URL or HTTP request object. Returns what have been read or `undef` on errors. See also `OpenURL`, `OpenRequest` and `ReadFile`.

Example:

```
$INET->OpenURL($URL, "http://www.yahoo.com/");
$file = $URL->ReadEntireFile();
```

ReadFile bytes

Reads *bytes* bytes of data from an opened URL or HTTP request object. Returns what have been read or `undef` on errors. See also `OpenURL`, `OpenRequest`, `QueryDataAvailable` and `ReadEntireFile`.

Note: be careful to keep *bytes* to an acceptable value (eg. don‘t tell him to swallow megabytes at once...). `ReadEntireFile` in fact uses `QueryDataAvailable` and `ReadFile` in a loop to read no more than 16k at a time.

Example:

```
$INET->OpenURL($URL, "http://www.yahoo.com/");
$chunk = $URL->ReadFile(16000);
```

SetOption option, value

Sets an Internet option. For the possible values of *option*, refer to the *"Microsoft Win32 Internet Functions"* document. See also `QueryOption`.

Example:

```
$INET->SetOption(INTERNET_OPTION_CONNECT_TIMEOUT,10000);
$HTTP->SetOption(INTERNET_OPTION_USERNAME,"dada");
```

SetStatusCallback

Initializes the callback routine used to return data about the progress of an asynchronous operation.

Example:

```
$INET->SetStatusCallback();
```

This is one of the step required to perform asynchronous operations; the complete procedure is:

```
# use the INTERNET_FLAG_ASYNC when initializing
$params{'flags'}=INTERNET_FLAG_ASYNC;
$INET = new Win32::Internet(\%params);

# initialize the callback routine
$INET->SetStatusCallback();

# specify the context parameter (the last 1 in this case)
$INET->HTTP($HTTP, "www.yahoo.com", "anonymous", "dada\@divinf.it", 80, 0, 1)
```

At this point, control returns immediately to Perl and `$INET`–Error() will return 997, which means an asynchronous I/O operation is pending. Now, you can call

```
$HTTP->GetStatusCallback(1);
```

in a loop to verify what‘s happening; see also `GetStatusCallback`.

TimeConvert time
TimeConvert seconds, minute, hours, day, month, year,
`day_of_week, RFC`

The first form takes a HTTP date/time string and returns the date/time converted in the following array: *seconds, minute, hours, day, month, year, day_of_week*.

The second form does the opposite (or at least it should, because actually seems to be malfunctioning): it takes the values and returns an HTTP date/time string, in the RFC format specified by the *RFC* parameter (OK, I didn‘t find yet any accepted value in the range 0..2000, let me know if you have more luck with it).

Example:

```
($sec, $min, $hour, $day, $mday, $year, $wday) =
   $INET->TimeConvert("Sun, 26 Jan 1997 20:01:52 GMT");

# the opposite DOESN'T WORK! which value should $RFC have???
$time = $INET->TimeConvert(52, 1, 20, 26, 1, 1997, 0, $RFC);
```

UserAgent [name]

Reads or sets the user agent used for HTTP requests. If no *name* parameter is specified, the current value is returned; otherwise, the user agent is set to *name*. See also `QueryOption` and `SetOption`.

Example:

```
$INET->UserAgent("Mozilla/3.0");
$useragent = $INET->UserAgent();
```

Username [name]

Reads or sets the username used for an `FTP` or `HTTP` connection. If no *name* parameter is specified, the current value is returned; otherwise, the username is set to *name*. See also `Password`, `QueryOption` and SetOption.

Example:

```
$HTTP->Username("dada");
$username = $HTTP->Username();
```

Version

Returns the version numbers for the Win32::Internet package and the WININET.DLL version, as an array or string, depending on the context. The string returned will contain "package_version/DLL_version", while the array will contain: "package_version", "DLL_version".

Example:

```
$version = $INET->Version(); # should return "0.06/4.70.1215"
@version = $INET->Version(); # should return ("0.06", "4.70.1215")
```

FTP Functions

General Note

All methods assume that you have the following lines:

```
use Win32::Internet;
$INET = new Win32::Internet();
$INET->FTP($FTP, "hostname", "username", "password");
```

somewhere before the method calls; in other words, we assume that you have an Internet object called `$INET` and an open FTP session called `$FTP`.

See `new` and `FTP` for more information.

Methods

Ascii
Asc Sets the ASCII transfer mode for this FTP session. It will be applied to the subsequent `Get` functions. See also the `Binary` and `Mode` function.

Example:

```
$FTP->Ascii();
```

Binary
Bin Sets the binary transfer mode for this FTP session. It will be applied to the subsequent `Get` functions. See also the `Ascii` and `Mode` function.

Example:

```
$FTP->Binary();
```

Cd path
Cwd path
Chdir path

Changes the current directory on the FTP remote host. Returns *path* or `undef` on error.

Example:

```
$FTP->Cd("/pub");
```

Delete file
Del file

Deletes a file on the FTP remote host. Returns `undef` on error.

Example:

```
$FTP->Delete("110-i86.zip");
```

Get remote, [local, overwrite, flags, context]

Gets the *remote* FTP file and saves it locally in *local*. If *local* is not specified, it will be the same name as *remote*. Returns `undef` on error. The parameters and their values are:

- remote

  The name of the remote file on the FTP server. Default: *none*.

- local

  The name of the local file to create. Default: remote.

- overwrite

  If 0, overwrites *local* if it exists; with any other value, the function fails if the *local* file already exists. Default: 0.

- flags

  Additional flags affecting the behavior of the function. Default: *none*.

- context

  A number to identify this operation if it is asynchronous. See `SetStatusCallback` and `GetStatusCallback` for more info on asynchronous operations. Default: *none*.

Refer to the *"Microsoft Win32 Internet Functions"* documentation for more details on those parameters.

Example:

```
$FTP->Get("110-i86.zip");
$FTP->Get("/pub/perl/languages/CPAN/00index.html", "CPAN_index.html");
```

List [pattern, listmode]
Ls [pattern, listmode]
Dir [pattern, listmode]

Returns a list containing the files found in this directory, eventually matching the given *pattern* (which, if omitted, is considered "*.*"). The content of the returned list depends on the *listmode* parameter, which can have the following values:

- listmode=1 (or omitted)

  the list contains the names of the files found. Example:

```
@files = $FTP->List();
@textfiles = $FTP->List("*.txt");
foreach $file (@textfiles) {
  print "Name: ", $file, "\n";
}
```

- listmode=2

  the list contains 7 values for each file, which respectively are:

  - the file name
  - the DOS short file name, aka 8.3
  - the size
  - the attributes
  - the creation time
  - the last access time
  - the last modified time

  Example:

```
@files = $FTP->List("*.*", 2);
for($i=0; $i<=$#files; $i+=7) {
  print "Name: ", @files[$i], "\n";
  print "Size: ", @files[$i+2], "\n";
  print "Attr: ", @files[$i+3], "\n";
}
```

- listmode=3

  the list contains a reference to an hash array for each found file; each hash contains:

  - name = the file name
  - altname = the DOS short file name, aka 8.3
  - size = the size
  - attr = the attributes
  - ctime = the creation time
  - atime = the last access time
  - mtime = the last modified time

  Example:

```
@files = $FTP->List("*.*", 3);
foreach $file (@files) {
  print $file->{'name'}, " ", $file->{'size'}, " ", $file->{'attr'}, "\n"
}
```

Note: all times are reported as strings of the following format: *second*, *hour*, *minute*, *day*, *month*, *year*.

Example:

```
    $file->{'mtime'} == "0,10,58,9,12,1996" stands for 09 Dec 1996 at 10:58:0
```

Mkdir name
Md name

Creates a directory on the FTP remote host. Returns `undef` on error.

Example:

```
    $FTP->Mkdir("NextBuild");
```

Mode [mode]

If called with no arguments, returns the current transfer mode for this FTP session ("asc" for ASCII or "bin" for binary). The *mode* argument can be "asc" or "bin", in which case the appropriate transfer mode is selected. See also the Ascii and Binary functions. Returns `undef` on errors.

Example:

```
    print "Current mode is: ", $FTP->Mode();
    $FTP->Mode("asc"); # ...  same as $FTP->Ascii();
```

Pasv [mode]

If called with no arguments, returns 1 if the current FTP session has passive transfer mode enabled, 0 if not.

You can call it with a *mode* parameter (0/1) only as a method of a Internet object (see `new`), in which case it will set the default value for the next `FTP` objects you create (read: set it before, because you can‘t change this value once you opened the FTP session).

Example:

```
    print "Pasv is: ", $FTP->Pasv();

    $INET->Pasv(1);
    $INET->FTP($FTP,"ftp.activeware.com", "anonymous", "dada\@divinf.it");
    $FTP->Pasv(0); # this will be ignored...
```

Put local, [remote, context]

Upload the *local* file to the FTP server saving it under the name *remote*, which if if omitted is the same name as *local*. Returns `undef` on error.

context is a number to identify this operation if it is asynchronous. See `SetStatusCallback` and `GetStatusCallback` for more info on asynchronous operations.

Example:

```
    $FTP->Put("internet.zip");
    $FTP->Put("d:/users/dada/temp.zip", "/temp/dada.zip");
```

Pwd

Returns the current directory on the FTP server or `undef` on errors.

Example:

```
    $path = $FTP->Pwd();
```

Rename oldfile, newfile
Ren oldfile, newfile

Renames a file on the FTP remote host. Returns `undef` on error.

Example:

```
    $FTP->Rename("110-i86.zip", "68i-011.zip");
```

Rmdir name
Rd name

Removes a directory on the FTP remote host. Returns `undef` on error.

Example:

```
    $FTP->Rmdir("CurrentBuild");
```

HTTP Functions

General Note

All methods assume that you have the following lines:

```
use Win32::Internet;
$INET = new Win32::Internet();
$INET->HTTP($HTTP, "hostname", "username", "password");
```

somewhere before the method calls; in other words, we assume that you have an Internet object called `$INET` and an open HTTP session called `$HTTP`.

See `new` and `HTTP` for more information.

Methods

AddHeader header, [flags]

Adds HTTP request headers to an HTTP request object created with `OpenRequest`. For the possible values of *flags* refer to the *"Microsoft Win32 Internet Functions"* document.

Example:

```
    $HTTP->OpenRequest($REQUEST,"/index.html");
    $REQUEST->AddHeader("If-Modified-Since:  Sunday, 17-Nov-96 11:40:03 GMT");
    $REQUEST->AddHeader("Accept: text/html\r\n", HTTP_ADDREQ_FLAG_REPLACE);
```

OpenRequest requestobject, [path, method, version, referer, accept, flags, context]
OpenRequest requestobject, hashref

Opens an HTTP request. Returns `undef` on errors or a number if the connection was succesful. You can then use one of the `AddHeader`, `SendRequest`, `QueryInfo`, `QueryDataAvailable` and `ReadFile` methods on the newly created *requestobject*. The parameters and their values are:

- path

  The object to request. This is generally a file name, an executable module, etc. Default: /

- method

  The method to use; can actually be GET, POST, HEAD or PUT. Default: GET

- version

  The HTTP version. Default: HTTP/1.0

- referer

  The URL of the document from which the URL in the request was obtained. Default: *none*

- accept

  The content types accepted. They must be separated by a "\0" (ASCII zero). Default: text/* image/gif image/jpeg

- flags

  Additional flags affecting the behavior of the function. Default: *none*

- context

  A number to identify this operation if it is asynchronous. See `SetStatusCallback` and `GetStatusCallback` for more info on asynchronous operations. Default: *none*

Refer to the *"Microsoft Win32 Internet Functions"* documentation for more details on those parameters. If you pass *hashref* (a reference to an hash array), the following values are taken from the array:

```
%hash=(
  "path"          => "path",
  "method"        => "method",
  "version"       => "version",
  "referer"       => "referer",
  "accept"        => "accept",
  "flags"         => flags,
  "context"       => context,
);
```

See also `Request`.

Example:

```
$HTTP->OpenRequest($REQUEST, "/index.html");
$HTTP->OpenRequest($REQUEST, "/index.html", "GET", "HTTP/0.9");

$params{"path"} = "/index.html";
$params{"flags"} = "
$HTTP->OpenRequest($REQUEST, \%params);
```

QueryInfo header, [flags]

Queries information about an HTTP request object created with `OpenRequest`. You can specify an *header* (for example, "Content−type") and/or one or more *flags*. If you don‘t specify *flags*, HTTP_QUERY_CUSTOM will be used by default; this means that *header* should contain a valid HTTP header name. For the possible values of *flags* refer to the *"Microsoft Win32 Internet Functions"* document.

Example:

```
$HTTP->OpenRequest($REQUEST,"/index.html");
$statuscode = $REQUEST->QueryInfo("", HTTP_QUERY_STATUS_CODE);
$headers = $REQUEST->QueryInfo("", HTTP_QUERY_RAW_HEADERS_CRLF); # will get a
$length = $REQUEST->QueryInfo("Content-length");
```

Request [path, method, version, referer, accept, flags]
Request hashref

Performs an HTTP request and returns an array containing the status code, the headers and the content of the file. It is a one−step procedure that makes an `OpenRequest`, a `SendRequest`, some `QueryInfo`, `ReadFile` and finally `Close`. For a description of the parameters, see `OpenRequest`.

Example:

```
($statuscode, $headers, $file) = $HTTP->Request("/index.html");
($s, $h, $f) = $HTTP->Request("/index.html", "GET", "HTTP/1.0");
```

SendRequest [postdata]

Send an HTTP request to the destination server. *postdata* are any optional data to send immediately after the request header; this is generally used for POST or PUT requests. See also `OpenRequest`.

Example:

```
            $HTTP->OpenRequest($REQUEST, "/index.html");
            $REQUEST->SendRequest();

            # A POST request...
            $HTTP->OpenRequest($REQUEST, "/cgi-bin/somescript.pl", "POST");

            #This line is a must -> (thanks Philip :)
            $REQUEST->AddHeader("Content-Type: application/x-www-form-urlencoded");

            $REQUEST->SendRequest("key1=value1&key2=value2&key3=value3");
```

APPENDIX

Microsoft Win32 Internet Functions

Complete documentation for the Microsoft Win32 Internet Functions can be found, in both HTML and zipped Word format, at this address:

```
http://www.microsoft.com/intdev/sdk/docs/wininet/
```

Functions Table

This table reports the correspondence between the functions offered by WININET.DLL and their implementation in the Win32::Internet extension. Functions showing a "—−" are not currently implemented. Functions enclosed in parens ( ) aren‘t implemented straightforwardly, but in a higher−level routine, eg. together with other functions.

WININET.DLL	Win32::Internet
InternetOpen	new Win32::Internet
InternetConnect	FTP / HTTP
InternetCloseHandle	Close
InternetQueryOption	QueryOption
InternetSetOption	SetOption
InternetSetOptionEx	---
InternetSetStatusCallback	SetStatusCallback
InternetStatusCallback	GetStatusCallback
InternetConfirmZoneCrossing	---
InternetTimeFromSystemTime	TimeConvert
InternetTimeToSystemTime	TimeConvert
InternetAttemptConnect	---
InternetReadFile	ReadFile
InternetSetFilePointer	---
InternetFindNextFile	(List)
InternetQueryDataAvailable	QueryDataAvailable
InternetGetLastResponseInfo	GetResponse
InternetWriteFile	---
InternetCrackUrl	CrackURL
InternetCreateUrl	CreateURL
InternetCanonicalizeUrl	CanonicalizeURL
InternetCombineUrl	CombineURL
InternetOpenUrl	OpenURL
FtpFindFirstFile	(List)
FtpGetFile	Get
FtpPutFile	Put
FtpDeleteFile	Delete
FtpRenameFile	Rename
FtpOpenFile	---
FtpCreateDirectory	Mkdir
FtpRemoveDirectory	Rmdir
FtpSetCurrentDirectory	Cd

`FtpGetCurrentDirectory`	`Pwd`
`HttpOpenRequest`	`OpenRequest`
`HttpAddRequestHeaders`	`AddHeader`
`HttpSendRequest`	`SendRequest`
`HttpQueryInfo`	`QueryInfo`
`InternetErrorDlg`	`---`

Actually, I don‘t plan to add support for Gopher, Cookie and Cache functions. I will if there will be consistent requests to do so.

There are a number of higher−level functions in the Win32::Internet that simplify some usual procedures, calling more that one WININET API function. This table reports those functions and the relative WININET functions they use.

`Win32::Internet`	`WININET.DLL`

`FetchURL`	`InternetOpenUrl`
	`InternetQueryDataAvailable`
	`InternetReadFile`
	`InternetCloseHandle`

`ReadEntireFile`	`InternetQueryDataAvailable` `InternetReadFile`

`Request`	`HttpOpenRequest`
	`HttpSendRequest`
	`HttpQueryInfo`
	`InternetQueryDataAvailable`
	`InternetReadFile`
	`InternetCloseHandle`

`List`	`FtpFindFirstFile` `InternetFindNextFile`

Constants

Those are the constants exported by the package in the main namespace (eg. you can use them in your scripts); for their meaning and proper use, refer to the Microsoft Win32 Internet Functions document.

```
HTTP_ADDREQ_FLAG_ADD
HTTP_ADDREQ_FLAG_REPLACE
HTTP_QUERY_ALLOW
HTTP_QUERY_CONTENT_DESCRIPTION
HTTP_QUERY_CONTENT_ID
HTTP_QUERY_CONTENT_LENGTH
HTTP_QUERY_CONTENT_TRANSFER_ENCODING
HTTP_QUERY_CONTENT_TYPE
HTTP_QUERY_COST
HTTP_QUERY_CUSTOM
HTTP_QUERY_DATE
HTTP_QUERY_DERIVED_FROM
HTTP_QUERY_EXPIRES
HTTP_QUERY_FLAG_REQUEST_HEADERS
HTTP_QUERY_FLAG_SYSTEMTIME
HTTP_QUERY_LANGUAGE
HTTP_QUERY_LAST_MODIFIED
HTTP_QUERY_MESSAGE_ID
HTTP_QUERY_MIME_VERSION
HTTP_QUERY_PRAGMA
HTTP_QUERY_PUBLIC
```

```
HTTP_QUERY_RAW_HEADERS
HTTP_QUERY_RAW_HEADERS_CRLF
HTTP_QUERY_REQUEST_METHOD
HTTP_QUERY_SERVER
HTTP_QUERY_STATUS_CODE
HTTP_QUERY_STATUS_TEXT
HTTP_QUERY_URI
HTTP_QUERY_USER_AGENT
HTTP_QUERY_VERSION
HTTP_QUERY_WWW_LINK
ICU_BROWSER_MODE
ICU_DECODE
ICU_ENCODE_SPACES_ONLY
ICU_ESCAPE
ICU_NO_ENCODE
ICU_NO_META
ICU_USERNAME
INTERNET_CONNECT_FLAG_PASSIVE
INTERNET_FLAG_ASYNC
INTERNET_FLAG_HYPERLINK
INTERNET_FLAG_KEEP_CONNECTION
INTERNET_FLAG_MAKE_PERSISTENT
INTERNET_FLAG_NO_AUTH
INTERNET_FLAG_NO_AUTO_REDIRECT
INTERNET_FLAG_NO_CACHE_WRITE
INTERNET_FLAG_NO_COOKIES
INTERNET_FLAG_READ_PREFETCH
INTERNET_FLAG_RELOAD
INTERNET_FLAG_RESYNCHRONIZE
INTERNET_FLAG_TRANSFER_ASCII
INTERNET_FLAG_TRANSFER_BINARY
INTERNET_INVALID_PORT_NUMBER
INTERNET_INVALID_STATUS_CALLBACK
INTERNET_OPEN_TYPE_DIRECT
INTERNET_OPEN_TYPE_PROXY
INTERNET_OPEN_TYPE_PROXY_PRECONFIG
INTERNET_OPTION_CONNECT_BACKOFF
INTERNET_OPTION_CONNECT_RETRIES
INTERNET_OPTION_CONNECT_TIMEOUT
INTERNET_OPTION_CONTROL_SEND_TIMEOUT
INTERNET_OPTION_CONTROL_RECEIVE_TIMEOUT
INTERNET_OPTION_DATA_SEND_TIMEOUT
INTERNET_OPTION_DATA_RECEIVE_TIMEOUT
INTERNET_OPTION_HANDLE_TYPE
INTERNET_OPTION_LISTEN_TIMEOUT
INTERNET_OPTION_PASSWORD
INTERNET_OPTION_READ_BUFFER_SIZE
INTERNET_OPTION_USER_AGENT
INTERNET_OPTION_USERNAME
INTERNET_OPTION_VERSION
INTERNET_OPTION_WRITE_BUFFER_SIZE
INTERNET_SERVICE_FTP
INTERNET_SERVICE_GOPHER
INTERNET_SERVICE_HTTP
```

```
INTERNET_STATUS_CLOSING_CONNECTION
INTERNET_STATUS_CONNECTED_TO_SERVER
INTERNET_STATUS_CONNECTING_TO_SERVER
INTERNET_STATUS_CONNECTION_CLOSED
INTERNET_STATUS_HANDLE_CLOSING
INTERNET_STATUS_HANDLE_CREATED
INTERNET_STATUS_NAME_RESOLVED
INTERNET_STATUS_RECEIVING_RESPONSE
INTERNET_STATUS_REDIRECT
INTERNET_STATUS_REQUEST_COMPLETE
INTERNET_STATUS_REQUEST_SENT
INTERNET_STATUS_RESOLVING_NAME
INTERNET_STATUS_RESPONSE_RECEIVED
INTERNET_STATUS_SENDING_REQUEST
```

VERSION HISTORY

- 0.081 (25 Sep 1999)
  - Documentation converted to pod format by Jan Dubois <jand@activestate.com.
  - Minor changes from Perl 5.005xx compatibility.
- 0.08 (14 Feb 1997)
  - fixed 2 more bugs in Option(s) related subs (thanks to Brian Helterline!).
  - `Error()` now gets error messages directly from WININET.DLL.
  - The PLL file now comes in 2 versions, one for Perl version 5.001 (build 100) and one for Perl version 5.003 (build 300 and higher). Everybody should be happy now :)
  - added an installation program.
- 0.07 (10 Feb 1997)
  - fixed a bug in `Version()` introduced with 0.06...
  - completely reworked PM file, fixed *lots* of minor bugs, and removed almost all the warnings with "perl −w".
- 0.06 (26 Jan 1997)
  - fixed another hideous bug in "new" (the ‘class’ parameter was still missing).
  - added support for asynchronous operations (work still in embryo).
  - removed the ending \0 (ASCII zero) from the DLL version returned by "Version".
  - added a lot of constants.
  - added `safefree()` after every `safemalloc()` in C... wonder why I didn‘t it before :)
  - added TimeConvert, which actually works one way only.
- 0.05f (29 Nov 1996)
  - fixed a bug in "new" (parameters passed were simply ignored).
  - fixed another bug: "Chdir" and "Cwd" were aliases of RMDIR instead of CD..
- 0.05 (29 Nov 1996)
  - added "CrackURL" and "CreateURL".
  - corrected an error in TEST.PL (there was a GetUserAgent instead of UserAgent).
- 0.04 (25 Nov 1996)
  - added "Version" to retrieve package and DLL versions.

  - added proxies and other options to "new".
  - changed "OpenRequest" and "Request" to read parameters from a hash.
  - added "SetOption/QueryOption" and a lot of relative functions (connect, username, password, useragent, etc.).
  - added "CanonicalizeURL" and "CombineURL".
  - "Error" covers a wider spectrum of errors.
- 0.02 (18 Nov 1996)
  - added support for HTTP sessions and requests.
- 0.01 (11 Nov 1996)
  - fetching of HTTP, FTP and GOPHER URLs.
  - complete set of commands to manage an FTP session.

AUTHOR

Version 0.08 (14 Feb 1997) by Aldo Calpini <a.calpini@romagiubileo.it

CREDITS

Win32::Internet is based on the Win32::Registry code written by Jesse Dougherty.

Additional thanks to: Carl Tichler for his help in the initial development; Tore Haraldsen, Brian Helterline for the bugfixes; Dave Roth for his great source code examples.

DISCLAIMER

This program is FREE; you can redistribute, modify, disassemble, or even reverse engineer this software at your will. Keep in mind, however, that NOTHING IS GUARANTEED to work and everything you do is AT YOUR OWN RISK − I will not take responsability for any damage, loss of money and/or health that may arise from the use of this program!

This is distributed under the terms of Larry Wall‘s Artistic License.

NAME

Win32::IPC − Base class for Win32 synchronization objects

SYNOPSIS

```
use Win32::Event 1.00 qw(wait_any);
#Create objects.

wait_any(@ListOfObjects,$timeout);
```

DESCRIPTION

This module is loaded by the other Win32 synchronization modules. You shouldn't need to load it yourself. It supplies the wait functions to those modules.

The synchronization modules are *"Win32::ChangeNotify"*, *"Win32::Event"*, *"Win32::Mutex"*, & *"Win32::Semaphore"*.

Methods

Win32::IPC supplies one method to all synchronization objects.

$obj−wait([$timeout])

Waits for `$obj` to become signalled. `$timeout` is the maximum time to wait (in milliseconds). If `$timeout` is omitted, waits forever. If `$timeout` is 0, returns immediately.

Returns:

```
   +1    The object is signalled
   -1    The object is an abandoned mutex
    0    Timed out
  undef  An error occurred
```

Functions

wait_any(@objects, [$timeout])

Waits for at least one of the `@objects` to become signalled. `$timeout` is the maximum time to wait (in milliseconds). If `$timeout` is omitted, waits forever. If `$timeout` is 0, returns immediately.

The return value indicates which object ended the wait:

```
   +N    $object[N-1] is signalled
   -N    $object[N-1] is an abandoned mutex
    0    Timed out
  undef  An error occurred
```

If more than one object became signalled, the one with the lowest index is used.

wait_all(@objects, [$timeout])

This is the same as `wait_any`, but it waits for all the `@objects` to become signalled. The return value indicates the last object to become signalled, and is negative if at least one of the `@objects` is an abandoned mutex.

Deprecated Functions and Methods

Win32::IPC still supports the ActiveWare syntax, but its use is deprecated.

INFINITE

Constant value for an infinite timeout. Omit the `$timeout` argument instead.

WaitForMultipleObjects(\@objects, $wait_all, $timeout)

Warning: `WaitForMultipleObjects` erases `@objects`! Use `wait_all` or `wait_any` instead.

`$obj–`Wait($timeout)

Similar to `not $obj->wait($timeout)`.

AUTHOR

Christopher J. Madsen <***chris_madsen@geocities.com***>

Loosely based on the original module by ActiveWare Internet Corp., ***http://www.ActiveWare.com***

NAME

Win32::Mutex − Use Win32 mutex objects from Perl

SYNOPSIS

```
require Win32::Mutex;

$mutex = Win32::Mutex->new($initial,$name);
$mutex->wait;
```

DESCRIPTION

This module allows access to the Win32 mutex objects. The `wait` method and `wait_all` & `wait_any` functions are inherited from the *"Win32::IPC"* module.

Methods

`$mutex` = Win32::Mutex−new([`$initial`, [`$name`]])

Constructor for a new mutex object. If `$initial` is true, requests immediate ownership of the mutex (default false). If `$name` is omitted, creates an unnamed mutex object.

If `$name` signifies an existing mutex object, then `$initial` is ignored and the object is opened.

`$mutex` = Win32::Mutex−open(`$name`)

Constructor for opening an existing mutex object.

`$mutex`−release

Release ownership of a `$mutex.` You should have obtained ownership of the mutex through `new` or one of the wait functions. Returns true if successful.

`$mutex`−wait([$timeout])

Wait for ownership of `$mutex.` See *"Win32::IPC"*.

Deprecated Functions and Methods

Win32::Mutex still supports the ActiveWare syntax, but its use is deprecated.

Create(`$MutObj,$Initial,$Name`)

Use `$MutObj = Win32::Mutex->new($Initial,$Name)` instead.

Open(`$MutObj,$Name`)

Use `$MutObj = Win32::Mutex->open($Name)` instead.

`$MutObj`−Release()

Use `$MutObj->release` instead.

AUTHOR

Christopher J. Madsen <***chris_madsen@geocities.com***>

Loosely based on the original module by ActiveWare Internet Corp., ***http://www.ActiveWare.com***

NAME

Win32::NetAdmin − manage network groups and users in perl

SYNOPSIS

```
use Win32::NetAdmin;
```

DESCRIPTION

This module offers control over the administration of groups and users over a network.

FUNCTIONS

NOTE

All of the functions return FALSE (0) if they fail, unless otherwise noted. `server` is optional for all the calls below. If not given the local machine is assumed.

GetDomainController(server, domain, returnedName)

Returns the name of the domain controller for server.

GetAnyDomainController(server, domain, returnedName)

Returns the name of any domain controller for a domain that is directly trusted by the server.

UserCreate(server, userName, password, passwordAge, privilege, homeDir, comment, flags, scriptPath)

Creates a user on server with password, passwordAge, privilege, homeDir, comment, flags, and scriptPath.

UserDelete(server, user)

Deletes a user from server.

UserGetAttributes(server, userName, password, passwordAge, privilege, homeDir, comment, flags, scriptPath)

Gets password, passwordAge, privilege, homeDir, comment, flags, and scriptPath for user.

UserSetAttributes(server, userName, password, passwordAge, privilege, homeDir, comment, flags, scriptPath)

Sets password, passwordAge, privilege, homeDir, comment, flags, and scriptPath for user.

UserChangePassword(domainname, username, oldpassword, newpassword)

Changes a users password. Can be run under any account.

UsersExist(server, userName)

Checks if a user exists.

GetUsers(server, filter, userRef)

Fills userRef with user names if it is an array reference and with the user names and the full names if it is a hash reference.

GroupCreate(server, group, comment)

Creates a group.

GroupDelete(server, group)

Deletes a group.

GroupGetAttributes(server, groupName, comment)

Gets the comment.

GroupSetAttributes(server, groupName, comment)

Sets the comment.

GroupAddUsers(server, groupName, users)

Adds a user to a group.

GroupDeleteUsers(server, groupName, users)

Deletes a users from a group.

GroupIsMember(server, groupName, user)

Returns TRUE if user is a member of groupName.

GroupGetMembers(server, groupName, userArrayRef)

Fills userArrayRef with the members of groupName.

LocalGroupCreate(server, group, comment)

Creates a local group.

LocalGroupDelete(server, group)

Deletes a local group.

LocalGroupGetAttributes(server, groupName, comment)

Gets the comment.

LocalGroupSetAttributes(server, groupName, comment)

Sets the comment.

LocalGroupIsMember(server, groupName, user)

Returns TRUE if user is a member of groupName.

LocalGroupGetMembers(server, groupName, userArrayRef)

Fills userArrayRef with the members of groupName.

LocalGroupGetMembersWithDomain(server, groupName, userRef)

This function is similar LocalGroupGetMembers but accepts an array or a hash reference. Unlike LocalGroupGetMembers it returns each user name as `DOMAIN\USERNAME`. If a hash reference is given, the function returns to each user or group name the type (group, user, alias etc.). The possible types are as follows:

```
$SidTypeUser = 1;
$SidTypeGroup = 2;
$SidTypeDomain = 3;
$SidTypeAlias = 4;
$SidTypeWellKnownGroup = 5;
$SidTypeDeletedAccount = 6;
$SidTypeInvalid = 7;
$SidTypeUnknown = 8;
```

LocalGroupAddUsers(server, groupName, users)

Adds a user to a group.

