



---

# 9

## *A Custom Module*

Throughout this book, we have been presenting scripts and script fragments that can be used to perform specific maintenance and configuration tasks on NT workstations. With a little effort, many of the scripts could be adapted to perform other tasks; in many circumstances, a mere change of registry key will suffice to harness the power of a script in a different situation than that for which it was written. Demonstrating the flexibility of scripting for administration has been one of the main points of the book. Unfortunately, however, there is a danger associated with altering scripts: careless alteration of a reasonably lengthy script can introduce bugs that, while rendering the code totally useless, are very hard to detect. In fact, you could be forgiven for thinking that a general problem with using Perl for scripting in the way that we have been promoting is that the code required (especially when using registry editing functions and the like) is generally rather verbose, repetitive, and error prone.

Does this mean that all reasonably complex maintenance and configuration scripts written in Perl are inherently unadaptable and take hours to write? The answer is a resounding, emphatic “No!”. There is a method of coding that allows for sophisticated error detection in scripts that can be written quickly, while actually reducing the likelihood of introducing bugs. The trick is to identify all of the scripting tasks involving repetitive code you use again and again and to wrap this code in a series of generic functions. If these functions are grouped as a module (see the Appendix, *Perl Module Functions*), any maintenance or configuration script can call them trivially after the invocation of a simple `use` statement. Effectively, the trick is to write a high-level custom API that is tailored to carry out the sorts of tasks that your scripts require on a regular basis. The module has to be written only once, but scripts can be written to take advantage of its functionality again and again. There are several huge advantages to be gained by writing a module:



- It enables the writing of scripts much more quickly, as much of the repetitive code is contained within the prewritten module.
- It allows sophisticated error checking, logging, and the like to be an integral part of all your scripts. The code for this can be self-contained in the module, and it will apply transparently to any script that subsequently uses the module.
- If changes to either the Perl language or the underlying workings of the operating system prevent module functions from working (for example, if registry-calling conventions change), only the code in the module itself needs to be altered, and all of the scripts that use it will suddenly work again. By contrast, if all code is hard-wired into scripts, every script would have to be rewritten to reflect the change.
- Wrapping up sophisticated functionality leads to noncluttered, easy-to-debug code that reads more like a simple list of instructions than a program written in a rather esoteric scripting language.

To be sure, writing a module takes some thought and planning, but we feel the effort is easily worthwhile. As we declared at the outset of this book, writing a script to carry out an automated task normally takes much longer than merely carrying out the task manually on a single workstation; the payoff for writing the script comes when you need to administer many workstations. The argument for writing a module is similar: it normally takes longer to write a generic module than a simple script designed to carry out a very specific task; the payoff is that the module needs to be written only once and can be used countless times.

This, the final chapter of the book, demonstrates the writing of a custom module. First, we introduce the specification for a `MaintainAndConfigure` module, outlining the functions it provides and giving examples of their use. In the second section, we present the code for the module and suggest how it could be expanded or adapted. By the time you reach the end of the chapter, we hope that you will be persuaded that a combination of a custom module, scripts, and automatic stub-loaders (see Chapter 3, *Remote Script Management*) provide an invaluable administration tool, making some of the most tedious tasks trivial and painless.

## *The MaintainAndConfigure Module*

The functions of the `MaintainAndConfigure` module will incorporate much of the functionality provided by scripts presented throughout this book. The main difference is that the module functions must be generic and should provide full error-detection and logging. The only really new territory is presented in the final function that we describe, `MConfigFromFile()`; this is a function that dramatically extends the possibilities for configuring workstations on an individual basis.

Following, each of the module functions is described and an example of its use demonstrated. In order to use any of these functions in a script, all that is required is for the module to reside somewhere where Perl can find it (the same directory as the Perl libraries or as the script using it) and for the script to begin with the following directive:

```
use MaintainAndConfigure;
```

The description of all the functions presented here is not a post hoc instruction manual; rather, it should be considered the specification around which the module was written. After all, if a module is to be useful it should be constructed around need, rather than be a solution in search of a problem.

The functions are split into three main groups. The first is concerned with the process of controlling script operations themselves, such as raising errors and writing to a log file; the second is concerned with workstation configuration tasks; the third is more about batch processing.

### *Script Control Functions*

#### *MCWriteLog()*

The function `MCWriteLog()` writes a single-line entry to a log file. It takes one parameter, namely a string value; this value is written to the log along with a timestamp. `MCWriteLog()` is used by virtually every other function in our module; it also can be used directly in any script that wishes to use the facility it provides. The filename of the log file is hardcoded into our module; as will become apparent later on in the chapter when the code is presented, this does not necessarily have to be the case. Some administrators might prefer that their scripts write to the NT Event Log rather than to a text-based log file; this could be accomplished reasonably trivially using the `Win32::EventLog` Perl module (see Chapter 4, *System Maintenance*, and the Appendix).

Following is an example of a short script that prints "Oh what a loggable world!" to standard output and uses `MCWriteLog()` to report that it has done so:

```
use MaintainAndConfigure;
Print "Oh what a loggable world!";
MCWriteLog("Written to STDOUT");
```

The log file entry produced would look very much like this:

```
01/01/98:12:52:05 - Written to STDOUT
```

#### *MCRaiseError()*

`MCRaiseError()` is similar to `MCWriteLog()` in that it takes a single string parameter, which it writes out to a log file (actually it invokes `MCWriteLog()` itself).

However, in addition, it emails an administrator and pops up a warning dialog on a designated admin machine reporting the error. Finally, it stops execution of the script that called it. Like `MCWriteLog()`, this function is used by most of the other functions in the `MaintainAndConfigure` module but also can be used directly by scripts that need to report errors rather than dying silently. One potentially useful feature that could be added to the function with a minimum of fuss would be a second parameter that allowed the calling code to specify whether the error should merely be reported or whether script execution should actually be halted.

