

Appendix: Perl Module Functions

If you have looked at any of the previous chapters of this book and haven't just randomly opened it on this page, you will notice that we make heavy use of Perl modules in almost all of our scripts. It is partly the huge number of modules shipped with ActiveState's Perl distribution that make it such a powerful administrative tool. The Perl modules that we have met in this book serve two main purposes:* First, they prevent people from having to reinvent the wheel by supplying commonly required functionality. Second, they provide a wrapper for so-called *native functions*—libraries of functions that are written in a language such as C and that interact directly with the operating system.

In order to use a module, it has to be imported into the script. This is done with a line such as the following:

```
use Win32::Registry;
```

The `use` keyword instructs the Perl interpreter to import the names of all constants and functions listed in the module's export list (listed as `@EXPORT` in the module itself—see Chapter 9 for an example of this). Once imported, all these entities will be present in the namespace of the calling script, so they can be referenced as though they were defined locally. In addition, `use` executes code in a special function called `BEGIN`; typically, this will initialize variables or possibly load external libraries that the module requires in order to work. Optionally, a `use <module>` directive can be followed by a list of names. In this case, only the named constants or functions are imported into a script's namespace; this can be useful in cases in which different modules contain names that conflict with each

* Here, we are referring to modules that ship with ActiveState and those third-party modules that provide generally useful functionality rather than the very specific kind of custom module that we met in Chapter 9, *A Custom Module*.



other. If you use this version of the incantation, names in a module's `@EXPORT_OK` list, as well as those in the `@EXPORT` list, can be imported.

Convention dictates that to prevent namespace contamination, names of constants can be exported from a module by default, but function names should not. This means that even if you have imported a module with `use`, you may well still have to refer to a function by its full "package" name when you use it. For example, if you wish to use the function `RegOpenKeyEx` (part of the registry module), you will have to refer to it as `Win32::Registry::RegOpenKeyEx`.

Many of the Win32-specific modules are wrappers for standard Windows API calls; these normally are invoked in a way that closely matches the native functions (as much as is possible, given the differences between Perl and C/C++). In addition to this conventional approach, many modules now present the Perl programmer with an object-oriented interface to native functions; often, such interfaces are easier to use and look more elegant than their conventional counterparts.* Throughout this book, we use the object-oriented versions of functions in almost all cases (where they exist). One notable exception concerns the `Win32::Registry` module, where we prefer the older, non-object-oriented versions. The reason for this is that this module appears to be in a state of flux at the time of writing: not all of the functions are implemented with the object-oriented interface, and some of the more important ones that are (e.g., the one to open a registry key) use deprecated Windows API calls. We are sure that in the near future this situation will change, but meanwhile it is safer to describe the nonwrapped versions of the functions in this book, because these are not liable to change (they reflect directly their equivalents in the Windows API).†

In the sections that follow, we describe the module functions that we have used elsewhere in the book. We warn you that this will probably not be a scintillating read, so we do not necessarily suggest you plow through it all in one sitting; instead, you should treat this as a quick reference guide for any function you want to use yourself. Note that we do not necessarily describe every single function contained within each module but restrict ourselves to the ones that we have used. However, where we do describe a function, we describe all its parameters in full even if we haven't used them. In other words, you can treat this as a complete reference guide for the functions that are listed at all!

* In fact, many of the modules bundled with the ActiveState distribution are exclusively concerned with providing an object-oriented interface to native functions and are not necessary if you wish to use the nonwrapped functions. However, it is normally worth importing these modules either way, because they deal with the hassle of actually loading the functions from the native libraries.

† If you would like to use the object-oriented versions of the registry functions and download a series of patches and hacks for them, Philippe Le Berre has an excellent web site at <http://www.inforoute.cgs.fr/leberre1/main.htm>.

Before we begin the module descriptions, there is one more general point to bear in mind: Perl is case sensitive, and that *includes* the names of modules. It is amazing how many times apparently inexplicable errors in scripts are caused by something like a failure to notice that the `MyModule` module is not the same thing as the `MyMODULE` module!