LocalGroupDeleteUsers(server, groupName, users)

Deletes a users from a group.

GetServers(server, domain, flags, serverRef)

Gets an array of server names or an hash with the server names and the comments as seen in the Network Neighborhood or the server manager. For flags, see SV_TYPE_* constants.

GetTransports(server, transportRef)

Enumerates the network transports of a computer. If transportRef is an array reference, it is filled with the transport names. If transportRef is a hash reference then a hash of hashes is filled with the data for the transports.

LoggedOnUsers(server, userRef)

Gets an array or hash with the users logged on at the specified computer. If userRef is a hash reference, the value is a semikolon separated string of username, logon domain and logon server.

GetAliasFromRID(server, RID, returnedName)
GetUserGroupFromRID(server, RID, returnedName)

Retrieves the name of an alias (i.e local group) or a user group for a RID from the specified server. These functions can be used for example to get the account name for the administrator account if it is renamed or localized.

Possible values for `RID`:

```
DOMAIN_ALIAS_RID_ACCOUNT_OPS
DOMAIN_ALIAS_RID_ADMINS
DOMAIN_ALIAS_RID_BACKUP_OPS
DOMAIN_ALIAS_RID_GUESTS
DOMAIN_ALIAS_RID_POWER_USERS
DOMAIN_ALIAS_RID_PRINT_OPS
DOMAIN_ALIAS_RID_REPLICATOR
DOMAIN_ALIAS_RID_SYSTEM_OPS
DOMAIN_ALIAS_RID_USERS
DOMAIN_GROUP_RID_ADMINS
DOMAIN_GROUP_RID_GUESTS
DOMAIN_GROUP_RID_USERS
DOMAIN_USER_RID_ADMIN
DOMAIN_USER_RID_GUEST
```

GetServerDisks(server, arrayRef)

Returns an array with the disk drives of the specified server. The array contains two−character strings (drive letter followed by a colon).

EXAMPLE

```
# Simple script using Win32::NetAdmin to set the login script for
# all members of the NT group "Domain Users".  Only works if you
# run it on the PDC. (From Robert Spier <rspier@seas.upenn.edu>)
#
# FILTER_TEMP_DUPLICATE_ACCOUNTS
#   Enumerates local user account data on a domain controller.
#
# FILTER_NORMAL_ACCOUNT
#   Enumerates global user account data on a computer.
#
# FILTER_INTERDOMAIN_TRUST_ACCOUNT
#   Enumerates domain trust account data on a domain controller.
#
# FILTER_WORKSTATION_TRUST_ACCOUNT
#   Enumerates workstation or member server account data on a domain
#   controller.
#
# FILTER_SERVER_TRUST_ACCOUNT
#   Enumerates domain controller account data on a domain controller.

use Win32::NetAdmin qw(GetUsers GroupIsMember
                       UserGetAttributes UserSetAttributes);
```

```
my %hash;
GetUsers("", FILTER_NORMAL_ACCOUNT , \%hash)
    or die "GetUsers() failed: $^E";

foreach (keys %hash) {
    my ($password, $passwordAge, $privilege,
        $homeDir, $comment, $flags, $scriptPath);
    if (GroupIsMember("", "Domain Users", $_)) {
        print "Updating $_ ($hash{$_})\n";
        UserGetAttributes("", $_, $password, $passwordAge, $privilege,
                          $homeDir, $comment, $flags, $scriptPath)
            or die "UserGetAttributes() failed: $^E";
        $scriptPath = "dnx_login.bat"; # this is the new login script
        UserSetAttributes("", $_, $password, $passwordAge, $privilege,
                          $homeDir, $comment, $flags, $scriptPath)
            or die "UserSetAttributes() failed: $^E";
    }
}
```

NAME

Win32::NetResource − manage network resources in perl

SYNOPSIS

```
use Win32::NetResource;

$ShareInfo = {
                'path' => "C:\\MyShareDir",
                'netname' => "MyShare",
                'remark' => "It is good to share",
                'passwd' => "",
                'current-users' =>0,
                'permissions' => 0,
                'maxusers' => -1,
                'type'  => 0,
                };

Win32::NetResource::NetShareAdd( $ShareInfo,$parm )
    or die "unable to add share";
```

DESCRIPTION

This module offers control over the network resources of Win32.Disks, printers etc can be shared over a network.

DATA TYPES

There are two main data types required to control network resources. In Perl these are represented by hash types.

%NETRESOURCE

```
KEY                       VALUE

'Scope'         =>  Scope of an Enumeration
                    RESOURCE_CONNECTED,
                    RESOURCE_GLOBALNET,
                    RESOURCE_REMEMBERED.

'Type'          =>  The type of resource to Enum
                    RESOURCETYPE_ANY    All resources
                    RESOURCETYPE_DISK    Disk resources
                    RESOURCETYPE_PRINT    Print resources

'DisplayType'   =>  The way the resource should be displayed.
                    RESOURCEDISPLAYTYPE_DOMAIN
                    The object should be displayed as a domain.
                    RESOURCEDISPLAYTYPE_GENERIC
                    The method used to display the object does not
                    RESOURCEDISPLAYTYPE_SERVER
                    The object should be displayed as a server.
                    RESOURCEDISPLAYTYPE_SHARE
                    The object should be displayed as a sharepoint.

'Usage'         =>  Specifies the Resources usage:
                    RESOURCEUSAGE_CONNECTABLE
                    RESOURCEUSAGE_CONTAINER.

'LocalName'     =>  Name of the local device the resource is
                    connected to.

'RemoteName'    =>  The network name of the resource.
```

```
                    'Comment'       =>  A string comment.

                    'Provider'      =>  Name of the provider of the resource.
```

%SHARE_INFO

This hash represents the SHARE_INFO_502 struct.

```
                    KEY                    VALUE
                    'netname'        =>    Name of the share.
                    'type'           =>    type of share.
                    'remark'         =>    A string comment.
                    'permissions'    =>    Permissions value
                    'maxusers'       =>    the max # of users.
                    'current-users'  =>    the current # of users.
                    'path'           =>    The path of the share.
                    'passwd'         =>    A password if one is req'd
```

FUNCTIONS

NOTE

All of the functions return FALSE (0) if they fail.

GetSharedResources(\@Resources,dwType)

Creates a list in @Resources of %NETRESOURCE hash references.

The return value indicates whether there was an error in accessing any of the shared resources. All the shared resources that were encountered (until the point of an error, if any) are pushed into @Resources as references to %NETRESOURCE hashes. See example below.

AddConnection(\%NETRESOURCE,`$Password`,`$UserName`,`$Connection`)

Makes a connection to a network resource specified by %NETRESOURCE

CancelConnection(`$Name`,`$Connection`,`$Force`)

Cancels a connection to a network resource connected to local device `$name`.`$Connection` is either 1 − persistent connection or 0, non−persistent.

WNetGetLastError(`$ErrorCode`,`$Description`,`$Name`)

Gets the Extended Network Error.

GetError(`$ErrorCode`)

Gets the last Error for a Win32::NetResource call.

GetUNCName(`$UNCName`, `$LocalPath`);

Returns the UNC name of the disk share connected to `$LocalPath` in `$UNCName`.

NOTE

`$servername` is optional for all the calls below. (if not given the local machine is assumed.)

NetShareAdd(\%SHARE,`$parm_err`,`$servername` = NULL)

Add a share for sharing.

NetShareCheck(`$device`,`$type`,`$servername` = NULL)

Check if a share is available for connection.

NetShareDel(`$netname`, `$servername` = NULL)

Remove a share from a machines list of shares.

NetShareGetInfo(**$netname, \%SHARE,$servername=NULL)**

Get the %SHARE_INFO information about the share `$netname` on the server `$servername`.

NetShareSetInfo(`$netname,\%SHARE,$parm_err,$servername=NULL`)

Set the information for share `$netname`.

EXAMPLE

```
#
# This example displays all the share points in the entire
# visible part of the network.
#

use strict;
use Win32::NetResource qw(:DEFAULT GetSharedResources GetError);
my $resources = [];
unless(GetSharedResources($resources, RESOURCETYPE_ANY)) {
    my $err = undef;
    GetError($err);
    warn Win32::FormatMessage($err);
}

foreach my $href (@$resources) {
    next if ($$href{DisplayType} != RESOURCEDISPLAYTYPE_SHARE);
    print "-----\n";
    foreach( keys %$href){
        print "$_: $href->{$_}\n";
    }
}
```

AUTHOR

Jesse Dougherty for Hip Communications.

Additional general cleanups and bug fixes by Gurusamy Sarathy <gsar@activestate.com.

NAME

Win32::ODBC − ODBC Extension for Win32

SYNOPSIS

To use this module, include the following statement at the top of your script:

```
use Win32::ODBC;
```

Next, create a data connection to your DSN:

```
$Data = new Win32::ODBC("MyDSN");
```

NOTE: *MyDSN* can be either the *DSN* as defined in the ODBC Administrator, *or* it can be an honest−to−God *DSN Connect String*.

```
Example: "DSN=My Database;UID=Brown Cow;PWD=Moo;"
```

You should check to see if `$Data` is indeed defined, otherwise there has been an error.

You can now send SQL queries and retrieve info to your heart‘s content! See the description of the methods provided by this module below and also the file ***TEST.PL*** as referred to in *INSTALLATION NOTES* to see how it all works.

Finally, **MAKE SURE** that you close your connection when you are finished:

```
$Data−>Close();
```

DESCRIPTION

Background

This is a hack of Dan DeMaggio‘s <dmag@umich.edu ***NTXS.C*** ODBC implementation. I have recoded and restructured most of it including most of the ***ODBC.PM*** package, but its very core is still based on Dan‘s code (thanks Dan!).

The history of this extension is found in the file ***HISTORY.TXT*** that comes with the original archive (see *INSTALLATION NOTES* below).

Benefits

And what are the benefits of this module?

- The number of ODBC connections is limited by memory and ODBC itself (have as many as you want!).
- The working limit for the size of a field is 10,240 bytes, but you can increase that limit (if needed) to a max of 2,147,483,647 bytes. (You can always recompile to increase the max limit.)
- You can open a connection by either specifing a DSN or a connection string!
- You can open and close the connections in any order!
- Other things that I can not think of right now... :)

CONSTANTS

This package defines a number of constants. You may refer to each of these constants using the notation `ODBC::xxxxx`, where `xxxxx` is the constant.

Example:

```
print ODBC::SQL_SQL_COLUMN_NAME, "\n";
```

SPECIAL NOTATION

For the method documentation that follows, an ***** following the method parameters indicates that that method is new or has been modified for this version.

CONSTRUCTOR

new (ODBC_OBJECT | DSN [, (OPTION1, VALUE1), (OPTION2, VALUE2) ...]) *

Creates a new ODBC connection based on `DSN`, or, if you specify an already existing ODBC object, then a new ODBC object will be created but using the ODBC Connection specified by `ODBC_OBJECT`. (The new object will be a new *hstmt* using the *hdbc* connection in `ODBC_OBJECT`.)

`DSN` is *Data Source Name* or a proper `ODBCDriverConnect` string.

You can specify SQL Connect Options that are implemented before the actual connection to the DSN takes place. These option/values are the same as specified in `GetConnectOption`/`SetConnectOption` (see below) and are defined in the ODBC API specs.

Returns a handle to the database on success, or *undef* on failure.

METHODS

Catalog (QUALIFIER, OWNER, NAME, TYPE)

Tells ODBC to create a data set that contains table information about the DSN. Use `Fetch` and `Data` or `DataHash` to retrieve the data. The returned format is:

```
[Qualifier] [Owner] [Name] [Type]
```

Returns *true* on error.

ColAttributes (ATTRIBUTE [, FIELD_NAMES])

Returns the attribute `ATTRIBUTE` on each of the fields in the list `FIELD_NAMES` in the current record set. If `FIELD_NAMES` is empty, then all fields are assumed. The attributes are returned as an associative array.

ConfigDSN (OPTION, DRIVER, ATTRIBUTE1 [, ATTRIBUTE2, ATTRIBUTE3, ...])

Configures a DSN. `OPTION` takes on one of the following values:

```
ODBC_ADD_DSN.......Adds a new DSN.
ODBC_MODIFY_DSN....Modifies an existing DSN.
ODBC_REMOVE_DSN....Removes an existing DSN.

ODBC_ADD_SYS_DSN.......Adds a new System DSN.
ODBC_MODIFY_SYS_DSN....Modifies an existing System DSN.
ODBC_REMOVE_SYS_DSN....Removes an existing System DSN.
```

You must specify the driver `DRIVER` (which can be retrieved by using `DataSources` or `Drivers`).

`ATTRIBUTE1` **should** be *"DSN=xxx"* where *xxx* is the name of the DSN. Other attributes can be any DSN attribute such as:

```
"UID=Cow"
"PWD=Moo"
"Description=My little bitty Data Source Name"
```

Returns *true* on success, *false* on failure.

NOTE 1: If you use `ODBC_ADD_DSN`, then you must include at least *"DSN=xxx"* and the location of the database.

Example: For MS Access databases, you must specify the *DatabaseQualifier*:

```
"DBQ=c:\\...\\MyDatabase.mdb"
```

NOTE 2: If you use `ODBC_MODIFY_DSN`, then you need only specify the *"DNS=xxx"* attribute. Any other attribute you include will be changed to what you specify.

NOTE 3: If you use `ODBC_REMOVE_DSN`, then you need only specify the *"DSN=xxx"* attribute.

Connection `()`

Returns the connection number associated with the ODBC connection.

Close `()`

Closes the ODBC connection. No return value.

Data ([FIELD_NAME])

Returns the contents of column name `FIELD_NAME` or the current row (if nothing is specified).

DataHash ([FIELD1, FIELD2, ...])

Returns the contents for `FIELD1, FIELD2, ...` or the entire row (if nothing is specified) as an associative array consisting of:

```
{Field Name} => Field Data
```

DataSources `()`

Returns an associative array of Data Sources and ODBC remarks about them. They are returned in the form of:

```
$ArrayName{'DSN'}=Driver
```

where *DSN* is the Data Source Name and ODBC Driver used.

Debug ([1 | 0])

Sets the debug option to on or off. If nothing is specified, then nothing is changed.

Returns the debugging value (*1* or).

Drivers `()`

Returns an associative array of ODBC Drivers and their attributes. They are returned in the form of:

```
$ArrayName{'DRIVER'}=Attrib1;Attrib2;Attrib3;...
```

where *DRIVER* is the ODBC Driver Name and *AttribX* are the driver−defined attributes.

DropCursor ([CLOSE_TYPE])

Drops the cursor associated with the ODBC object. This forces the cursor to be deallocated. This overrides `SetStmtCloseType`, but the ODBC object does not lose the `StmtCloseType` setting. `CLOSE_TYPE` can be any valid `SmtpCloseType` and will perform a close on the stmt using the specified close type.

Returns *true* on success, *false* on failure.

DumpData `()`

Dumps to the screen the fieldnames and all records of the current data set. Used primarily for debugging. No return value.

Error `()`

Returns the last encountered error. The returned value is context dependent:

If called in a *scalar* context, then a *3−element array* is returned:

```
( ERROR_NUMBER, ERROR_TEXT, CONNECTION_NUMBER )
```

If called in a *string* context, then a *string* is returned:

```
"[ERROR_NUMBER] [CONNECTION_NUMBER] [ERROR_TEXT]"
```

If debugging is on then two more variables are returned:

```
( ..., FUNCTION, LEVEL )
```

where `FUNCTION` is the name of the function in which the error occurred, and `LEVEL` represents extra information about the error (usually the location of the error).

FetchRow ([ROW [, TYPE]])

Retrieves the next record from the keyset. When `ROW` and/or `TYPE` are specified, the call is made using `SQLExtendedFetch` instead of `SQLFetch`.

NOTE 1: If you are unaware of `SQLExtendedFetch` and its implications, stay with just regular `FetchRow` with no parameters.

NOTE 2: The ODBC API explicitly warns against mixing calls to `SQLFetch` and `SQLExtendedFetch`; use one or the other but not both.

If *ROW* is specified, it moves the keyset **RELATIVE** `ROW` number of rows.

If *ROW* is specified and `TYPE` is **not**, then the type used is **RELATIVE**.

Returns *true* when another record is available to read, and *false* when there are no more records.

FieldNames `()`

Returns an array of fieldnames found in the current data set. There is no guarantee on order.

GetConnections `()`

Returns an array of connection numbers showing what connections are currently open.

GetConnectOption (OPTION)

Returns the value of the specified connect option `OPTION`. Refer to ODBC documentation for more information on the options and values.

Returns a string or scalar depending upon the option specified.

GetCursorName `()`

Returns the name of the current cursor as a string or *undef*.

GetData `()`

Retrieves the current row from the dataset. This is not generally used by users; it is used internally.

Returns an array of field data where the first element is either *false* (if successful) and *true* (if **not** successful).

getDSN ([DSN])

Returns an associative array indicating the configuration for the specified DSN.

If no DSN is specified then the current connection is used.

The returned associative array consists of:

```
keys=DSN keyword; values=Keyword value. $Data{$Keyword}=Value
```

GetFunctions ([FUNCTION1, FUNCTION2, ...])

Returns an associative array indicating the ability of the ODBC Driver to support the specified functions. If no functions are specified, then a 100 element associative array is returned containing all possible functions and their values.

`FUNCTION` must be in the form of an ODBC API constant like `SQL_API_SQLTRANSACT`.

The returned array will contain the results like:

```
$Results{SQL_API_SQLTRANSACT}=Value
```

Example:

```
$Results = $O->GetFunctions(
                            $O->SQL_API_SQLTRANSACT,
                            SQL_API_SQLSETCONNECTOPTION
                           );
$ConnectOption = $Results{SQL_API_SQLSETCONNECTOPTION};
$Transact = $Results{SQL_API_SQLTRANSACT};
```

GetInfo (OPTION)

Returns a string indicating the value of the particular option specified.

GetMaxBufSize `()`

Returns the current allocated limit for *MaxBufSize*. For more info, see `SetMaxBufSize`.

GetSQLState `()` *

Returns a string indicating the SQL state as reported by ODBC. The SQL state is a code that the ODBC Manager or ODBC Driver returns after the execution of a SQL function. This is helpful for debugging purposes.

GetStmtCloseType ([CONNECTION])

Returns a string indicating the type of closure that will be used everytime the *hstmt* is freed. See `SetStmtCloseType` for details.

By default, the connection of the current object will be used. If `CONNECTION` is a valid connection number, then it will be used.

GetStmtOption (OPTION)

Returns the value of the specified statement option `OPTION`. Refer to ODBC documentation for more information on the options and values.

Returns a string or scalar depending upon the option specified.

MoreResults `()`

This will report whether there is data yet to be retrieved from the query. This can happen if the query was a multiple select.

Example:

```
"SELECT * FROM [foo] SELECT * FROM [bar]"
```

NOTE: Not all drivers support this.

Returns *1* if there is more data, *undef* otherwise.

RowCount (CONNECTION)

For *UPDATE*, *INSERT* and *DELETE* statements, the returned value is the number of rows affected by the request or *−1* if the number of affected rows is not available.

NOTE 1: This function is not supported by all ODBC drivers! Some drivers do support this but not for all statements (e.g., it is supported for *UPDATE*, *INSERT* and *DELETE* commands but not for the *SELECT* command).

NOTE 2: Many data sources cannot return the number of rows in a result set before fetching them; for maximum interoperability, applications should not rely on this behavior.

Returns the number of affected rows, or *−1* if not supported by the driver in the current context.

Run (SQL)

Executes the SQL command **SQL** and dumps to the screen info about it. Used primarily for debugging.

No return value.

SetConnectOption (OPTION) *

Sets the value of the specified connect option **OPTION**. Refer to ODBC documentation for more information on the options and values.

Returns *true* on success, *false* otherwise.

SetCursorName (NAME) *

Sets the name of the current cursor.

Returns *true* on success, *false* otherwise.

SetPos (ROW [, OPTION, LOCK]) *

Moves the cursor to the row `ROW` within the current keyset (**not** the current data/result set).

Returns *true* on success, *false* otherwise.

SetMaxBufSize (SIZE)

This sets the *MaxBufSize* for a particular connection. This will most likely never be needed but...

The amount of memory that is allocated to retrieve the field data of a record is dynamic and changes when it need to be larger. I found that a memo field in an MS Access database ended up requesting 4 Gig of space. This was a bit much so there is an imposed limit (2,147,483,647 bytes) that can be allocated for data retrieval.

Since it is possible that someone has a database with field data greater than 10,240, you can use this function to increase the limit up to a ceiling of 2,147,483,647 (recompile if you need more).

Returns the max number of bytes.

SetStmtCloseType (TYPE [, CONNECTION])

Sets a particular *hstmt* close type for the connection. This is the same as `ODBCFreeStmt(hstmt, TYPE)`. By default, the connection of the current object will be used. If `CONNECTION` is a valid connection number, then it will be used.

`TYPE` may be one of:

```
SQL_CLOSE
SQL_DROP
SQL_UNBIND
SQL_RESET_PARAMS
```

Returns a string indicating the newly set type.

SetStmtOption (OPTION) *

Sets the value of the specified statement option `OPTION`. Refer to ODBC documentation for more information on the options and values.

Returns *true* on success, *false* otherwise.

ShutDown `()`

Closes the ODBC connection and dumps to the screen info about it. Used primarily for debugging.

No return value.

Sql (SQL_STRING)

Executes the SQL command `SQL_STRING` on the current connection.

Returns *?* on success, or an error number on failure.

TableList (QUALIFIER, OWNER, NAME, TYPE)

Returns the catalog of tables that are available in the DSN. For an unknown parameter, just specify the empty string "".

Returns an array of table names.

Transact (TYPE) *

Forces the ODBC connection to perform a *rollback* or *commit* transaction.

`TYPE` may be one of:

```
SQL_COMMIT
SQL_ROLLBACK
```

NOTE: This only works with ODBC drivers that support transactions. Your driver supports it if *true* is returned from:

```
$O->GetFunctions($O->SQL_API_SQLTRANSACT)[1]
```

(See `GetFunctions` for more details.)

Returns *true* on success, *false* otherwise.

Version (PACKAGES)

Returns an array of version numbers for the requested packages (***ODBC.pm*** or ***ODBC.PLL***). If the list `PACKAGES` is empty, then all version numbers are returned.

LIMITATIONS

What known problems does this thing have?

- If the account under which the process runs does not have write permission on the default directory (for the process, not the ODBC DSN), you will probably get a runtime error during a `SQLConnection`. I don‘t think that this is a problem with the code, but more like a problem with ODBC. This happens because some ODBC drivers need to write a temporary file. I noticed this using the MS Jet Engine (Access Driver).
- This module has been neither optimized for speed nor optimized for memory consumption.

INSTALLATION NOTES

If you wish to use this module with a build of Perl other than ActivePerl, you may wish to fetch the original source distribution for this module at:

```
ftp://ftp.roth.net:/pub/ntperl/ODBC/970208/Bin/Win32_ODBC_Build_CORE.zip
```

or one of the other archives at that same location. See the included README for hints on installing this module manually, what to do if you get a *parse exception*, and a pointer to a test script for this module.

OTHER DOCUMENTATION

Find a FAQ for Win32::ODBC at:

```
http://www.roth.net/odbc/odbcfaq.htm
```

AUTHOR

Dave Roth <rothd@roth.net

CREDITS

Based on original code by Dan DeMaggio <dmag@umich.edu

DISCLAIMER

I do not guarantee **ANYTHING** with this package. If you use it you are doing so **AT YOUR OWN RISK**! I may or may not support this depending on my time schedule.

HISTORY

Last Modified 1999.09.25.

COPYRIGHT

NAME

Win32::OLE − OLE Automation extensions

SYNOPSIS

```
$ex = Win32::OLE->new('Excel.Application') or die "oops\n";
$ex->Amethod("arg")->Bmethod->{'Property'} = "foo";
$ex->Cmethod(undef,undef,$Arg3);
$ex->Dmethod($RequiredArg1, {NamedArg1 => $Value1, NamedArg2 => $Value2});

$wd = Win32::OLE->GetObject("D:\\Data\\Message.doc");
$xl = Win32::OLE->GetActiveObject("Excel.Application");
```

DESCRIPTION

This module provides an interface to OLE Automation from Perl. OLE Automation brings VisualBasic like scripting capabilities and offers powerful extensibility and the ability to control many Win32 applications from Perl scripts.

The Win32::OLE module uses the IDispatch interface exclusively. It is not possible to access a custom OLE interface. OLE events and OCX‘s are currently not supported.

Actually, that‘s no longer strictly true. This module now contains **ALPHA** level support for OLE events. This is largely untested and the specific interface might still change in the future.

Methods

Win32::OLE−new(PROGID[, DESTRUCTOR])

The `new()` class method starts a new instance of an OLE Automation object. It returns a reference to this object or `undef` if the creation failed.

The PROGID argument must be either the OLE *program id* or the *class id* of the required application. The optional DESTRUCTOR specifies a DESTROY−like method. This can be either a CODE reference or a string containing an OLE method name. It can be used to cleanly terminate OLE applications in case the Perl program dies.

To create an object via DCOM on a remote server you can use an array reference in place of PROGID. The referenced array must contain the machine name and the *program id* or *class id*. For example:

```
my $obj = Win32::OLE->new(['my.machine.com', 'Program.Id']);
```

If the PROGID is a *program id* then Win32::OLE will try to resolve the corresponding *class id* locally. If the *program id* is not registered locally then the remote registry is queried. This will only succeed if the local process has read access to the remote registry. The safest (and fastest) method is to specify the `class id` directly.

Win32::OLE−EnumAllObjects([CALLBACK])

This class method returns the number Win32::OLE objects currently in existance. It will call the optional CALLBACK function for each of these objects:

```
$Count = Win32::OLE->EnumAllObjects(sub {
    my $Object = shift;
    my $Class = Win32::OLE->QueryObjectType($Object);
    printf "# Object=%s Class=%s\n", $Object, $Class;
});
```

The `EnumAllObjects()` method is primarily a debugging tool. It can be used e.g. in an END block to check if all external connections have been properly destroyed.

Win32::OLE−FreeUnusedLibraries()

The `FreeUnusedLibraries()` class method unloads all unused OLE resources. These are the libraries of those classes of which all existing objects have been destroyed. The unloading of object libraries is really only important for long running processes that might instantiate a huge number of **different** objects over time.

Be aware that objects implemented in Visual Basic have a buggy implementation of this functionality: They pretend to be unloadable while they are actually still running their cleanup code. Unloading the DLL at that moment typically produces an access violation. The probability for this problem can be reduced by calling the `SpinMessageLoop()` method and `sleep()`ing for a few seconds.

Win32::OLE−GetActiveObject(CLASS[, DESTRUCTOR])

The `GetActiveObject()` class method returns an OLE reference to a running instance of the specified OLE automation server. It returns `undef` if the server is not currently active. It will croak if the class is not even registered. The optional DESTRUCTOR method takes either a method name or a code reference. It is executed when the last reference to this object goes away. It is generally considered `impolite` to stop applications that you did not start yourself.

Win32::OLE−GetObject(MONIKER[, DESTRUCTOR])

The `GetObject()` class method returns an OLE reference to the specified object. The object is specified by a pathname optionally followed by additional item subcomponent separated by exclamation marks '!'. The optional DESTRUCTOR argument has the same semantics as the DESTRUCTOR in `new()` or `GetActiveObject()`.

Win32::OLE−Initialize([COINIT])

The `Initialize()` class method can be used to specify an alternative apartment model for the Perl thread. It must be called **before** the first OLE object is created. If the `Win32::OLE::Const` module is used then the call to the `Initialize()` method must be made from a BEGIN block before the first `use` statement for the `Win32::OLE::Const` module.

Valid values for COINIT are:

```
  Win32::OLE::COINIT_APARTMENTTHREADED  - single threaded
  Win32::OLE::COINIT_MULTITHREADED      - the default
  Win32::OLE::COINIT_OLEINITIALIZE      - single threaded, additional OLE stu
```

COINIT_OLEINITIALIZE is sometimes needed when an OLE object uses additional OLE compound document technologies not available from the normal COM subsystem (for example MAPI.Session seems to require it). Both COINIT_OLEINITIALIZE and COINIT_APARTMENTTHREADED create a hidden top level window and a message queue for the Perl process. This may create problems with other application, because Perl normally doesn‘t process its message queue. This means programs using synchronous communication between applications (such as DDE initiation), may hang until Perl makes another OLE method call/property access or terminates. This applies to InstallShield setups and many things started to shell associations. Please try to utilize the `Win32::OLE->SpinMessageLoop` and `Win32::OLE->Uninitialize` methods if you can not use the default COINIT_MULTITHREADED model.

OBJECT−Invoke(METHOD[, ARGS])

The `Invoke()` object method is an alternate way to invoke OLE methods. It is normally equivalent to `$OBJECT-METHOD(@ARGS)`. This function must be used if the METHOD name contains characters not valid in a Perl variable name (like foreign language characters). It can also be used to invoke the default method of an object even if the default method has not been given a name in the type library. In this case use <undef or `''` as the method name. To invoke an OLE objects native `Invoke()` method (if such a thing exists), please use:

```
$Object->Invoke('Invoke', @Args);
```

`Win32::OLE`−LastError()

The `LastError()` class method returns the last recorded OLE error. This is a dual value like the `$!` variable: in a numeric context it returns the error number and in a string context it returns the error message. The error number is a signed HRESULT value. Please use the *HRESULT(ERROR)* function to convert an unsigned hexadecimal constant to a signed HRESULT.

The last OLE error is automatically reset by a successful OLE call. The numeric value can also explicitly be set by a call (which will discard the string value):

```
Win32::OLE->LastError(0);
```

OBJECT−LetProperty(NAME,ARGS,VALUE)

In Win32::OLE property assignment using the hash syntax is equivalent to the Visual Basic `Set` syntax (*by reference* assignment):

```
$Object->{Property} = $OtherObject;
```

corresponds to this Visual Basic statement:

```
Set Object.Property = OtherObject
```

To get the *by value* treatment of the Visual Basic `Let` statement

```
Object.Property = OtherObject
```

you have to use the `LetProperty()` object method in Perl:

```
$Object->LetProperty($Property, $OtherObject);
```

`LetProperty()` also supports optional arguments for the property assignment. See *OBJECT−*SetProperty(NAME,ARGS,VALUE) for details.

`Win32::OLE`−MessageLoop()

The `MessageLoop()` class method will run a standard Windows message loop, dispatching messages until the `QuitMessageLoop()` class method is called. It is used to wait for OLE events.

Win32::OLE−Option(OPTION)

The `Option()` class method can be used to inspect and modify *Module Options*. The single argument form retrieves the value of an option:

```
my $CP = Win32::OLE->Option('CP');
```

A single call can be used to set multiple options simultaneously:

```
Win32::OLE->Option(CP => CP_ACP, Warn => 3);
```

Win32::OLE−QueryObjectType(OBJECT)

The `QueryObjectType()` class method returns a list of the type library name and the objects class name. In a scalar context it returns the class name only. It returns `undef` when the type information is not available.

`Win32::OLE`−QuitMessageLoop()

The `QuitMessageLoop()` class method posts a (user−level) "Quit" message to the current threads message loop. `QuitMessageLoop()` is typically called from an event handler. The `MessageLoop()` class method will return when it receives this "Quit" method.