#### *MCSetAdminMachine()*

This function simply sets a string value corresponding to the *UNC (uniform naming convention)* name of the workstation on which a pop up should be presented when `MCRaiseError()` is called. A default value is hardcoded into the module, so it is unnecessary for `MCSetAdminMachine()` to be called in order for the error-raising function to work; however, it is useful in situations in which the machine to be notified depends on the script that happens to be running.\*

#### *MCSetAdminEmail()*

As might be guessed, this function sets a string value corresponding to the email address of the person who should be informed of an error raised by `MCRaiseError()`. Everything that applies to `MCSetAdminMachine()` applies equally to this function.

### *Configuration Functions*

#### *MCWriteRegistry()*

The function `MCWriteRegistry()`, as the name suggests, is used to write to the registry. The Perl `Win32::Registry` module, which has made appearances throughout this book (and is discussed in detail in the Appendix) provides ample facilities for creating, modifying, and deleting registry keys and values, so the function `MCWriteRegistry()` and its companion, `MCDeleteRegistry()`, are not intended to replace these; in fact, they use them extensively. Instead, they are meant to supplement these facilities by providing a higher-level interface that is easier to use and avoids the need for repetitive coding. `MCWriteRegistry()` has

---

\* Given that this function merely sets the value of a variable in the module's namespace, it is actually superfluous to requirements, as there is no reason why calling code cannot set the value of the variable directly (`MaintainAndConfigure::$AdminMachine = '\\SENECA'`;). However, the function exists for two reasons: (1) the implementation of the module might change, and using a function (rather than setting a variable directly) in calling code prevents it from being affected by the change; (2) one of the authors of this book has a background in object-oriented programming and is therefore obsessed with the concept of encapsulation. For evidence of (2), see (1)!

the following features that makes it particularly useful compared with the Perl registry functions:

- If an attempt is made to write a value to a registry key that doesn't exist, the key will be created automatically.
- A whole set of name/value pairs can be written to a registry key with just a single invocation of the function.
- All creation and modification of keys and values is transparently logged (using `MCWriteLog()`).
- Failure to write a value will cause an error to be raised.

The function takes two parameters: the first is a string corresponding to the registry key to be opened or created; the second is a *reference* to an array of name/value/type triplets. This second parameter deserves some further explanation. The underlying Win32 API registry functions (and therefore the Perl registry functions) require four pieces of information to write a registry value: the key under which the value is to be written, the name of the value, the value itself, and the type (`REG_DWORD`, `REG_SZ`, and so forth. As the `MCWriteRegistry()` function can write several values with one invocation, this information cannot be passed to it as a simple list of scalars. Instead, a single set of names, values, and types has to be passed as an array, so that multiple sets can be passed as an array of arrays! The reason an array *reference* is needed (rather than an array itself) is that when parameters are passed to functions, arrays are flattened to lists of scalars.

If this all sounds confusing, that is because it is! However, `MCWriteRegistry()` really does make life easier. A code example should make everything much clearer:

```
$key = 'SOFTWARE\CoolBookInfo';
@values =
(
    ["Publisher", "Oreilly", REG_SZ],
    ["NumberOfChapters", 10, REG_DWORD],
    ["Subject", "WindowsNT", REG_SZ]
);
MCWriteRegistry($key, \@values);
```

The preceding snippet of code will write three values (`Publisher`, `NumberOfChapters`, `Subject`) to the key `SOFTWARE\CoolbookInfo` under the `HKEY_LOCAL_MACHINE` hive. It should now be clear what we mean by arrays of arrays. The first line of code here simply sets up a string containing the value of the key to open or create; `HKEY_LOCAL_MACHINE` is not explicitly mentioned anywhere, as our function *always* writes to this hive.\* The next six lines (actually only

---

\* We always write to `HKEY_LOCAL_MACHINE` because we use our module for workstation maintenance and configuration rather than user-profile manipulation. This behavior can be adapted if required with a simple modification to the module.

one line, but split so that it is easier to read) set the array of name/value/type triplets to be written to the registry key. The constants used to specify type (here only the most popular, `REG_SZ` and `REG_DWORD` are used) are exported by the `Win32::Registry` module; our module also exports them so that scripts that use it do not have to use `Win32::Registry` specifically in addition to our module. The final line of code is the call to the function itself.

If the preceding example seems a little long-winded, it should be remembered that writing another 10 values to the same key would merely involve the addition of another 10 lines to the list. Of course, if you need to write only one value, the syntax is comparable to using the `Win32::Registry` functions, but you have the added bonus of error raising and logging. Here is an example of a single-value call to the function:

```
MCWriteRegistry('SYSTEM\MyKey', (["MyValue", "hello", REG_SZ]));
```

This creates the key `MyKey` under `HKLM\SYSTEM` and writes a string value (`MyValue`) with the value “hello”.

### *MCDeleteRegistry()*

`MCDeleteRegistry()` is the companion function to `MCWriteRegistry()`. Unsurprisingly, its purpose is to remove registry values. The rationale for its existence is very similar to that of `MCWriteRegistry()`, namely, it provides a higher-level interface than do the native functions, it provides error raising and logging for free, and allows multiple registry values to be deleted with a single invocation of the function. The syntax required to use `MCDeleteRegistry()` is much simpler than that for `MCWriteRegistry()`; this reflects the fact that the Windows API needs considerably less information to delete a value than it does to create one. Specifically, all that is needed is the name of the key in which the value to be deleted resides and the name of the value itself. Our function can take *any* number of string-value parameters provided it is given at least two: a keyname and a value name. Any parameters passed to it in addition to these first two are taken to be further values requiring deletion.

