The Regina Rexx Interpreter – Win32 Functions

Patrick TJ McPhee (ptjm@interlog.com)

version 1.5.1, 19 December 2003

Contents

1 Introduction

This paper describes a small collection of functions which provide access to the NT registry, event log, service control manager, clipboard and shell, and to a large set of applications through OLE Automation. Although this documentation consistently refers to NT, the functions which are not NT-specific (ie, registry, shell, and OLE) should also work with lesser Win32 implementations.

Many of the functions originally appeared as part of the Regina port performed by Ataman software, which has been shipped with the NT resource kit. I have changed the syntax of some of the functions. In most cases, the changes are upwardly compatible (new optional arguments), however there is one fairly pervasive change which will require changes to existing scripts. Ataman's port raises a syntax error under certain odd circumstances, for instance when you query the value of a registry entry, but there is no registry entry to be found. This version of the functions will generally set the special variable RC to 1 on failure or 0 on success.

The effect of this change is that code like this:

```
signal on syntax errone
failed = 0
call function arg1,arg2,arg3
contone:
  if failed
    say 'it failed'
...

errone:
  failed=1
  signal contone
```

will no longer work, and has to be replaced by code like this

```
call function arg1,arg2,arg3

if RC
  say 'it failed'
...
```

I did this because I think it's an improvement. In general, the current version of Regina maintained by Mark Hessling has many improvements in terms of stability, functionality, and ANSI compliance over the version on the resource kit, and the resource kit version should never be used.

I initially dropped w32expandenvironmentstrings because I believed it to duplicate the (more portable) value function. value('PATH',,'ENVIRONMENT') returns the path, for instance, but then I found that w32expandenvironmentstrings was quite useful after all, so it's back as of release 1.3.0.

The functions are provided in hopes that they will be useful, but there is no warranty.

1.1 Reporting Bugs

Theorem A: Every program can be reduced by at least one line.

Theorem B: Every program contains at least one bug.

Corollary: Every program can be reduced to one line which doesn't work correctly.

w32funcs undergoes very little testing before new releases are shipped. I have not had the time to produce a regression test, for instance, and although it is on my list of things to do, the pressures of work and life keep me from doing it. Since the first release of w32funcs (1.0.0) in February 1998, there have

been surprisingly few bugs discovered, given the amount of testing it undergoes at this end. When bugs are reported, I do my best to fix them and to get a new release out within a short time. My time tends to be very tight, though, so I can't make any guarantees.

If you do find a bug, an error in the documentation, or you simply have a suggestion for improving the distribution, please send me details at ptjm@interlog.com. It's useful to know the operating system you're using, the version of Regina, and the version of w32funcs, and to have a set of steps for reproducing the bug.

If you are using w32funcs for a serious purpose and therefore take the time to produce a test suite for your own use, I would appreciate it if you'd contribute it to the cause.

1.2 Using RxFuncAdd

All the routines in w32funcs can be loaded either directly using RxFuncAdd, or indirectly using w32-LoadFuncs. RxFuncAdd takes three arguments – the name of the function as it will be used in the rexx program, the name of the library from which to load the function, and the name of the function as it appears in the library. This last argument should match the case of the actual function library (all function names in w32funcs use lower-case only). The names of the functions in the rexx script itself can take any case.

RxFuncAdd returns 0 on success, or 1 on failure. Regina has a function called RxFuncErrMsg which can give useful information about the reason for a load failure. A few common reasons for failure are:

Path issues: w32util.dll needs to be in the path, or in the directory containing regina.exe.

Windows 95: early releases of windows 95 did not include msvcrt.dll, the C run-time library used by w32funcs. This library is sometimes installed with applications software. It can also be obtained through service packs, or from the Microsoft web site.

Rexx.exe: Regina includes two executables, one called 'rexx', and the other called 'regina'. The difference is that 'rexx' includes the Rexx interpreter as part of the executable, while 'regina' loads the interpreter from a shared library. RxFuncAdd works only with the 'regina' version of the interpreter (the 'rexx' version is slightly faster, though).