Registry Module Functions

The registry module now comes as part of ActiveState's distribution. To make use of the functions, include the following line at the beginning of your Perl script:

```
use Win32::Registry;
```

As we use the non-object-oriented versions (thereby using the module mainly as a loading mechanism), it is these functions that we explain here.

Win32::Registry::RegOpenKeyEx()

Description

Opens a specified registry key and returns a handle to it.

Usage

```
Win32::Registry::RegOpenKeyEx($Hkey, $Subkey, $Reserved, $Sam, $RegHandle);
```

`$Hkey` is a handle to a key that is already open. At first glance this may seem to present a chicken-and-egg problem; after all, the first time you call the function you won't have opened any keys. However, it's actually not a problem because the registry module exports references to a number of open keys that you can use in your own code. These are as follows:

```
HKEY_CLASSES_ROOT  
HKEY_CURRENT_CONFIG  
HKEY_CURRENT_USER  
HKEY_LOCAL_MACHINE  
HKEY_PERFORMANCE_DATA  
HKEY_PERFORMANCE_NLSTEXT  
HKEY_PERFORMANCE_TEXT  
HKEY_USERS
```

Their meaning is clear from the names: each refers to a different registry hive.

`$Subkey` specifies the name of the registry key you wish to open; this must reside within the key defined by `$Hkey`.

`$Reserved` is currently unused and should be set to `NULL`.

`$Sam` is an access modifier, used to specify the level of access you require. Typical values for this variable are defined by the following constants:

```
KEY_READ
KEY_WRITE
KEY_EXECUTE
KEY_ALL_ACCESS
```

If the attempt to open a registry key has been successful, `$RegHandle` will be a handle to the opened key. This handle can be used as an `$Hkey` parameter if you wish to open a further subkey or can be passed to any of the other registry functions.

Win32::Registry::RegCloseKey()

Description

Closes a key opened with `Win32::Registry::RegOpenKeyEx()`.

Usage

```
Win32::Registry::RegCloseKey($RegHandle);
```

`$RegHandle` is the handle of the open key that you wish to close. You should always close a key once you have finished working with it.

Win32::Registry::RegSetValueEx()

Description

Used to either create or modify a value within a registry key.

Usage

```
Win32::Registry::RegSetValueEx($RegHandle, $Valuename, $Reserved, $DataType,
    $Data);
```

`$RegHandle` is the handle of the open key within which you wish to create or modify a value.

`$Valuename` is a string specifying the name of the value that you wish to set.

`$Reserved` is unused and should be set to `NULL`.

`$DataType` specifies the type of data the value will hold (e.g., string, binary, etc.—see Table 2-1 for the list of constants that can be used).

`$Data` specifies the actual data that will be assigned to the `$ValueName` key. The form this value should take depends on `$DataType`.

Win32::Registry::RegDeleteValue()

Description

Deletes a value from an open key. This function removes the value entirely; it does *not* simply set it to NULL.

Usage

```
Win32::Registry::RegDeleteValue($RegHandle, $ValueName);
```

`$RegHandle` is a reference to an open key.

`$ValueName` specifies the value to be removed.

Win32::Registry::RegCreateKey()

Description

Creates a new key and returns a handle to it.

Usage

```
Win32::Registry::RegCreateKey($Hkey, $Subkey, $RegHandle);
```

`$Hkey` is a handle to an open key.

`$Subkey` specifies the name of the new key you wish to create within `$Hkey`.

To create the key `HKEY_LOCAL_MACHINE\SOFTWARE\Animals\Goose`, set `$Hkey` to `HKEY_LOCAL_MACHINE` and `$Subkey` to `SOFTWARE\Animals\Goose`.

If the attempt to create a registry key has been successful, `$RegHandle` will be a handle to the opened key.

All of the preceding functions return success or failure, so it is possible to use them as the condition in an `if` statement. For example:

```
If (Win32::Registry::RegCreateKey(HKEY_LOCAL_MACHINE, 'SOFTWARE\animals',  
$RegHandle)  
{  
    # Things in here will be executed if the key was created successfully.  
}
```

Process Module Functions

The Process module comes as part of ActiveState's Perl distribution. It provides access to a set of Windows API functions for dealing with process management; it also exports several constants that are needed to use these functions. At the time

of writing, the module does not present an object-oriented interface. To use the module, include the following line at the start of your Perl script:

```
use Win32::Process;
```

In this book, we use only a single function from this module. It is described here:

Win32::Process::Create()

Description

Creates a new process.