A call to `MCDeleteRegistry()` that deletes the value `MyValue` from `SYSTEM\MyKey` in the `HKEY_LOCAL_MACHINE` hive would look like this:

```
MCDeleteRegistry('SYSTEM\MyKey', 'MyValue');
```

A call to the function that deletes four values (`MyValue`, `HisValue`, `HerValue`, and `ItsValue`) from the same registry key would look like this:

```
MCDeleteRegistry('SYSTEM\MyKey', 'MyValue', 'HisValue', 'HerValue',  
'ItsValue');
```

You will notice that there is no mention here of the actual values or the types; for deletion purposes, this information is totally irrelevant.

### *MCInstallAutoLogon()*

The `MCInstallAutoLogon()` function sets all the registry keys required to ensure that the next time a workstation is rebooted, automatic logon will be triggered and a specified script run. In other words, it wraps up the functionality first presented in Chapter 2, *Running a Script Without User Intervention*, where automatic logons are discussed. It is envisaged that this function will be most useful in contexts in which a stub is used to run a script whose purpose is to install another script that will, in turn, carry out some configuration changes after a reboot (see Chapter 3).

The function requires three parameters: the fully qualified path and name of a script to run during autologon and the username and password of a logon account. If security requirements dictate that the account used for autologons should be kept in a disabled state, this poses no problems for `MCInstallAutoLogon()`, as it will enable the account before it attempts to use it.\*

The fragment of script presented next sets up an automatic logon to run a script called *pterodactyl.pl*, which resides in *C:\scripts*. The account name for logon is *triceratops* and the password is *d1pl0d0cu5*:

```
use MaintainAndConfigure;  
MCInstallAutoLogon('C:\scripts\pterodactyl.pl', 'triceratops', 'd1pl0d0cu5');
```

If any aspect of the installation process fails, an error is raised.

### *MCRemoveAutoLogon()*

Once a workstation is set up to carry out automatic logon, it will continue to do this until explicitly disabled. A configuration script that runs in an automatic-logon context should normally ensure that this disabling is carried out. `MCRemoveAutoLogon()` wraps up the process of writing the necessary registry keys and disabling an autologon account.

The function takes up to three optional parameters. The first is the username of an account to disable (this should be the name of the account that was used for automatic logon). The second two are the text strings for the `LegalNoticeCaption` and `LegalNoticeText` registry values (see Chapter 2).

If any aspect of the removal process fails, an error is raised.

### *MCInstallService()*

The function `MCInstallService()` performs a similar function to `MCInstallAutoLogon()` except, as the name implies, it installs a script to run as a service. Here only one parameter is necessary: the fully qualified path and name of the

---

\* This functionality requires that the script invoking this function is running with administrative privileges.

script to be installed as a service. The reason for this is that `MCInstallService()` always installs scripts to run in the `LocalMachine` security context. The location of the Perl installation itself, information that the function requires in order to install any script as a service, is hardcoded into the module. For some environments it may be appropriate to alter the `MCInstallService()` function to accept the location of Perl as a parameter; such an alteration would be trivial to implement.

If any aspect of the installation process fails, an error is raised. In order for the function to work at all, the two NT Service Pack Utilities `srvany.exe` and `instsrv.exe` must be present on the system in a location that is part of the *path* (see Chapter 2).

The following script fragment installs the script `strawberry.pl`, residing in `C:\scripts`, to run as a service:

```
use MaintainAndConfigure;
MCInstallService('C:\scripts\strawberry.pl');
```

### *MCRemoveService()*

`MCRemoveService()` is the companion function to `MCInstallService()`. Its purpose is to uninstall a script that is currently running on a workstation as a service, but as it is merely a wrapper for `instsrv.exe` (a program that must be present on the system for the function to work), it will actually uninstall any service. It takes a single parameter, the name of the service to remove. By default, any failure to remove the service will result in an error being raised. Such behavior is not necessarily desirable and can be changed simply by deleting a couple of lines of code; it will be obvious which ones need to be removed when the code is presented later in this chapter.

### *MCGetIdentity()*

The function `MCGetIdentity()` returns the computer name and IP address of the workstation on which it is being run. It can be used in any situation in which a script needs to find out workstation identity, as the technique used by the function to extract the information from the computer depends on the way it is called. If no parameters are passed, `MCGetIdentity()` extracts the computer name from NT's environment table and IP address using the `ipconfig.exe` utility\* (see Chapter 6, *Machine-Specific Scripting*). However, if a fully qualified path and name of a file in the `address.mac` format (see Chapter 6) is passed as a parameter to `MCGetIdentity()`, then the function will extract a computer name and IP

---

\* On a multihomed system, only the first IP address is extracted.

address from the *address.mac* file by attempting to find a match with the workstation's MAC address. If no MAC address match is found, an error is raised.

The first incantation of the function is the most practical in almost every scripting situation. A script fragment that employs it might look something like this:

```
($name, $IP) = MCGetIdentity;
```

Alternatively:

```
@identity = MCGetIdentity;
```

The second incantation is most useful in a situation in which the script that needs to use it is trying to *change* the identity of a workstation. In such a situation, it is irrelevant what the workstation *thinks* its identity is; what matters is what new identity has been assigned to it, information that resides in an *address.mac* file. Following is an example of a script fragment that uses this incantation of the function to extract identity from a file located in *C:\localconfig*:

```
($AssignedName, $AssignedIP) = MCGetIdentity('C:\localconfig\address.mac');
```

## Batch Processing Functions

### *MCUpdateScripts()*

*MCUpdateScripts()* has been written specifically to be used by stub scripts (see Chapter 3). Its task is to synchronize workstation and server script directories to ensure the following:

- That scripts being run by the stub on a workstation are up-to-date versions
- That the workstation has local copies of all the scripts on the server drive
- That the workstation is not running any old scripts that are no longer required

The function carries out this functionality by connecting to a specified server drive and comparing the contents of specified directories. It takes three parameters: the fully qualified path of the local script directory, the server and share name to connect to (in UNC format), and the pathname on the share of the server's script directory.