1.3 Licencing

w32funcs is distributed free of charge in the hopes that it will be useful, but without any warranty. The original code by Ataman and my modifications to it are distributed under the terms of the GNU Library General Public License. As of version 1.3.1, my original code is distributed under the terms of the Mozilla Public License. The precise details of the licence are found in the file MPL-1.0.txt in the distribution.

If you use the library purely as distributed by me, then you can cheerfully ignore the licencing change. If you modify the source code or adopt portions of it in your own programs or libraries, you should be aware of which licence applies to which code, and fulfill your obligations under the licences. I believe that the restrictions placed by the Mozilla licence are less onerous than the ones in the GNU Library licence, and they are more in the spirit in which I would like my work to be distributed.

Although there are no obligations or restrictions related to use of the library, I would prefer that you do not use w32funcs in applications which cause injury or hardship to others. Also, if you derive a significant monetary benefit from the use of w32funcs, please share a portion with someone less fortunate (for instance, you could give money to Unicef).

1.4 Numeric Index Convention

Some functions in this library process stem variables using the 'numeric index convention'. This is a common method of treating stems as arrays with integer indices. The tail 0 contains the number of elements in the array, *n*, and the tails 1 to *n* are the array elements. For instance, you can print all the elements of an array *a* with

```
do i = 1 to a.0
  say a.i
  end
```

2 Housekeeping Functions

2.1 W32Load/DropFuncs

Unlike the original Ataman functions, these w32 functions are not built-in to Regina. To use any of the functions, they must be loaded using RxFuncAdd. If you are using more than one or two functions, it will be more convenient to simply load all the functions by loading and calling w32LoadFuncs, as follows:

```
call rxfuncadd 'w32loadfuncs', 'w32util', 'w32loadfuncs'
call w32LoadFuncs
```

If you feel the need to unload the functions, you can do so by calling w32DropFuncs:

```
call w32DropFuncs
```

As long as w32DropFuncs has been loaded itself, you can safely call it, even if all the functions you've loaded were loaded using RxFuncAdd. It will unload any functions which have been loaded, and silently ignore the ones that haven't been. I don't know of any advantage to loading each function individually over calling w32LoadFuncs.

2.2 W32Version

Beginning with version 1.2.4, the function w32version( ) returns a version tag, and the full path to the utility library. The version tag has the format *version.release.modification*. It is separated from the file name by a space. You may want to use the version number, for instance, to take alternate code paths if there is a bug in some version of the library which you might have to support. The file name is meant to help identify conflicts when more than one version of a library is on a machine.

```
parse value w32version() with ver '.' rel '.' mod file

say 'w32 functions loaded from' file
if ver > 1 | rel > 1 | (rel = 1 & mod > 3) then
   say 'huzzah!'
```

3 Error Messages

The functions generally report success or failure without giving details. Two functions are provided which might be helpful in determining the cause of function failures, and one which might be helpful in tracing the progress of your programs.

3.1 w32geterror

```
w32geterror() -> string
```

w32geterror returns error text related to the last failure of a WIN32 API function. This can sometimes be helpful in debugging applications.

The format of the error message is *number*: *text*, where *number* is the return code from the API function GetLastError(), and *text* is the generic system error message for that error code, which is supposed to be in the native language of the user.

3.2 w32olegeterror

```
w32olegeterror() -> string
```

w32olegeterror returns error text related to the last failure of the OLE functions. This can sometimes be helpful in debugging applications. I expect the format of this error string to change in the future.

3.3 w32debug

```
w32debug(message) -> 0
```

w32debug sends its argument to a program which has registered as a debugger either for the system or for your program. If there is no such program, the function does nothing. It can be used in conjunction with a program such as DbgView from http://www.sysinternals.com to provide a rudimentary tracing facility.