Usage

```
Win32::Process::Create($Object, $Applic, $Cmdline, $Iflags, $Cflags, $Curdir)
```

`$Object` will be set as a handle to the process object once it has been created.

`$Applic` specifies the full path and executable name of the application to be launched.

`$Cmdline` specifies the command line to be executed.

`$Iflags` is a flag used to specify whether the new process inherits the process handle of the calling process.

`$Cflags` specifies one or more creation flags that determine the behavior and state of the newly created process. The module exports constants defining these flags for use in your code. These constants are as follows:

```
CREATE_DEFAULT_ERROR_MODE  
CREATE_NEW_CONSOLE  
CREATE_NEW_PROCESS_GROUP  
CREATE_NO_WINDOW  
CREATE_SEPARATE_WOW_VDM  
CREATE_SUSPENDED  
CREATE_UNICODE_ENVIRONMENT  
DEBUG_ONLY_THIS_PROCESS  
DEBUG_PROCESS  
DETACHED_PROCESS  
HIGH_PRIORITY_CLASS  
IDLE_PRIORITY_CLASS  
INFINITE  
NORMAL_PRIORITY_CLASS  
REALTIME_PRIORITY_CLASS  
THREAD_PRIORITY_ABOVE_NORMAL  
THREAD_PRIORITY_BELOW_NORMAL  
THREAD_PRIORITY_ERROR_RETURN  
THREAD_PRIORITY_HIGHEST  
THREAD_PRIORITY_IDLE  
THREAD_PRIORITY_LOWEST  
THREAD_PRIORITY_NORMAL  
THREAD_PRIORITY_TIME_CRITICAL
```

As you can see, the names are self-explanatory. If you wish to specify more than one of these flags, you use the `|` (logical or) operator to join them. For example, to create a high-priority process without a window, you would specify `THREAD_PRIORITY_HIGHEST | CREATE_NO_WINDOW`.

`$CurDir` specifies the working directory for the new process.

The following is an example of this function in use. It is lifted from the *self.pl* script in Chapter 3, *Remote Script Management*:

```
Win32::Process::Create(  
    $process, "C:\\Perl\\5.00502\\  
    bin\\MSWin32-x86-object\\perl.exe", "perl $SelfName",  
    0, CREATE_NO_WINDOW, ".");
```

This function will create a process, whose image is *perl.exe* and whose command line is `perl` plus the value of `$SelfName`. The process will not inherit the process handles from its parent (it will be an independent process) and it will have no window. The working directory will be the current directory.

Net::SMTP Module Functions

This module provides a set of functions for sending email with the SMTP protocol. It uses the object-oriented type of interface. At the time of writing it does not come as part of ActiveState's distribution. Assuming that you are connected to the Internet, however, it can be installed easily using the *Perl Package Manager (PPM)*, which does come as standard. `Net::SMTP` is one of a number of modules providing networking functionality; these are bundled together under the collective package name *Libnet*.^{*} To run the package manager, type the following at the command prompt:

```
C:\>ppm.pl
```

If you are using the same release of Perl as we are, the following will be displayed on your screen:

```
PPM interactive shell (0.9.1) - type 'help' for available commands.  
PPM>
```

At this point, simply type `install libnet`. PPM will transparently go through the process of downloading the module over the Net. Next, a series of onscreen instructions take you through the rest of the installation process. During this process, you will be required to provide details of your SMTP server, NNTP server, and so forth, as these details are essential to the operation of the Net modules. If you do not have all the details at hand when you install the package, it is not

^{*} These Win32 versions of these Net modules can be installed only as one bundled package. By contrast, the Unix versions are all available on an individual basis from CPAN.

serious, however, as you can always change these settings later. This is done either by locating and editing the configuration files directly or by using the `Net::Config` module (not described here).