The following script fragment connects to a share called *GRAPE* on the server *FIG*. The local updateable-script path is *C:\scripts\updateable*; the path on the share is *\updates*:

```
MCUpdateScripts('C:\scripts\updateable', '\\FIG\GRAPE', '\updates');
```

Unlike many of the other functions contained in the *MaintainAndConfigure* module, *MCUpdateScripts()* does not raise an error if it fails to accomplish its task. The reason is that in most circumstances it is not at all appropriate for a stub

script to stop executing if its script directory cannot be updated. After all, a major point in keeping local copies of maintenance scripts rather than running them directly from a server drive is to ensure that scripts can still run if the server is unavailable. On the other hand, there may well be circumstances in which raising an error is desirable. Therefore, `MCUpdateScripts()` generates a return value, which can be read to determine if its operation were successful. A return value of zero indicates that the update was accomplished successfully; a nonzero value indicates otherwise.

The following fragment of script performs exactly the same task as the one presented previously, but here the return value of `MCUpdateScripts()` is checked, and an error raised if the update was unsuccessful:

```
if(MCUpdateScripts('C:\scripts\updatable', '\\FIG\GRAPE', '\updates'))
{
    MCRaiseError("Update failed dismally");
}
```

In an environment in which an error should always be raised on update failure, it would, of course, be possible to amend the `MCUpdateScripts()` function to incorporate error raising rather than—or in addition to—providing a return value.

### *MCConfigFromFile()*

Whereas all the functions so far discussed in script control and configuration are effectively wrappers for techniques introduced at earlier points in the book, `MCConfigFromFile()` offers genuinely new functionality. Its purpose is dramatically to extend capabilities for machine-specific workstation configuration. As you will remember from Chapter 6, *Machine-Specific Scripting*, and Chapter 7, *Changing Network Identity*, a Perl script can be written to carry out certain actions depending on the identity of the machine on which it is being run. It can do this by finding out the identity of a workstation (either by extracting the information from the workstation directly or using a combination of MAC address and a lookup file) and then using this information in one of two ways. The first way is to execute blocks of statements conditionally. For example:

```
if($name eq "ALBATROSS"){... do this stuff ...}
else {... do this stuff ...}
```

The second way is to write the information retrieved from an *address.mac* file into various parts of the registry (or elsewhere). For example:

```
($name, $IPAddress) = MCGetIdentity;
@values = (['name', $name, REG_SZ], ['IP', $IPAddress, REG_SZ]);
MCWriteRegistry('SYSTEM\VitalInformation', \@values);
```

This example writes the computer name and IP address into a fictitious part of the registry. You will note that to save coding, it uses some of the module functions

described earlier. If more machine-specific information than simple name and IP address is required for a script to carry out its task, then the easiest solution is to add a few more lines to *address.mac* and modify the `MCGetIdentity()` function to cope with this.\*

The problem with such a solution, however, is that if more than a few parameters need to be specified in *address.mac*, the file would soon become unmanageably large and confusing. A further limitation is that it can still supply only static parameters, whose meaning is determined solely by position in the file; this is hardly flexible. `MCConfigFromFile()` is designed to allow for flexible machine-specific configuration. It does this by reading a configuration file passed to it as a parameter and acting according to the contents. The file can contain registry entries to add (or to remove) and even a list of system calls to carry out. While this may not seem that revolutionary a concept in itself, its power comes from the fact that every single workstation, or a group of workstations, can have a different configuration file associated with it; each can therefore be configured by the same script in a totally different way.

How does the script know which configuration file is associated with which machines? It depends on the context. In many situations, the configuration file can simply have the name of the workstation or the workgroup for which it applies. In this case, `MCConfigFromFile()` would be invoked as follows (assuming that `$path` points to the location of the configuration files and that `$name` contains the value of the computer name):

```
MCConfigFromFile($path.$name);
```

Or, if the variable name has not yet been set, syntax such as the following would suffice:

```
MCConfigFromFile($path.$ENV{ComputerName});
```

An alternative would be to have the fully qualified path to the configuration file included as an entry in an *address.mac* file.

`MCConfigFromFile()` may need only one parameter, the name of the configuration file, which must exist and must be in the correct format! The format of the configuration file is extremely simple. It consists of three types of entry:

#### *Section header*

Defines the start of a command section. It is always enclosed in square brackets and contains one of two things: the fully qualified name of a registry key within the `HKEY_LOCAL_MACHINE` hive or the word `COMMANDS`.

---

\* An interesting possibility would be to implement `MCGetIdentity` in such a way that it could read *any* number of parameters from an *address.mac* file. As the function simply returns an array, this would not affect code that did not need extra parameters.

*Section entry*

Defines an action to be carried out. The way in which it is interpreted and its proper format depend on context.

If the entry is situated below a registry key header, it defines a registry value within that key that will be added or removed. The format of such an entry is either name, value, and type (where these terms mean exactly the same as in the `MCWriteRegistry()` function) or simply name. In the former case, the registry value will be written to the key specified in the header; in the latter case, it will be deleted.