4 Registry Functions

The registry is a hierarchical database used to store configuration information. It is made up of keys, each of which can contain one or more keys and values. If you think of the registry as a sort-of file system, the keys correspond to directories, and the values to files. One uses registry functions by first opening a key, querying or setting values using the key, and then closing the key. To get a feel for what the keys are all about, run regedit (or regedt32) and look around.

There are four standard keys which are the roots of all the other keys. These four keys are always open. They are identified by the strings HKEY_LOCAL_MACHINE, HKEY_CLASSES_ROOT, HKEY_CURRENT_USER, and HKEY_USERS. Since the 'HKEY_' prefix only exists to differentiate manifest constants used to define these keys in the Windows SDK, they are optional in the Rexx interface (*e.g.*, you can refer to the local machine key as LOCAL_MACHINE). All other keys must be opened, and the 'key' parameter returned from w32RegOpenKey or w32RegCreateKey must be used in any function call referring to the key.

Each key and each value has a name, which must be unique within its key. The name of a value can be an empty string. The names are not case sensitive.

You can use the registry to store configuration information for your applications. By convention, use a location such as HKEY_LOCAL_MACHINE\Software*your company name**your application name*. Under some circumstances you can also change parameters for other people's applications, but this is not a good idea unless you know what you're doing. You can also connect to registries on other machines, which makes it possible to perform centralised configuration.

4.1 List of Registry Functions

w32regopenkey	(key, [subkeyname]) → key: opens a key
w32regcreatekey	(key, subkeyname) → key: creates a key
w32regconnectregistry	(hostname,key) → key: open a registry on another machine
w32regclosekey	(key) → 0 or 1: closes a key
w32regqueryvalue	(key, entry, [type], [stem]) → value: retrieves a value
w32regqueryvaluetype	(key, entry) → type: retrieves a value's type
w32regsetvalue	(hkey, entry, type, data, [stem]) → 0 or 1: sets a value
w32regdeletekey	(hkey, subkeyname) → 0 or 1: deletes a key

w32regdeletevalue (hkey, valuename) → 0 or 1: deletes a value

w32regenumkey (hkey, [index], [stem]) → keyname: gets the names of a key's keys

w32regenumvalue (hkey, [index], [stem]) → value: gets the names of a key's values

w32regflushkey (hkey) → 0 or 1: flushes values to the registry

w32reggetkeysecdesc (hkey) → secdesc: gets a key's security descriptor

w32regsetkeysecdesc (hkey, secdesc) → 0 or 1: sets a key's security descriptor

w32regqueryinfokey (hkey, infokey, [stem]) → desired info: retrieves a variety of information about a key

w32regsavekey (hkey, filename) → 0 or 1: save a key and its sub-keys to a file

w32regrestorekey (hkey, filename) → 0 or 1: restore a key and its subkeys from a file

w32regloadkey (hkey, subkey, filename) → 0 or 1: attaches a 'hive' as a key;

w32regunloadkey (hkey, subkey) → 0 or 1: detaches a hive

w32expandenvironmentstrings (string) → string: expands environment variables in a string

4.2 Example

Here's a script which sets Regina up to run files with extension .rex, from the command-line.

```
/* make Regina the default processor for .rex files */

call rxfuncadd 'w32addfuncs', 'w32util', 'w32addfuncs'
call w32addfuncs

/* here's where regina is installed */
regexe='c:\bin\regina.exe'

/* here's the extension I want */
regext = '.rex'

/* open up the classes root key, and see if there's already a .rex
   there */
crkey = w32RegOpenKey("CLASSES_ROOT", regext)
if rc = 0 then do
  call w32RegCloseKey(crkey)
  call charout , regext 'is already defined. Overwrite it? '
  pull yesno
  if left(yesno,1) \= 'Y' then exit 1
  end

 /* now create the necessary sub-keys. I'll ignore rc until
    we set the value */
 crkey = w32RegCreateKey("CLASSES_ROOT", regext)
 if rc then do
   say 'failed to create HKEY_CLASSES_ROOT/'regext
   exit 2
   end
```

```
crskey = w32RegCreateKey(crkey, "Shell")
crsokey = w32RegCreateKey(crskey, "Open")
crsockey = w32RegCreateKey(crsokey, "Command")
if w32RegSetValue(crsockey, '', "REG_SZ", regexe '%1 %*') then do
  say 'failed to set up the registry key'
  exit 3
  end

call w32DropFuncs

exit 0
```