Once installed, `Net::SMTP` can be used in the normal way. Ensure that the following is included at the start of your Perl script:

```
use Net::SMTP;
```

Net::SMTP->new()

Description

This function is the constructor for a new SMTP object. This method must be called before any other SMTP methods can be accessed, as it is responsible for opening a connection to the server.

Usage

```
$smtp = Net::SMTP->new($Mailhost, [OPTIONS]);
```

`$smtp` is a handle to the new SMTP object created by the constructor. As will be seen later, it is required for all subsequent calls to SMTP methods, as it refers to the specific server connection specified to be set up by `new()`.

`$Mailhost` is the name of the SMTP server to be used for the outgoing mail.

[`OPTIONS`] specifies one of a number of optional parameters that determine aspects of behavior of the new SMTP object. They are passed to the constructor as a set of hash pairs, like this:

```
Hello=>yourhost.domain  
Timeout=>20  
Debug=>1
```

The `Hello` option is used to identify your host to the mail server when a connection is made. `Timeout` dictates how long to wait (in seconds) for a response from the SMTP server before giving up. The `Debug` option is used to indicate whether debug information should be generated for a connection session. If no options are specified, then a default value is sent for the timeout, no debug information is generated, and the server resolves the hostname for itself.

Example

```
$smtp = Net::SMTP->new('smtp.myorganization.com');
```

Net::SMTP->mail()

Description

This method sends the source address (return path). This method is called using the object handle created by the previous function. Its purpose is to inform the SMTP server as to who is sending the message (a requirement of the SMTP protocol).

Usage

```
$smtp->mail($Sender, [OPTIONS]);
```

`$Sender` is the source address.

Just as with the `new()` method, you can optionally specify some extra parameters here, as hash pairs. These options are part of the family of SMTP extensions (ESMTP) specified in a number of RFCs.*

Example

```
$smtp->mail('fred@myorganization.com');
```

Net::SMTP->recipient()

Description

This method tells the SMTP server who should receive the mail message. It is essential to call this method before trying to send any message content. A call to this method *must* be preceded by a call to `mail()`.

Usage

```
$smtp->recipient($Recipient, [$NextRecipient]);
```

`$Recipient` specifies the address of, you guessed it, the recipient. Optionally, a whole list of recipients can be specified.

Example

```
$smtp->recipient('frank@zappa.com', 'darlington@nohope.co.uk');
```

Net::SMTP->to()

Description

This method is identical to `recipient()` and carries out the exact same function. Everything described in the preceding section applies equally here.

* An RFC, or Request for Comment, document specifies new standards within the networking world. The SMTP protocol is specified by RFC 812, while ESMTP is defined variously in RFCs 1425–28..

Net::SMTP->data()

Description

This method is used to initiate the sending of the actual message data. Specifically, it sends a `DATA` command to the SMTP server. A call to this method *must* be preceded by a call to `recipient()` or `to()`.

Usage

```
$smtp->data([LIST]);
```

If [LIST] is not empty, its contents are sent to the server as message content.

Examples

```
$smtp->data("A rusty spoon is not a very interesting archaeological find\n");
```

OR:

```
$smtp->data($Message);
```

OR:

```
$smtp->data();
```

Net::SMTP->datasend()

Description

This method is used to send some data to the waiting server. A call to this method *must* be preceded by a call to `data()`.

Usage

```
$smtp->datasend($SomeData);
```

`$SomeData` is simply message data.

Example

```
$smtp->datasend("Tim, how's the running going?\n");
```

Net::SMTP->dataend()

Description

This method is used to end the message. Once it has been called, no more data can be sent.

Usage

```
$smtp->dataend();
```

There are no parameters required for this method.

Example

```
$smtp->dataend();
```

Net::SMTP->quit()

Description

This method ends the session with the SMTP server and kills the connection.

Usage

```
$smtp->quit();
```

This method takes no parameters.

Example

```
$smtp->quit();
```

Event Module Functions

The Event module comes as a part of ActiveState's distribution. It provides Perl with the functionality required to manipulate NT's event logs. It is first seen in this book in Chapter 4, *System Maintenance*. In order to use it, the following line must be included at the start of your script:

```
use Win32::EventLog;
```

Following are the functions from this module that we have used in this book. As you can see, this module provides an object-oriented interface to the Windows API functions.