If the entry is situated below a `COMMANDS` section header, the entry defines a command to be executed by the system (e.g., `del C:\temp\*.*`). The format of such an entry clearly depends on the command to be executed; `MCConfigFromFile()` reads such entries as literal strings.

*Comment*

Defines a line to be ignored by `MCConfigFromFile()`. The purpose of such a line is purely to allow a configuration file to be documented internally. Any line that starts with a `#` or a `;` is treated as a comment.

To clarify all this, an example file is presented here:

```
# This file, called goose, contains machine-specific information for the
# workstation of that name.

# The section below writes the default username and removes the legal notice
# from the ...winlogon registry key.

[SOFTWARE\Microsoft\Windows NT\Current Version\winlogon]
DefaultUserName, Mike, REG_SZ
LegalNoticeCaption
LegalNoticeText

# Here, two bits of information needed by some bespoke software are written

[SOFTWARE\MyCompany\Bespoke]
User, Mike, REG_SZ
Department, Trading, REG_SZ

# Mike is given full control over all files on his computer.

[COMMANDS]
cacls C:\*.* /t /e /c /g Mike:F
```

When a configuration script is run on Mike's machine (called *goose*), a single line of code using `MCConfigFromFile()` will ensure that the contents of the configuration file are executed.

## The Module Code

Now we present the module code. As this is a significantly larger chunk of code than we have presented so far in the text, we will show each function separately, preceding each with a short explanation and suggestions as to where alterations might be useful. Unlike in previously shown scripts, we will not split the functions themselves with text comments; instead, all within-function comments will be shown within the source code. We hope that by now much of the code will be familiar, so it should be reasonably clear what is going on.

The first few lines of code constitute the header—lines without which it would not *be* a module. The only interesting bit here is the `@EXPORT` list, the list of names that are exported into the namespace of any Perl scripts that include the invocation `use MaintainAndConfigure`. You will notice that this list includes a list of all our functions and three constants from the registry module. Although Perl convention dictates that function names should *not* be exported because doing so contaminates the main namespace, we take the view that in this particular case, the convenience of being able to refer to a function without the `MaintainAndConfigure::` prefix easily makes potential namespace contamination worth the risk. The fact that all our function names start with the letters MC makes it very unlikely that name conflicts will pose a problem!

```
package MaintainAndConfigure;
require Exporter;
use Win32::Registry;
use Win32;
use Net::SMTP;

@ISA = qw(Exporter);
@EXPORT = qw(
    MCWriteLog
    MCRaiseError
    MCSetAdminMachine
    MCSetAdminEmail
    MCWriteRegistry
    MCDeleteRegistry
    MCInstallAutoLogon
    MCRemoveAutoLogon
    MCInstallService
    MCRemoveService
    MCGetIdentity
    MCUpdateScripts
    MCConfigFromFile
    REG_SZ
    REG_DWORD
    REG_BINARY
);
```

The first function that appears in the module is `MCWriteLog()`. The reason it comes first is simple: virtually everything else uses it. You will notice that the function (like many others) is prototyped; if anything other than a single scalar is passed as a parameter, an error condition will result. The contents of the function are very simple: it increments a variable, notes the current time, and writes this and the information passed as a parameter to the logfile. The variables `$LogFileName` and `$LogCount` are set up when the module loads (see the `BEGIN` function near the end of this chapter).

```
sub MCWriteLog($)
{
    my @t = localtime;
    my $time = "@t[3]\/@t[4]\/@t[5]:@t[2]:@t[1]:@t[0]";
    open LOGFILE, ">> $LogFileName";
    print LOGFILE "${logcount}\.$time - @_ [0]\n";
    close LOGFILE;
    $logcount++;
}
```

`MCRaiseError()` is another function used extensively by other code in the module. It takes one parameter, a string detailing the error condition, and performs a variety of operations on it: first, it writes a log entry (using the preceding function), then it sends a pop-up message to a nominated administration machine and emails the administrator. The name of the machine to send a message to and the administrator to email are set up as global variables by the `BEGIN` function. Of course, you could totally rewrite this function to get it to do something different; everything that uses it would then exhibit the new behavior. This is the great advantage of modules! In some circumstances, it might be desirable to add an extra parameter to the function that is used to determine whether an error should really cause a script to stop running or whether another, less-fatal reaction is required.

```
sub MCRaiseError($)
{
    MCWriteLog ($_[0]);
    my $errmsg = "A script on this workstation has failed.\n
                Please see log file for details.";
    `net send $admin_machine $errmsg`;
    print "$errmsg\n";

    # The lines below require the netutil module to be
    # installed for sending SMTP.
    # If it is not installed on your system, rem out these lines.

    my $smtp = Net::SMTP->new($smtp_host);
    $smtp->mail($script_address);
    $smtp->to($admin_email);
    $smtp->data();
    $smtp->datasend("Scripting has failed on $ENV{ComputerName}");
}
```

```

    $smtp->datasend($errmsg);
    $smtp->dataend();
    $smtp->quit;
    exit 0;
}

```

The next two functions, `MCSetAdminMachine()` and `MCSetAdminEmail()`, simply set two variables and log the change. We provide such wrapper functions only for `$AdminMachine` and `$AdminEmail` because these are the only two variables that you may well need to set to different values depending on the script that is running. Both the variables have default values; these are set in the `BEGIN` function.

```

sub MCSetAdminMachine($)
{
    $AdminMachine = ($_[0]);
    MCWriteLog("Script error messages will be sent to $AdminMachine");
}

sub MCSetAdminEmail($)
{
    $AdminEmail = ($_[0]);
    MCWriteLog("Script errors will be e-mailed to $AdminEmail");
}