OBJECT−SetProperty(NAME,ARGS,VALUE)

The `SetProperty()` method allows to modify properties with arguments, which is not supported by the hash syntax. The hash form

```
$Object->{Property} = $Value;
```

is equivalent to

```
$Object->SetProperty('Property', $Value);
```

Arguments must be specified between the property name and the new value:

```
$Object->SetProperty('Property', @Args, $Value);
```

It is not possible to use "named argument" syntax with this function because the new value must be the last argument to `SetProperty()`.

This method hides any native OLE object method called `SetProperty()`. The native method will still be available through the `Invoke()` method:

```
$Object->Invoke('SetProperty', @Args);
```

Win32::OLE−SpinMessageLoop

This class method retrieves all pending messages from the message queue and dispatches them to their respective window procedures. Calling this method is only necessary when not using the COINIT_MULTITHREADED model. All OLE method calls and property accesses automatically process the message queue.

Win32::OLE−Uninitialize

The `Uninitialize()` class method uninitializes the OLE subsystem. It also destroys the hidden top level window created by OLE for single threaded apartments. All OLE objects will become invalid after this call! It is possible to call the `Initialize()` class method again with a different apartment model after shutting down OLE with `Uninitialize()`.

Win32::OLE−WithEvents(OBJECT[, HANDLER[, INTERFACE]])

This class method enables and disables the firing of events by the specified OBJECT. If no HANDLER is specified, then events are disconnected. For some objects Win32::OLE is not able to automatically determine the correct event interface. In this case the INTERFACE argument must contain either the COCLASS name of the OBJECT or the name of the event DISPATCH interface. Please read the *Events* section below for detailed explanation of the Win32::OLE event support.

Whenever Perl does not find a method name in the Win32::OLE package it is automatically used as the name of an OLE method and this method call is dispatched to the OLE server.

There is one special hack built into the module: If a method or property name could not be resolved with the OLE object, then the default method of the object is called with the method name as its first parameter. So

```
my $Sheet = $Worksheets->Table1;
```

or

```
my $Sheet = $Worksheets-{Table1};
```

is resolved as

```
my $Sheet = $Worksheet->Item('Table1');
```

provided that the `$Worksheets` object doesnot have a `Table1` method or property. This hack has been introduced to call the default method of collections which did not name the method in their type library. The recommended way to call the "unnamed" default method is:

```
my $Sheet = $Worksheets->Invoke('', 'Table1');
```

This special hack is disabled under `use strict 'subs';`.

Object methods and properties

The object returned by the `new()` method can be used to invoke methods or retrieve properties in the same fashion as described in the documentation for the particular OLE class (eg. Microsoft Excel documentation describes the object hierarchy along with the properties and methods exposed for OLE access).

Optional parameters on method calls can be omitted by using `undef` as a placeholder. A better way is to use named arguments, as the order of optional parameters may change in later versions of the OLE server application. Named parameters can be specified in a reference to a hash as the last parameter to a method call.

Properties can be retrieved or set using hash syntax, while methods can be invoked with the usual perl method call syntax. The `keys` and `each` functions can be used to enumerate an object's properties. Beware that a property is not always writable or even readable (sometimes raising exceptions when read while being undefined).

If a method or property returns an embedded OLE object, method and property access can be chained as shown in the examples below.

Functions

The following functions are not exported by default.

HRESULT(ERROR)

The `HRESULT()` function converts an unsigned number into a signed HRESULT error value as used by OLE internally. This is necessary because Perl treats all hexadecimal constants as unsigned. To check if the last OLE function returned "Member not found" (0x80020003) you can write:

```
if (Win32::OLE->LastError == HRESULT(0x80020003)) {
    # your error recovery here
}
```

in(COLLECTION)

If COLLECTION is an OLE collection object then `in $COLLECTION` returns a list of all members of the collection. This is a shortcut for `Win32::OLE::Enum`−All(`$COLLECTION)`. It is most commonly used in a `foreach` loop:

```
foreach my $value (in $collection) {
    # do something with $value here
}
```

valof(OBJECT)

Normal assignment of Perl OLE objects creates just another reference to the OLE object. The `valof()` function explictly dereferences the object (through the default method) and returns the value of the object.

```
my $RefOf = $Object;
my $ValOf = valof $Object;
$Object->{Value} = $NewValue;
```

Now `$ValOf` still contains the old value wheras `$RefOf` would resolve to the `$NewValue` because it is still a reference to `$Object`.

The `valof()` function can also be used to convert Win32::OLE::Variant objects to Perl values.

with(OBJECT, PROPERTYNAME = VALUE, ...)

This function provides a concise way to set the values of multiple properties of an object. It iterates over its arguments doing `$OBJECT−{PROPERTYNAME} = $VALUE` on each trailing pair.

Overloading

The Win32::OLE objects can be overloaded to automatically convert to their values whenever they are used in a bool, numeric or string context. This is not enabled by default. You have to request it through the OVERLOAD pseudoexport:

```
use Win32::OLE qw(in valof with OVERLOAD);
```

You can still get the original string representation of an object (`Win32::OLE=0xDEADBEEF`), e.g. for debugging, by using the `overload::StrVal()` method:

```
print overload::StrVal($object), "\n";
```

Please note that `OVERLOAD` is a global setting. If any module enables Win32::OLE overloading then it‘s active everywhere.

Events

The Win32::OLE module now contains **ALPHA** level event support. This support is only available when Perl is running in a single threaded apartment. This can most easily be assured by using the `EVENTS` pseudo−import:

```
use Win32::OLE qw(EVENTS);
```

which implicitly does something like:

```
use Win32::OLE;
Win32::OLE->Initialize(Win32::OLE::COINIT_OLEINITIALIZE);
```

The current interface to OLE events should be considered experimental and is subject to change. It works as expected for normal OLE applications, but OLE control events often don‘t seem to work yet.

Events must be enabled explicitly for an OLE object through the `Win32::OLE`−WithEvents() class method. The Win32::OLE module uses the IProvideClassInfo2 interface to determine the default event source of the object. If this interface is not supported, then the user must specify the name of the event source explicitly in the `WithEvents()` method call. It is also possible to specify the class name of the object as the third parameter. In this case Win32::OLE will try to look up the default source interface for this COCLASS.

The HANDLER argument to `Win32::OLE`−WithEvents() can either be a CODE reference or a package name. In the first case, all events will invoke this particular function. The first two arguments to this function will be the OBJECT itself and the name of the event. The remaining arguments will be event specific.

```
sub Event {
    my ($Obj,$Event,@Args) = @_;
    print "Event triggered: '$Event'\n";
}
Win32::OLE->WithEvents($Obj, \&Event);
```

Alternatively the HANDLER argument can specify a package name. When the OBJECT fires an event, Win32::OLE will try to find a function of the same name as the event in this package. This function will be called with the OBJECT as the first argument followed again by the event specific parameters:

```
package MyEvents;
sub EventName1 {
    my ($Obj,@Args) = @_;
    print "EventName1 event triggered\n";
}

package main;
Win32::OLE->WithEvents($Obj, 'MyEvents', 'IEventInterface');
```

If Win32::OLE doesn‘t find a function with the name of the event then nothing happens.

Event parameters passed *by reference* are handled specially. They are not converted to the corresponding Perl datatype but passed as Win32::OLE::Variant objects. You can assign a new value to these objects with the help of the `Put()` method. This value will be passed back to the object when the event function returns:

```
package MyEvents;
sub BeforeClose {
    my ($self,$Cancel) = @_;
    $Cancel->Put(1) unless $MayClose;
}
```

Direct assignment to `$Cancel` would have no effect on the original value and would therefore not command the object to abort the closing action.

Module Options

The following module options can be accessed and modified with the `Win32::OLE`−Option class method. In earlier versions of the Win32::OLE module these options were manipulated directly as class variables. This practice is now deprecated.

CP
: This variable is used to determine the codepage used by all translations between Perl strings and Unicode strings used by the OLE interface. The default value is CP_ACP, which is the default ANSI codepage. Other possible values are CP_OEMCP, CP_MACCP, CP_UTF7 and CP_UTF8. These constants are not exported by default.

LCID
: This variable controls the locale idnetifier used for all OLE calls. It is set to LOCALE_NEUTRAL by default. Please check the *Win32::OLE::NLS* module for other locale related information.

Warn
: This variable determines the behavior of the Win32::OLE module when an error happens. Valid values are:

```
        Ignore error, return undef
1       Carp::carp if $^W is set (-w option)
2       always Carp::carp
3       Carp::croak
```

The error number and message (without Carp line/module info) are available through the `Win32::OLE`−LastError class method.

Alternatively the Warn option can be set to a CODE reference. E.g.

```
Win32::OLE->Option(Warn => 3);
```

is equivalent to

```
Win32::OLE->Option(Warn => \&Carp::croak);
```

This can even be used to emulate the VisualBasic `On Error Goto Label` construct:

```
    Win32::OLE->Option(Warn =>  sub {goto CheckError});
    # ... your normal OLE code here ...

  CheckError:
    # ... your error handling code here ...
```

_NewEnum
: This option enables additional enumeration support for collection objects. When the `_NewEnum` option is set, all collections will receive one additional property: `_NewEnum`. The value of this property will be a reference to an array containing all the elements of the collection. This option can be useful when used in conjunction with an automatic tree traversal program, like `Data::Dumper` or an object tree browser. The value of this option should be either 1 (enabled) or 0 (disabled, default).

```
Win32::OLE->Option(_NewEnum => 1);
# ...
my @sheets = @{$Excel->Worksheets->{_NewEnum}};
```

In normal application code, this would be better written as:

```
use Win32::OLE qw(in);
# ...
my @sheets = in $Excel->Worksheets;
```

_Unique The `_Unique` options guarantees that Win32::OLE will maintain a one−to−one mapping between Win32::OLE objects and the native COM/OLE objects. Without this option, you can query the same property twice and get two different Win32::OLE objects for the same underlying COM object.

Using a unique proxy makes life easier for tree traversal algorithms to recognize they already visited a particular node. This option comes at a price: Win32::OLE has to maintain a global hash of all outstanding objects and their corresponding proxies. Identity checks on COM objects can also be expensive if the objects reside out−of−process or even on a different computer. Therefore this option is off by default unless the program is being run in the debugger.

Unfortunately, this option doesn‘t always help. Some programs will return new COM objects for even the same property when asked for it multiple times (especially for collections). In this case, there is nothing Win32::OLE can do to detect that these objects are in fact identical (because they aren‘t at the COM level).

The `_Unique` option can be set to either 1 (enabled) or 0 (disabled, default).

EXAMPLES

Here is a simple Microsoft Excel application.

```
use Win32::OLE;

# use existing instance if Excel is already running
eval {$ex = Win32::OLE->GetActiveObject('Excel.Application')};
die "Excel not installed" if $@;
unless (defined $ex) {
    $ex = Win32::OLE->new('Excel.Application', sub {$_[0]->Quit;})
            or die "Oops, cannot start Excel";
}

# get a new workbook
$book = $ex->Workbooks->Add;

# write to a particular cell
$sheet = $book->Worksheets(1);
$sheet->Cells(1,1)->{Value} = "foo";

# write a 2 rows by 3 columns range
$sheet->Range("A8:C9")->{Value} = [[ undef, 'Xyzzy', 'Plugh' ],
                                   [ 42,    'Perl',  3.1415  ]];

# print "XyzzyPerl"
$array = $sheet->Range("A8:C9")->{Value};
for (@$array) {
    for (@$_) {
        print defined($_) ? "$_|" : "<undef>|";
    }
    print "\n";
}

# save and exit
$book->SaveAs( 'test.xls' );
undef $book;
undef $ex;
```

Please note the destructor specified on the Win32::OLE−new method. It ensures that Excel will shutdown properly even if the Perl program dies. Otherwise there could be a process leak if your application dies after having opened an OLE instance of Excel. It is the responsibility of the module user to make sure that all OLE objects are cleaned up properly!

Here is an example of using Variant data types.

```
use Win32::OLE;
use Win32::OLE::Variant;
$ex = Win32::OLE->new('Excel.Application', \&OleQuit) or die "oops\n";
$ex->{Visible} = 1;
$ex->Workbooks->Add;
# should generate a warning under -w
$ovR8 = Variant(VT_R8, "3 is a good number");
$ex->Range("A1")->{Value} = $ovR8;
$ex->Range("A2")->{Value} = Variant(VT_DATE, 'Jan 1,1970');

sub OleQuit {
    my $self = shift;
    $self->Quit;
}
```

The above will put value "3" in cell A1 rather than the string "3 is a good number". Cell A2 will contain the date.

Similarly, to invoke a method with some binary data, you can do the following:

```
$obj->Method( Variant(VT_UI1, "foo\000b\001a\002r") );
```

Here is a wrapper class that basically delegates everything but `new()` and `DESTROY()`. The wrapper class shown here is another way to properly shut down connections if your application is liable to die without proper cleanup. Your own wrappers will probably do something more specific to the particular OLE object you may be dealing with, like overriding the methods that you may wish to enhance with your own.

```
package Excel;
use Win32::OLE;

sub new {
    my $s = {};
    if ($s->{Ex} = Win32::OLE->new('Excel.Application')) {
        return bless $s, shift;
    }
    return undef;
}

sub DESTROY {
    my $s = shift;
    if (exists $s->{Ex}) {
        print "# closing connection\n";
        $s->{Ex}->Quit;
        return undef;
    }
}

sub AUTOLOAD {
    my $s = shift;
    $AUTOLOAD =~ s/^.*:://;
    $s->{Ex}->$AUTOLOAD(@_);
}
```

```
1;
```

The above module can be used just like Win32::OLE, except that it takes care of closing connections in case of abnormal exits. Note that the effect of this specific example can be easier accomplished using the optional destructor argument of Win32::OLE::new:

```
my $Excel = Win32::OLE->new('Excel.Application', sub {$_[0]->Quit;});
```

Note that the delegation shown in the earlier example is not the same as true subclassing with respect to further inheritance of method calls in your specialized object. See *perlobj*, *perltoot* and *perlbot* for details. True subclassing (available by setting `@ISA`) is also feasible, as the following example demonstrates:

```
#
# Add error reporting to Win32::OLE
#

package Win32::OLE::Strict;
use Carp;
use Win32::OLE;

use strict qw(vars);
use vars qw($AUTOLOAD @ISA);
@ISA = qw(Win32::OLE);

sub AUTOLOAD {
    my $obj = shift;
    $AUTOLOAD =~ s/^.*:://;
    my $meth = $AUTOLOAD;
    $AUTOLOAD = "SUPER::" . $AUTOLOAD;
    my $retval = $obj->$AUTOLOAD(@_);
    unless (defined($retval) || $AUTOLOAD eq 'DESTROY') {
        my $err = Win32::OLE::LastError();
        croak(sprintf("$meth returned OLE error 0x%08x",$err))
          if $err;
    }
    return $retval;
}

1;
```

This package inherits the constructor `new()` from the Win32::OLE package. It is important to note that you cannot later rebless a Win32::OLE object as some information about the package is cached by the object. Always invoke the `new()` constructor through the right package!

Here's how the above class will be used:

```
use Win32::OLE::Strict;
my $Excel = Win32::OLE::Strict->new('Excel.Application', 'Quit');
my $Books = $Excel->Workbooks;
$Books->UnknownMethod(42);
```

In the sample above the call to `UnknownMethod()` will be caught with

```
UnknownMethod returned OLE error 0x80020009 at test.pl line 5
```

because the Workbooks object inherits the class `Win32::OLE::Strict` from the `$Excel` object.

NOTES

Hints for Microsoft Office automation

Documentation

The object model for the Office applications is defined in the Visual Basic reference guides for the various applications. These are typically not installed by default during the standard installation. They can be added later by rerunning the setup program with the custom install option.

Class, Method and Property names

The names have been changed between different versions of Office. For example `Application` was a method in Office 95 and is a property in Office97. Therefore it will not show up in the list of property names `keys %$object` when querying an Office 95 object.

The class names are not always identical to the method/property names producing the object. E.g. the `Workbook` method returns an object of type `Workbook` in Office 95 and `_Workbook` in Office 97.

Moniker (GetObject support)

Office applications seem to implement file monikers only. For example it seems to be impossible to retrieve a specific worksheet object through `GetObject("File.XLS!Sheet")`. Furthermore, in Excel 95 the moniker starts a Worksheet object and in Excel 97 it returns a Workbook object. You can use either the Win32::OLE::QueryObjectType class method or the `$object-`{Version} property to write portable code.

Enumeration of collection objects

Enumerations seem to be incompletely implemented. Office 95 application don‘t seem to support neither the `Reset()` nor the `Clone()` methods. The `Clone()` method is still unimplemented in Office 97. A single walk through the collection similar to Visual Basics `for each` construct does work however.

Localization

Starting with Office 97 Microsoft has changed the localized class, method and property names back into English. Note that string, date and currency arguments are still subject to locale specific interpretation. Perl uses the system default locale for all OLE transaction whereas Visual Basic uses a type library specific locale. A Visual Basic script would use "R1C1" in string arguments to specify relative references. A Perl script running on a German language Windows would have to use "Z1S1". Set the LCID module option to an English locale to write portable scripts. This variable should not be changed after creating the OLE objects; some methods seem to randomly fail if the locale is changed on the fly.

SaveAs method in Word 97 doesn‘t work

This is an known bug in Word 97. Search the MS knowledge base for Word / Foxpro incompatibility. That problem applies to the Perl OLE interface as well. A workaround is to use the WordBasic compatibility object. It doesn‘t support all the options of the native method though.

```
    $Word->WordBasic->FileSaveAs($file);
```

The problem seems to be fixed by applying the Office 97 Service Release 1.

Randomly failing method calls

It seems like modifying objects that are not selected/activated is sometimes fragile. Most of these problems go away if the chart/sheet/document is selected or activated before being manipulated (just like an interactive user would automatically do it).

Incompatibilities

There are some incompatibilities with the version distributed by Activeware (as of build 306).

1 The package name has changed from "OLE" to "Win32::OLE".

2 All functions of the form "Win32::OLEFoo" are now "Win32::OLE::Foo", though the old names are temporarily accomodated. `Win32::OLECreateObject()` was changed to `Win32::OLE::CreateObject()`, and is now called `Win32::OLE::new()` bowing to established convention for naming constructors. The old names should be considered deprecated, and will be removed in the next version.

3 Package "OLE::Variant" is now "Win32::OLE::Variant".

4 The Variant function is new, and is exported by default. So are all the VT_XXX type constants.

5 The support for collection objects has been moved into the package Win32::OLE::Enum. The `keys %$object` method is now used to enumerate the properties of the object.

Bugs and Limitations

- To invoke a native OLE method with the same name as one of the Win32::OLE methods (`Dispatch`, `Invoke`, `SetProperty`, `DESTROY`, etc.), you have to use the `Invoke` method:

```
        $Object->Invoke('Dispatch', @AdditionalArgs);
```

  The same is true for names exported by the Exporter or the Dynaloader modules, e.g.: `export`, `export_to_level`, `import`, `_push_tags`, `export_tags`, `export_ok_tags`, `export_fail`, `require_version`, `dl_load_flags`, `croak`, `bootstrap`, `dl_findfile`, `dl_expandspec`, `dl_find_symbol_anywhere`, `dl_load_file`, `dl_find_symbol`, `dl_undef_symbols`, `dl_install_xsub` and `dl_error`.

SEE ALSO

The documentation for *Win32::OLE::Const*, *Win32::OLE::Enum*, *Win32::OLE::NLS* and *Win32::OLE::Variant* contains additional information about OLE support for Perl on Win32.

AUTHORS

Originally put together by the kind people at Hip and Activeware.

Gurusamy Sarathy <gsar@activestate.com subsequently fixed several major bugs, memory leaks, and reliability problems, along with some redesign of the code.

Jan Dubois <jand@activestate.com pitched in with yet more massive redesign, added support for named parameters, and other significant enhancements. He‘s been hacking on it ever since.

Please send questions about problems with this module to the Perl−Win32−Users mailinglist at ActiveState.com. The mailinglist charter requests that you put an [OLE] tag somewhere on the subject line (for OLE related questions only, of course).

COPYRIGHT

VERSION

Version 0.1403 21 November 2000

NAME

Win32::PerfLib − accessing the Windows NT Performance Counter

SYNOPSIS

```
use Win32::PerfLib;
my $server = "";
Win32::PerfLib::GetCounterNames($server, \%counter);
%r_counter = map { $counter{$_} => $_ } keys %counter;
# retrieve the id for process object
$process_obj = $r_counter{Process};
# retrieve the id for the process ID counter
$process_id = $r_counter{'ID Process'};

# create connection to $server
$perflib = new Win32::PerfLib($server);
$proc_ref = {};
# get the performance data for the process object
$perflib->GetObjectList($process_obj, $proc_ref);
$perflib->Close();
$instance_ref = $proc_ref->{Objects}->{$process_obj}->{Instances};
foreach $p (sort keys %{$instance_ref})
{
    $counter_ref = $instance_ref->{$p}->{Counters};
    foreach $i (keys %{$counter_ref})
    {
        if($counter_ref->{$i}->{CounterNameTitleIndex} == $process_id)
        {
            printf( "% 6d %s\n", $counter_ref->{$i}->{Counter},
                    $instance_ref->{$p}->{Name}
                  );
        }
    }
}
```

DESCRIPTION

This module allows to retrieve the performance counter of any computer (running Windows NT) in the network.

FUNCTIONS

NOTE

All of the functions return FALSE (0) if they fail, unless otherwise noted. If the `$server` argument is undef the local machine is assumed.

Win32::PerfLib::GetCounterNames(`$server,$hashref`)

Retrieves the counter names and their indices from the registry and stores them in the hash reference

Win32::PerfLib::GetCounterHelp(`$server,$hashref`)

Retrieves the counter help strings and their indices from the registry and stores them in the hash reference

`$perflib` = Win32::PerfLib−new (`$server`)

Creates a connection to the performance counters of the given server

`$perflib−`GetObjectList($objectid,$hashref)

retrieves the object and counter list of the given performance object.

`$perflib−`Close($hashref)

closes the connection to the performance counters

Win32::PerfLib::GetCounterType(countertype)

converts the counter type to readable string as referenced in *calc.html* so that it is easier to find the appropriate formula to calculate the raw counter data.

Datastructures

The performance data is returned in the following data structure:

Level 1

```
$hashref = {
    'NumObjectTypes'    => VALUE
    'Objects'           => HASHREF
    'PerfFreq'          => VALUE
    'PerfTime'          => VALUE
    'PerfTime100nSec'   => VALUE
    'SystemName'        => STRING
    'SystemTime'        => VALUE
}
```

Level 2 The hash reference `$hashref−`{Objects} has the returned object ID(s) as keys and a hash reference to the object counter data as value. Even there is only one object requested in the call to GetObjectList there may be more than one object in the result.

```
$hashref->{Objects} = {
    <object1>  => HASHREF
    <object2>  => HASHREF
    ...
}
```

Level 3 Each returned object ID has object−specific performance information. If an object has instances like the process object there is also a reference to the instance information.

```
$hashref->{Objects}->{<object1>} = {
    'DetailLevel'            => VALUE
    'Instances'              => HASHREF
    'Counters'               => HASHREF
    'NumCounters'            => VALUE
    'NumInstances'           => VALUE
    'ObjectHelpTitleIndex'   => VALUE
    'ObjectNameTitleIndex'   => VALUE
    'PerfFreq'               => VALUE
    'PerfTime'               => VALUE
}
```

Level 4 If there are instance information for the object available they are stored in the ‘Instances’ hashref. If the object has no instances there is an ‘Counters’ key instead. The instances or counters are numbered.

```
$hashref->{Objects}->{<object1>}->{Instances} = {
    <1>      => HASHREF
    <2>      => HASHREF
    ...
    <n>      => HASHREF
}
or
$hashref->{Objects}->{<object1>}->{Counters} = {
    <1>     => HASHREF
    <2>     => HASHREF
    ...
    <n>     => HASHREF
}
```

Level 5

```
$hashref->{Objects}->{<object1>}->{Instances}->{<1>} = {
    Counters                => HASHREF
    Name                    => STRING
    ParentObjectInstance    => VALUE
    ParentObjectTitleIndex => VALUE
}
or
$hashref->{Objects}->{<object1>}->{Counters}->{<1>} = {
    Counter                => VALUE
    CounterHelpTitleIndex => VALUE
    CounterNameTitleIndex => VALUE
    CounterSize            => VALUE
    CounterType            => VALUE
    DefaultScale           => VALUE
    DetailLevel            => VALUE
    Display                => STRING
}
```

Level 6

```
$hashref->{Objects}->{<object1>}->{Instances}->{<1>}->{Counters} = {
    <1>     => HASHREF
    <2>     => HASHREF
    ...
    <n>     => HASHREF
}
```

Level 7

```
$hashref->{Objects}->{<object1>}->{Instances}->{<1>}->{Counters}->{<1>} =
    Counter                => VALUE
    CounterHelpTitleIndex => VALUE
    CounterNameTitleIndex => VALUE
    CounterSize            => VALUE
    CounterType            => VALUE
    DefaultScale           => VALUE
    DetailLevel            => VALUE
    Display                => STRING
}
```

Depending on the **CounterType** there are calculations to do (see calc.html).

AUTHOR

Jutta M. Klebe, jmk@bybyte.de

SEE ALSO

perl(1).

NAME

Win32::Pipe − Win32 Named Pipe

SYNOPSIS

To use this extension, follow these basic steps. First, you need to ‘use’ the pipe extension:

```
use Win32::Pipe;
```

Then you need to create a server side of a named pipe:

```
$Pipe = new Win32::Pipe("My Pipe Name");
```

or if you are going to connect to pipe that has already been created:

```
$Pipe = new Win32::Pipe("\\\\server\\pipe\\My Pipe Name");

NOTE: The "\\\\server\\pipe\\" is necessary when connecting
      to an existing pipe! If you are accessing the same
      machine you could use "\\\\.\\pipe\\" but either way
      works fine.
```

You should check to see if `$Pipe` is indeed defined otherwise there has been an error.

Whichever end is the server, it must now wait for a connection...

```
$Result = $Pipe->Connect();

NOTE: The client end does not do this! When the client creates
      the pipe it has already connected!
```

Now you can read and write data from either end of the pipe:

```
$Data = $Pipe->Read();

$Result = $Pipe->Write("Howdy! This is cool!");
```

When the server is finished it must disconnect:

```
$Pipe->Disconnect();
```

Now the server could `Connect` again (and wait for another client) or it could destroy the named pipe...

```
$Data->Close();
```

The client should `Close` in order to properly end the session.

DESCRIPTION

General Use

This extension gives Win32 Perl the ability to use Named Pipes. Why? Well considering that Win32 Perl does not (yet) have the ability to `fork` I could not see what good the `pipe(X,Y)` was. Besides, where I am as an admin I must have several perl daemons running on several NT Servers. It dawned on me one day that if I could pipe all these daemons’ output to my workstation (across the net) then it would be much easier to monitor. This was the impetus for an extension using Named Pipes. I think that it is kinda cool. :)

Benefits

And what are the benefits of this module?

- You may create as many named pipes as you want (uh, well, as many as your resources will allow).
- Currently there is a limit of 256 instances of a named pipe (once a pipe is created you can have 256 client/server connections to that name).
- The default buffer size is 512 bytes; this can be altered by the `ResizeBuffer` method.

- All named pipes are byte streams. There is currently no way to alter a pipe to be message based.
- Other things that I cannot think of right now... :)

CONSTRUCTOR

new (NAME)

Creates a named pipe if used in server context or a connection to the specified named pipe if used in client context. Client context is determined by prepending `$Name` with "\\\\".

Returns *true* on success, *false* on failure.

METHODS

BufferSize `()`

Returns the size of the instance of the buffer of the named pipe.

Connect `()`

Tells the named pipe to create an instance of the named pipe and wait until a client connects. Returns *true* on success, *false* on failure.

Close `()`

Closes the named pipe.

Disconnect `()`

Disconnects (and destroys) the instance of the named pipe from the client. Returns *true* on success, *false* on failure.

Error `()`

Returns the last error messages pertaining to the named pipe. If used in context to the package. Returns a list containing `ERROR_NUMBER` and `ERROR_TEXT`.

Read `()`

Reads from the named pipe. Returns data read from the pipe on success, undef on failure.

ResizeBuffer (SIZE)

Sets the size of the buffer of the instance of the named pipe to `SIZE`. Returns the size of the buffer on success, *false* on failure.

Write (DATA)

Writes `DATA` to the named pipe. Returns *true* on success, *false* on failure.

LIMITATIONS

What known problems does this thing have?

- If someone is waiting on a `Read` and the other end terminates then you will wait for one **REALLY** long time! (If anyone has an idea on how I can detect the termination of the other end let me know!)
- All pipes are blocking. I am considering using threads and callbacks into Perl to perform async IO but this may be too much for my time stress. ;)
- There is no security placed on these pipes.
- This module has neither been optimized for speed nor optimized for memory consumption. This may run into memory bloat.

INSTALLATION NOTES

If you wish to use this module with a build of Perl other than ActivePerl, you may wish to fetch the source distribution for this module. The source is included as part of the `libwin32` bundle, which you can find in any CPAN mirror here:

```
modules/by-authors/Gurusamy_Sarathy/libwin32-0.151.tar.gz
```

The source distribution also contains a pair of sample client/server test scripts. For the latest information on this module, consult the following web site:

```
http://www.roth.net/perl
```

AUTHOR

Dave Roth <rothd@roth.net

DISCLAIMER

I do not guarantee **ANYTHING** with this package. If you use it you are doing so **AT YOUR OWN RISK**! I may or may not support this depending on my time schedule.

COPYRIGHT

NAME

Win32::Process − Create and manipulate processes.

SYNOPSIS

```
use Win32::Process;
use Win32;

sub ErrorReport{
        print Win32::FormatMessage( Win32::GetLastError() );
}

Win32::Process::Create($ProcessObj,
                        "D:\\winnt35\\system32\\notepad.exe",
                        "notepad temp.txt",
                        0,
                        NORMAL_PRIORITY_CLASS,
                        ".")|| die ErrorReport();

$ProcessObj->Suspend();
$ProcessObj->Resume();
$ProcessObj->Wait(INFINITE);
```

DESCRIPTION

This module allows for control of processes in Perl.

METHODS

Win32::Process::Create(`$obj,$appname,$cmdline,$iflags,$cflags,$curdir)`

Creates a new process.

```
    Args:

        $obj            container for process object
        $appname        full path name of executable module
        $cmdline        command line args
        $iflags         flag: inherit calling processes handles or not
        $cflags         flags for creation (see exported vars below)
        $curdir         working dir of new process
```

Win32::Process::KillProcess(`$pid, $exitcode)`

Terminates any process identified by `$pid.` The process will exit with `$exitcode.`

`$ProcessObj–`Suspend()

Suspend the process associated with the `$ProcessObj`.

`$ProcessObj–`Resume()

Resume a suspended process.

`$ProcessObj–`Kill(`$ExitCode`)

Kill the associated process, have it die with exit code `$ExitCode.`

`$ProcessObj–`GetPriorityClass($class)

Get the priority class of the process.

`$ProcessObj–`SetPriorityClass(`$class`)

Set the priority class of the process (see exported values below for options).

`$ProcessObj–`GetProcessAffinitymask(`$processAffinityMask, $systemAffinitymask)`

Get the process affinity mask. This is a bitvector in which each bit represents the processors that a process is allowed to run on.

$ProcessObj−SetProcessAffinitymask($processAffinityMask)
Set the process affinity mask. Only available on Windows NT.