4.3 w32regopenkey

```
w32regopenkey(key, [subkeyname]) -> key
```

Opens a sub-key of an already-open key, or opens a new handle to an already-open key. *key* can be one of the special key names listed above, or the return code from a previous call to one of the key opening routines. If *subkeyname* is not specified, a new handle to the key specified by *key* will be returned.

If the function is not successful, it returns 0.

4.4 w32regcreatekey

```
w32regcreatekey(key, subkeyname) -> key
```

Creates or opens a sub-key of an already-open key. *key* can be one of the special key names listed in section 4 above, or the return code from a previous call to one of the key opening routines. If the sub-key already exists, it is opened, rather than created.

If the function is not successful, it returns 0.

4.5 w32regconnectregistry

```
w32regconnectregistry(hostname,key) -> key
```

Opens a connection to a registry key (only LOCAL_MACHINE and USERS are allowed) on the machine specified by *hostname*. Returns a handle to the key.

If the function is not successful, it returns 0.

4.6 w32regclosekey

```
w32regclosekey(key) -> 0 or 1
```

Closes a previously opened registry key. *key* must be the return code from a previous call to one of the key opening routines. If the function is not successful, it returns 1, otherwise, it returns 0.

4.7 w32regqueryvalue

```
w32regqueryvalue(key, entry, [{type}], [{stem}]) -> the value
```

Retrieves the value of the value named *entry* from the key identified by *key*. If the third argument is specified, it must evaluate to a variable name, and the type of the value will be written to that variable. See w32regqueryvaluetype, section 4.8, for a list of the possible types. If the *stem* variable is specified, it must evaluate to the name of a stem variable. If the value is of type REG_MULTI_SZ, the values will

be written to the stem variable, with the number of values in stem.0. Other values are always returned as strings, regardless of the type of data in the registry. For instance, a REG_DWORD value will be returned as an ordinary rexx integer, rather than as a 4-byte binary value.

If there is no such entry in the key, sets RC to 1 and returns the empty string. Otherwise it sets RC to 0.

4.8 w32regqueryvaluetype

```
w32regqueryvaluetype(key, entry) -> type
```

Returns the type of the value named *entry* from the key identified by *key*. There is no reason to use this function instead of the optional argument to w32regqueryvalue.

The possible types are

REG_BINARY	binary data
REG_DWORD	integer data
EG_DWORD_BIG_ENDIAN	integer data
REG_EXPAND_SZ	character data with environment variables
REG_LINK	symbolic link to another registry entry
REG_MULTI_SZ	multiple-string character data
REG_NONE	no type
REG_RESOURCE_LIST	resource list(?)
REG_SZ	character data

If there is no such entry in the key, sets RC to 1 and returns the empty string. Otherwise it sets RC to 0.

4.9 w32regsetvalue

```
w32regsetvalue(hkey, entry, type, data [, stemdata]) -> 0 or 1
```

Prefacatory remark: someone at Microsoft decided that the hierarchical nodes in the registry should be called keys, the key part of the key/value pairs should be called values and there should be no special name for the value part of the key/value pairs. That is the reason the next sentence reads the way it does.

Sets the data part of the value identified by *entry* in the key identified by *hkey*, or creates a new value. *type* must be one of the types specified in section 4.8 above, and must match the type of the value if it already exists in the registry. *stemdata* is the name of a stem variable which contains data for a REG_MULTI_SZ value. It is ignored for other types.

If the function is not successful, returns 1, otherwise it returns 0.