```

`MCWriteRegistry()` is the wrapper for the registry functions. Apart from the calling convention (see the description earlier in this chapter), the main advantage of using this function over the built-in registry function is that error raising and logging are included here. Also, if you try to write a value to a nonexistent key, the key is automatically created; in some circumstances you may wish to alter this behavior.

```

sub MCWriteRegistry($$)
{
    # Create usable variable names.
    my $key = $_[0];
    my $RegHandle; #declare to force "my"; this is not initialized yet
    my @values = @($_[1]); # an array of arrayreferences

    if(!Win32::Registry::RegOpenKeyEx
        (HKEY_LOCAL_MACHINE,$key,NULL,KEY_ALL_ACCESS,$RegHandle))
    {
        # If opening of registry key is unsuccessful,
        # try creating a new key.
        if(!Win32::Registry::RegCreateKey
            (HKEY_LOCAL_MACHINE,$key,$RegHandle))
        {
            # Failing that, generate a fatal error.
            MCRaiseError("MCWriteRegistry was unable to open or create
                the required registry key");
        }
        MCWriteLog("Registry Key $key not found, so has been created");
    }
}

```

```

MCWriteLog("Registry key $key opened successfully");

# If we got this far, the key has been opened successfully.
# Therefore we can continue.
# So write each value pair, aborting mission if we can't.

# Each entry in @values is a name, value, type array.
foreach $arrayref(@values)
{
    # Dereference the three-part array and put the values into
    # useful variables.
    my ($name, $value, $type) = @$arrayref;

    # Write the value. If that fails, CLOSE THE KEY and raise an error.
    if(!Win32::Registry::RegSetValueEx
        ($RegHandle, $name, NULL, $type, $value))
    {
        Win32::Registry::RegCloseKey($RegHandle);
        MCRaiseError("MCWriteRegistry was unable
            to write the required registry value");
    }
    MCWriteLog("Value $name written");
}
Win32::Registry::RegCloseKey($RegHandle);
return 0; # Return an error code of 0 for success,
        # rather than letting the value of @_ be returned.
}

```

MCDeleteRegistry() is used to remove sets of registry values from a key; it cannot remove keys themselves (although this functionality could be added). Attempting to remove a value from a nonexistent key will cause an error to be raised.

```

sub MCDeleteRegistry
{
    my $key = shift @_;
    my $RegHandle; # Declare to force "my"; this is not initialized yet.
    my $value; # Declare to force a "my".

    if(!Win32::Registry::RegOpenKeyEx(HKEY_LOCAL_MACHINE, $key,
        NULL, KEY_ALL_ACCESS, $RegHandle))
    {
        # If key didn't exist, raise an error.
        MCRaiseError("MCWriteRegistry was unable to open
            the required registry key");
    }
    MCWriteLog("Registry key $key opened successfully");

    # If we got this far, the key has been opened successfully.
    # Therefore we can continue.
    # So delete each value:

    foreach $value(@_)
    {
        if(!Win32::Registry::RegDeleteValue($RegHandle, $value))

```

```

    {
    # Uncomment the following lines to make deletion failure a fatal error.
    # Win32::Registry::RegCloseKey($RegHandle);
    # MCRaiseError("MCDeleteRegistry
    #   was unable to delete the specified key");
    }
    MCWriteLog("Value $value deleted successfully");
}
Win32::Registry::RegCloseKey($RegHandle);
}

```

The following functions wrap code first seen in Chapter 2 into a module format. First is a function to set up autologon; the second disables it. An obvious improvement to be made to the former is to check whether the username, password, and filename sent as parameters actually exist. In the form presented here, registry keys are blindly written, regardless of whether they are sane!

```

sub MCInstallAutoLogon($$$)
{
    # First turn command-line parameters into a usable format.
    (my $scriptpath, my $username, my $password) = @_;

    # Ensure the logon account is enabled.
    `net user $username /ACTIVE:Yes`;
    if($?) # If this is not 0, account cannot be enabled,raise an error.
    {
        MCRaiseError("Unable to activate user account for autologon");
    }
    else
    {
        MCWriteLog("User account $username activated");
    }

    # Now write the relevant registry entries
    # These functions need no error checking as this is built in
    # to the registry function
    $key = 'SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon';
    @values=
    (
        ['AutoAdminLogon',1,REG_SZ],
        ['DefaultUserName',$username,REG_SZ],
        ['DefaultPassword',$password,REG_SZ],
        ['DontDisplayLastUserName',0,REG_SZ],
        ['RunLogonScriptSync',0,REG_DWORD]
    );
    MCWriteRegistry($key,\@values);
    MCDeleteRegistry
        ($key,'DefaultDomainName','LegalNoticeCaption','LegalNoticeText');
    MCWriteRegistry('SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce',
        (['AutoInstalledScript',$scriptpath,REG_SZ]));
    MCWriteLog("Script $script installed for autologon");
}

```

```

sub MCRemoveAutoLogon
{
    # First disable the logon account if the parameter is provided.
    if(@_)
    {
        `net user $_[0] /ACTIVE:No`;
        if($?) # if net command was unsuccessful
        {
            # If your environment does not require a fatal error here
            # rem out the line below, or replace with an
            # MCWriteLog call.
            MCRaiseError("Unable to disable user account");
        }
        else
        {
            MCWriteLog("User account $_[0] disabled");
        }
    }

    # Now set registry keys to disable autologon.
    my $key = 'SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon';
    my @values=
    (
        ['AutoAdminLogon',0,REG_SZ],
        ['DefaultUserName','','REG_SZ'],
        ['DefaultPassword','','REG_SZ'],
        ['DontDisplayLastUserName',1,REG_SZ],
        ['RunLogonScriptSync',1,REG_DWORD]
    );
    MCWriteRegistry($key,\@values);

    # If params 2 & 3 are provided, write a LegalNotice.
    if($_[1] && $_[2])
    {
        my @values=
        (
            ['LegalNoticeCaption',$_[1],REG_SZ],
            ['LegalNoticeText',$_[2],REG_SZ]
        );
        MCWriteRegistry($key,\@values);
    }
    MCWriteLog("Autologon disabled");
}