Win32::EventLog->new()

Description

This method is the constructor for a new event log object. As is the case with all constructors, this method must be called before any of the other methods in the module, as it provides the access point to the module's functionality.

Usage

```
$evtlog = Win32::EventLog->new($LogType, $ComputerName);
```

`$LogType` is a string specifying which of the three event logs to access (system, security, or application).

`$ComputerName` is the a string containing the name of the workstation (or server) whose logs should be accessed.

`$evtlog` will become the object handle used to reference the open event log and access all other methods in the module.

Example

```
$evtlog = Win32::EventLog->new('system', $ENV{COMPUTERNAME});
```

Win32::EventLog->Read()

Description

This method is used to read entries from an open event log.

Usage

```
$evtlog->Read(FLAGS, $Record, \%eventinfo);
```

The `FLAGS`, in conjunction with `$Record`, determine the read behavior. Four constants exported by the module can be passed as flags. These are as follows:

- EVENTLOG_SEQUENTIAL_READ
- EVENTLOG_SEEK_READ
- EVENTLOG_FORWARDS_READ
- EVENTLOG_BACKWARDS_READ

The first two specify whether events should be read from the log in sequence each time the method is called (`EVENTLOG_SEQUENTIAL_READ`) or whether the function should jump straight to the record specified by `$Record`. The second pair of constants determine the read direction, either forward (`EVENTLOG_FORWARDS_READ`) or backward (`EVENTLOG_BACKWARDS_READ`).

These constants operate as masks, so if you wish to specify more than one, the `|` operator can be used. For example, to specify sequential read in a forward direction, you would pass `EVENTLOG_SEQUENTIAL_READ | EVENTLOG_FORWARDS_READ` as the `FLAGS` parameter. In this case, you would set `$Record` to be 0. The order in which the flags are listed is unimportant.

`\%eventinfo` is a reference to a hash table that will contain the record data for each record once it has been retrieved. The information is stored with the following hash keys:

Length

Specifies the length of an event record. This is measured in bytes.

RecordNumber

Specifies the record number of the event entry.

TimeGenerated

When an event is submitted to the event log service, it is timestamped with a value corresponding to the number of seconds that have elapsed since January 1, 1970. The time value is stored here.

TimeWritten

The event is timestamped a second time, indicating when the event log service actually wrote the event to the log. Again, the time is given as a number of elapsed seconds since January 1, 1970.

EventID

This number uniquely identifies the event type. It is meaningful only in term of the event **Source** (see that later entry), and can be used to determine the nature of the event that caused the log to be written.

EventType

Specifies the type of event, which can be one of the following: Error, Warning, or Information. In addition, the security log supports two more event types: Success Audit and Failure Audit.

Category

This field is source specific.

Source

This string variable contains the name of the source that generated the event.

Computer

This string represents the name of the source computer that generated the event.

Strings

This list of strings contains information relevant to the event. These usually are meaningless on their own and need to be merged with an event *message* if they are to be of any use (see Chapter 4).

Data

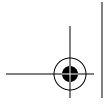
A variable amount of binary information. This could be anything, really, depending on the event source.

Example

```
$evtlog->Read(EVENTLOG_FORWARDS_READ| EVENTLOG_SEQUENTIAL_READ,0,\%eventinfo)
```

As the `read()` method accesses a single record each time it is invoked, typical usage would employ a loop, as shown here:

```
While($evtlog->Read(EVENTLOG_FORWARDS_READ| EVENTLOG_SEQUENTIAL_READ,0,\%eventinfo))  
{  
    print "The event source is $eventinfo{Source}";  
}
```



Clear()

Description

This method archives the contents of all three event logs to a file and then clears the logs.

Usage

```
$evtlog->Clear($LogName);
```

\$LogName specifies a full path and filename for the archive file. If the specified file already exists, the method fails.

Example

```
$evtlog->Clear('C:\evtback\security.evt');
```