$ProcessObj−GetExitCode($ExitCode)
Retrieve the exitcode of the process.

$ProcessObj−Wait($Timeout)
Wait for the process to die. forever = INFINITE

$ProcessObj−GetProcessID()
Returns the Process ID.

NAME

Win32::Registry − accessing the Windows registry [obsolete, use Win32::TieRegistry]

SYNOPSIS

```
use Win32::Registry;
my $tips;
$::HKEY_LOCAL_MACHINE->Open("SOFTWARE\\Microsoft\\Windows"
                           ."\\CurrentVersion\\Explorer\\Tips", $tips)
    or die "Can’t open tips: $^E";
my ($type, $value);
$tips->QueryValueEx("18", $type, $value) or die "No tip #18: $^E";
print "Here’s a tip: $value\n";
```

DESCRIPTION

```
NOTE: This module provides a very klunky interface to access the
Windows registry, and is not currently being developed actively.  It
only exists for backward compatibility with old code that uses it.
For more powerful and flexible ways to access the registry, use
Win32::TieRegistry.
```

Win32::Registry provides an object oriented interface to the Windows Registry.

The following "root" registry objects are exported to the main:: name space. Additional keys must be opened by calling the provided methods on one of these.

```
$HKEY_CLASSES_ROOT
$HKEY_CURRENT_USER
$HKEY_LOCAL_MACHINE
$HKEY_USERS
$HKEY_PERFORMANCE_DATA
$HKEY_CURRENT_CONFIG
$HKEY_DYN_DATA
```

Methods

The following methods are supported. Note that subkeys can be specified as a path name, separated by backslashes (which may need to be doubled if you put them in double quotes).

Open

```
$reg_obj->Open($sub_key_name, $sub_reg_obj);
```

Opens a subkey of a registry object, returning the new registry object in `$sub_reg_obj`.

Close

```
$reg_obj->Close();
```

Closes an open registry key.

Connect

```
$reg_obj->Connect($node_name, $new_reg_obj);
```

Connects to a remote Registry on the node specified by `$node_name`, returning it in `$new_reg_obj`. Returns false if it fails.

Create

```
$reg_obj->Create($sub_key_name, $new_reg_obj);
```

Opens the subkey specified by `$sub_key_name`, returning the new registry object in `$new_reg_obj`. If the specified subkey doesn‘t exist, it is created.

SetValue

```
    $reg_obj->SetValue($sub_key_name, $type, $value);
```

Sets the default value for a subkey specified by `$sub_key_name`.

SetValueEx

```
    $reg_obj->SetValueEx($value_name, $reserved, $type, $value);
```

Sets the value for the value name identified by `$value_name` in the key specified by `$reg_obj`.

QueryValue

```
    $reg_obj->QueryValue($sub_key_name, $value);
```

Gets the default value of the subkey specified by `$sub_key_name`.

QueryKey

```
    $reg_obj->QueryKey($classref, $number_of_subkeys, $number_of_values);
```

Gets information on a key specified by `$reg_obj`.

QueryValueEx

```
    $reg_obj->QueryValueEx($value_name, $type, $value);
```

Gets the value for the value name identified by `$value_name` in the key specified by `$reg_obj`.

GetKeys

```
    my @keys;
    $reg_obj->GetKeys(\@keys);
```

Populates the supplied array reference with the names of all the keys within the registry object `$reg_obj`.

GetValues

```
    my %values;
    $reg_obj->GetValues(\%values);
```

Populates the supplied hash reference with entries of the form

```
    $value_name => [ $value_name, $type, $data ]
```

for each value in the registry object `$reg_obj`.

DeleteKey

```
    $reg_obj->DeleteKey($sub_key_name);
```

Deletes a subkey specified by `$sub_key_name` from the registry.

DeleteValue

```
    $reg_obj->DeleteValue($value_name);
```

Deletes a value identified by `$value_name` from the registry.

Save

```
    $reg_obj->Save($filename);
```

Saves the hive specified by `$reg_obj` to a file.

Load

```
    $reg_obj->Load($sub_key_name, $file_name);
```

Loads a key specified by `$sub_key_name` from a file.

UnLoad

```
$reg_obj->Unload($sub_key_name);
```

Unloads a registry hive.

NAME

Win32::Semaphore − Use Win32 semaphore objects from Perl

SYNOPSIS

```
        require Win32::Semaphore;

        $sem = Win32::Semaphore->new($initial,$maximum,$name);
        $sem->wait;
```

DESCRIPTION

This module allows access to Win32 semaphore objects. The `wait` method and `wait_all` & `wait_any` functions are inherited from the *"Win32::IPC"* module.

Methods

`$semaphore` = Win32::Semaphore−new(`$initial, $maximum, [$name]`)

Constructor for a new semaphore object. `$initial` is the initial count, and `$maximum` is the maximum count for the semaphore. If `$name` is omitted, creates an unnamed semaphore object.

If `$name` signifies an existing semaphore object, then `$initial` and `$maximum` are ignored and the object is opened.

`$semaphore` = Win32::Semaphore−open(`$name`)

Constructor for opening an existing semaphore object.

`$semaphore−`release([$increment, `[$previous]]`)

Increment the count of `$semaphore` by `$increment` (default 1). If `$increment` plus the semaphore‘s current count is more than its maximum count, the count is not changed. Returns true if the increment is successful.

The semaphore‘s count (before incrementing) is stored in the second argument (if any).

It is not necessary to wait on a semaphore before calling `release`, but you‘d better know what you‘re doing.

`$semaphore−`wait([$timeout])

Wait for `$semaphore‘s` count to be nonzero, then decrement it by 1. See *"Win32::IPC"*.

Deprecated Functions and Methods

Win32::Semaphore still supports the ActiveWare syntax, but its use is deprecated.

Win32::Semaphore::Create(`$SemObject,$Initial,$Max,$Name`)

Use `$SemObject = Win32::Semaphore->new($Initial,$Max,$Name)` instead.

Win32::Semaphore::Open(`$SemObject, $Name`)

Use `$SemObject = Win32::Semaphore->open($Name)` instead.

`$SemObj−`Release($Count,$LastVal)

Use `$SemObj->release($Count,$LastVal)` instead.

AUTHOR

Christopher J. Madsen <***chris_madsen@geocities.com***>

Loosely based on the original module by ActiveWare Internet Corp., ***http://www.ActiveWare.com***

NAME

Win32::Service − manage system services in perl

SYNOPSIS

```
        use Win32::Service;
```

DESCRIPTION

This module offers control over the administration of system services.

FUNCTIONS

NOTE:

All of the functions return FALSE (0) if they fail, unless otherwise noted. If hostName is an empty string, the local machine is assumed.

StartService(hostName, serviceName)

Start the service serviceName on machine hostName.

StopService(hostName, serviceName)

Stop the service serviceName on the machine hostName.

GetStatus(hostName, serviceName, status)

Get the status of a service. The third argument must be a hash reference that will be populated with entries corresponding to the SERVICE_STATUS structure of the Win32 API. See the Win32 Platform SDK documentation for details of this structure.

PauseService(hostName, serviceName)
ResumeService(hostName, serviceName)
GetServices(hostName, hashref)

Enumerates both active and inactive Win32 services at the specified host. The hashref is populated with the descriptive service names as keys and the short names as the values.

NAME

Win32::Sound − An extension to play with Windows sounds

SYNOPSIS

```
use Win32::Sound;
Win32::Sound::Volume('100%');
Win32::Sound::Play("file.wav");
Win32::Sound::Stop();

# ...and read on for more fun ;-)
```

FUNCTIONS

Win32::Sound::Play(SOUND, [FLAGS])

Plays the specified sound: SOUND can the be name of a WAV file or one of the following predefined sound names:

```
SystemDefault
SystemAsterisk
SystemExclamation
SystemExit
SystemHand
SystemQuestion
SystemStart
```

Additionally, if the named sound could not be found, the function plays the system default sound (unless you specify the `SND_NODEFAULT` flag). If no parameters are given, this function stops the sound actually playing (see also Win32::Sound::Stop).

FLAGS can be a combination of the following constants:

`SND_ASYNC`

The sound is played asynchronously and the function returns immediately after beginning the sound (if this flag is not specified, the sound is played synchronously and the function returns when the sound ends).

`SND_LOOP`

The sound plays repeatedly until it is stopped. You must also specify `SND_ASYNC` flag.

`SND_NODEFAULT`

No default sound is used. If the specified *sound* cannot be found, the function returns without playing anything.

`SND_NOSTOP`

If a sound is already playing, the function fails. By default, any new call to the function will stop previously playing sounds.

`Win32::Sound::Stop()`

Stops the sound currently playing.

`Win32::Sound::Volume()`

Returns the wave device volume; if called in an array context, returns left and right values. Otherwise, returns a single 32 bit value (left in the low word, right in the high word). In case of error, returns `undef` and sets `$!`.

Examples:

```
($L, $R) = Win32::Sound::Volume();
if( not defined Win32::Sound::Volume() ) {
    die "Can't get volume: $!";
```

```
        }
```

Win32::Sound::Volume(LEFT, [RIGHT])

Sets the wave device volume; if two arguments are given, sets left and right channels independently, otherwise sets them both to LEFT (eg. RIGHT=LEFT). Values range from 0 to 65535 (0xFFFF), but they can also be given as percentage (use a string containing a number followed by a percent sign).

Returns `undef` and sets `$!` in case of error, a true value if successful.

Examples:

```
    Win32::Sound::Volume('50%');
    Win32::Sound::Volume(0xFFFF, 0x7FFF);
    Win32::Sound::Volume('100%', '50%');
    Win32::Sound::Volume(0);
```

Win32::Sound::Format(filename)

Returns information about the specified WAV file format; the array contains:

- sample rate (in Hz)
- bits per sample (8 or 16)
- channels (1 for mono, 2 for stereo)

Example:

```
    ($hz, $bits, $channels)
        = Win32::Sound::Format("file.wav");
```

Win32::Sound::Devices()

Returns all the available sound devices; their names contain the type of the device (WAVEOUT, WAVEIN, MIDIOUT, MIDIIN, AUX or MIXER) and a zero−based ID number: valid devices names are for example:

```
    WAVEOUT0
    WAVEOUT1
    WAVEIN0
    MIDIOUT0
    MIDIIN0
    AUX0
    AUX1
    AUX2
```

There are also two special device names, `WAVE_MAPPER` and `MIDI_MAPPER` (the default devices for wave output and midi output).

Example:

```
    @devices = Win32::Sound::Devices();
```

Win32::Sound::DeviceInfo(DEVICE)

Returns an associative array of information about the sound device named DEVICE (the same format of Win32::Sound::Devices).

The content of the array depends on the device type queried. Each device type returns **at least** the following information:

```
    manufacturer_id
    product_id
    name
    driver_version
```

For additional data refer to the following table:

```
WAVEIN..... formats
            channels

WAVEOUT.... formats
            channels
            support

MIDIOUT.... technology
            voices
            notes
            channels
            support

AUX........ technology
            support

MIXER...... destinations
            support
```

The meaning of the fields, where not obvious, can be evinced from the Microsoft SDK documentation (too long to report here, maybe one day... :−).

Example:

```
%info = Win32::Sound::DeviceInfo('WAVE_MAPPER');
print "$info{name} version $info{driver_version}\n";
```

THE WaveOut PACKAGE

Win32::Sound also provides a different, more powerful approach to wave audio data with its `WaveOut` package. It has methods to load and then play WAV files, with the additional feature of specifying the start and end range, so you can play only a portion of an audio file.

Furthermore, it is possible to load arbitrary binary data to the soundcard to let it play and save them back into WAV files; in a few words, you can do some sound synthesis work.

FUNCTIONS

new Win32::Sound::WaveOut(FILENAME)
new Win32::Sound::WaveOut(SAMPLERATE, BITS, CHANNELS)
new `Win32::Sound::WaveOut()`

This function creates a `WaveOut` object; the first form opens the specified wave file (see also `Open()`), so you can directly `Play()` it.

The second (and third) form opens the wave output device with the format given (or if none given, defaults to 44.1kHz, 16 bits, stereo); to produce something audible you can either `Open()` a wave file or `Load()` binary data to the soundcard and then `Write()` it.

`Close()`

Closes the wave file currently opened.

`CloseDevice()`

Closes the wave output device; you can change format and reopen it with `OpenDevice()`.

GetErrorText(ERROR)

Returns the error text associated with the specified ERROR number; note it only works for wave−output−specific errors.

Load(DATA)

Loads the DATA buffer in the soundcard. The format of the data buffer depends on the format used; for example, with 8 bit mono each sample is one character, while with 16 bit stereo each sample is four characters long (two 16 bit values for left and right channels). The sample rate defines how much samples are in one second of sound. For example, to fit one second at 44.1kHz 16 bit stereo your buffer must contain 176400 bytes (44100 * 4).

Open(FILE)

Opens the specified wave FILE.

`OpenDevice()`

Opens the wave output device with the current sound format (not needed unless you used `CloseDevice()`).

`Pause()`

Pauses the sound currently playing; use `Restart()` to continue playing.

Play([FROM, TO])

Plays the opened wave file. You can optionally specify a FROM − TO range, where FROM and TO are expressed in samples (or use FROM=0 for the first sample and TO=−1 for the last sample). Playback happens always asynchronously, eg. in the background.

`Position()`

Returns the sample number currently playing; note that the play position is not zeroed when the sound ends, so you have to call a `Reset()` between plays to receive the correct position in the current sound.

`Reset()`

Stops playing and resets the play position (see `Position()`).

`Restart()`

Continues playing the sound paused by `Pause()`.

Save(FILE, [DATA])

Writes the DATA buffer (if not given, uses the buffer currently loaded in the soundcard) to the specified wave FILE.

`Status()`

Returns 0 if the soundcard is currently playing, 1 if it‘s free, or `undef` on errors.

`Unload()`

Frees the soundcard from the loaded data.

Volume([LEFT, RIGHT])

Gets or sets the volume for the wave output device. It works the same way as Win32::Sound::Volume.

`Write()`

Plays the data currently loaded in the soundcard; playback happens always asynchronously, eg. in the background.

THE SOUND FORMAT

The sound format is stored in three properties of the `WaveOut` object: `samplerate`, `bits` and `channels`. If you need to change them without creating a new object, you should close before and reopen afterwards the device.

```
$WAV->CloseDevice();
$WAV->{samplerate} = 44100; # 44.1kHz
$WAV->{bits}       = 8;     # 8 bit
$WAV->{channels}   = 1;     # mono
$WAV->OpenDevice();
```

You can also use the properties to query the sound format currently used.

EXAMPLE

This small example produces a 1 second sinusoidal wave at 440Hz and saves it in ***sinus.wav***:

```
use Win32::Sound;

# Create the object
$WAV = new Win32::Sound::WaveOut(44100, 8, 2);

$data = "";
$counter = 0;
$increment = 440/44100;

# Generate 44100 samples ( = 1 second)
for $i (1..44100) {

    # Calculate the pitch
    # (range 0..255 for 8 bits)
    $v = sin($counter/2*3.14) * 128 + 128;

    # "pack" it twice for left and right
    $data .= pack("cc", $v, $v);

    $counter += $increment;
}

$WAV->Load($data);       # get it
$WAV->Write();           # hear it
1 until $WAV->Status();  # wait for completion
$WAV->Save("sinus.wav"); # write to disk
$WAV->Unload();          # drop it
```

VERSION

Win32::Sound version 0.46, 25 Sep 1999.

AUTHOR

Aldo Calpini, `dada@divinf.it`

Parts of the code provided and/or suggested by Dave Roth.

NAME

Win32::TieRegistry − Powerful and easy ways to manipulate a registry [on Win32 for now].

SYNOPSIS

```
use Win32::TieRegistry 0.20 ( UseOptionName=>UseOptionValue[,...] );

$Registry->SomeMethodCall(arg1,...);

$subKey= $Registry->{"Key\\SubKey\\"};
$valueData= $Registry->{"Key\\SubKey\\\\ValueName"};
$Registry->{"Key\\SubKey\\"}= { "NewSubKey" => {...} };
$Registry->{"Key\\SubKey\\\\ValueName"}= "NewValueData";
$Registry->{"\\ValueName"}= [ pack("fmt",$data), REG_DATATYPE ];
```

EXAMPLES

```
use Win32::TieRegistry( Delimiter=>"#", ArrayValues=>0 );
$pound= $Registry->Delimiter("/");
$diskKey= $Registry->{"LMachine/System/Disk/"}
  or  die "Can't read LMachine/System/Disk key: $^E\n";
$data= $key->{"/Information"}
  or  die "Can't read LMachine/System/Disk//Information value: $^E\n";
$remoteKey= $Registry->{"//ServerA/LMachine/System/"}
  or  die "Can't read //ServerA/LMachine/System/ key: $^E\n";
$remoteData= $remoteKey->{"Disk//Information"}
  or  die "Can't read ServerA's System/Disk//Information value: $^E\n";
foreach $entry (  keys(%$diskKey)  ) {
    ...
}
foreach $subKey (  $diskKey->SubKeyNames  ) {
    ...
}
$diskKey->AllowSave( 1 );
$diskKey->RegSaveKey( "C:/TEMP/DiskReg", [] );
```

DESCRIPTION

The *Win32::TieRegistry* module lets you manipulate the Registry via objects [as in "object oriented"] or via tied hashes. But you will probably mostly use a combination reference, that is, a reference to a tied hash that has also been made an object so that you can mix both access methods [as shown above].

If you did not get this module as part of *libwin32*, you might want to get a recent version of *libwin32* from CPAN which should include this module and the *Win32API::Registry* module that it uses.

Skip to the *SUMMARY* section if you just want to dive in and start using the Registry from Perl.

Accessing and manipulating the registry is extremely simple using *Win32::TieRegistry*. A single, simple expression can return you almost any bit of information stored in the Registry. *Win32::TieRegistry* also gives you full access to the "raw" underlying API calls so that you can do anything with the Registry in Perl that you could do in C. But the "simple" interface has been carefully designed to handle almost all operations itself without imposing arbitrary limits while providing sensible defaults so you can list only the parameters you care about.

But first, an overview of the Registry itself.

The Registry

The Registry is a forest: a collection of several tree structures. The root of each tree is a key. These root keys are identified by predefined constants whose names start with "HKEY_". Although all keys have a few attributes associated with each [a class, a time stamp, and security information], the most important aspect of keys is that each can contain subkeys and can contain values.

Each subkey has a name: a string which cannot be blank and cannot contain the delimiter character [backslash: `'\\'`] nor nul [`'\0'`]. Each subkey is also a key and so can contain subkeys and values [and has a class, time stamp, and security information].

Each value has a name: a string which be blank and contain the delimiter character [backslash: `'\\'`] and any character except for null, `'\0'`. Each value also has data associated with it. Each value‘s data is a contiguous chunk of bytes, which is exactly what a Perl string value is so Perl strings will usually be used to represent value data.

Each value also has a data type which says how to interpret the value data. The primary data types are:

REG_SZ

A null−terminated string.

REG_EXPAND_SZ

A null−terminated string which contains substrings consisting of a percent sign [`'%'`], an environment variable name, then a percent sign, that should be replaced with the value associate with that environment variable. The system does *not* automatically do this substitution.

REG_BINARY

Some arbitrary binary value. You can think of these as being "packed" into a string.

If your system has the *SetDualVar* module installed, the `DualBinVals()` option wasn‘t turned off, and you fetch a `REG_BINARY` value of 4 bytes or fewer, then you can use the returned value in a numeric context to get at the "unpacked" numeric value. See `GetValue()` for more information.

REG_MULTI_SZ

Several null−terminated strings concatenated together with an extra trailing `'\0'` at the end of the list. Note that the list can include empty strings so use the value‘s length to determine the end of the list, not the first occurrence of `'\0\0'`. It is best to set the `SplitMultis()` option so *Win32::TieRegistry* will split these values into an array of strings for you.

REG_DWORD

A long [4−byte] integer value. These values are expected either packed into a 4−character string or as a hex string of 4 characters [but *not* as a numeric value, unfortunately, as there is no sure way to tell a numeric value from a packed 4−byte string that just happens to be a string containing a valid numeric value].

How such values are returned depends on the `DualBinVals()` and `DWordsToHex()` options. See `GetValue()` for details.

In the underlying Registry calls, most places which take a subkey name also allow you to pass in a subkey "path" — a string of several subkey names separated by the delimiter character, backslash [`'\\'`]. For example, doing `RegOpenKeyEx(HKEY_LOCAL_MACHINE,"SYSTEM\\DISK",...)` is much like opening the `"SYSTEM"` subkey of `HKEY_LOCAL_MACHINE`, then opening its `"DISK"` subkey, then closing the `"SYSTEM"` subkey.

All of the *Win32::TieRegistry* features allow you to use your own delimiter in place of the system‘s delimiter, [`'\\'`]. In most of our examples we will use a forward slash [`'/'`] as our delimiter as it is easier to read and less error prone to use when writing Perl code since you have to type two backslashes for each backslash you want in a string. Note that this is true even when using single quotes — `'\\HostName\LMachine\'` is an invalid string and must be written as `'\\\\HostName\\LMachine\\'`.

You can also connect to the registry of other computers on your network. This will be discussed more later.

Although the Registry does not have a single root key, the *Win32::TieRegistry* module creates a virtual root key for you which has all of the *HKEY_** keys as subkeys.

Tied Hashes Documentation

Before you can use a tied hash, you must create one. One way to do that is via:

```
use Win32::TieRegistry ( TiedHash => '%RegHash' );
```

which exports a `%RegHash` variable into your package and ties it to the virtual root key of the Registry. An alternate method is:

```
my %RegHash;
use Win32::TieRegistry ( TiedHash => \%RegHash );
```

There are also several ways you can tie a hash variable to any other key of the Registry, which are discussed later.

Note that you will most likely use `$Registry` instead of using a tied hash. `$Registry` is a reference to a hash that has been tied to the virtual root of your computer‘s Registry [as if, `$Registry= \%RegHash`]. So you would use `$Registry->{Key}` rather than `$RegHash{Key}` and use `keys %{$Registry}` rather than `keys %RegHash`, for example.

For each hash which has been tied to a Registry key, the Perl `keys` function will return a list containing the name of each of the key‘s subkeys with a delimiter character appended to it and containing the name of each of the key‘s values with a delimiter prepended to it. For example:

```
keys( %{ $Registry->{"HKEY_CLASSES_ROOT\\batfile\\"} } )
```

might yield the following list value:

```
( "DefaultIcon\\",  # The subkey named "DefaultIcon"
  "shell\\",        # The subkey named "shell"
  "shellex\\",      # The subkey named "shellex"
  "\\",             # The default value [named ""]
  "\\EditFlags" )   # The value named "EditFlags"
```

For the virtual root key, short−hand subkey names are used as shown below. You can use the short−hand name, the regular *HKEY_** name, or any numeric value to access these keys, but the short−hand names are all that will be returned by the `keys` function.

"Classes" for HKEY_CLASSES_ROOT

Contains mappings between file name extensions and the uses for such files along with configuration information for COM [MicroSoft‘s Common Object Model] objects. Usually a link to the `"SOFTWARE\\Classes"` subkey of the `HKEY_LOCAL_MACHINE` key.

"CUser" for HKEY_CURRENT_USER

Contains information specific to the currently logged−in user. Mostly software configuration information. Usually a link to a subkey of the `HKEY_USERS` key.

"LMachine" for HKEY_LOCAL_MACHINE

Contains all manner of information about the computer.

"Users" for HKEY_USERS

Contains one subkey, `".DEFAULT"`, which gets copied to a new subkey whenever a new user is added. Also contains a subkey for each user of the system, though only those for active users [usually only one] are loaded at any given time.

"PerfData" for HKEY_PERFORMANCE_DATA

Used to access data about system performance. Access via this key is "special" and all but the most carefully constructed calls will fail, usually with `ERROR_INSUFFICIENT_BUFFER`. For example, you can‘t enumerate key names without also enumerating values which require huge buffers but the exact buffer size required cannot be determined beforehand because `RegQueryInfoKey()` fails with `ERROR_INSUFFICIENT_BUFFER` for `HKEY_PERFORMANCE_DATA` no matter how it is called. So it is currently not very useful to tie a hash to this key. You can use it to create an object to use for making carefully constructed calls to the underlying `Reg*()` routines.

"CConfig" for HKEY_CURRENT_CONFIG

Contains minimal information about the computer‘s current configuration that is required very early in the boot process. For example, setting for the display adapter such as screen resolution and refresh rate are found in here.

"DynData" for HKEY_DYN_DATA

Dynamic data. We have found no documentation for this key.

A tied hash is much like a regular hash variable in Perl — you give it a key string inside braces, [{ and }], and it gives you back a value [or lets you set a value]. For *Win32::TieRegistry* hashes, there are two types of values that will be returned.

SubKeys

If you give it a string which represents a subkey, then it will give you back a reference to a hash which has been tied to that subkey. It can‘t return the hash itself, so it returns a reference to it. It also blesses that reference so that it is also an object so you can use it to call method functions.

Values

If you give it a string which is a value name, then it will give you back a string which is the data for that value. Alternately, you can request that it give you both the data value string and the data value type [we discuss how to request this later]. In this case, it would return a reference to an array where the value data string is element `[0]` and the value data type is element `[1]`.

The key string which you use in the tied hash must be interpreted to determine whether it is a value name or a key name or a path that combines several of these or even other things. There are two simple rules that make this interpretation easy and unambiguous:

```
Put a delimiter after each key name.
Put a delimiter in front of each value name.
```

Exactly how the key string will be intepreted is governed by the following cases, in the order listed. These cases are designed to "do what you mean". Most of the time you won‘t have to think about them, especially if you follow the two simple rules above. After the list of cases we give several examples which should be clear enough so feel free to skip to them unless you are worried about the details.

Remote machines

If the hash is tied to the virtual root of the registry [or the virtual root of a remote machine‘s registry], then we treat hash key strings which start with the delimiter character specially.

If the hash key string starts with two delimiters in a row, then those should be immediately followed by the name of a remote machine whose registry we wish to connect to. That can be followed by a delimiter and more subkey names, etc. If the machine name is not following by anything, then a virtual root for the remote machine‘s registry is created, a hash is tied to it, and a reference to that hash it is returned.

Hash key string starts with the delimiter

If the hash is tied to a virtual root key, then the leading delimiter is ignored. It should be followed by a valid Registry root key name [either a short−hand name like `"LMachine"`, an *HKEY_** value, or a numeric value]. This alternate notation is allowed in order to be more consistant with the `Open()` method function.

For all other Registry keys, the leading delimiter indicates that the rest of the string is a value name. The leading delimiter is stripped and the rest of the string [which can be empty and can contain more delimiters] is used as a value name with no further parsing.

Exact match with direct subkey name followed by delimiter

If you have already called the Perl `keys` function on the tied hash [or have already called `MemberNames` on the object] and the hash key string exactly matches one of the strings returned, then no further parsing is done. In other words, if the key string exactly matches the name of a direct subkey with a delimiter appended, then a reference to a hash tied to that subkey is returned [but only if `keys` or `MemberNames` has already been called for that tied hash].

This is only important if you have selected a delimiter other than the system default delimiter and one of the subkey names contains the delimiter you have chosen. This rule allows you to deal with subkeys which contain your chosen delimiter in their name as long as you only traverse subkeys one level at a time and always enumerate the list of members before doing so.

The main advantage of this is that Perl code which recursively traverses a hash will work on hashes tied to Registry keys even if a non−default delimiter has been selected.

Hash key string contains two delimiters in a row

If the hash key string contains two [or more] delimiters in a row, then the string is split between the first pair of delimiters. The first part is interpreted as a subkey name or a path of subkey names separated by delimiters and with a trailing delimiter. The second part is interpreted as a value name with one leading delimiter [any extra delimiters are considered part of the value name].

Hash key string ends with a delimiter

If the key string ends with a delimiter, then it is treated as a subkey name or path of subkey names separated by delimiters.

Hash key string contains a delimiter

If the key string contains a delimiter, then it is split after the last delimiter. The first part is treated as a subkey name or path of subkey names separated by delimiters. The second part is ambiguous and is treated as outlined in the next item.

Hash key string contains no delimiters

If the hash key string contains no delimiters, then it is ambiguous.

If you are reading from the hash [fetching], then we first use the key string as a value name. If there is a value with a matching name in the Registry key which the hash is tied to, then the value data string [and possibly the value data type] is returned. Otherwise, we retry by using the hash key string as a subkey name. If there is a subkey with a matching name, then we return a reference to a hash tied to that subkey. Otherwise we return `undef`.

If you are writing to the hash [storing], then we use the key string as a subkey name only if the value you are storing is a reference to a hash value. Otherwise we use the key string as a value name.

=head3 Examples

Here are some examples showing different ways of accessing Registry information using references to tied hashes:

Canonical value fetch

```
$tip18= $Registry->{"HKEY_LOCAL_MACHINE\\Software\\Microsoft\\"
             . 'Windows\\CurrentVersion\\Explorer\\Tips\\\\18'};
```

Should return the text of important tip number 18. Note that two backslashes, `"\\"`, are required to get a single backslash into a Perl double−quoted or single−qouted string. Note that `"\\"` is appended to each key name [`"HKEY_LOCAL_MACHINE"` through `"Tips"`] and `"\\"` is prepended to the value name, `"18"`.

Changing your delimiter

```
$Registry->Delimiter("/");
$tip18= $Registry->{"HKEY_LOCAL_MACHINE/Software/Microsoft/"
```

```
                  . 'Windows/CurrentVersion/Explorer/Tips//18'};
```

This usually makes things easier to read when working in Perl. All remaining examples will assume the delimiter has been changed as above.

Using intermediate keys

```
$ms= $Registry->{"LMachine/Software/Microsoft/"};
$tips= $ms->{"Windows/CurrentVersion/Explorer/Tips/"};
$tip18= $winlogon->{"/18"};
```

Same as above but opens more keys into the Registry which lets you efficiently re−access those intermediate keys. This is slightly less efficient if you never reuse those intermediate keys.

Chaining in a single statement

```
$tip18= $Registry->{"LMachine/Software/Microsoft/"}->
          {"Windows/CurrentVersion/Explorer/Tips/"}->{"/18"};
```

Like above, this creates intermediate key objects then uses them to access other data. Once this statement finishes, the intermediate key objects are destroyed. Several handles into the Registry are opened and closed by this statement so it is less efficient but there are times when this will be useful.

Even less efficient example of chaining

```
$tip18= $Registry->{"LMachine/Software/Microsoft"}->
          {"Windows/CurrentVersion/Explorer/Tips"}->{"/18"};
```

Because we left off the trailing delimiters, *Win32::TieRegistry* doesn‘t know whether final names, `"Microsoft"` and `"Tips"`, are subkey names or value names. So this statement ends up executing the same code as the next one.

What the above really does

```
$tip18= $Registry->{"LMachine/Software/"}->{"Microsoft"}->
          {"Windows/CurrentVersion/Explorer/"}->{"Tips"}->{"/18"};
```

With more chains to go through, more temporary objects are created and later destroyed than in our first chaining example. Also, when `"Microsoft"` is looked up, *Win32::TieRegistry* first tries to open it as a value and fails then tries it as a subkey. The same is true for when it looks up `"Tips"`.