```

More module wrappers for Chapter 2 functions follow. The main thing of interest here is the regular expression that creates the service name by extracting the raw name from a fully qualified path and extension.

```

sub MCInstallService($)
{
    # Extract the "name" part of the parameter (e.g. strip the path and .pl).
    my $script_name = $_[0];
    $script_name =~ s/.*\\(\w+)\.pl/$1/;
}

```

```

# Use instsrv to install an instance of srvany (with a plausible name).
`instsrv $script_name srvany.exe`;

# Now write the registry parameters used by srvany.
$key = "SYSTEM\CurrentControlSet\Services\$script_name";
@values =
(
    ['Application', 'srvany.exe', REG_SZ],
    ['AppParameters', $_[0], REG_SZ]
);
MCWriteRegistry($key, \@values);
MCWriteLog("$script_name installed as a service");
}

sub MCRemoveService($)
{
    `instsrv $_[0] /delete`; # Use instsrv to remove service by name.
    if($?)
    {
        # If a fatal error is not appropriate here, use MCWriteLog instead.
        MCRaiseError("Unable to remove $_[0] service");
    }
    else
    {
        MCWriteLog("@_[0] service successfully deleted");
    }
}

```

MCGetIdentity() returns a computer name and IP address either by extracting this information from a workstation or reading from an *address.mac* file. Many of the code details should by now be familiar. Again the most interesting things to note are the various regular expressions used in name and MAC address matching.

```

sub MCGetIdentity
{
    # First check to see if a parameter was passed.
    if($_[0] != null)
    {
        # It wasn't, so get identity from the workstation.
        # First, the computer name - no error checking here
        # as this is NOT going to fail!
        my @name = `net config workstation`;
        $name[0]=~ s/.*\(\(.*)/$1/;
        my @lines = `ipconfig`;
        # Now get the ipaddress.
        foreach my $IP(@lines)
        {
            chomp($IP);
            if ($IP =~ /^s+IP.*/)
            {
                $IP =~ s/.*:\s(.*)/$1/;
                goto FOUND_IP; # Leap out of loop.
            }
        }
    }
}

```

```

    }
    # The following line will only execute if no IP address was found.
    MCRaiseError("Unable to deduce IP address");
    # A successful ipconfig lookup - return the Computer Name and
    # IP address.
    FOUND_IP: return ($name[0], $IP);
}
else # We use the address.mac file to get the information.
{
    # Open the registry key where the MAC address is stored.
    my $key = 'SOFTWARE\Description\Microsoft\Rpc\UuidTemporaryData';
    my $RegHandle;
    if(!Win32::Registry::RegOpenKeyEx
        (HKEY_LOCAL_MACHINE,$key,NULL,KEY_ALL_ACCESS,$RegHandle))
    {
        MCRaiseError("MCGetIdentity unable to open Registry
            key where MAC address is stored");
    }

    # Retrieve MAC address from the key.
    if(Win32::Registry::RegQueryValueEx
        ($RegHandle, 'NetworkAddress',NULL, $type, $rawmac))
    {
        $machinemac = unpack "H12",$rawmac;
        $machinemac =~ s/(\w{2})/$1-/g;
        chop($machinemac);
    }
    else
    {
        Win32::Registry::RegCloseKey($RegHandle);
        MCRaiseError("MCGetIdentity unable to retrieve MAC address");
    }

    # Finally, close the key.
    Win32::Registry::RegCloseKey($RegHandle);

    # Open the MAC address lookup file.
    open MACFILE, $_[0] || MCRaiseError("MCGetIdentity cannot
        open lookup file");

    # Loop through each line, looking for an entry
    # corresponding to @machinemac.
    $match=0;
    while (($line = <MACFILE>) && (!$match))
    {
        chomp($line);
        ($computername,$ipnumber,$macaddress) =
            split(/\s*[,;:]\s*/,$line);
        if($macaddress eq $machinemac)
        {
            $match = 1;
        }
    }
}

```

```

        close MACFILE;

        if($match)
        {
            MCWriteLog("Entry for MAC address $macaddress found");
            return ($computername,$ipnumber);
        }
        else
        {
            MCRaiseError("MCGetIdentity unable to find
                a MAC address match");
        }
    }
}

```

`MCUpdateScripts()` is effectively a module version of the stub code first seen in the second half of Chapter 3; the main difference is that (being a function) it takes parameters. In many circumstances it might be beneficial (or even necessary) to modify this function so that it takes an extra two parameters that specify a username and password with which to connect to a remote share. In its current guise, the function uses the security context in which it is running to connect; this is no good for a script running in `LocalSystem` context.

```

sub MCUpdateScripts($$$)
{
    my $localpath = $_[0];
    my $remoteshare = $_[1];
    my $remotepath = $_[2];
    # Open the local directory, and create a hash table.
    opendir LOCAL, $localpath;
    @local_files = readdir LOCAL;
    foreach $file(@local_files)
    {
        if($file =~ /\.*\.pl$/) # We're only interested in .pl files.
        {
            $local_files{$file} = "x";
            #remember its the keys that matter
        }
    }
    @local_files = ""; # Be nice and conserve memory.
    closedir LOCAL;

    `net use z: /delete`; # Just to be sure.
    `net use z: $remoteshare`;
    if(!$?) # If we got the share mapped successfully.
    {
        opendir REMOTE, "z:$remotepath";
        @remote_files = readdir REMOTE;
        closedir REMOTE;
    }
    else
    {
        # Returning a nonzero value means the update isn't successful.
    }
}