Data formats: REG_SZ, REG_EXPAND_SZ, REG_DWORD and REG_DWORD_BIG_ENDIAN data are entered as normal strings. The DWORD types should evaluate to integers. If the type is REG_MULTI_SZ and the stemdata argument is passed, the .0 element of the stem should hold the number of strings, *n*, and elements 1 to *n* should contain normal strings. Otherwise it will be treated as binary data. Other data types will be treated as binary data, which should be entered in hex. That is, the string 1234 would have the format `1234` if it were to be saved as REG_SZ, but `31323334` if it were to be saved as binary data.

4.10 w32regdeletekey

```
w32regdeletekey(hkey, subkeyname) -> 0 or 1
```

Deletes the sub-key named *subkeyname*. Returns 0 on success or 1 on failure. NT's delete key function will not delete keys unless they have no sub-keys. On Windows 95 and 98, keys can be deleted if they have sub-keys. As of version 1.2.3 of w32funcs, w32regdeletekey deletes keys recursively on all systems, except possibly Windows 2000.

4.11 w32regdeletevalue

```
w32regdeletevalue(hkey, valuename) -> 0 or 1
```

Deletes the value named *valuename*. Returns 0 on success or 1 on failure.

4.12 w32regenumkey

```
w32regenumkey(hkey, [index], [{stem}]) -> keyname
```

Either retrieves the name of the *index*th key from *hkey* and returns it, or retrieves the names of all keys in *hkey* and puts them in the stem variable identified by *stem*. *stem*.0 is set to the number of keys.

If the stem variable is used, returns 0 for success or 1 for failure. Otherwise, sets RC to 0 for success or 1 for failure.

4.13 w32regenumvalue

```
w32regenumvalue(hkey, [index], [{stem}]) -> value, 0 or 1 (sets rc)
```

Either retrieves the name of the *index*th value from *hkey* and returns it, or retrieves the names of all values in *hkey* and puts them in the stem variable identified by *stem*. *stem*.0 is set to the number of values.

If the stem variable is used, returns 0 for success or 1 for failure. Otherwise, sets RC to 0 for success or 1 for failure.

4.14 w32regflushkey

```
w32regflushkey(hkey) -> 0 or 1
```

Writes changes to a *hkey* to the registry. Normally, the registry routines cache changes, and they're not guaranteed to be written to disk at any particular time. Call w32regflushkey to ensure that the changes are written out.

Returns 0 for success or 1 for failure.

4.15 w32reggetkeysecdesc

```
w32reggetkeysecdesc(hkey) -> secdesc
```

Retrieves a pointer to a C structure which you absolutely cannot alter using Rexx. This function could be used in conjunction with w32regsetkeydesc to set a registry entry's security to be the same as some other registry entry's security. Otherwise it's useless.

Sets RC to 0 for success or 1 for failure.

4.16 w32regsetkeysecdesc

```
w32regsetkeysecdesc(hkey, secdesc) -> 0 or 1
```

Sets the security for *hkey* according to a C structure pointed to by *secdesc*. This function could be used in conjunction with w32reggetkeydesc to set a registry entry's security to be the same as some other registry entry's security. Otherwise it's useless.

Returns 0 for success or 1 for failure.

4.17 w32regqueryinfokey

```
w32regqueryinfokey(hkey, infokey, [{stem}]) -> desired info, 0 or 1
```

Retrieves information about a key. *infokey* can be one of NumSubKeys, MaxSubKeyName, NumValues, MaxValueName, or MaxValueData, meaning to return the number of subkeys, maximum length of a sub-key, number of values, maximum length of a value's name, or maximum length of a value's data, respectively. If the optional third argument is specified, it must be a stem variable name, and the members NumSubKeys, MaxSubKeyName, NumValues, MaxValueName, and MaxValueData of the stem variable will be set to the corresponding values instead.

If you use the stem variable, returns 0 for success or 1 for failure. Otherwise it sets RC to these values.

4.18 w32regsavekey

```
w32regsavekey(hkey, filename) -> 0 or 1
```

Saves the contents of *hkey* and its sub-keys to the file specified by *filename*.