Getting all of the tips

```
$tips= $Registry->{"LMachine/Software/Microsoft/"}->
          {"Windows/CurrentVersion/Explorer/Tips/"}
  or  die "Can't find the Windows tips: $^E\n";
foreach(  keys %$tips  ) {
    print "$_: ", $tips->{$_}, "\n";
}
```

First notice that we actually check for failure for the first time. We are assuming that the `"Tips"` key contains no subkeys. Otherwise the `print` statement would show something like `"Win32::TieRegistry=HASH(0xc03ebc)"` for each subkey.

The output from the above code will start something like:

```
/0: If you don't know how to do something,[...]
```

=head3 Deleting items

You can use the Perl `delete` function to delete a value from a Registry key or to delete a subkey as long that subkey contains no subkeys of its own. See *More Examples*, below, for more information.

=head3 Storing items

You can use the Perl assignment operator [=] to create new keys, create new values, or replace values. The values you store should be in the same format as the values you would fetch from a tied hash. For example, you can use a single assignment statement to copy an entire Registry tree. The following statement:

```
    $Registry->{"LMachine/Software/Classes/Tie_Registry/"}=
      $Registry->{"LMachine/Software/Classes/batfile/"};
```

creates a `"Tie_Registry"` subkey under the `"Software\\Classes"` subkey of the `HKEY_LOCAL_MACHINE` key. Then it populates it with copies of all of the subkeys and values in the `"batfile"` subkey and all of its subkeys. Note that you need to have called `$Registry->ArrayValues(1)` for the proper value data type information to be copied. Note also that this release of *Win32::TieRegistry* does not copy key attributes such as class name and security information [this is planned for a future release].

The following statement creates a whole subtree in the Registry:

```
    $Registry->{"LMachine/Software/FooCorp/"}= {
        "FooWriter/" => {
            "/Version" => "4.032",
            "Startup/" => {
                "/Title" => "Foo Writer Deluxe ][",
                "/WindowSize" => [ pack("LL",$wid,$ht), "REG_BINARY" ],
                "/TaskBarIcon" => [ "0x0001", "REG_DWORD" ],
            },
            "Compatibility/" => {
                "/AutoConvert" => "Always",
                "/Default Palette" => "Windows Colors",
            },
        },
        "/License", => "0123-9C8EF1-09-FC",
    };
```

Note that all but the last Registry key used on the left−hand side of the assignment [that is, "LMachine/Software/" but not "FooCorp/"] must already exist for this statement to succeed.

By using the leading a trailing delimiters on each subkey name and value name, *Win32::TieRegistry* will tell you if you try to assign subkey information to a value or visa−versa.

=head3 More examples

Adding a new tip

```
        $tips= $Registry->{"LMachine/Software/Microsoft/"}->
                  {"Windows/CurrentVersion/Explorer/Tips/"}
          or  die "Can't find the Windows tips: $^E\n";
        $tips{'/186'}= "Be very careful when making changes to the Registry!";
```

Deleting our new tip

```
        $tips= $Registry->{"LMachine/Software/Microsoft/"}->
                  {"Windows/CurrentVersion/Explorer/Tips/"}
          or  die "Can't find the Windows tips: $^E\n";
        $tip186= delete $tips{'/186'};
```

Note that Perl's `delete` function returns the value that was deleted.

Adding a new tip differently

```
        $Registry->{"LMachine/Software/Microsoft/" .
                    "Windows/CurrentVersion/Explorer/Tips//186"}=
          "Be very careful when making changes to the Registry!";
```

Deleting differently

```
        $tip186= delete $Registry->{"LMachine/Software/Microsoft/Windows/" .
                                    "CurrentVersion/Explorer/Tips//186"};
```

Note that this only deletes the tail of what we looked up, the `"186"` value, not any of the keys listed.

Deleting a key

WARNING: The following code will delete all information about the current user‘s tip preferences. Actually executing this command would probably cause the user to see the Welcome screen the next time they log in and may cause more serious problems. This statement is shown as an example only and should not be used when experimenting.

```
$tips= delete $Registry->{"CUser/Software/Microsoft/Windows/" .
                           "CurrentVersion/Explorer/Tips/"};
```

This deletes the `"Tips"` key and the values it contains. The `delete` function will return a reference to a hash [not a tied hash] containing the value names and value data that were deleted.

The information to be returned is copied from the Registry into a regular Perl hash before the key is deleted. If the key has many subkeys, this copying could take a significant amount of memory and/or processor time. So you can disable this process by calling the `FastDelete` member function:

```
$prevSetting= $regKey->FastDelete(1);
```

which will cause all subsequent delete operations via `$regKey` to simply return a true value if they succeed. This optimization is automatically done if you use `delete` in a void context.

Technical notes on deleting

If you use `delete` to delete a Registry key or value and use the return value, then *Win32::TieRegistry* usually looks up the current contents of that key or value so they can be returned if the deletion is successful. If the deletion succeeds but the attempt to lookup the old contents failed, then the return value of `delete` will be `$^E` from the failed part of the operation.

Undeleting a key

```
$Registry->{"LMachine/Software/Microsoft/Windows/" .
            "CurrentVersion/Explorer/Tips/"}= $tips;
```

This adds back what we just deleted. Note that this version of *Win32::TieRegistry* will use defaults for the key attributes [such as class name and security] and will not restore the previous attributes.

Not deleting a key

WARNING: Actually executing the following code could cause serious problems. This statement is shown as an example only and should not be used when experimenting.

```
$res= delete $Registry->{"CUser/Software/Microsoft/Windows/"}
defined($res)  ||  die "Can’t delete URL key: $^E\n";
```

Since the "Windows" key should contain subkeys, that `delete` statement should make no changes to the Registry, return `undef`, and set `$^E` to "Access is denied".

Not deleting again

```
$tips= $Registry->{"CUser/Software/Microsoft/Windows/" .
                   "CurrentVersion/Explorer/Tips/"};
delete $tips;
```

The Perl `delete` function requires that its argument be an expression that ends in a hash element lookup [or hash slice], which is not the case here. The `delete` function doesn‘t know which hash `$tips` came from and so can‘t delete it.

Objects Documentation

The following member functions are defined for use on *Win32::TieRegistry* objects:

new The `new` method creates a new *Win32::TieRegistry* object. `new` is mostly a synonym for `Open()` so see `Open()` below for information on what arguments to pass in. Examples:

```
    $machKey= new Win32::TieRegistry "LMachine"
      or  die "Can't access HKEY_LOCAL_MACHINE key: $^E\n";
    $userKey= Win32::TieRegistry->new("CUser")
      or  die "Can't access HKEY_CURRENT_USER key: $^E\n";
```

Note that calling `new` via a reference to a tied hash returns a simple object, not a reference to a tied hash.

Open

`$subKey= $key-`Open(`$sSubKey, $rhOptions`)

The `Open` method opens a Registry key and returns a new *Win32::TieRegistry* object associated with that Registry key. If `Open` is called via a reference to a tied hash, then `Open` returns another reference to a tied hash. Otherwise `Open` returns a simple object and you should then use `TiedRef` to get a reference to a tied hash.

`$sSubKey` is a string specifying a subkey to be opened. Alternately `$sSubKey` can be a reference to an array value containing the list of increasingly deep subkeys specifying the path to the subkey to be opened.

`$rhOptions` is an optional reference to a hash containing extra options. The `Open` method supports two options, `"Delimiter"` and `"Access"`, and `$rhOptions` should have only have zero or more of these strings as keys. See the "Examples" section below for more information.

The `"Delimiter"` option specifies what string [usually a single character] will be used as the delimiter to be appended to subkey names and prepended to value names. If this option is not specified, the new key [`$subKey`] inherits the delimiter of the old key [`$key`] .

The `"Access"` option specifies what level of access to the Registry key you wish to have once it has been opened. If this option is not specified, the new key [`$subKey`] is opened with the same access level used when the old key [`$key`] was opened. The virtual root of the Registry pretends it was opened with access `KEY_READ()|KEY_WRITE()` so this is the default access when opening keys directory via `$Registry`. If you don‘t plan on modifying a key, you should open it with `KEY_READ` access as you may not have `KEY_WRITE` access to it or some of its subkeys.

If the `"Access"` option value is a string that starts with `"KEY_"`, then it should match of the predefined access levels [probably `"KEY_READ"`, `"KEY_WRITE"`, or `"KEY_ALL_ACCESS"`] exported by the *Win32API::Registry* module. Otherwise, a numeric value is expected. For maximum flexibility, include `use Win32::TieRegistry qw(:KEY_);`, for example, near the top of your script so you can specify more complicated access levels such as `KEY_READ()|KEY_WRITE()`.

If `$sSubKey` does not begin with the delimiter [or `$sSubKey` is an array reference], then the path to the subkey to be opened will be relative to the path of the original key [`$key`] . If `$sSubKey` begins with a single delimiter, then the path to the subkey to be opened will be relative to the virtual root of the Registry on whichever machine the original key resides. If `$sSubKey` begins with two consecutive delimiters, then those must be followed by a machine name which causes the `Connect()` method function to be called.

Examples:

```
    $machKey= $Registry->Open( "LMachine", {Access=>KEY_READ(),Delimiter=>"/"} )
      or  die "Can't open HKEY_LOCAL_MACHINE key: $^E\n";
    $swKey= $machKey->Open( "Software" );
    $logonKey= $swKey->Open( "Microsoft/Windows NT/CurrentVersion/Winlogon/" );
    $NTversKey= $swKey->Open( ["Microsoft","Windows NT","CurrentVersion"] );
    $versKey= $swKey->Open( qw(Microsoft Windows CurrentVersion) );

    $remoteKey= $Registry->Open( "//HostA/LMachine/System/", {Delimiter=>"/"} )
      or  die "Can't connect to HostA or can't open subkey: $^E\n";
```

Clone
`$copy= $key-`Clone

Creates a new object that is associated with the same Registry key as the invoking object.

Connect
`$remoteKey= $Registry-`Connect(`$sMachineName, $sKeyPath, $rhOptions`)

The `Connect` method connects to the Registry of a remote machine, and opens a key within it, then returns a new *Win32::TieRegistry* object associated with that remote Registry key. If `Connect` was called using a reference to a tied hash, then the return value will also be a reference to a tied hash [or `undef`]. Otherwise, if you wish to use the returned object as a tied hash [not just as an object], then use the `TiedRef` method function after `Connect`.

`$sMachineName` is the name of the remote machine. You don‘t have to preceed the machine name with two delimiter characters.

`$sKeyPath` is a string specifying the remote key to be opened. Alternately `$sKeyPath` can be a reference to an array value containing the list of increasingly deep keys specifying the path to the key to be opened.

`$rhOptions` is an optional reference to a hash containing extra options. The `Connect` method supports two options, `"Delimiter"` and `"Access"`. See the `Open` method documentation for more information on these options.

`$sKeyPath` is already relative to the virtual root of the Registry of the remote machine. A single leading delimiter on `sKeyPath` will be ignored and is not required.

`$sKeyPath` can be empty in which case `Connect` will return an object representing the virtual root key of the remote Registry. Each subsequent use of `Open` on this virtual root key will call the system `RegConnectRegistry` function.

The `Connect` method can be called via any *Win32::TieRegistry* object, not just `$Registry.` Attributes such as the desired level of access and the delimiter will be inherited from the object used but the `$sKeyPath` will always be relative to the virtual root of the remote machine‘s registry.

Examples:

```
$remMachKey= $Registry->Connect( "HostA", "LMachine", {Delimiter->"/"} )
  or  die "Can’t connect to HostA’s HKEY_LOCAL_MACHINE key: $^E\n";

$remVersKey= $remMachKey->Connect( "www.microsoft.com",
               "LMachine/Software/Microsoft/Inetsrv/CurrentVersion/",
               { Access=>KEY_READ, Delimiter=>"/" } )
  or  die "Can’t check what version of IIS Microsoft is running: $^E\n";

$remVersKey= $remMachKey->Connect( "www",
               qw(LMachine Software Microsoft Inetsrv CurrentVersion) )
  or  die "Can’t check what version of IIS we are running: $^E\n";
```

ObjectRef
`$object_ref= $obj_or_hash_ref-`ObjectRef

For a simple object, just returns itself [`$obj == $obj-ObjectRef`].

For a reference to a tied hash [if it is also an object], `ObjectRef` returns the simple object that the hash is tied to.

This is primarilly useful when debugging since typing `x $Registry` will try to display your *entire* registry contents to your screen. But the debugger command `x $Registry-ObjectRef` will just dump the implementation details of the underlying object to your screen.

Flush(**`$bFlush`**)

Flushes all cached information about the Registry key so that future uses will get fresh data from the Registry.

If the optional `$bFlush` is specified and a true value, then `RegFlushKey()` will be called, which is almost never necessary.

GetValue
`$ValueData=` `$key-`GetValue(`$sValueName`)
(`$ValueData,$ValueType)=` `$key-`GetValue(`$sValueName`)

Gets a Registry value‘s data and data type.

`$ValueData` is usually just a Perl string that contains the value data [packed into it]. For certain types of data, however, `$ValueData` may be processed as described below.

`$ValueType` is the `REG_*` constant describing the type of value data stored in `$ValueData.` If the `DualTypes()` option is on, then `$ValueType` will be a dual value. That is, when used in a numeric context, `$ValueType` will give the numeric value of a `REG_*` constant. However, when used in a non−numeric context, `$ValueType` will return the name of the `REG_*` constant, for example `"REG_SZ"` [note the quotes]. So both of the following can be true at the same time:

```
$ValueType == REG_SZ()
$ValueType eq "REG_SZ"
```

REG_SZ and REG_EXPAND_SZ

If the `FixSzNulls()` option is on, then the trailing `'\0'` will be stripped [unless there isn‘t one] before values of type `REG_SZ` and `REG_EXPAND_SZ` are returned. Note that `SetValue()` will add a trailing `'\0'` under similar circumstances.

REG_MULTI_SZ

If the `SplitMultis()` option is on, then values of this type are returned as a reference to an array containing the strings. For example, a value that, with `SplitMultis()` off, would be returned as:

```
"Value1\000Value2\000\000"
```

would be returned, with `SplitMultis()` on, as:

```
[ "Value1", "Value2" ]
```

REG_DWORD

If the `DualBinVals()` option is on, then the value is returned as a scalar containing both a string and a number [much like the `$!` variable — see the *SetDualVar* module for more information] where the number part is the "unpacked" value. Use the returned value in a numeric context to access this part of the value. For example:

```
$num= 0 + $Registry->{"CUser/Console//ColorTable01"};
```

If the `DWordsToHex()` option is off, the string part of the returned value is a packed, 4−byte string [use `unpack("L",$value)` to get the numeric value.

If `DWordsToHex()` is on, the string part of the returned value is a 10−character hex strings [with leading "0x"]. You can use `hex($value)` to get the numeric value.

Note that `SetValue()` will properly understand each of these returned value formats no matter how `DualBinVals()` is set.

ValueNames
@names= `$key-`ValueNames

Returns the list of value names stored directly in a Registry key. Note that the names returned do *not* have a delimiter prepended to them like with `MemberNames()` and tied hashes.

Once you request this information, it is cached in the object and future requests will always return the same list unless `Flush()` has been called.

SubKeyNames
@key_names= `$key−`SubKeyNames

Returns the list of subkey names stored directly in a Registry key. Note that the names returned do *not* have a delimiter appended to them like with `MemberNames()` and tied hashes.

Once you request this information, it is cached in the object and future requests will always return the same list unless `Flush()` has been called.

SubKeyClasses
@classes= `$key−`SubKeyClasses

Returns the list of classes for subkeys stored directly in a Registry key. The classes are returned in the same order as the subkey names returned by `SubKeyNames()`.

SubKeyTimes
@times= `$key−`SubKeyTimes

Returns the list of last−modified times for subkeys stored directly in a Registry key. The times are returned in the same order as the subkey names returned by `SubKeyNames()`. Each time is a `FILETIME` structure packed into a Perl string.

Once you request this information, it is cached in the object and future requests will always return the same list unless `Flush()` has been called.

MemberNames
@members= `$key−`MemberNames

Returns the list of subkey names and value names stored directly in a Registry key. Subkey names have a delimiter appended to the end and value names have a delimiter prepended to the front.

Note that a value name could end in a delimiter [or could be `""` so that the member name returned is just a delimiter] so the presence or absence of the leading delimiter is what should be used to determine whether a particular name is for a subkey or a value, not the presence or absence of a trailing delimiter.

Once you request this information, it is cached in the object and future requests will always return the same list unless `Flush()` has been called.

Information
%info= `$key−`Information
@items= `$key−`Information(@itemNames);

Returns the following information about a Registry key:

LastWrite

A `FILETIME` structure indicating when the key was last modified and packed into a Perl string.

CntSubKeys

The number of subkeys stored directly in this key.

CntValues

The number of values stored directly in this key.

SecurityLen

The length [in bytes] of the largest[?] `SECURITY_DESCRIPTOR` associated with the Registry key.

MaxValDataLen

The length [in bytes] of the longest value data associated with a value stored in this key.

MaxSubKeyLen

The length [in chars] of the longest subkey name associated with a subkey stored in this key.

MaxSubClassLen

The length [in chars] of the longest class name associated with a subkey stored directly in this key.

MaxValNameLen

The length [in chars] of the longest value name associated with a value stored in this key.

With no arguments, returns a hash [not a reference to a hash] where the keys are the names for the items given above and the values are the information describe above. For example:

```
%info= ( "CntValues" => 25,         # Key contains 25 values.
         "MaxValNameLen" => 20,     # One of which has a 20-char name.
         "MaxValDataLen" => 42,     # One of which has a 42-byte value.
         "CntSubKeys" => 1,         # Key has 1 immediate subkey.
         "MaxSubKeyLen" => 13,      # One of which has a 12-char name.
         "MaxSubClassLen" => 0,     # All of which have class names of "".
         "SecurityLen" => 232,      # One SECURITY_DESCRIPTOR is 232 bytes.
         "LastWrite" => "\x90mZ\cX{\xA3\xBD\cA\c@\cA"
                      # Key was last modifed 1998/06/01 16:29:32 GMT
       );
```

With arguments, each one must be the name of a item given above. The return value is the information associated with the listed names. In other words:

```
return $key->Information( @names );
```

returns the same list as:

```
%info= $key->Information;
return @info{@names};
```

Delimiter
`$oldDelim= $key-`Delimiter
`$oldDelim= $key-`Delimiter(`$newDelim`)

Gets and possibly changes the delimiter used for this object. The delimiter is appended to subkey names and prepended to value names in many return values. It is also used when parsing keys passed to tied hashes.

The delimiter defaults to backslash (`'\\'`) but is inherited from the object used to create a new object and can be specified by an option when a new object is created.

Handle
`$handle= $key-`Handle

Returns the raw `HKEY` handle for the associated Registry key as an integer value. This value can then be used to `Reg*()` calls from *Win32API::Registry*. However, it is usually easier to just call the *Win32API::Registry* calls directly via:

```
$key->RegNotifyChangeKeyValue( ... );
```

For the virtual root of the local or a remote Registry, `Handle()` return `"NONE"`.

Path
`$path= $key-`Path

Returns a string describing the path of key names to this Registry key. The string is built so that if it were passed to `$Registry-Open()`, it would reopen the same Registry key [except in the rare case where one of the key names contains `$key-Delimiter`].

Machine
`$computerName= $key-`Machine

Returns the name of the computer [or "machine"] on which this Registry key resides. Returns `""` for local Registry keys.

Access

Returns the numeric value of the bit mask used to specify the types of access requested when this Registry key was opened. Can be compared to `KEY_*` values.

OS_Delimiter

Returns the delimiter used by the operating system's `RegOpenKeyEx()` call. For Win32, this is always backslash (`"\\"`).

Roots

Returns the mapping from root key names like `"LMachine"` to their associated `HKEY_*` constants. Primarily for internal use and subject to change.

Tie
`$key-`Tie(\%hash);

Ties the referenced hash to that Registry key. Pretty much the same as

```
    tie %hash, ref($key), $key;
```

Since `ref($key)` is the class [package] to tie the hash to and `TIEHASH()` just returns its argument, `$key,` [without calling `new()`] when it sees that it is already a blessed object.

TiedRef
`$TiedHashRef= $hash_or_obj_ref-`TiedRef

For a simple object, returns a reference to a hash tied to the object. Used to promote a simple object into a combined object and hash ref.

If already a reference to a tied hash [that is also an object], it just returns itself [`$ref == $ref-TiedRef`].

Mostly used internally.

ArrayValues
`$oldBool= $key-`ArrayValues
`$oldBool= $key-`ArrayValues(`$newBool`)

Gets the current setting of the `ArrayValues` option and possibly turns it on or off.

When off, Registry values fetched via a tied hash are returned as just a value scalar [the same as `GetValue()` in a scalar context]. When on, they are returned as a reference to an array containing the value data as the `[0]` element and the data type as the `[1]` element.

TieValues
`$oldBool=` TieValues
`$oldBool=` TieValues(`$newBool`)

Gets the current setting of the `TieValues` option and possibly turns it on or off.

Turning this option on is not yet supported in this release of *Win32::TieRegistry*. In a future release, turning this option on will cause Registry values returned from a tied hash to be a tied array that you can use to modify the value in the Registry.

FastDelete
`$oldBool= $key-`FastDelete
`$oldBool= $key-`FastDelete(`$newBool`)

Gets the current setting of the `FastDelete` option and possibly turns it on or off.

When on, successfully deleting a Registry key [via a tied hash] simply returns `1`.

When off, successfully deleting a Registry key [via a tied hash and not in a void context] returns a reference to a hash that contains the values present in the key when it was deleted. This hash is just like that returned when referencing the key before it was deleted except that it is an ordinary hash, not one tied to the *Win32::TieRegistry* package.

Note that deleting either a Registry key or value via a tied hash *in a void context* prevents any overhead in trying to build an appropriate return value.

Note that deleting a Registry *value* via a tied hash [not in a void context] returns the value data even if <FastDelete is on.

SplitMultis
`$oldBool= $key−`SplitMultis
`$oldBool= $key−`SplitMultis(`$newBool`)

Gets the current setting of the `SplitMultis` option and possibly turns it on or off.

If on, Registry values of type `REG_MULTI_SZ` are returned as a reference to an array of strings. See `GetValue()` for more information.

DWordsToHex
`$oldBool= $key−`DWordsToHex
`$oldBool= $key−`DWordsToHex(`$newBool`)

Gets the current setting of the `DWordsToHex` option and possibly turns it on or off.

If on, Registry values of type `REG_DWORD` are returned as a hex string with leading `"0x"` and longer than 4 characters. See `GetValue()` for more information.

FixSzNulls
`$oldBool= $key−`FixSzNulls
`$oldBool= $key−`FixSzNulls(`$newBool`)

Gets the current setting of the `FixSzNulls` option and possibly turns it on or off.

If on, Registry values of type `REG_SZ` and `REG_EXPAND_SZ` have trailing `'\0'`s added before they are set and stripped before they are returned. See `GetValue()` and `SetValue()` for more information.

DualTypes
`$oldBool= $key−`DualTypes
`$oldBool= $key−`DualTypes(`$newBool`)

Gets the current setting of the `DualTypes` option and possibly turns it on or off.

If on, data types are returned as a combined numeric/string value holding both the numeric value of a `REG_*` constant and the string value of the constant‘s name. See `GetValue()` for more information.

DualBinVals
`$oldBool= $key−`DualBinVals
`$oldBool= $key−`DualBinVals(`$newBool`)

Gets the current setting of the `DualBinVals` option and possibly turns it on or off.

If on, Registry value data of type `REG_BINARY` and no more than 4 bytes long and Registry values of type `REG_DWORD` are returned as a combined numeric/string value where the numeric value is the "unpacked" binary value as returned by:

```
        hex reverse unpack( "h*", $valData )
```

on a "little−endian" computer. [Would be `hex unpack("H*",$valData)` on a "big−endian" computer if this module is ever ported to one.]

See `GetValue()` for more information.

GetOptions
@oldOptValues= `$key-`GetOptions(@optionNames)
`$refHashOfOldOpts= $key-`GetOptions()
`$key-`GetOptions(\%hashForOldOpts)

Returns the current setting of any of the following options:

```
    Delimiter     FixSzNulls    DWordsToHex
    ArrayValues   SplitMultis   DualBinVals
    TieValues     FastDelete    DualTypes
```

Pass in one or more of the above names (as strings) to get back an array of the corresponding current settings in the same order:

```
  my( $fastDel, $delim )= $key->GetOptions("FastDelete","Delimiter");
```

Pass in no arguments to get back a reference to a hash where the above option names are the keys and the values are the corresponding current settings for each option:

```
  my $href= $key->GetOptions();
  my $delim= $href->{Delimiter};
```

Pass in a single reference to a hash to have the above key/value pairs *added* to the referenced hash. For this case, the return value is the original object so further methods can be chained after the call to GetOptions:

```
  my %oldOpts;
  $key->GetOptions( \%oldOpts )->SetOptions( Delimiter => "/" );
```

SetOptions
@oldOpts= `$key-`SetOptions(optNames=`$optValue,...`)

Changes the current setting of any of the following options, returning the previous setting(s):

```
    Delimiter     FixSzNulls    DWordsToHex   AllowLoad
    ArrayValues   SplitMultis   DualBinVals   AllowSave
    TieValues     FastDelete    DualTypes
```

For `AllowLoad` and `AllowSave`, instead of the previous setting, `SetOptions` returns whether or not the change was successful.

In a scalar context, returns only the last item. The last option can also be specified as `"ref"` or `"r"` [which doesn‘t need to be followed by a value] to allow chaining:

```
    $key->SetOptions(AllowSave=>1,"ref")->RegSaveKey(...)
```

SetValue
`$okay= $key-`SetValue(`$ValueName, $ValueData`);
`$okay= $key-`SetValue(`$ValueName, $ValueData, $ValueType`);

Adds or replaces a Registry value. Returns a true value if successfully, false otherwise.

`$ValueName` is the name of the value to add or replace and should *not* have a delimiter prepended to it. Case is ignored.

`$ValueType` is assumed to be `REG_SZ` if it is omitted. Otherwise, it should be one the `REG_*` constants.

`$ValueData` is the data to be stored in the value, probably packed into a Perl string. Other supported formats for value data are listed below for each posible `$ValueType`.

REG_SZ or REG_EXPAND_SZ

The only special processing for these values is the addition of the required trailing `'\0'` if it is missing. This can be turned off by disabling the `FixSzNulls` option.

REG_MULTI_SZ

These values can also be specified as a reference to a list of strings. For example, the following two lines are equivalent:

```
    $key->SetValue( "Val1\000Value2\000LastVal\000\000", "REG_MULTI_SZ" );
    $key->SetValue( ["Val1","Value2","LastVal"], "REG_MULTI_SZ" );
```

Note that if the required two trailing nulls (`"\000\000"`) are missing, then this release of `SetValue()` will *not* add them.

REG_DWORD

These values can also be specified as a hex value with the leading `"0x"` included and totaling *more than* 4 bytes. These will be packed into a 4−byte string via:

```
    $data= pack( "L", hex($data) );
```

REG_BINARY

This value type is listed just to emphasize that no alternate format is supported for it. In particular, you should *not* pass in a numeric value for this type of data. `SetValue()` cannot distinguish such from a packed string that just happens to match a numeric value and so will treat it as a packed string.

An alternate calling format:

```
    $okay= $key->SetValue( $ValueName, [ $ValueData, $ValueType ] );
```

[two arguments, the second of which is a reference to an array containing the value data and value type] is supported to ease using tied hashes with `SetValue()`.

CreateKey
`$newKey= $key−`CreateKey(`$subKey`);
`$newKey= $key−`CreateKey(`$subKey`, { Option=OptVal,... });

Creates a Registry key or just updates attributes of one. Calls `RegCreateKeyEx()` then, if it succeeded, creates an object associated with the [possibly new] subkey.

`$subKey` is the name of a subkey [or a path to one] to be created or updated. It can also be a reference to an array containing a list of subkey names.

The second argument, if it exists, should be a reference to a hash specifying options either to be passed to `RegCreateKeyEx()` or to be used when creating the associated object. The following items are the supported keys for this options hash:

Delimiter

Specifies the delimiter to be used to parse `$subKey` and to be used in the new object. Defaults to `$key−Delimiter`.

Access

Specifies the types of access requested when the subkey is opened. Should be a numeric bit mask that combines one or more `KEY_*` constant values.

Class

The name to assign as the class of the new or updated subkey. Defaults to `""` as we have never seen a use for this information.

Disposition

Lets you specify a reference to a scalar where, upon success, will be stored either `REG_CREATED_NEW_KEY()` or `REG_OPENED_EXISTING_KEY()` depending on whether a new key was created or an existing key was opened.

If you, for example, did `use Win32::TieRegistry qw(REG_CREATED_NEW_KEY)` then you can use `REG_CREATED_NEW_KEY()` to compare against the numeric value stored in the referenced scalar.

If the `DualTypes` option is enabled, then in addition to the numeric value described above, the referenced scalar will also have a string value equal to either `"REG_CREATED_NEW_KEY"` or `"REG_OPENED_EXISTING_KEY"`, as appropriate.

Security

Lets you specify a `SECURITY_ATTRIBUTES` structure packed into a Perl string. See `Win32API::Registry::RegCreateKeyEx()` for more information.

Volatile

If true, specifies that the new key should be volatile, that is, stored only in memory and not backed by a hive file [and not saved if the computer is rebooted]. This option is ignored under Windows 95. Specifying `Volatile=1` is the same as specifying `Options=REG_OPTION_VOLATILE`.

Backup

If true, specifies that the new key should be opened for backup/restore access. The `Access` option is ignored. If the calling process has enabled `"SeBackupPrivilege"`, then the subkey is opened with `KEY_READ` access as the `"LocalSystem"` user which should have access to all subkeys. If the calling process has enabled `"SeRestorePrivilege"`, then the subkey is opened with `KEY_WRITE` access as the `"LocalSystem"` user which should have access to all subkeys.

This option is ignored under Windows 95. Specifying `Backup=1` is the same as specifying `Options=REG_OPTION_BACKUP_RESTORE`.

Options

Lets you specify options to the `RegOpenKeyEx()` call. The value for this option should be a numeric value combining zero or more of the `REG_OPTION_*` bit masks. You may with to used the `Volatile` and/or `Backup` options instead of this one.

StoreKey

`$newKey= $key–`StoreKey(`$subKey,` \%Contents);

Primarily for internal use.

Used to create or update a Registry key and any number of subkeys or values under it or its subkeys.

`$subKey` is the name of a subkey to be created [or a path of subkey names separated by delimiters]. If that subkey already exists, then it is updated.

`\%Contents` is a reference to a hash containing pairs of value names with value data and/or subkey names with hash references similar to `\%Contents`. Each of these cause a value or subkey of `$subKey` to be created or updated.

If `$Contents{""}` exists and is a reference to a hash, then it used as the options argument when `CreateKey()` is called for `$subKey`. This allows you to specify ...

```
    if(  defined( $$data{""} )  &&  "HASH" eq ref($$data{""})  ) {
        $self= $this->CreateKey( $subKey, delete $$data{""} );
```

Load

`$newKey= $key–`Load(`$file`)
`$newKey= $key–`Load(`$file, $newSubKey`)
`$newKey= $key–`Load(`$file, $newSubKey,` { Option=OptVal... })
`$newKey= $key–`Load(`$file,` { Option=OptVal... })

Loads a hive file into a Registry. That is, creates a new subkey and associates a hive file with it.

`$file` is a hive file, that is a file created by calling `RegSaveKey()`. The `$file` path is interpreted relative to `%SystemRoot%/System32/config` on the machine where `$key` resides.