```

```

        # If your circumstances require, you can use
        # MCRaiseError here instead.
        return 1;
    }

    foreach $remote_file(@remote_files)
    {
        # Check if matches a local file (hash key).
        if($local_files{$remote_file} ne undef)
        {
            #The file exists, so do the version checking.
            print "Found a match on $remote_file\n";
            #First get version number of local file.
            open FILE, "$localpath\\$remote_file";
            <FILE>; $localversion = <FILE>;
            $localversion =~ s/\$version=(\d+)/$1/;
            close FILE;
            # Now get version number of remote file.
            open FILE, "z:$remotepath\\$remote_file";
            <FILE>; $remoteversion = <FILE>;
            $remoteversion =~ s/\$version=(\d+)/$1/;
            close FILE;
            if($remoteversion > $localversion)
            {
                # If the server version is newer, replace it.
                `xcopy z:\\$remotepath\\$remote_file
                    $localpath\\$local_file`;
            }
            delete $local_files{$remote_file}; # delete hash entry
        }
    }

    # Every entry still in the hash table has its file deleted.
    # (They weren't on the server so must be obsolete.)
    @deletables = keys(%local_files);
    foreach $deletable(@deletables)
    {
        print "$deletable\n";
        `del $localpath\\$deletable`;
    }
    # Return a "success" code.
    return 0;
}

```

The following function, `MCConfigFromFile()`, is genuinely new in that the code has not appeared anywhere else in any guise. However, by this time, it should be fairly obvious how it works. Basically, we read the config file, ignore lines that start with `rem` characters, and set a variable called `$status` with the content of any lines that start with square brackets. We then assume that any other line is either a command to execute (if `$status` contains the word `COMMAND`) or that it is a registry value to write or delete. In the latter case, whether to write or delete

depends on how many commas the line contains. If we are in the least bit confused, an error is raised!

```

sub MConfigFromFile($)
{
    open FILE, $_[0];
    my $status = 0;
    foreach $entry(<FILE>)
    {
        chomp($entry);
        if(!($entry =~ /^#[\s]/))
        {
            # It's not a comment, so work out what to do with it.
            if($entry =~ /^\[\/)
            {
                # It's a section header, so strip the brackets.
                $entry =~ s/\[([.]*\)]/$1/;
                $status = $entry; # set the status.
            }
            else
            {
                if($status eq "COMMANDS") # It's a normal entry.
                {
                    ` $entry `; # Carry out the command,
                                # with error-checking.
                    # If you don't want this to cause an error,
                    # rem out this line.
                    if($?) {MCRaiseError("Unable to execute $entry");}
                }
                else # It's a registry entry.
                {
                    my ($name, $value, $type_string) =
                        split(/\s*,\s*/, $entry);
                    if($type_string) # It's a value to add.
                    {
                        # Now set the type of entry to write.
                        # The code only supports DWORD AND SZ.
                        # You may wish to change this
                        # by adding
                        # something else - it's here!
                        if($type_string == REG_DWORD)
                        {
                            $type = REG_DWORD;
                        }
                        else { $type = REG_SZ; }
                        MWriteRegistry($status, [$name, $value, $type]);
                    }
                    else # It's a value to remove.
                    {
                        MDeleteValue($status, $entry);
                    }
                }
            }
        }
    }
    close FILE;
}

```

Finally, we present two special functions: `BEGIN` and `END`. These are called when a package is invoked (e.g., a module with the `use` keyword) and when it is finished. As you can see, we use the former to set up some global variables and write a fairly useless log entry. The latter is used only to write a log entry; it exists more for reasons of demonstration than utility. The values of the variables in `BEGIN` will need to be customized for a specific environment.

```
BEGIN
{
  # Set up some generally useful (essential) variables.
  $LogFileName = 'C:\ghostlog.txt';
  $AdminMachine = 'myNTbox';
  $AdminEmail = 'root@mymachine.mydomain.com';
  $smtp_host = 'smtp.myorganisation.com';
  $script_address = 'AutoScripter@myserver.myorganisation.com';
  $logcount = 1;

  # Write a log entry - any other useful startup code can be added here.
  open LOGFILE, ">> $LogFileName";
  MCWriteLog ("MaintainAndConfigure module invoked");
  close LOGFILE;
}

END
{
  # Write a log entry - any other useful tidyup code can be added here.
  open LOGFILE, ">> $LogFileName";
  MCWriteLog ("MaintainAndConfigure module released");
  close LOGFILE;
}
```

## *Summary*

We started this chapter by introducing the concept of a custom module and explaining why you might want to write one. We then gave the specification for our own module, `MaintainAndConfigure`, in some detail before finally showing the implementation code itself. And this brings us to the end of the book.

We could, of course, have avoided presenting the module code in full here, suggesting instead that you download it from our web site or something along those lines. However, we have not done this and did not even consider doing so, as this would be totally contrary to the ethos of the book. The interest here is categorically *not* the module itself; instead, it lies in the ideas contained within it. If reading our code provokes you into writing your own variations on a theme or even writing a totally fresh module of your own, our effort would have been worth it. Throughout the book, our emphasis has been on techniques and ideas, general approaches rather than solutions to specific problems. We hope that by taking some of these ideas further, you can develop sophisticated configuration and



maintenance solutions of your own, tailored to your environment. Just as importantly, we hope you'll agree with us that a combination of Perl, coffee, and common sense can not only make your working practices more efficient but also can be rather good fun.