Returns 0 for success or 1 for failure.

4.19 w32regrestorekey

```
w32regrestorekey(hkey, filename) -> 0 or 1
```

Adds the sub-keys and values listed in the file specified by *filename* to *hkey*.

Returns 0 for success or 1 for failure.

4.20 w32regloadkey

```
w32regloadkey(hkey, subkey, filename) -> 0 or 1
```

Creates a new sub-key named *subkey* under key *hkey*, which points at the file specified by *filename*. *hkey* must be one of LOCAL_MACHINE, USERS, or a key returned by w32RegConnectRegistry. The file must have been created by w32RegSaveKey. They call a key created with w32RegLoadKey a 'hive'. The data is stored n the file *filename*, rather than the normal registry files.

Historically, this function hasn't worked correctly, but it can be useful if you need to transfer registry information between two machines or access registry information associated with a defunct user. You need to have both the registry data file (*e.g.* software) and its associated log file (software.log).

Returns 0 for success or 1 for failure.

4.21 w32regunloadkey

```
w32regunloadkey(hkey, subkeyname) -> 0 or 1
```

Removes the sub-key called *subkeyname* from the registry tree. The sub-key must be a hive (*i.e.*, it must have been created using w32regloadkey).

Returns 0 for success or 1 for failure.

4.22 w32expandenvironmentstrings

```
w32expandenvironmentstrings(string) -> string
```

Returns the input string with all environment variables replaced by their values. The environment variables must be surrounded by percent-signs. The function is useful for processing registry entries which include embedded environment variables.

5 Event Log Functions

The event log is a centralised location for writing information about the progress made, and tribulations faced, by applications. In order to log events to the event log, you need to create an executable with message identifiers in its resources, and put the appropriate information in the registry. Doing all that goes beyond the scope of this documentation.

Once the messages have been installed, you can use the event log functions to write and retrieve event information.

5.1 List of Event Log Functions

w32openeventlog (host, source) → handle, or 0: open an event log

w32closeeventlog (handle) → 0 or 1: close an event log

2getnumberofeventlogrecords (handle) → the number, or -1: returns the number of events in a log

w32getoldesteventlogrecord (handle) → the number, or -1: returns the number of the first event in a log

w32openbackupeventlog ([hostname], filename) → handle, or 0: opens an event log back-up

w32backupeventlog (handle, filename) → 0 or 1: creates a back-up of an event log

w32cleareventlog (handle[, filename]) → 0 or 1: clears the contents of an event log

w32findeventlogentry (handle, recnum, [stem]) → 0 or 1: retrieves the information for an event

w32geteventid ( ) → id: returns the ID for the last event returned by w32findeventlogentry

w32geteventtype ( ) → type: returns the type for the last event returned by w32findeventlogentry

w32geteventcategory ( ) → category: returns the type for the last event returned by w32findeventlogentry

w32geteventnumstrings ( ) → num strings: returns the number of arguments for the last event returned by w32findeventlogentry

w32geteventtimewritten ( ) → time written: returns the time written for the last event returned by w32findeventlogentry

w32geteventtimegenerated ( ) → time generated: returns the time generated for the last event returned by w32findeventlogentry

w32geteventstring (index) → string: returns the *index*th argument for the last event returned by w32findeventlogentry

w32geteventdata ( ) → data: returns the message for the last event returned by w32findeventlogentry

w32writeeventlog ([hostname], source, [eventtype], [category], eventid, [data], [string1, ...]) → 0 or 1: writes an event log entry

5.2 w32openeventlog

```
w32openeventlog(host, source) -> handle, or 0
```

Opens the event log indicated by *source* on host *host*. If *host* is omitted, uses the local machine. Source is an entry from somewhere under the key LOCAL_MACHINE\System\CurrentControlSet\Services\Eventlog. For instance, it could be Application, Security, System, DrWatson, MSSQLServer, or some other application-dependent value.

On failure, it returns 0, otherwise it returns a handle to the event log for use in the event reading functions.