`$newSubKey` is the name to be given to the new subkey. If `$newSubKey` is specified, then `$key` must be `HKEY_LOCAL_MACHINE` or `HKEY_USERS` of the local computer or a remote computer and `$newSubKey` should not contain any occurrences of either the delimiter or the OS delimiter.

If `$newSubKey` is not specified, then it is as if `$key` was `$Registry−{LMachine}` and `$newSubKey` is `"PerlTie:999"` where `"999"` is actually a sequence number incremented each time this process calls `Load()`.

You can specify as the last argument a reference to a hash containing options. You can specify the same options that you can specify to `Open()`. See `Open()` for more information on those. In addition, you can specify the option `"NewSubKey"`. The value of this option is interpretted exactly as if it was specified as the `$newSubKey` parameter and overrides the `$newSubKey` if one was specified.

The hive is automatically unloaded when the returned object [`$newKey]` is destroyed. Registry key objects opened within the hive will keep a reference to the `$newKey` object so that it will not be destroyed before these keys are closed.

UnLoad
`$okay= $key−`UnLoad

Unloads a hive that was loaded via `Load()`. Cannot unload other hives. `$key` must be the return from a previous call to `Load()`. `$key` is closed and then the hive is unloaded.

AllowSave
`$okay=` AllowSave(`$bool`)

Enables or disables the `"ReBackupPrivilege"` privilege for the current process. You will probably have to enable this privilege before you can use `RegSaveKey()`.

The return value indicates whether the operation succeeded, not whether the privilege was previously enabled.

AllowLoad
`$okay=` AllowLoad(`$bool`)

Enables or disables the `"ReRestorePrivilege"` privilege for the current process. You will probably have to enable this privilege before you can use `RegLoadKey()`, `RegUnLoadKey()`, `RegReplaceKey()`, or `RegRestoreKey` and thus `Load()` and `UnLoad()`.

The return value indicates whether the operation succeeded, not whether the privilege was previously enabled.

Exports [`use` and `import()`]

To have nothing imported into your package, use something like:

```
    use Win32::TieRegistry 0.20 ();
```

which would verify that you have at least version 0.20 but wouldn‘t call `import()`. The ***Changes*** file can be useful in figuring out which, if any, prior versions of *Win32::TieRegistry* you want to support in your script.

The code

```
    use Win32::TieRegistry;
```

imports the variable `$Registry` into your package and sets it to be a reference to a hash tied to a copy of the master Registry virtual root object with the default options. One disadvantage to this "default" usage is that Perl does not support checking the module version when you use it.

Alternately, you can specify a list of arguments on the `use` line that will be passed to the `Win32::TieRegistry−import()` method to control what items to import into your package. These arguments fall into the following broad categories:

Import a reference to a hash tied to a Registry virtual root

You can request that a scalar variable be imported (possibly) and set to be a reference to a hash tied to a Registry virtual root using any of the following types of arguments or argument pairs:

"TiedRef", '`$scalar`'
"TiedRef", '`$pack::scalar`'
"TiedRef", 'scalar'
"TiedRef", 'pack::scalar'

All of the above import a scalar named `$scalar` into your package (or the package named "pack") and then sets it.

'`$scalar`'
'`$pack::scalar`'

These are equivalent to the previous items to support a more traditional appearance to the list of exports. Note that the scalar name cannot be "RegObj" here.

"TiedRef", \`$scalar`
\`$scalar`

These versions don't import anything but set the referenced `$scalar`.

Import a hash tied to the Registry virtual root

You can request that a hash variable be imported (possibly) and tied to a Registry virtual root using any of the following types of arguments or argument pairs:

"TiedHash", '%hash'
"TiedHash", '%pack::hash'
"TiedHash", 'hash'
"TiedHash", 'pack::hash'

All of the above import a hash named `%hash` into your package (or the package named "pack") and then sets it.

'%hash'
'%pack::hash'

These are equivalent to the previous items to support a more traditional appearance to the list of exports.

"TiedHash", \%hash
\%hash

These versions don't import anything but set the referenced `%hash`.

Import a Registry virtual root object

You can request that a scalar variable be imported (possibly) and set to be a Registry virtual root object using any of the following types of arguments or argument pairs:

"ObjectRef", '`$scalar`'
"ObjectRef", '`$pack::scalar`'
"ObjectRef", 'scalar'
"ObjectRef", 'pack::scalar'

All of the above import a scalar named `$scalar` into your package (or the package named "pack") and then sets it.

'`$RegObj`'

This is equivalent to the previous items for backward compatibility.

"ObjectRef", \`$scalar`

This version doesn't import anything but sets the referenced `$scalar`.

Import constant(s) exported by *Win32API::Registry*

You can list any constants that are exported by *Win32API::Registry* to have them imported into your package. These constants have names starting with "KEY_" or "REG_" (or even "HKEY_").

You can also specify `":KEY_"`, `":REG_"`, and even `":HKEY_"` to import a whole set of constants.

See *Win32API::Registry* documentation for more information.

Options

You can list any option names that can be listed in the `SetOptions()` method call, each folowed by the value to use for that option. A Registry virtual root object is created, all of these options are set for it, then each variable to be imported/set is associated with this object.

In addition, the following special options are supported:

ExportLevel

Whether to import variables into your package or some package that uses your package. Defaults to the value of `$Exporter::ExportLevel` and has the same meaning. See the *Exporter* module for more information.

ExportTo

The name of the package to import variables and constants into. Overrides *ExportLevel*.

=head3 Specifying constants in your Perl code

This module was written with a strong emphasis on the convenience of the module user. Therefore, most places where you can specify a constant like `REG_SZ()` also allow you to specify a string containing the name of the constant, `"REG_SZ"`. This is convenient because you may not have imported that symbolic constant.

Perl also emphasizes programmer convenience so the code `REG_SZ` can be used to mean `REG_SZ()` or `"REG_SZ"` or be illegal. Note that using `&REG_SZ` (as we‘ve seen in much Win32 Perl code) is not a good idea since it passes the current `@_` to the `constant()` routine of the module which, at the least, can give you a warning under **−w**.

Although greatly a matter of style, the "safest" practice is probably to specifically list all constants in the `use Win32::TieRegistry` statement, specify `use strict` [or at least `use strict qw(subs)`], and use bare constant names when you want the numeric value. This will detect mispelled constant names at compile time.

```
use strict;
my $Registry;
use Win32::TieRegistry 0.20 (
    TiedRef => \$Registry,  Delimiter => "/",  ArrayValues => 1,
    SplitMultis => 1,  AllowLoad => 1,
    qw( REG_SZ REG_EXPAND_SZ REG_DWORD REG_BINARY REG_MULTI_SZ
        KEY_READ KEY_WRITE KEY_ALL_ACCESS ),
);
$Registry->{"LMachine/Software/FooCorp/"}= {
    "FooWriter/" => {
        "/Fonts" => [ ["Times","Courier","Lucinda"], REG_MULTI_SZ ],
        "/WindowSize" => [ pack("LL",24,80), REG_BINARY ],
        "/TaskBarIcon" => [ "0x0001", REG_DWORD ],
    },
}  or  die "Can't create Software/FooCorp/: $^E\n";
```

If you don‘t want to `use strict qw(subs)`, the second safest practice is similar to the above but use the `REG_SZ()` form for constants when possible and quoted constant names when required. Note that `qw()` is a form of quoting.

```
    use Win32::TieRegistry 0.20 qw(
        TiedRef $Registry
        Delimiter /  ArrayValues 1  SplitMultis 1  AllowLoad 1
        REG_SZ REG_EXPAND_SZ REG_DWORD REG_BINARY REG_MULTI_SZ
        KEY_READ KEY_WRITE KEY_ALL_ACCESS
    );
    $Registry->{"LMachine/Software/FooCorp/"}= {
        "FooWriter/" => {
            "/Fonts" => [ ["Times","Courier","Lucinda"], REG_MULTI_SZ() ],
            "/WindowSize" => [ pack("LL",24,80), REG_BINARY() ],
            "/TaskBarIcon" => [ "0x0001", REG_DWORD() ],
        },
    }  or  die "Can't create Software/FooCorp/: $^E\n";
```

The examples in this document mostly use quoted constant names (`"REG_SZ"`) since that works regardless of which constants you imported and whether or not you have `use strict` in your script. It is not the best choice for you to use for real scripts (vs. examples) because it is less efficient and is not supported by most other similar modules.

SUMMARY

Most things can be done most easily via tied hashes. Skip down to the the *Tied Hashes Summary* to get started quickly.

Objects Summary

Here are quick examples that document the most common functionality of all of the method functions [except for a few almost useless ones].

```
    # Just another way of saying Open():
    $key= new Win32::TieRegistry "LMachine\\Software\\",
      { Access=>KEY_READ()|KEY_WRITE(), Delimiter=>"\\" };

    # Open a Registry key:
    $subKey= $key->Open( "SubKey/SubSubKey/",
      { Access=>KEY_ALL_ACCESS, Delimiter=>"/" } );

    # Connect to a remote Registry key:
    $remKey= $Registry->Connect( "MachineName", "LMachine/",
      { Access=>KEY_READ, Delimiter=>"/" } );

    # Get value data:
    $valueString= $key->GetValue("ValueName");
    ( $valueString, $valueType )= $key->GetValue("ValueName");

    # Get list of value names:
    @valueNames= $key->ValueNames;

    # Get list of subkey names:
    @subKeyNames= $key->SubKeyNames;

    # Get combined list of value names (with leading delimiters)
    # and subkey names (with trailing delimiters):
    @memberNames= $key->MemberNames;

    # Get all information about a key:
    %keyInfo= $key->Information;
    # keys(%keyInfo)= qw( Class LastWrite SecurityLen
    #   CntSubKeys MaxSubKeyLen MaxSubClassLen
    #   CntValues MaxValNameLen MaxValDataLen );

    # Get selected information about a key:
```

```
( $class, $cntSubKeys )= $key->Information( "Class", "CntSubKeys" );

# Get and/or set delimiter:
$delim= $key->Delimiter;
$oldDelim= $key->Delimiter( $newDelim );

# Get "path" for an open key:
$path= $key->Path;
# For example, "/CUser/Control Panel/Mouse/"
# or "//HostName/LMachine/System/DISK/".

# Get name of machine where key is from:
$mach= $key->Machine;
# Will usually be "" indicating key is on local machine.

# Control different options (see main documentation for descriptions):
$oldBool= $key->ArrayValues( $newBool );
$oldBool= $key->FastDelete( $newBool );
$oldBool= $key->FixSzNulls( $newBool );
$oldBool= $key->SplitMultis( $newBool );
$oldBool= $key->DWordsToHex( $newBool );
$oldBool= $key->DualBinVals( $newBool );
$oldBool= $key->DualTypes( $newBool );
@oldBools= $key->SetOptions( ArrayValues=>1, FastDelete=>1, FixSzNulls=>0,
  Delimiter=>"/", AllowLoad=>1, AllowSave=>1 );
@oldBools= $key->GetOptions( ArrayValues, FastDelete, FixSzNulls );

# Add or set a value:
$key->SetValue( "ValueName", $valueDataString );
$key->SetValue( "ValueName", pack($format,$valueData), "REG_BINARY" );

# Add or set a key:
$key->CreateKey( "SubKeyName" );
$key->CreateKey( "SubKeyName",
  { Access=>"KEY_ALL_ACCESS", Class=>"ClassName",
    Delimiter=>"/", Volatile=>1, Backup=>1 } );

# Load an off-line Registry hive file into the on-line Registry:
$newKey= $Registry->Load( "C:/Path/To/Hive/FileName" );
$newKey= $key->Load( "C:/Path/To/Hive/FileName", "NewSubKeyName",
                  { Access=>"KEY_READ" } );
# Unload a Registry hive file loaded via the Load() method:
$newKey->UnLoad;

# (Dis)Allow yourself to load Registry hive files:
$success= $Registry->AllowLoad( $bool );

# (Dis)Allow yourself to save a Registry key to a hive file:
$success= $Registry->AllowSave( $bool );

# Save a Registry key to a new hive file:
$key->RegSaveKey( "C:/Path/To/Hive/FileName", [] );
```

=head3 Other Useful Methods

See *Win32API::Registry* for more information on these methods. These methods are provided for coding convenience and are identical to the *Win32API::Registry* functions except that these don‘t take a handle to a Registry key, instead getting the handle from the invoking object [`$key`] .

```
$key->RegGetKeySecurity( $iSecInfo, $sSecDesc, $lenSecDesc );
$key->RegLoadKey( $sSubKeyName, $sPathToFile );
```

```
$key->RegNotifyChangeKeyValue(
  $bWatchSubtree, $iNotifyFilter, $hEvent, $bAsync );
$key->RegQueryMultipleValues(
  $structValueEnts, $cntValueEnts, $Buffer, $lenBuffer );
$key->RegReplaceKey( $sSubKeyName, $sPathToNewFile, $sPathToBackupFile );
$key->RegRestoreKey( $sPathToFile, $iFlags );
$key->RegSetKeySecurity( $iSecInfo, $sSecDesc );
$key->RegUnLoadKey( $sSubKeyName );
```

Tied Hashes Summary

For fast learners, this may be the only section you need to read. Always append one delimiter to the end of each Registry key name and prepend one delimiter to the front of each Registry value name.

=head3 Opening keys

```
use Win32::TieRegistry ( Delimiter=>"/", ArrayValues=>1 );
$Registry->Delimiter("/");                  # Set delimiter to "/".
$swKey= $Registry->{"LMachine/Software/"};
$winKey= $swKey->{"Microsoft/Windows/CurrentVersion/"};
$userKey= $Registry->
  {"CUser/Software/Microsoft/Windows/CurrentVersion/"};
$remoteKey= $Registry->{"//HostName/LMachine/"};
```

=head3 Reading values

```
$progDir= $winKey->{"/ProgramFilesDir"};    # "C:\\Program Files"
$tip21= $winKey->{"Explorer/Tips//21"};     # Text of tip #21.

$winKey->ArrayValues(1);
( $devPath, $type )= $winKey->{"/DevicePath"};
# $devPath eq "%SystemRoot%\\inf"
# $type eq "REG_EXPAND_SZ"  [if you have SetDualVar.pm installed]
# $type == REG_EXPAND_SZ()  [if did C<use Win32::TieRegistry qw(:REG_)>]
```

=head3 Setting values

```
$winKey->{"Setup//SourcePath"}= "\\\\SwServer\\SwShare\\Windows";
# Simple.  Assumes data type of REG_SZ.

$winKey->{"Setup//Installation Sources"}=
  [ "D:\x00\\\\SwServer\\SwShare\\Windows\0\0", "REG_MULTI_SZ" ];
# "\x00" and "\0" used to mark ends of each string and end of list.

$winKey->{"Setup//Installation Sources"}=
  [ ["D:","\\\\SwServer\\SwShare\\Windows"], "REG_MULTI_SZ" ];
# Alternate method that is easier to read.

$userKey->{"Explorer/Tips//DisplayInitialTipWindow"}=
  [ pack("L",0), "REG_DWORD" ];
$userKey->{"Explorer/Tips//Next"}= [ pack("S",3), "REG_BINARY" ];
$userKey->{"Explorer/Tips//Show"}= [ pack("L",0), "REG_BINARY" ];
```

=head3 Adding keys

```
$swKey->{"FooCorp/"}= {
    "FooWriter/" => {
        "/Version" => "4.032",
        "Startup/" => {
            "/Title" => "Foo Writer Deluxe ][",
            "/WindowSize" => [ pack("LL",$wid,$ht), "REG_BINARY" ],
            "/TaskBarIcon" => [ "0x0001", "REG_DWORD" ],
        },
        "Compatibility/" => {
            "/AutoConvert" => "Always",
            "/Default Palette" => "Windows Colors",
        },
    },
    "/License", => "0123-9C8EF1-09-FC",
};
```

=head3 Listing all subkeys and values

```
    @members= keys( %{$swKey} );
    @subKeys= grep(  m#^/#,  keys( %{$swKey->{"Classes/batfile/"}} )  );
    # @subKeys= ( "/", "/EditFlags" );
    @valueNames= grep(  ! m#^/#,  keys( %{$swKey->{"Classes/batfile/"}} )  );
    # @valueNames= ( "DefaultIcon/", "shell/", "shellex/" );
```

=head3 Deleting values or keys with no subkeys

```
    $oldValue= delete $userKey->{"Explorer/Tips//Next"};

    $oldValues= delete $userKey->{"Explorer/Tips/"};
    # $oldValues will be reference to hash containing deleted keys values.
```

=head3 Closing keys

```
    undef $swKey;                # Explicit way to close a key.
    $winKey= "Anything else";    # Implicitly closes a key.
    exit 0;                      # Implicitly closes all keys.
```

Tie::Registry

This module was originally called *Tie::Registry*. Changing code that used *Tie::Registry* over to *Win32::TieRegistry* is trivial as the module name should only be mentioned once, in the `use` line. However, finding all of the places that used *Tie::Registry* may not be completely trivial so we have included ***Tie/Registry.pm*** which you can install to provide backward compatibility.

AUTHOR

Tye McQueen. See http://www.metronet.com/~tye/ or e−mail tye@metronet.com with bug reports.

SEE ALSO

Win32API::Registry − Provides access to `Reg*()`, `HKEY_*`, `KEY_*`, `REG_*` [required].

Win32::WinError − Defines `ERROR_*` values [optional].

SetDualVar − For returning `REG_*` values as combined string/integer values [optional].

BUGS

Perl5.004_02 has bugs that make *Win32::TieRegistry* fail in strange and subtle ways.

Using *Win32::TieRegistry* with versions of Perl prior to 5.005 can be tricky or impossible. Most notes about this have been removed from the documentation (they get rather complicated and confusing). This includes references to `$^E` perhaps not being meaningful.

Because Perl hashes are case sensitive, certain lookups are also case sensistive. In particular, the root keys ("Classes", "CUser", "LMachine", "Users", "PerfData", "CConfig", "DynData", and HKEY_*) must always be entered without changing between upper and lower case letters. Also, the special rule for matching subkey names that contain the user−selected delimiter only works if case is matched. All other key name and value name lookups should be case insensitive because the underlying `Reg*()` calls ignore case.

Information about each key is cached when using a tied hash. This cache is not flushed nor updated when changes are made, *even when the same tied hash is used* to make the changes.

Current implementations of Perl‘s "global destruction" phase can cause objects returned by `Load()` to be destroyed while keys within the hive are still open, if the objects still exist when the script starts to exit. When this happens, the automatic `UnLoad()` will report a failure and the hive will remain loaded in the Registry.

Trying to `Load()` a hive file that is located on a remote network share may silently delete all data from the hive. This is a bug in the Win32 APIs, not any Perl code or modules. This module does not try to protect you from this bug.

There is no test suite.

FUTURE DIRECTIONS

The following items are desired by the author and may appear in a future release of this module.

TieValues option

Currently described in main documentation but no yet implemented.

AutoRefresh option

Trigger use of `RegNotifyChangeKeyValue()` to keep tied hash caches up−to−date even when other programs make changes.

Error options

Allow the user to have unchecked calls (calls in a "void context") to automatically report errors via `warn` or `die`.

For complex operations, such a copying an entire subtree, provide access to detailed information about errors (and perhaps some warnings) that were encountered. Let the user control whether the complex operation continues in spite of errors.

NAME

Win32::OLE::Const − Extract constant definitions from TypeLib

SYNOPSIS

```
use Win32::OLE::Const 'Microsoft Excel';
printf "xlMarkerStyleDot = %d\n", xlMarkerStyleDot;

my $wd = Win32::OLE::Const->Load("Microsoft Word 8\\.0 Object Library");
foreach my $key (keys %$wd) {
    printf "$key = %s\n", $wd->{$key};
}
```

DESCRIPTION

This modules makes all constants from a registered OLE type library available to the Perl program. The constant definitions can be imported as functions, providing compile time name checking. Alternatively the constants can be returned in a hash reference which avoids defining lots of functions of unknown names.

Functions/Methods

use Win32::OLE::Const

The `use` statement can be used to directly import the constant names and values into the users namespace.

```
use Win32::OLE::Const (TYPELIB,MAJOR,MINOR,LANGUAGE);
```

The TYPELIB argument specifies a regular expression for searching through the registry for the type library. Note that this argument is implicitly prefixed with ^ to speed up matches in the most common cases. Use a typelib name like ".*Excel" to match anywhere within the description. TYPELIB is the only required argument.

The MAJOR and MINOR arguments specify the requested version of the type specification. If the MAJOR argument is used then only typelibs with exactly this major version number will be matched. The MINOR argument however specifies the minimum acceptable minor version. MINOR is ignored if MAJOR is undefined.

If the LANGUAGE argument is used then only typelibs with exactly this language id will be matched.

The module will select the typelib with the highest version number satisfying the request. If no language id is specified then a the default language (0) will be preferred over the others.

Note that only constants with valid Perl variable names will be exported, i.e. names matching this regexp: `/^[a−zA−Z_][a−zA−Z0−9_]*$/`.

Win32::OLE::Const−Load

The Win32::OLE::Const−Load method returns a reference to a hash of constant definitions.

```
my $const = Win32::OLE::Const->Load(TYPELIB,MAJOR,MINOR,LANGUAGE);
```

The parameters are the same as for the `use` case.

This method is generally preferrable when the typelib uses a non−english language and the constant names contain locale specific characters not allowed in Perl variable names.

Another advantage is that all available constants can now be enumerated.

The load method also accepts an OLE object as a parameter. In this case the OLE object is queried about its containing type library and no registry search is done at all. Interestingly this seems to be slower.

EXAMPLES

The first example imports all Excel constants names into the main namespace and prints the value of xlMarkerStyleDot (−4118).

```
use Win32::OLE::Const ('Microsoft Excel 8.0 Object Library');
print "xlMarkerStyleDot = %d\n", xlMarkerStyleDot;
```

The second example returns all Word constants in a hash ref.

```
use Win32::OLE::Const;
my $wd = Win32::OLE::Const->Load("Microsoft Word 8.0 Object Library");
foreach my $key (keys %$wd) {
    printf "$key = %s\n", $wd->{$key};
}
printf "wdGreen = %s\n", $wd->{wdGreen};
```

The last example uses an OLE object to specify the type library:

```
use Win32::OLE;
use Win32::OLE::Const;
my $Excel = Win32::OLE->new('Excel.Application', 'Quit');
my $xl = Win32::OLE::Const->Load($Excel);
```

AUTHORS/COPYRIGHT

This module is part of the Win32::OLE distribution.

NAME

Win32::OLE::Enum − OLE Automation Collection Objects

SYNOPSIS

```
    my $Sheets = $Excel->Workbooks(1)->Worksheets;
    my $Enum = Win32::OLE::Enum->new($Sheets);
    my @Sheets = $Enum->All;

    while (defined(my $Sheet = $Enum->Next)) { ... }
```

DESCRIPTION

This module provides an interface to OLE collection objects from Perl. It defines an enumerator object closely mirroring the functionality of the IEnumVARIANT interface.

Please note that the `Reset()` method is not available in all implementations of OLE collections (like Excel 7). In that case the Enum object is good only for a single walk through of the collection.

Functions/Methods

Win32::OLE::Enum−new(`$object`)

Creates an enumerator for `$object`, which must be a valid OLE collection object. Note that correctly implemented collection objects must support the `Count` and `Item` methods, so creating an enumerator is not always necessary.

`$Enum`−All()

Returns a list of all objects in the collection. You have to call `$Enum`−Reset() before the enumerator can be used again. The previous position in the collection is lost.

This method can also be called as a class method:

```
        my @list = Win32::OLE::Enum->All($Collection);
```

`$Enum`−Clone()

Returns a clone of the enumerator maintaining the current position within the collection (if possible). Note that the `Clone` method is often not implemented. Use `$Enum`−Clone() in an eval block to avoid dying if you are not sure that Clone is supported.

`$Enum`−Next([`$count`])

Returns the next element of the collection. In a list context the optional `$count` argument specifies the number of objects to be returned. In a scalar context only the last of at most `$count` retrieved objects is returned. The default for `$count` is 1.

`$Enum`−Reset()

Resets the enumeration sequence to the beginning. There is no guarantee that the exact same set of objects will be enumerated again (e.g. when enumerating files in a directory). The methods return value indicates the success of the operation. (Note that the `Reset()` method seems to be unimplemented in some applications like Excel 7. Use it in an eval block to avoid dying.)

`$Enum`−Skip([`$count`])

Skip the next `$count` elements of the enumeration. The default for `$count` is 1. The functions returns TRUE if at least `$count` elements could be skipped. It returns FALSE if not enough elements were left.

AUTHORS/COPYRIGHT

This module is part of the Win32::OLE distribution.

NAME

Win32::OLE::NEWS − What‘s new in Win32::OLE

This file contains a history of user visible changes to the Win32::OLE::* modules. Only new features and major bug fixes that might affect backwards compatibility are included.

Version 0.13

`nothing()` method in Win32::OLE::Variant

The `nothing()` function returns an empty VT_DISPATCH variant. It can be used to clear an object reference stored in a property

```
        use Win32::OLE::Variant qw(:DEFAULT nothing);
        # ...
        $object->{Property} = nothing;
```

This has the same effect as the Visual Basic statement

```
        Set object.Property = Nothing
```

new _NewEnum and _Unique options

There are two new options available for the Win32::OLE−Option class method: `_NewEnum` provides the elements of a collection object directly as the value of a `_NewEnum` property. The `_Unique` option guarantees that Win32::OLE will not create multiple proxy objects for the same underlying COM/OLE object.

Both options are only really useful to tree traversal programs or during debugging.

Version 0.12

Additional error handling functionality

The Warn option can now be set to a CODE reference too. For example,

```
    Win32::OLE->Option(Warn => 3);
```

could now be written as

```
    Win32::OLE->Option(Warn => \&Carp::croak);
```

This can even be used to emulate the VisualBasic `On Error Goto Label` construct:

```
    Win32::OLE->Option(Warn =>  sub {goto CheckError});
    # ... your normal OLE code here ...

  CheckError:
    # ... your error handling code here ...
```

Builtin event loop

Processing OLE events required a polling loop before, e.g.

```
    my $Quit;
    #...
    until ($Quit) {
        Win32::OLE->SpinMessageLoop;
        Win32::Sleep(100);
    }
    package BrowserEvents;
    sub OnQuit { $Quit = 1 }
```

This is inefficient and a bit odd. This version of Win32::OLE now supports a standard messageloop:

```
    Win32::OLE->MessageLoop();
    package BrowserEvents;
    sub OnQuit { Win32::OLE->QuitMessageLoop }
```

Free unused OLE libraries

Previous versions of Win32::OLE would call the `CoFreeUnusedLibraries()` API whenever an OLE object was destroyed. This made sure that OLE libraries would be unloaded as soon as they were no longer needed. Unfortunately, objects implemented in Visual Basic tend to crash during this call, as they pretend to be ready for unloading, when in fact, they aren‘t.

The unloading of object libraries is really only important for long running processes that might instantiate a huge number of **different** objects over time. Therefore this API is no longer called automatically. The functionality is now available explicitly to those who want or need it by calling a Win32::OLE class method:

```
    Win32::OLE->FreeUnusedLibraries();
```

The "Win32::OLE" article from "The Perl Journal #10"

The article is Copyright 1998 by *The Perl Journal*. http://www.tpj.com

It originally appeared in *The Perl Journal* # 10 and appears here courtesy of Jon Orwant and *The Perl Journal*. The sample code from the article is in the ***eg/tpj.pl*** file.

`VARIANT->`Put() bug fixes

The `Put()` method didn‘t work correctly for arrays of type VT_BSTR, VT_DISPATH or VT_UNKNOWN. This has been fixed.

Error message fixes

Previous versions of Win32::OLE gave a wrong argument index for some OLE error messages (the number was too large by 1). This should be fixed now.

VT_DATE and VT_ERROR return values handled differently

Method calls and property accesses returning a VT_DATE or VT_ERROR value would previously translate the value to string or integer format. This has been changed to return a Win32::OLE::Variant object. The return values will behave as before if the Win32::OLE::Variant module is being used. This module overloads the conversion of the objects to strings and numbers.

Version 0.11 (changes since 0.1008)

new DHTML typelib browser

The Win32::OLE distribution now contains a type library browser. It is written in PerlScript, generating dynamic HTML. It requires Internet Explorer 4.0 or later. You‘ll find it in ***browser/Browser.html***. It should be available in the ActivePerl HTML help under Win32::OLE::Browser.

After selecting a library, type or member you can press F1 to call up the corresponding help file at the appropriate location.

VT_DECIMAL support

The Win32::OLE::Variant module now supports VT_DECIMAL variants too. They are not "officially" allowed in OLE Automation calls, but even Microsoft‘s "ActiveX Data Objects" sometimes returns VT_DECIMAL values.

VT_DECIMAL variables are stored as 96−bit integers scaled by a variable power of 10. The power of 10 scaling factor specifies the number of digits to the right of the decimal point, and ranges from 0 to 28. With a scale of 0 (no decimal places), the largest possible value is +/−79,228,162,514,264,337,593,543,950,335. With a 28 decimal places, the largest value is +/−7.9228162514264337593543950335 and the smallest, non−zero value is +/−0.0000000000000000000000000001.

Version 0.1008

new `LetProperty()` object method

In Win32::OLE property assignment using the hash syntax is equivalent to the Visual Basic `Set` syntax (*by reference* assignment):

```
$Object->{Property} = $OtherObject;
```

corresponds to this Visual Basic statement:

```
Set Object.Property = OtherObject
```

To get the *by value* treatment of the Visual Basic `Let` statement

```
Object.Property = OtherObject
```

you have to use the `LetProperty()` object method in Perl:

```
$Object->LetProperty($Property, $OtherObject);
```

new `HRESULT()` function

The `HRESULT()` function converts an unsigned number into a signed HRESULT error value as used by OLE internally. This is necessary because Perl treats all hexadecimal constants as unsigned. To check if the last OLE function returned "Member not found" (0x80020003) you can write:

```
if (Win32::OLE->LastError == HRESULT(0x80020003)) {
    # your error recovery here
}
```

Version 0.1007 (changes since 0.1005)

OLE Event support

This version of Win32::OLE contains **ALPHA** level support for OLE events. The userinterface is still subject to change. There are ActiveX objects / controls that don‘t fire events under the current implementation.

Events are enabled for a specific object with the `Win32::OLE`−WithEvents() class method:

```
Win32::OLE->WithEvents(OBJECT, HANDLER, INTERFACE)
```

Please read further documentation in Win32::OLE.

`GetObject()` and `GetActiveObject()` now support optional DESTRUCTOR argument

It is now possible to specify a DESTRUCTOR argument to the `GetObject()` and
`GetActiveObject()` class methods. They work identical to the `new()` DESTRUCTOR argument.

Remote object instantiation via DCOM

This has actually been in Win32::OLE since 0.0608, but somehow never got documented. You can provide an array reference in place of the usual PROGID parameter to `Win32::OLE`−new():

```
OBJ = Win32::OLE->new([MACHINE, PRODID]);
```

The array must contain two elements: the name of the MACHINE and the PROGID. This will try to create the object on the remote MACHINE.

Enumerate all Win32::OLE objects

This class method returns the number Win32::OLE objects currently in existance. It will call the optional CALLBACK function for each of these objects:

```
$Count = Win32::OLE->EnumAllObjects(sub {
    my $Object = shift;
    my $Class = Win32::OLE->QueryObjectType($Object);
    printf "# Object=%s Class=%s\n", $Object, $Class;
```

```
  });
```

The `EnumAllObjects()` method is primarily a debugging tool. It can be used e.g. in an END block to check if all external connections have been properly destroyed.