5.3 w32closeeventlog

```
w32closeeventlog(handle) -> 0 or 1
```

Closes an event log previously opened using w32openeventlog. Returns 0 on success or 1 on failure.

5.4 w32getnumberofeventlogrecords

```
w32getnumberofeventlogrecords(handle) -> the number, or -1
```

Returns the number of events in the log referred to by handle. On error, returns -1.

5.5 w32getoldesteventlogrecord

```
w32getoldesteventlogrecord(handle) -> the number, or -1
```

Returns the number of the oldest event in the log referred to by handle. Events are retrieved by record number, which doesn't necessarily start at 1. There are w32getnumberofeventlogrecords( ) records, numbered from w32getoldesteventlogrecord( ) to w32getnumberofeventlogrecords( )–1. On error, returns -1.

5.6 w32openbackupeventlog

```
w32openbackupeventlog([hostname], filename) -> handle, or 0
```

Opens an event log stored in the file referred to by *filename*. If *hostname* is not specified, the file is on the current host.

Returns 0 on failure, or a handle for use in the event reading functions on success.

5.7 w32backupeventlog

```
w32backupeventlog(handle, filename) -> 0 or 1
```

Saves an open event log to a file. Do this before clearing the event log. w32backupeventlog( ) will generally create a smaller output file than using the optional filename argument to w32cleareventlog( ).

Returns 1 on failure or 0 on success.

5.8 w32cleareventlog

```
w32cleareventlog(handle[, filename]) -> 0 or 1
```

Clears the contents of an event log, optionally backing it up first. w32backupeventlog( ) will generally create a smaller output file than using the optional filename argument to w32cleareventlog( ).

Returns 1 on failure or 0 on success.

5.9 w32findeventlogentry

```
w32findeventlogentry(handle, recnum, [{stem}]) -> 0 or 1
```

Retrieve the *recnum*th record (see section 5.5) from the event log referred to by *handle*. If the optional third argument is specified, it must be the name of a stem which will be populated with the event log information. Currently the indexes which are populated are: *source*, *id*, *type*, *category*, *numstrings*, *timewritten*, *timegenerated*, *data*, *description*, and *strings*.

id is the numeric identifier of the message. Together with *source*, it identifies the message. *Strings* is a stem variable containing the events arguments. *strings*.0 is the same as *numstrings*, and *strings*.1 to *strings*.numstrings are the arguments. *description* gives the full formatted text for the message in the default language of the user. *category* gives the application-specific category of the message, in the default language of the user, or 'None' if there is no application-specified category. *type* gives the type, which is one of 'Success', 'Information', 'Warning', 'Error', 'Success Audit', or 'Failure Audit'.

If the optional third argument is not specified, the event information can be retrieved using the w32getevent* routines. The format of the values returned by the w32getevent* routines is different from the format of the values returned by this routine. For instance, *timewritten* and *timegenerated* members is yyyymmdd hh:mm:ss (that is, date('S') time('N') format), and the *type* and *category* fields have their values converted to text. There are no corresponding w32getevent* functions for *source* or *description*, and none will be added.

Returns 1 on failure or 0 on success.

5.10 w32geteventid

```
w32geteventid() -> id
```

Returns the event id for the record retrieved in the last call to w32findeventlogentry. Use the stem argument to w32findeventlogentry instead. The id specifies the message text which should be used to display the event, plus a mask which indicates the kind of error. To get the same event id as returned by the event viewer and w32findeventlogentry, do something like this:

```
idx = x2c(right(d2x(w32geteventid()),8,'0'))
id = c2d(bitand(x2c('0000ffff'),idx))
```

On success, sets RC to 0, and on failure, sets it to 1.

5.11 w32geteventtype

```
w32geteventtype() -> type
```

Returns the event type for the record retrieved in the last call to w32findeventlogentry. Use the stem argument to w32findeventlogentry instead.

The possible values are 0: success, 1: error, 2: warning, 4: information, 8: audit success, and 16: audit failure.