The `VARIANT−`Put() method now returns the VARIANT object itself

This allows chaining of `Put()` method calls to set multiple values in an array variant:

```
  $Array->Put(0,0,$First_value)->Put(0,1,$Another_value);
```

The VARIANT−Put(ARRAYREF) form allows assignment to a complete SAFEARRAY

This allows automatic conversion from a list of lists to a SAFEARRAY. You can now write:

```
  my $Array = Variant(VT_ARRAY|VT_R8, [1,2], 2);
  $Array->Put([[1,2], [3,4]]);
```

instead of the tedious:

```
  $Array->Put(1,0,1);
  $Array->Put(1,1,2);
  $Array->Put(2,0,3);
  $Array->Put(2,1,4);
```

New Variant formatting methods

There are four new methods for formatting variant values: `Currency()`, `Date()`, `Number()` and `Time()`. For example:

```
  my $v = Variant(VT_DATE, "April 1 99");
  print $v->Date(DATE_LONGDATE), "\n";
  print $v->Date("ddd',' MMM dd yy"), "\n";
```

will print:

```
  Thursday, April 01, 1999
  Thu, Apr 01 99
```

new Win32::OLE::NLS methods: `SendSettingChange()` and `SetLocaleInfo()`

`SendSettingChange()` sends a WM_SETTINGCHANGE message to all top level windows.

`SetLocaleInfo()` allows changing elements in the user override section of the locale database. Unfortunately these changes are not automatically available to further Variant formatting; you have to call `SendSettingChange()` first.

Win32::OLE::Const now correctly treats version numbers as hex

The minor and major version numbers of type libraries have been treated as decimal. This was wrong. They are now correctly decoded as hex.

more robust global destruction of Win32::OLE objects

The final destruction of Win32::OLE objects has always been somewhat fragile. The reason for this is that Perl doesn‘t honour reference counts during global destruction but destroys objects in seemingly random order. This can lead to leaked database connections or unterminated external objects. The only solution was to make all objects lexical and hope that no object would be trapped in a closure. Alternatively all objects could be explicitly set to `undef`, which doesn‘t work very well with exception handling.

With version 0.1007 of Win32::OLE this problem should be gone: The module keeps a list of active Win32::OLE objects. It uses an END block to destroy all objects at program termination *before* the Perl‘s global destruction starts. Objects still existing at program termination are now destroyed in reverse order of creation. The effect is similar to explicitly calling `Win32::OLE−`Uninitialize() just prior to termination.

Version 0.1005 (changes since 0.1003)

Win32::OLE 0.1005 has been release with ActivePerl build 509. It is also included in the *Perl Resource Kit for Win32* Update.

optional DESTRUCTOR for `GetActiveObject()` `GetObject()` class methods

The `GetActiveObject()` and `GetObject()` class method now also support an optional DESTRUCTOR parameter just like `Win32::OLE`−new(). The DESTRUCTOR is executed when the last reference to this object goes away. It is generally considered `impolite` to stop applications that you did not start yourself.

new Variant object method: `$object`−Copy()

See *Win32::OLE::Variant/Copy([DIM])*.

new `Win32::OLE`−Option() class method

The `Option()` class method can be used to inspect and modify *Win32::OLE/Module Options*. The single argument form retrieves the value of an option:

```
my $CP = Win32::OLE->Option('CP');
```

A single call can be used to set multiple options simultaneously:

```
Win32::OLE->Option(CP => CP_ACP, Warn => 3);
```

Currently the following options exist: CP, LCID and `Warn`.

NAME

Win32::OLE::NLS − OLE National Language Support

SYNOPSIS

```
missing
```

DESCRIPTION

This module provides access to the national language support features in the ***OLENLS.DLL***.

Functions

CompareString(LCID,FLAGS,STR1,STR2)

Compare STR1 and STR2 in the LCID locale. FLAGS indicate the character traits to be used or ignored when comparing the two strings.

```
NORM_IGNORECASE         Ignore case
NORM_IGNOREKANATYPE     Ignore hiragana/katakana character difference
NORM_IGNORENONSPACE     Ignore accents, diacritics, and vowel marks
NORM_IGNORESYMBOLS      Ignore symbols
NORM_IGNOREWIDTH        Ignore character width
```

Possible return values are:

```
        Function failed
1       STR1 is less than STR2
2       STR1 is equal to STR2
3       STR1 is greater than STR2
```

Note that you can subtract 2 from the return code to get values comparable to the `cmp` operator.

LCMapString(LCID,FLAGS,STR)

LCMapString translates STR using LCID dependent translation. Flags contains a combination of the following options:

```
LCMAP_LOWERCASE         Lowercase
LCMAP_UPPERCASE         Uppercase
LCMAP_HALFWIDTH         Narrow characters
LCMAP_FULLWIDTH         Wide characters
LCMAP_HIRAGANA          Hiragana
LCMAP_KATAKANA          Katakana
LCMAP_SORTKEY           Character sort key
```

The following normalization options can be combined with `LCMAP_SORTKEY`:

```
NORM_IGNORECASE         Ignore case
NORM_IGNOREKANATYPE     Ignore hiragana/katakana character difference
NORM_IGNORENONSPACE     Ignore accents, diacritics, and vowel marks
NORM_IGNORESYMBOLS      Ignore symbols
NORM_IGNOREWIDTH        Ignore character width
```

The return value is the translated string.

GetLocaleInfo(LCID,LCTYPE)

Retrieve locale setting LCTYPE from the locale specified by LCID. Use LOCALE_NOUSEROVERRIDE | LCTYPE to always query the locale database. Otherwise user changes to `win.ini` through the windows control panel take precedence when retrieving values for the system default locale. See the documentation below for a list of valid LCTYPE values.

The return value is the contents of the requested locale setting.

GetStringType(LCID,TYPE,STR)

Retrieve type information from locale LCID about each character in STR. The requested TYPE can be one of the following 3 levels:

CT_CTYPE1	ANSI C and POSIX type information
CT_CTYPE2	Text layout type information
CT_CTYPE3	Text processing type information

The return value is a list of values, each of wich is a bitwise OR of the applicable type bits from the corresponding table below:

```
@ct = GetStringType(LOCALE_SYSTEM_DEFAULT, CT_CTYPE1, "String");
```

ANSI C and POSIX character type information:

C1_UPPER	Uppercase
C1_LOWER	Lowercase
C1_DIGIT	Decimal digits
C1_SPACE	Space characters
C1_PUNCT	Punctuation
C1_CNTRL	Control characters
C1_BLANK	Blank characters
C1_XDIGIT	Hexadecimal digits
C1_ALPHA	Any letter

Text layout type information:

C2_LEFTTORIGHT	Left to right
C2_RIGHTTOLEFT	Right to left
C2_EUROPENUMBER	European number, European digit
C2_EUROPESEPARATOR	European numeric separator
C2_EUROPETERMINATOR	European numeric terminator
C2_ARABICNUMBER	Arabic number
C2_COMMONSEPARATOR	Common numeric separator
C2_BLOCKSEPARATOR	Block separator
C2_SEGMENTSEPARATOR	Segment separator
C2_WHITESPACE	White space
C2_OTHERNEUTRAL	Other neutrals
C2_NOTAPPLICABLE	No implicit direction (e.g. ctrl codes)

Text precessing type information:

C3_NONSPACING	Nonspacing mark
C3_DIACRITIC	Diacritic nonspacing mark
C3_VOWELMARK	Vowel nonspacing mark
C3_SYMBOL	Symbol
C3_KATAKANA	Katakana character
C3_HIRAGANA	Hiragana character
C3_HALFWIDTH	Narrow character
C3_FULLWIDTH	Wide character
C3_IDEOGRAPH	Ideograph
C3_ALPHA	Any letter
C3_NOTAPPLICABLE	Not applicable

`GetSystemDefaultLangID()`

Returns the system default language identifier.

`GetSystemDefaultLCID()`

Returns the system default locale identifier.

`GetUserDefaultLangID()`

Returns the user default language identifier.

`GetUserDefaultLCID()`

Returns the user default locale identifier.

`SendSettingChange()`

Sends a WM_SETTINGCHANGE message to all top level windows.

SetLocaleInfo(LCID, LCTYPE, LCDATA)

Changes an item in the user override part of the locale setting LCID. It doesn‘t change the system default database. The following LCTYPEs are changeable:

```
LOCALE_ICALENDARTYPE    LOCALE_SDATE
LOCALE_ICURRDIGITS      LOCALE_SDECIMAL
LOCALE_ICURRENCY        LOCALE_SGROUPING
LOCALE_IDIGITS          LOCALE_SLIST
LOCALE_IFIRSTDAYOFWEEK  LOCALE_SLONGDATE
LOCALE_IFIRSTWEEKOFYEAR LOCALE_SMONDECIMALSEP
LOCALE_ILZERO           LOCALE_SMONGROUPING
LOCALE_IMEASURE         LOCALE_SMONTHOUSANDSEP
LOCALE_INEGCURR         LOCALE_SNEGATIVESIGN
LOCALE_INEGNUMBER       LOCALE_SPOSITIVESIGN
LOCALE_IPAPERSIZE       LOCALE_SSHORTDATE
LOCALE_ITIME            LOCALE_STHOUSAND
LOCALE_S1159            LOCALE_STIME
LOCALE_S2359            LOCALE_STIMEFORMAT
LOCALE_SCURRENCY        LOCALE_SYEARMONTH
```

You have to call `SendSettingChange()` to activate these changes for subsequent Win32::OLE::Variant object formatting because the OLE subsystem seems to cache locale information.

MAKELANGID(LANG,SUBLANG)

Creates a lnguage identifier from a primary language and a sublanguage.

PRIMARYLANGID(LANGID)

Retrieves the primary language from a language identifier.

SUBLANGID(LANGID)

Retrieves the sublanguage from a language identifier.

MAKELCID(LANGID)

Creates a locale identifies from a language identifier.

LANGIDFROMLCID(LCID)

Retrieves a language identifier from a locale identifier.

Locale Types

LOCALE_ILANGUAGE

The language identifier (in hex).

LOCALE_SLANGUAGE

The localized name of the language.

LOCALE_SENGLANGUAGE

The ISO Standard 639 English name of the language.

LOCALE_SABBREVLANGNAME

The three−letter abbreviated name of the language. The first two letters are from the ISO Standard 639 language name abbreviation. The third letter indicates the sublanguage type.

LOCALE_SNATIVELANGNAME

The native name of the language.

LOCALE_ICOUNTRY

The country code, which is based on international phone codes.

LOCALE_SCOUNTRY

The localized name of the country.

LOCALE_SENGCOUNTRY

The English name of the country.

LOCALE_SABBREVCTRYNAME

The ISO Standard 3166 abbreviated name of the country.

LOCALE_SNATIVECTRYNAME

The native name of the country.

LOCALE_IDEFAULTLANGUAGE

Language identifier for the principal language spoken in this locale.

LOCALE_IDEFAULTCOUNTRY

Country code for the principal country in this locale.

LOCALE_IDEFAULTANSICODEPAGE

The ANSI code page associated with this locale. Format: 4 Unicode decimal digits plus a Unicode null terminator.

XXX This should be translated by GetLocaleInfo. XXX

LOCALE_IDEFAULTCODEPAGE

The OEM code page associated with the country.

LOCALE_SLIST

Characters used to separate list items (often a comma).

LOCALE_IMEASURE

Default measurement system:

```
                metric system (S.I.)
        1       U.S. system
```

LOCALE_SDECIMAL

Characters used for the decimal separator (often a dot).

LOCALE_STHOUSAND

Characters used as the separator between groups of digits left of the decimal.

LOCALE_SGROUPING

Sizes for each group of digits to the left of the decimal. An explicit size is required for each group. Sizes are separated by semicolons. If the last value is 0, the preceding value is repeated. To group thousands, specify 3;0.

LOCALE_IDIGITS

The number of fractional digits.

LOCALE_ILZERO

Whether to use leading zeros in decimal fields. A setting of 0 means use no leading zeros; 1 means use leading zeros.

LOCALE_SNATIVEDIGITS

The ten characters that are the native equivalent of the ASCII 0−9.

LOCALE_INEGNUMBER

Negative number mode.

0	(1.1)
1	-1.1
2	-1.1
3	1.1
4	1.1

LOCALE_SCURRENCY

The string used as the local monetary symbol.

LOCALE_SINTLSYMBOL

Three characters of the International monetary symbol specified in ISO 4217, Codes for the Representation of Currencies and Funds, followed by the character separating this string from the amount.

LOCALE_SMONDECIMALSEP

Characters used for the monetary decimal separators.

LOCALE_SMONTHOUSANDSEP

Characters used as monetary separator between groups of digits left of the decimal.

LOCALE_SMONGROUPING

Sizes for each group of monetary digits to the left of the decimal. An explicit size is needed for each group. Sizes are separated by semicolons. If the last value is 0, the preceding value is repeated. To group thousands, specify 3;0.

LOCALE_ICURRDIGITS

Number of fractional digits for the local monetary format.

LOCALE_IINTLCURRDIGITS

Number of fractional digits for the international monetary format.

LOCALE_ICURRENCY

Positive currency mode.

	Prefix, no separation.
1	Suffix, no separation.
2	Prefix, 1-character separation.
3	Suffix, 1-character separation.

LOCALE_INEGCURR

Negative currency mode.

	($1.1)
1	-$1.1
2	$-1.1
3	$1.1-

```
4       $(1.1$)
5       -1.1$
6       1.1-$
7       1.1$-
8       -1.1 $ (space before $)
9       -$ 1.1 (space after $)
10      1.1 $- (space before $)
```

LOCALE_ICALENDARTYPE

The type of calendar currently in use.

```
1       Gregorian (as in U.S.)
2       Gregorian (always English strings)
3       Era: Year of the Emperor (Japan)
4       Era: Year of the Republic of China
5       Tangun Era (Korea)
```

LOCALE_IOPTIONALCALENDAR

The additional calendar types available for this LCID. Can be a null−separated list of all valid optional calendars. Value is 0 for "None available" or any of the LOCALE_ICALENDARTYPE settings.

XXX null separated list should be translated by GetLocaleInfo XXX

LOCALE_SDATE

Characters used for the date separator.

LOCALE_STIME

Characters used for the time separator.

LOCALE_STIMEFORMAT

Time−formatting string.

LOCALE_SSHORTDATE

Short Date_Time formatting strings for this locale.

LOCALE_SLONGDATE

Long Date_Time formatting strings for this locale.

LOCALE_IDATE

Short Date format−ordering specifier.

```
        Month - Day - Year
1       Day - Month - Year
2       Year - Month - Day
```

LOCALE_ILDATE

Long Date format ordering specifier. Value can be any of the valid LOCALE_IDATE settings.

LOCALE_ITIME

Time format specifier.

```
        AM/PM 12-hour format.
1       24-hour format.
```

LOCALE_ITIMEMARKPOSN

Whether the time marker string (AM|PM) precedes or follows the time string.
0 Suffix (9:15 AM).
1 Prefix (AM 9:15).

LOCALE_ICENTURY

Whether to use full 4−digit century.

```
                 Two digit.
        1        Full century.
```

LOCALE_ITLZERO

Whether to use leading zeros in time fields.

```
                 No leading zeros.
        1        Leading zeros for hours.
```

LOCALE_IDAYLZERO

Whether to use leading zeros in day fields. Values as for LOCALE_ITLZERO.

LOCALE_IMONLZERO

Whether to use leading zeros in month fields. Values as for LOCALE_ITLZERO.

LOCALE_S1159

String for the AM designator.

LOCALE_S2359

String for the PM designator.

LOCALE_IFIRSTWEEKOFYEAR

Specifies which week of the year is considered first.

```
                 Week containing 1/1 is the first week of the year.
        1        First full week following 1/1is the first week of the year.
        2        First week with at least 4 days is the first week of the year
```

LOCALE_IFIRSTDAYOFWEEK

Specifies the day considered first in the week. Value "0" means SDAYNAME1 and value "6" means SDAYNAME7.

LOCALE_SDAYNAME1 .. LOCALE_SDAYNAME7

Long name for Monday .. Sunday.

LOCALE_SABBREVDAYNAME1 .. LOCALE_SABBREVDAYNAME7

Abbreviated name for Monday .. Sunday.

LOCALE_SMONTHNAME1 .. LOCALE_SMONTHNAME12

Long name for January .. December.

LOCALE_SMONTHNAME13

Native name for 13th month, if it exists.

LOCALE_SABBREVMONTHNAME1 .. LOCALE_SABBREVMONTHNAME12

Abbreviated name for January .. December.

LOCALE_SABBREVMONTHNAME13

Native abbreviated name for 13th month, if it exists.

LOCALE_SPOSITIVESIGN

String value for the positive sign.

LOCALE_SNEGATIVESIGN

String value for the negative sign.

LOCALE_IPOSSIGNPOSN

Formatting index for positive values.

```
0 Parentheses surround the amount and the monetary symbol.
1 The sign string precedes the amount and the monetary symbol.
2 The sign string precedes the amount and the monetary symbol.
3 The sign string precedes the amount and the monetary symbol.
4 The sign string precedes the amount and the monetary symbol.
```

LOCALE_INEGSIGNPOSN

Formatting index for negative values. Values as for LOCALE_IPOSSIGNPOSN.

LOCALE_IPOSSYMPRECEDES

If the monetary symbol precedes, 1. If it succeeds a positive amount, 0.

LOCALE_IPOSSEPBYSPACE

If the monetary symbol is separated by a space from a positive amount, 1. Otherwise, 0.

LOCALE_INEGSYMPRECEDES

If the monetary symbol precedes, 1. If it succeeds a negative amount, 0.

LOCALE_INEGSEPBYSPACE

If the monetary symbol is separated by a space from a negative amount, 1. Otherwise, 0.

AUTHORS/COPYRIGHT

This module is part of the Win32::OLE distribution.

Win32::OLE

Suppose you‘re composing a document with Microsoft Word. You want to include an Excel spreadsheet. You could save the spreadsheet in some image format that Word can understand, and import it into your document. But if the spreadsheet changes, your document will be out of date.

Microsoft‘s OLE (Object Linking and Embedding, pronounced "olay") lets one program use objects from another. In the above scenario, the spreadsheet is the object. As long as Excel makes that spreadsheet available as an OLE object, and Word knows to treat it like one, your document will always be current.

You can control OLE objects from Perl with the Win32::OLE module, and that‘s what this article is about. First, I‘ll show you how to "think OLE," which mostly involves a lot of jargon. Next, I‘ll show you the mechanics involved in using Win32::OLE. Then we‘ll go through a Perl program that uses OLE to manipulate Microsoft Excel, Microsoft Access, and Lotus Notes. Finally, I‘ll talk about Variants, an internal OLE data type.

THE OLE MINDSET

When an application makes an OLE object available for other applications to use, that‘s called OLE *automation*. The program using the object is called the *controller*, and the application providing the object is called the *server*. OLE automation is guided by the OLE Component Object Model (COM) which specifies how those objects must behave if they are to be used by other processes and machines.

There are two different types of OLE automation servers. *In−process* servers are implemented as dynamic link libraries (DLLs) and run in the same process space as the controller. *Out−of−process* servers are more interesting; they are standalone executables that exist as separate processes − possibly on a different computer.

The Win32::OLE module lets your Perl program act as an OLE controller. It allows Perl to be used in place of other languages like Visual Basic or Java to control OLE objects. This makes all OLE automation servers immediately available as Perl modules.

Don‘t confuse ActiveState OLE with Win32::OLE. ActiveState OLE is completely different, although future builds of ActiveState Perl (500 and up) will work with Win32::OLE.

Objects can expose OLE methods, properties, and events to the outside world. Methods are functions that the controller can call to make the object do something; properties describe the state of the object; and events let the controller know about external events affecting the object, such as the user clicking on a button. Since events involve asynchronous communication with their objects, they require either threads or an event loop. They are not yet supported by the Win32::OLE module, and for the same reason ActiveX controls (OCXs) are currently unsupported as well.

WORKING WITH WIN32::OLE

The Win32::OLE module doesn‘t let your Perl program create OLE objects. What it does do is let your Perl program act like a remote control for other applications−it lets your program be an OLE controller. You can take an OLE object from another application (Access, Notes, Excel, or anything else that speaks OLE) and invoke its methods or manipulate its properties.

THE FIRST STEP: CREATING AN OLE SERVER OBJECT

First, we need to create a Perl object to represent the OLE server. This is a weird idea; what it amounts to is that if we want to control OLE objects produced by, say, Excel, we have to create a Perl object that represents Excel. So even though our program is an OLE controller, it‘ll contain objects that represent OLE servers.

You can create a new OLE *server object* with `< Win32::OLE−new` . This takes a program ID (a human readable string like `'Speech.VoiceText'`) and returns a server object:

```
  my $server = Win32::OLE->new('Excel.Application', 'Quit');
```

Some server objects (particularly those for Microsoft Office applications) don‘t automatically terminate when your program no longer needs them. They need some kind of Quit method, and that‘s just what our second argument is. It can be either a code reference or a method name to be invoked when the object is destroyed. This lets you ensure that objects will be properly cleaned up even when the Perl program dies abnormally.

To access a server object on a different computer, replace the first argument with a reference to a list of the server name and program ID:

```
my $server = Win32::OLE->new(['foo.bar.com',
                              'Excel.Application']);
```

(To get the requisite permissions, you‘ll need to configure your security settings with ***DCOMCNFG.EXE***.)

You can also directly attach your program to an already running OLE server:

```
my $server = Win32::OLE->GetActiveObject('Excel.Application');
```

This fails (returning `undef`) if no server exists, or if the server refuses the connection for some reason. It is also possible to use a persistent object moniker (usually a filename) to start the associated server and load the object into memory:

```
my $doc = Win32::OLE->GetObject("MyDocument.Doc");
```

METHOD CALLS

Once you‘ve created one of these server objects, you need to call its methods to make the OLE objects sing and dance. OLE methods are invoked just like normal Perl object methods:

```
$server->Foo(@Arguments);
```

This is a Perl method call − but it also triggers an OLE method call in the object. After your program executes this statement, the `$server` object will execute its `Foo()` method. The available methods are typically documented in the application‘s *object model*.

Parameters. By default, all parameters are positional (e.g. `foo($first, $second, $third)`) rather than named (e.g. `<  foo(-name  =` "Yogi", −title = "Coach")). The required parameters come first, followed by the optional parameters; if you need to provide a dummy value for an optional parameter, use undef.

Positional parameters get cumbersome if a method takes a lot of them. You can use named arguments instead if you go to a little extra trouble − when the last argument is a reference to a hash, the key/value pairs of the hash are treated as named parameters:

```
$server->Foo($Pos1, $Pos2, {Name1 => $Value1,
                            Name2 => $Value2});
```

Foreign Languages and Default Methods. Sometimes OLE servers use method and property names that are specific to a non−English locale. That means they might have non−ASCII characters, which aren‘t allowed in Perl variable names. In German, you might see `Öffnen` used instead of `Open`. In these cases, you can use the `Invoke()` method:

```
$server->Invoke('Öffnen', @Arguments);
```

This is necessary because `<  $Server−Öffnen(@Arguments)` is a syntax error in current versions of Perl.

PROPERTIES

As I said earlier, objects can expose three things to the outside world: methods, properties, and events. We‘ve covered methods, and Win32::OLE can‘t handle events. That leaves properties. But as it turns out, properties and events are largely interchangeable. Most methods have corresponding properties, and vice versa.

An object‘s properties can be accessed with a hash reference:

```
$server->{Bar} = $value;
$value = $server->{Bar};
```

This example sets and queries the value of the property named `Bar`. You could also have called the object‘s `Bar()` method to achieve the same effect:

```
  $value = $server->Bar;
```

However, you can‘t write the first line as < `$server-Bar` = `$value` , because you can‘t assign to the return value of a method call. In Visual Basic, OLE automation distinguishes between assigning the name of an object and assigning its value:

```
  Set Object = OtherObject
```

```
  Let Value = Object
```

The `Set` statement shown here makes `Object` refer to the same object as `OtherObject`. The `Let` statement copies the value instead. (The value of an OLE object is what you get when you call the object‘s default method.

In Perl, saying < `$server1 = $server2` always creates another reference, just like the `Set` in Visual Basic. If you want to assign the value instead, use the `valof()` function:

```
  my $value = valof $server;
```

This is equivalent to

```
  my $value = $server->Invoke('');
```

SAMPLE APPLICATION

Let‘s look at how all of this might be used. In Listing: 1 you‘ll see ***T−Bond.pl***, a program that uses Win32::OLE for an almost−real world application.

The developer of this application, Mary Lynch, is a financial futures broker. Every afternoon, she connects to the Chicago Board of Trade (CBoT) web site at http://www.cbot.com and collects the time and sales information for U.S. T−bond futures. She wants her program to create a chart that depicts the data in 15−minute intervals, and then she wants to record the data in a database for later analysis. Then she wants her program to send mail to her clients.

Mary‘s program will use Microsoft Access as a database, Microsoft Excel to produce the chart, and Lotus Notes to send the mail. It will all be controlled from a single Perl program using OLE automation. In this section, we‘ll go through T−Bond. pl step by step so you can see how Win32::OLE lets you control these applications.

DOWNLOADING A WEB PAGE WITH LWP

However, Mary first needs to amass the raw T−bond data by having her Perl program automatically download and parse a web page. That‘s the perfect job for LWP, the libwww−perl bundle available on the CPAN. LWP has nothing to do with OLE. But this is a real−world application, and it‘s just what Mary needs to download her data from the Chicago Board of Trade.

```
  use LWP::Simple;
  my $URL = 'http://www.cbot.com/mplex/quotes/tsfut';
  my $text = get("$URL/tsf$Contract.htm");
```

She could also have used the Win32::Internet module:

```
  use Win32::Internet;
  my $URL = 'http://www.cbot.com/mplex/quotes/tsfut';
  my $text = $Win32::Internet->new->FetchURL("$URL/tsf$Contract.htm");
```

Mary wants to condense the ticker data into 15 minute bars. She‘s interested only in lines that look like this:

```
  03/12/1998 US 98Mar 12116 15:28:34 Open
```

A regular expression can be used to determine whether a line looks like this. If it does, the regex can split it up into individual fields. The price quoted above, `12116`, really means 121 16/32, and needs to be converted to 121.5. The data is then condensed into 15 minute intervals and only the first, last, highest, and lowest price during each interval are kept. The time series is stored in the array `@Bars`. Each entry in `@Bars` is a reference to a list of 5 elements: Time, Open, High, Low, and Close.

```
foreach (split "\n", $text) {
    # 03/12/1998 US 98Mar 12116 15:28:34 Open
    my ($Date,$Price,$Hour,$Min,$Sec,$Ind) =
         m|^\s*(\d+/\d+/\d+) # " 03/12/1998"
            \s+US\s+\S+\s+(\d+) # " US 98Mar 12116"
            \s+(\d+):(\d+):(\d+) # " 12:42:40"
            \s*(.*)$|x; # " Ask"
    next unless defined $Date;
    $Day = $Date;

    # Convert from fractional to decimal format
    $Price = int($Price/100) + ($Price%100)/32;

    # Round up time to next multiple of 15 minutes
    my $NewTime = int(($Sec+$Min*60+$Hour*3600)/900+1)*900;
    unless (defined $Time && $NewTime == $Time) {
        push @Bars, [$hhmm, $Open, $High, $Low, $Close]
                                          if defined $Time;
        $Open = $High = $Low = $Close = undef;
        $Time = $NewTime;
        my $Hour = int($Time/3600);
        $hhmm = sprintf "%02d:%02d", $Hour, $Time/60-$Hour*60;
    }

    # Update 15 minute bar values
    $Close = $Price;
    $Open = $Price unless defined $Open;
    $High = $Price unless defined $High && $High > $Price;
    $Low = $Price unless defined $Low && $Low > $Price;
}

die "No data found" unless defined $Time;
push @Bars, [$hhmm, $Open, $High, $Low, $Close];
```

MICROSOFT ACCESS

Now that Mary has her T−bond quotes, she‘s ready to use Win32::OLE to store them into a Microsoft Access database. This has the advantage that she can copy the database to her lap−top and work with it on her long New York commute. She‘s able to create an Access database as follows:

```
use Win32::ODBC;
use Win32::OLE;

# Include the constants for the Microsoft Access
# "Data Access Object".

use Win32::OLE::Const 'Microsoft DAO';

my $DSN      = 'T-Bonds';
my $Driver   = 'Microsoft Access Driver (*.mdb)';
my $Desc     = 'US T-Bond Quotes';
my $Dir      = 'i:\tmp\tpj';
my $File     = 'T-Bonds.mdb';
my $Fullname = "$Dir\\$File";

# Remove old database and dataset name
unlink $Fullname if -f $Fullname;
Win32::ODBC::ConfigDSN(ODBC_REMOVE_DSN, $Driver, "DSN=$DSN")
```

```
                           if Win32::ODBC::DataSources($DSN);

# Create new database
my $Access = Win32::OLE->new('Access.Application', 'Quit');
my $Workspace = $Access->DBEngine->CreateWorkspace('', 'Admin', '');
my $Database = $Workspace->CreateDatabase($Fullname, dbLangGeneral);

# Add new database name
Win32::ODBC::ConfigDSN(ODBC_ADD_DSN, $Driver,
        "DSN=$DSN", "Description=$Desc", "DBQ=$Fullname",
        "DEFAULTDIR=$Dir", "UID=", "PWD=");
```

This uses Win32::ODBC (described in TPJ #9) to remove and create ***T−Bonds.mdb***. This lets Mary use the same script on her workstation and on her laptop even when the database is stored in different locations on each. The program also uses Win32::OLE to make Microsoft Access create an empty database.

Every OLE server has some constants that your Perl program will need to use, made accessible by the Win32::OLE::Const module. For instance, to grab the Excel constants, say `use Win32::OLE::Const ‘Microsoft Excel'`.

In the above example, we imported the Data Access Object con−stants just so we could use `dbLangGeneral`.

MICROSOFT EXCEL

Now Mary uses Win32::OLE a second time, to have Microsoft Excel create the chart shown below.

```
Figure 1: T-Bond data generated by MicroSoft Excel via Win32::OLE

# Start Excel and create new workbook with a single sheet
use Win32::OLE qw(in valof with);
use Win32::OLE::Const 'Microsoft Excel';
use Win32::OLE::NLS qw(:DEFAULT :LANG :SUBLANG);

my $lgid = MAKELANGID(LANG_ENGLISH, SUBLANG_DEFAULT);
$Win32::OLE::LCID = MAKELCID($lgid);

$Win32::OLE::Warn = 3;
```

Here, Mary sets the locale to American English, which lets her do things like use American date formats (e.g. `"12−30−98"` rather than `"30−12−98"`) in her program. It will continue to work even when she‘s visiting one of her international customers and has to run this program on their computers.

The value of `$Win32::OLE::Warn` determines what happens when an OLE error occurs. If it‘s 0, the error is ignored. If it‘s 2, or if it‘s 1 and the script is running under −w, the Win32::OLE module invokes `Carp::carp()`. If `$Win32::OLE::Warn` is set to 3, `Carp::croak()` is invoked and the program dies immediately.

Now the data can be put into an Excel spreadsheet to produce the chart. The following section of the program launches Excel and creates a new workbook with a single worksheet. It puts the column titles (‘Time‘, ‘Open‘, ‘High‘, ‘Low‘, and ‘Close’) in a bold font on the first row of the sheet. The first column displays the timestamp in *hh:mm* format; the next four display prices.

```
my $Excel = Win32::OLE->new('Excel.Application', 'Quit');
$Excel->{SheetsInNewWorkbook} = 1;
my $Book = $Excel->Workbooks->Add;
my $Sheet = $Book->Worksheets(1);
$Sheet->{Name} = 'Candle';

# Insert column titles
my $Range = $Sheet->Range("A1:E1");
$Range->{Value} = [qw(Time Open High Low Close)];
$Range->Font->{Bold} = 1;
```

```
$Sheet->Columns("A:A")->{NumberFormat} = "h:mm";
# Open/High/Low/Close to be displayed in 32nds
$Sheet->Columns("B:E")->{NumberFormat} = "# ?/32";

# Add 15 minute data to spreadsheet
print "Add data\n";
$Range = $Sheet->Range(sprintf "A2:E%d", 2+$#Bars);
$Range->{Value} = \@Bars;
```

The last statement shows how to pass arrays to OLE objects. The Win32::OLE module automatically translates each array reference to a `SAFEARRAY`, the internal OLE array data type. This translation first determines the maximum nesting level used by the Perl array, and then creates a `SAFEARRAY` of the same dimension. The `@Bars` array already contains the data in the correct form for the spreadsheet:

```
([Time1, Open1, High1, Low1, Close1],
...
[TimeN, OpenN, HighN, LowN, CloseN])
```

Now the table in the spreadsheet can be used to create a candle stick chart from our bars. Excel automatically chooses the time axis labels if they are selected before the chart is created:

```
# Create candle stick chart as new object on worksheet
$Sheet->Range("A:E")->Select;

my $Chart = $Book->Charts->Add;
$Chart->{ChartType} = xlStockOHLC;
$Chart->Location(xlLocationAsObject, $Sheet->{Name});
# Excel bug: the old $Chart is now invalid!
$Chart = $Excel->ActiveChart;
```

We can change the type of the chart from a separate sheet to a chart object on the spreadsheet page with the `< $Chart-Location` method. (This invalidates the chart object handle, which might be considered a bug in Excel.) Fortunately, this new chart is still the ‘active’ chart, so an object handle to it can be reclaimed simply by asking Excel.

At this point, our chart still needs a title, the legend is meaningless, and the axis has decimals instead of fractions. We can fix those with the following code:

```
# Add title, remove legend
with($Chart, HasLegend => 0, HasTitle => 1);
$Chart->ChartTitle->Characters->{Text} = "US T-Bond";

# Set up daily statistics
$Open  = $Bars[0][1];
$High  = $Sheet->Evaluate("MAX(C:C)");
$Low   = $Sheet->Evaluate("MIN(D:D)");
$Close = $Bars[$#Bars][4];
```

The `with()` function partially mimics the Visual Basic With statement, but allows only property assignments. It‘s a convenient shortcut for this:

```
{ # open new scope
  my $Axis = $Chart->Axes(xlValue);
  $Axis->{HasMajorGridlines} = 1;
  $Axis->{HasMinorGridlines} = 1;
  # etc ...
}
```

The `$High` and `$Low` for the day are needed to determine the minimum and maximum scaling levels. MIN and MAX are spreadsheet functions, and aren‘t automatically available as methods. However, Excel provides an `Evaluate()` method to calculate arbitrary spreadsheet functions, so we can use that.

We want the chart to show major gridlines at every fourth tick and minor gridlines at every second tick. The minimum and maximum are chosen to be whatever multiples of 1/16 we need to do that.

```
# Change tickmark spacing from decimal to fractional
with($Chart->Axes(xlValue),
    HasMajorGridlines => 1,
    HasMinorGridlines => 1,
    MajorUnit => 1/8,
    MinorUnit => 1/16,
    MinimumScale => int($Low*16)/16,
    MaximumScale => int($High*16+1)/16
);

# Fat candles with only 5% gaps
$Chart->ChartGroups(1)->{GapWidth} = 5;

sub RGB { $_[0] | ($_[1] >> 8) | ($_[2] >> 16) }

# White background with a solid border

$Chart->PlotArea->Border->{LineStyle} = xlContinuous;
$Chart->PlotArea->Border->{Color} = RGB(0,0,0);
$Chart->PlotArea->Interior->{Color} = RGB(255,255,255);

# Add 1 hour moving average of the Close series
my $MovAvg = $Chart->SeriesCollection(4)->Trendlines
      ->Add({Type => xlMovingAvg, Period => 4});
$MovAvg->Border->{Color} = RGB(255,0,0);
```

Now the finished workbook can be saved to disk as ***i:\tmp\tpj\data.xls***. That file most likely still exists from when the program ran yesterday, so we‘ll remove it. (Otherwise, Excel would pop up a dialog with a warning, because the `SaveAs()` method doesn‘t like to overwrite files.)

```
# Save workbook to file my $Filename = 'i:\tmp\tpj\data.xls';
unlink $Filename if -f $Filename;
$Book->SaveAs($Filename);
$Book->Close;
```

ACTIVEX DATA OBJECTS

Mary stores the daily prices in her T−bonds database, keeping the data for the different contracts in separate tables. After creating an ADO (ActiveX Data Object) connection to the database, she tries to connect a record set to the table for the current contract. If this fails, she assumes that the table doesn‘t exists yet and tries to create it:

```
use Win32::OLE::Const 'Microsoft ActiveX Data Objects';

my $Connection = Win32::OLE->new('ADODB.Connection');
my $Recordset = Win32::OLE->new('ADODB.Recordset');
$Connection->Open('T-Bonds');

# Open a record set for the table of this contract
{
  local $Win32::OLE::Warn = 0;
  $Recordset->Open($Contract, $Connection, adOpenKeyset,
                       adLockOptimistic, adCmdTable);
}

# Create table and index if it doesn't exist yet
if (Win32::OLE->LastError) {
    $Connection->Execute(>>"SQL");
      CREATE TABLE $Contract
```

```
          (
            Day DATETIME,
            Open DOUBLE, High DOUBLE, Low DOUBLE, Close DOUBLE
          )
  SQL
      $Connection->Execute(>>"SQL");
          CREATE INDEX $Contract
          ON $Contract (Day) WITH PRIMARY
  SQL
      $Recordset->Open($Contract, $Connection, adOpenKeyset,
                                  adLockOptimistic, adCmdTable);
  }
```

$Win32::OLE::Warn is temporarily set to zero, so that if $Recordset−Open fails, the failure will be recorded silently without terminating the program. Win32::OLE−LastError shows whether the Open failed or not. LastError returns the OLE error code in a numeric context and the OLE error message in a string context, just like Perl‘s $! variable.

Now Mary can add today‘s data:

```
  # Add new record to table
  use Win32::OLE::Variant;
  $Win32::OLE::Variant::LCID = $Win32::OLE::LCID;

  my $Fields = [qw(Day Open High Low Close)];
  my $Values = [Variant(VT_DATE, $Day),
                $Open, $High, $Low, $Close];
```

Mary uses the Win32::OLE::Variant module to store $Day as a date instead of a mere string. She wants to make sure that it‘s stored as an American−style date, so in the third line shown here she sets the locale ID of the Win32::OLE::Variant module to match the Win32::OLE module. ($Win32::OLE::LCID had been set earlier to English, since that‘s what the Chicago Board of Trade uses.)

```
  {
      local $Win32::OLE::Warn = 0;
      $Recordset->AddNew($Fields, $Values);
  }

  # Replace existing record
  if (Win32::OLE->LastError) {
      $Recordset->CancelUpdate;
      $Recordset->Close;
      $Recordset->Open(>>"SQL",
                       $Connection, adOpenDynamic);
          SELECT * FROM $Contract
          WHERE Day = #$Day#
  SQL
      $Recordset->Update($Fields, $Values);
  }

  $Recordset->Close;
  $Connection->Close;
```

The program expects to be able to add a new record to the table. It fails if a record for this date already exists, because the Day field is the primary index and therefore must be unique. If an error occurs, the update operation started by AddNew() must first be cancelled with < $Recordset−CancelUpdate ; otherwise the record set won‘t close.

LOTUS NOTES

Now Mary can use Lotus Notes to mail updates to all her customers interested in the T−bond data. (Lotus Notes doesn‘t provide its constants in the OLE type library, so Mary had to determine them by playing around with LotusScript.) The actual task is quite simple: A Notes session must be started, the mail database must be opened and the mail message must be created. The body of the message is created as a rich text field, which lets her mix formatted text with object attachments.

In her program, Mary extracts the email addresses from her customer database and sends separate message to each. Here, we‘ve simplified it somewhat.

```
sub EMBED_ATTACHMENT {1454;}     # from LotusScript

my $Notes = Win32::OLE->new('Notes.NotesSession');
my $Database = $Notes->GetDatabase('', '');
$Database->OpenMail;
my $Document = $Database->CreateDocument;

$Document->{Form} = 'Memo';
$Document->{SendTo} = ['Jon Orwant >orwant@tpj.com>',
                       'Jan Dubois >jan.dubois@ibm.net>'];
$Document->{Subject} = "US T-Bonds Chart for $Day";

my $Body = $Document->CreateRichtextItem('Body');
$Body->AppendText(>>"EOT");
I\'ve attached the latest US T-Bond data and chart for $Day.
The daily statistics were:

\tOpen\t$Open
\tHigh\t$High
\tLow\t$Low
\tClose\t$Close

Kind regards,

Mary
EOT

$Body->EmbedObject(EMBED_ATTACHMENT, '', $Filename);

$Document->Send(0);
```

VARIANTS

In this final section, I‘ll talk about Variants, which are the data types that you use to talk to OLE objects. We talked about this line earlier:

```
my $Values = [Variant(VT_DATE, $Day),
              $Open, $High, $Low, $Close];
```

Here, the `Variant()` function creates a Variant object, of type `VT_DATE` and with the value `$Day`. Variants are similar in many ways to Perl scalars. Arguments to OLE methods are transparently converted from their internal Perl representation to Variants and back again by the Win32::OLE module.

OLE automation uses a generic `VARIANT` data type to pass parameters. This data type contains type information in addition to the actual data value. Only the following data types are valid for OLE automation:

```
B<Data Type     Meaning>
VT_EMPTY       Not specified
VT_NULL        Null
VT_I2          2 byte signed integer
VT_I4          4 byte signed integer
VT_R4          4 byte real
```

`VT_R8`	`8 byte real`
`VT_CY`	`Currency`
`VT_DATE`	`Date`
`VT_BSTR`	`Unicode string`
`VT_DISPATCH`	`OLE automation interface`
`VT_ERROR`	`Error`
`VT_BOOL`	`Boolean`
`VT_VARIANT`	`(only valid with VT_BYREF)`
`VT_UNKNOWN`	`Generic COM interface`
`VT_UI1`	`Unsigned character`

The following two flags can also be used:

`VT_ARRAY`	`Array of values`
`VT_BYREF`	`Pass by reference (instead of by value)`

The Perl to Variant transformation. The following conversions are performed automatically whenever a Perl value must be translated into a Variant:

`Perl value`	`Variant`
`Integer values`	`VT_I4`
`Real values`	`VT_R8`
`Strings`	`VT_BSTR`
`undef`	`VT_ERROR (DISP_E_PARAMNOTFOUND)`
`Array reference`	`VT_VARIANT | VT_ARRAY`
`Win32::OLE object`	`VT_DISPATCH`
`Win32::OLE::Variant object`	`Type of the Variant object`

What if your Perl value is a list of lists? Those can be irregularly shaped in Perl; that is, the subsidiary lists needn‘t have the same number of elements. In this case, the structure will be converted to a "rectangular" `SAFEARRAY` of Variants, with unused slots set to `VT_EMPTY`. Consider this Perl 2−D array:

```
[ ["Perl" ],             # one element
  [1, 3.1215, undef]     # three elements
]
```

This will be translated to a 2 by 3 `SAFEARRAY` that looks like this:

```
VT_BSTR("Perl") VT_EMPTY      VT_EMPTY
VT_I4(1) VT_R8(3.1415)        VT_ERROR(DISP_E_PARAMNOTFOUND)
```

The Variant To Perl Transformation. Automatic conversion from Variants to Perl values happens as follows:

`Variant`	`Perl value`
`VT_BOOL, VT_ERROR`	`Integer`
`VT_UI1, VT_I2, VT_I4`	`Integer`
`VT_R4, VT_R8`	`Float value`
`VT_BSTR`	`String`
`VT_DISPATCH`	`Win32::OLE object`

The Win32::OLE::Variant module. This module provides access to the Variant data type, which gives you more control over how these arguments to OLE methods are encoded. (This is rarely necessary if you have a good grasp of the default conversion rules.) A Variant object can be created with the < `Win32::OLE::Variant−new` method or the equivalent `Variant()` function:

```
use Win32::OLE::Variant;
my $var1 = Win32::OLE::Variant->new(VT_DATE, 'Jan 1,1970');
my $var2 = Variant(VT_BSTR, 'This is an Unicode string');
```

Several methods let you inspect and manipulate Variant objects: The `Type()` and `Value()` methods return the variant type and value; the `As()` method returns the value after converting it to a different variant type; `ChangeType()` coerces the Variant into a different type; and `Unicode()` returns the value of a Variant object as an object of the Unicode::String class.

These conversions are more interesting if they can be applied directly to the return value of an OLE method call without first mutilating the value with default conversions. This is possible with the following trick:

```
my $RetVal = Variant(VT_EMPTY, undef);
$Object->Dispatch($Method, $RetVal, @Arguments);
```

Normally, you wouldn‘t call `Dispatch()` directly; it‘s executed implicitly by either `AUTOLOAD()` or `Invoke()`. If `Dispatch()` realizes that the return value is already a Win32::OLE::Variant object, the return value is not translated into a Perl representation but rather copied verbatim into the Variant object.

Whenever a Win32::OLE::Variant object is used in a numeric or string context it is automatically converted into the corresponding format.

```
printf "Number: %f and String: %s\n",
       $Var, $Var;
```

This is equivalent to:

```
printf "Number: %f and String: %s\n",
       $Var->As(VT_R8), $Var->As(VT_BSTR);
```

For methods that modify their arguments, you need to use the `VT_BYREF` flag. This lets you create number and string Variants that can be modified by OLE methods. Here, Corel‘s `GetSize()` method takes two integers and stores the `x` and `y` dimensions in them:

```
my $x = Variant( VT_I4 | VT_BYREF, 0);
my $y = Variant( VT_I4 | VT_BYREF, 0);
$Corel->GetSize($x, $y);
```

`VT_BYREF` support for other Variant types might appear in future releases of Win32::OLE.

FURTHER INFORMATION

DOCUMENTATION AND EXAMPLE CODE

More information about the OLE modules can be found in the documentation bundled with Win32::OLE. The distribution also contains other code samples.

The object model for Microsoft Office applications can be found in the Visual Basic Reference for Microsoft Access, Excel, Word, or PowerPoint. These help files are not installed by default, but they can be added later by rerunning ***setup.exe*** and choosing *custom setup*. The object model for Microsoft Outlook can be found on the Microsoft Office Developer Forum at: http://www.microsoft.com/OutlookDev/.

Information about the LotusScript object model can be found at:
http://www.lotus.com/products/lotusscript.nsf.

OLE AUTOMATION ON OTHER PLATFORMS

Microsoft also makes OLE technology available for the Mac. DCOM is already included in Windows NT 4.0 and can be downloaded for Windows 95. MVS and some Unix systems can use EntireX to get OLE functionality; see http://www.softwareag.com/corporat/solutions/entirex/entirex.htm.

COPYRIGHT

NAME

Win32::OLE::Variant − Create and modify OLE VARIANT variables

SYNOPSIS

```
use Win32::OLE::Variant;
my $var = Variant(VT_DATE, 'Jan 1,1970');
$OleObject->{value} = $var;
$OleObject->Method($var);
```

DESCRIPTION

The IDispatch interface used by the Perl OLE module uses a universal argument type called VARIANT. This is basically an object containing a data type and the actual data value. The data type is specified by the VT_xxx constants.

Functions

nothing()

The `nothing()` function returns an empty VT_DISPATCH variant. It can be used to clear an object reference stored in a property

```
use Win32::OLE::Variant qw(:DEFAULT nothing);
# ...
$object->{Property} = nothing;
```

This has the same effect as the Visual Basic statement

```
Set object.Property = Nothing
```

The `nothing()` function is **not** exported by default.

Variant(TYPE, DATA)

This is just a function alias of the `Win32::OLE::Variant-new()` method (see below). This function is exported by default.

Methods

new(TYPE, DATA)

This method returns a Win32::OLE::Variant object of the specified TYPE that contains the given DATA. The Win32::OLE::Variant object can be used to specify data types other than IV, NV or PV (which are supported transparently). See *Variants* below for details.

For VT_EMPTY and VT_NULL variants, the DATA argument may be omitted. For all non−VT_ARRAY variants DATA specifies the initial value.

To create a SAFEARRAY variant, you have to specify the VT_ARRAY flag in addition to the variant base type of the array elemnts. In this cases DATA must be a list specifying the dimensions of the array. Each element can be either an element count (indices 0 to count−1) or an array reference pointing to the lower and upper array bounds of this dimension:

```
my $Array = Win32::OLE::Variant->new(VT_ARRAY|VT_R8, [1,2], 2);
```

This creates a 2−dimensional SAFEARRAY of doubles with 4 elements: (1,0), (1,1), (2,0) and (2,1).

A special case is the the creation of one−dimensional VT_UI1 arrays with a string DATA argument:

```
my $String = Variant(VT_ARRAY|VT_UI1, "String");
```

This creates a 6 element character array initialized to "String". For backward compatibility VT_UI1 with a string initializer automatically implies VT_ARRAY. The next line is equivalent to the previous example:

```
my $String = Variant(VT_UI1, "String");
```

If you really need a single character VT_UI1 variant, you have to create it using a numeric intializer:

```
my $Char = Variant(VT_UI1, ord('A'));
```

As(TYPE)

`As` converts the VARIANT to the new type before converting to a Perl value. This take the current LCID setting into account. For example a string might contain a ',' as the decimal point character. Using `$variant-As(VT_R8)` will correctly return the floating point value.

The underlying variant object is NOT changed by this method.

ChangeType(TYPE)

This method changes the type of the contained VARIANT in place. It returns the object itself, not the converted value.

Copy([DIM])

This method creates a copy of the object. If the original variant had the VT_BYREF bit set then the new object will contain a copy of the referenced data and not a reference to the same old data. The new object will not have the VT_BYREF bit set.

```
my $Var = Variant(VT_I4|VT_ARRAY|VT_BYREF, [1,5], 3);
my $Copy = $Var->Copy;
```

The type of `$Copy` is now VT_I4|VT_ARRAY and the value is a copy of the other SAFEARRAY. Changes to elements of `$Var` will not be reflected in `$Copy` and vice versa.

The `Copy` method can also be used to extract a single element of a VT_ARRAY | VT_VARIANT object. In this case the array indices must be specified as a list DIM:

```
my $Int = $Var->Copy(1, 2);
```

`$Int` is now a VT_I4 Variant object containing the value of element (1,2).

Currency([FORMAT[, LCID]])

This method converts the VARIANT value into a formatted curency string. The FORMAT can be either an integer constant or a hash reference. Valid constants are 0 and LOCALE_NOUSEROVERRIDE. You get the value of LOCALE_NOUSEROVERRIDE from the Win32::OLE::NLS module:

```
use Win32::OLE::NLS qw(:LOCALE);
```

LOCALE_NOUSEROVERRIDE tells the method to use the system default currency format for the specified locale, disregarding any changes that might have been made through the control panel application.

The hash reference could contain the following keys:

```
NumDigits       number of fractional digits
LeadingZero     whether to use leading zeroes in decimal fields
Grouping        size of each group of digits to the left of the decim
DecimalSep      decimal separator string
ThousandSep     thousand separator string
NegativeOrder   see L<Win32::OLE::NLS/LOCALE_ICURRENCY>
PositiveOrder   see L<Win32::OLE::NLS/LOCALE_INEGCURR>
CurrencySymbol  currency symbol string
```

For example:

```
use Win32::OLE::Variant;
use Win32::OLE::NLS qw(:DEFAULT :LANG :SUBLANG :DATE :TIME);
```

```
my $lcidGerman = MAKELCID(MAKELANGID(LANG_GERMAN, SUBLANG_NEUTRAL));
my $v = Variant(VT_CY, "-922337203685477.5808");
print $v->Currency({CurrencySymbol => "Tuits"}, $lcidGerman), "\n";
```

will print:

```
-922.337.203.685.477,58 Tuits
```

Date([FORMAT[, LCID]])

Converts the VARIANT into a formatted date string. FORMAT can be either one of the following integer constants or a format string:

LOCALE_NOUSEROVERRIDE	system default date format for this locale
DATE_SHORTDATE	use the short date format (default)
DATE_LONGDATE	use the long date format
DATE_YEARMONTH	use the year/month format
DATE_USE_ALT_CALENDAR	use the alternate calendar, if one exists
DATE_LTRREADING	left-to-right reading order layout
DATE_RTLREADING	right-to left reading order layout

The constants are available from the Win32::OLE::NLS module:

```
use Win32::OLE::NLS qw(:LOCALE :DATE);
```

The following elements can be used to construct a date format string. Characters must be specified exactly as given below (e.g. "dd" **not** "DD"). Spaces can be inserted anywhere between formating codes, other verbatim text should be included in single quotes.

d	day of month
dd	day of month with leading zero for single-digit days
ddd	day of week: three-letter abbreviation (LOCALE_SABBREVDAYNAME
dddd	day of week: full name (LOCALE_SDAYNAME)
M	month
MM	month with leading zero for single-digit months
MMM	month: three-letter abbreviation (LOCALE_SABBREVMONTHNAME)
MMMM	month: full name (LOCALE_SMONTHNAME)
y	year as last two digits
yy	year as last two digits with leading zero for years less than
yyyy	year represented by full four digits
gg	period/era string

For example:

```
my $v = Variant(VT_DATE, "April 1 99");
print $v->Date(DATE_LONGDATE), "\n";
print $v->Date("ddd',' MMM dd yy"), "\n";
```

will print:

```
Thursday, April 01, 1999
Thu, Apr 01 99
```

`Dim()` Returns a list of array bounds for a VT_ARRAY variant. The list contains an array reference for each dimension of the variant's SAFEARRAY. This reference points to an array containing the lower and upper bounds for this dimension. For example:

```
my @Dim = $Var->Dim;
```

Now `@Dim` contains the following list: `([1,5], [0,2])`.

Get(DIM) For normal variants `Get` returns the value of the variant, just like the `Value` method. For VT_ARRAY variants `Get` retrieves the value of a single array element. In this case `DIM` must be a list of array indices. E.g.

```
    my $Val = $Var->Get(2,0);
```

As a special case for one dimensional VT_UI1|VT_ARRAY variants the `Get` method without arguments returns the character array as a Perl string.

```
    print $String->Get, "\n";
```

`LastError()`

The use of the `Win32::OLE::Variant-LastError()` method is deprecated. Please use the `Win32::OLE-LastError()` class method instead.

Number([FORMAT[, LCID]])

This method converts the VARIANT value into a formatted number string. The FORMAT can be either an integer constant or a hash reference. Valid constants are 0 and LOCALE_NOUSEROVERRIDE. You get the value of LOCALE_NOUSEROVERRIDE from the Win32::OLE::NLS module:

```
    use Win32::OLE::NLS qw(:LOCALE);
```

LOCALE_NOUSEROVERRIDE tells the method to use the system default number format for the specified locale, disregarding any changes that might have been made through the control panel application.

The hash reference could contain the following keys:

```
    NumDigits       number of fractional digits
    LeadingZero     whether to use leading zeroes in decimal fields
    Grouping        size of each group of digits to the left of the decim
    DecimalSep      decimal separator string
    ThousandSep     thousand separator string
    NegativeOrder   see L<Win32::OLE::NLS/LOCALE_INEGNUMBER>
```

Put(DIM, VALUE)

The `Put` method is used to assign a new value to a variant. The value will be coerced into the current type of the variant. E.g.:

```
    my $Var = Variant(VT_I4, 42);
    $Var->Put(3.1415);
```

This changes the value of the variant to `3` because the type is VT_I4.

For VT_ARRAY type variants the indices for each dimension of the contained SAFEARRAY must be specified in front of the new value:

```
    $Array->Put(1, 1, 2.7);
```

It is also possible to assign values to *every* element of the SAFEARRAY at once using a single `Put()` method call:

```
    $Array->Put([[1,2], [3,4]]);
```

In this case the argument to `Put()` must be an array reference and the dimensions of the Perl list−of−lists must match the dimensions of the SAFEARRAY exactly.

The are a few special cases for one−dimensional VT_UI1 arrays: The VALUE can be specified as a string instead of a number. This will set the selected character to the first character of the string or to '\0' if the string was empty:

```
my $String = Variant(VT_UI1|VT_ARRAY, "ABCDE");
$String->Put(1, "123");
$String->Put(3, ord('Z'));
$String->Put(4, '');
```

This will set the value of `$String` to `"A1CZ\0"`. If the index is omitted then the string is copied to the value completely. The string is truncated if it is longer than the size of the VT_UI1 array. The result will be padded with '\0's if the string is shorter:

```
$String->Put("String");
```

Now `$String` contains the value "Strin".

`Put` returns the Variant object itself so that multiple `Put` calls can be chained together:

```
$Array->Put(0,0,$First_value)->Put(0,1,$Another_value);
```

Time([FORMAT[, LCID]])

Converts the VARIANT into a formatted time string. FORMAT can be either one of the following integer constants or a format string:

```
LOCALE_NOUSEROVERRIDE   system default time format for this locale
TIME_NOMINUTESORSECONDS don't use minutes or seconds
TIME_NOSECONDS          don't use seconds
TIME_NOTIMEMARKER       don't use a time marker
TIME_FORCE24HOURFORMAT  always use a 24-hour time format
```

The constants are available from the Win32::OLE::NLS module:

```
use Win32::OLE::NLS qw(:LOCALE :TIME);
```

The following elements can be used to construct a time format string. Characters must be specified exactly as given below (e.g. "dd" **not** "DD"). Spaces can be inserted anywhere between formating codes, other verbatim text should be included in single quotes.

```
h       hours; 12-hour clock
hh      hours with leading zero for single-digit hours; 12-hour clock
H       hours; 24-hour clock
HH      hours with leading zero for single-digit hours; 24-hour clock
m       minutes
mm      minutes with leading zero for single-digit minutes
s       seconds
ss      seconds with leading zero for single-digit seconds
t       one character time marker string, such as A or P
tt      multicharacter time marker string, such as AM or PM
```

For example:

```
my $v = Variant(VT_DATE, "April 1 99 2:23 pm");
print $v->Time, "\n";
print $v->Time(TIME_FORCE24HOURFORMAT|TIME_NOTIMEMARKER), "\n";
print $v->Time("hh.mm.ss tt"), "\n";
```

will print:

```
2:23:00 PM
14:23:00
02.23.00 PM
```

`Type()` The `Type` method returns the variant type of the contained VARIANT.

`Unicode()`

The `Unicode` method returns a `Unicode::String` object. This contains the BSTR value of the variant in network byte order. If the variant is not currently in VT_BSTR format then a VT_BSTR copy will be produced first.

`Value()` The `Value` method returns the value of the VARIANT as a Perl value. The conversion is performed in the same manner as all return values of Win32::OLE method calls are converted.

Overloading

The Win32::OLE::Variant package has overloaded the conversion to string and number formats. Therefore variant objects can be used in arithmetic and string operations without applying the `Value` method first.

Class Variables

The Win32::OLE::Variant class used to have its own set of class variables like `$CP`, `$LCID` and `$Warn`. In version 0.1003 and later of the Win32::OLE module these variables have been eleminated. Now the settings of Win32::OLE are used by the Win32::OLE::Variant module too. Please read the documentation of the `Win32::OLE-&gt;Option` class method.

Constants

These constants are exported by default:

```
VT_EMPTY
VT_NULL
VT_I2
VT_I4
VT_R4
VT_R8
VT_CY
VT_DATE
VT_BSTR
VT_DISPATCH
VT_ERROR
VT_BOOL
VT_VARIANT
VT_UNKNOWN
VT_DECIMAL
VT_UI1

VT_ARRAY
VT_BYREF
```

VT_DECIMAL is not on the official list of allowable OLE Automation datatypes. But even Microsoft ADO seems to sometimes return values of Recordset fields in VT_DECIMAL format.

Variants

A Variant is a data type that is used to pass data between OLE connections.

The default behavior is to convert each perl scalar variable into an OLE Variant according to the internal perl representation. The following type correspondence holds:

```
C type          Perl type       OLE type
------          ---------       --------
  int              IV            VT_I4
double             NV            VT_R8
char *             PV            VT_BSTR
void *          ref to AV        VT_ARRAY
   ?              undef          VT_ERROR
   ?        Win32::OLE object    VT_DISPATCH
```

Note that VT_BSTR is a wide character or Unicode string. This presents a problem if you want to pass in binary data as a parameter as 0x00 is inserted between all the bytes in your data. The `Variant()` method provides a solution to this. With Variants the script writer can specify the OLE variant type that the parameter should be converted to. Currently supported types are:

VT_UI1	unsigned char
VT_I2	signed int (2 bytes)
VT_I4	signed int (4 bytes)
VT_R4	float (4 bytes)
VT_R8	float (8 bytes)
VT_DATE	OLE Date
VT_BSTR	OLE String
VT_CY	OLE Currency
VT_BOOL	OLE Boolean

When VT_DATE and VT_CY objects are created, the input parameter is treated as a Perl string type, which is then converted to VT_BSTR, and finally to VT_DATE of VT_CY using the `VariantChangeType()` OLE API function. See *Win32::OLE/EXAMPLES* for how these types can be used.

Variant arrays

A variant can not only contain a single value but also a multi−dimensional array of values (called a SAFEARRAY). In this case the VT_ARRAY flag must be added to the base variant type, e.g. `VT_I4 | VT_ARRAY` for an array of integers. The VT_EMPTY and VT_NULL types are invalid for SAFEARRAYs. It is possible to create an array of variants: `VT_VARIANT | VT_ARRAY`. In this case each element of the array can have a different type (including VT_EMPTY and VT_NULL). The elements of a VT_VARIANT SAFEARRAY cannot have either of the VT_ARRAY or VT_BYREF flags set.

The lower and upper bounds for each dimension can be specified separately. They do not have to have all the same lower bound (unlike Perl‘s arrays).

Variants by reference

Some OLE servers expect parameters passed by reference so that they can be changed in the method call. This allows methods to easily return multiple values. There is preliminary support for this in the Win32::OLE::Variant module:

```
my $x = Variant(VT_I4|VT_BYREF, 0);
my $y = Variant(VT_I4|VT_BYREF, 0);
$Corel->GetSize($x, $y);
print "Size is $x by $y\n";
```

After the `GetSize` method call `$x` and `$y` will be set to the respective sizes. They will still be variants. In the print statement the overloading converts them to string representation automatically.

VT_BYREF is now supported for all variant types (including SAFEARRAYs). It can also be used to pass an OLE object by reference:

```
my $Results = $App->CreateResultsObject;
$Object->Method(Variant(VT_DISPATCH|VT_BYREF, $Results));
```

AUTHORS/COPYRIGHT

This module is part of the Win32::OLE distribution.

TABLE OF CONTENTS