On success, sets RC to 0, and on failure, sets it to 1.

5.12 w32geteventcategory

```
w32geteventcategory() -> category
```

Returns the event category for the record retrieved in the last call to w32findeventlogentry. Use the stem argument to w32findeventlogentry instead.

On success, sets RC to 0, and on failure, sets it to 1.

5.13 w32geteventnumstrings

```
w32geteventnumstrings() -> num strings
```

Returns the number of arguments for the record retrieved in the last call to w32findeventlogentry. Use the stem argument to w32findeventlogentry instead.

On success, sets RC to 0, and on failure, sets it to 1.

5.14 w32geteventtimewritten

```
w32geteventtimewritten() -> time written
```

Returns the time the record retrieved in the last call to w32findeventlogentry was written to the log. Use the stem argument to w32findeventlogentry instead.

The format returned by w32geteventtimewritten is ddd mmm dd yy:mm:ss yyyy, eg Fri Sep 26 17:42:23 1997.

On success, sets RC to 0, and on failure, sets it to 1.

5.15 w32geteventtimegenerated

```
w32geteventtimegenerated() -> time generated
```

Returns the time the record retrieved in the last call to w32findeventlogentry was generated. Use the stem argument to w32findeventlogentry instead.

The format returned by w32geteventtimegenerated is ddd mmm dd yy:mm:ss yyyy, eg Fri Sep 26 17:42:23 1997.

On success, sets RC to 0, and on failure, sets it to 1.

5.16 w32geteventstring

```
w32geteventstring(index) -> string
```

Returns the *index*th argument from the record retrieved in the last call to w32findeventlogentry. Use the stem argument to w32findeventlogentry instead.

On success, sets RC to 0, and on failure, sets it to 1.

5.17 w32geteventdata

```
w32geteventdata() -> data
```

Returns the optinal binary data from the record retrieved in the last call to w32findeventlogentry. Use the stem argument to w32findeventlogentry instead.

On success, sets RC to 0, and on failure, sets it to 1.

5.18 w32writeeventlog

```
w32writeeventlog([hostname], source, [eventtype], [category],
                 eventid, [data], [string1, ...]) -> 0 or 1
```

Writes an entry for application *source* to the event log for machine *hostname*. *Hostname* is the name of a machine with the event log, using the windows network naming convention. *Source* is the name under which a message catalogue has been registered.

Eventtype indicates the severity of the event. It can be one of ‘error’, ’warning’, ‘information’, or the corresponding numeric values 1, 2, and 4. The default value is ‘error’.

Category is an application-specific identifier which describes the type of message, or the application component which generated the message. If you define separate categories, you should create a category message file. The default value is 0.

Eventid is the numeric identifier of the event being logged.

Data is arbitrary binary data to store with the message. This might include information required to debug programming errors.

All arguments after *data* are strings which will be interpolated into the error message by the event viewer.

See section 5.19 for information about creating and registering message files. See http://www.kixscripts.com/resources/eventlogdll.asp for a generic event log message file (thanks to Mark Vilez for letting me know about this). I can't assist in using it, as I haven't looked at it yet.

5.19 Creating message files

To use w32WriteEvent effectively, you need to be able to create message files, which are dynamic link libraries containing specific resources. This is normally done using tools provided with your development environment: a message compiler, a resource compiler, and a linker.

If you have, for instance, the Microsoft C compiler, the steps are to create a message source file as described below, create a binary resource from it, and link the resource into a .dll. This set of commands is sufficient:

```
mc x.mc
rc x.rc
link /dll x.res msvcrt.lib
```

If you don't have a compiler, some of the necessary tools are available for free as part of the MinGW distribution of the GNU Compiler Collection. MinGW doesn't include a message compiler, though, so I've written and included one with this package: ptjmc. The equivalent command line is:

```
ptjmc x.mc
windres x.rc x.o
ld -dll x.o -o x.dll
```

5.19.1 Message Definitions

The remainder of this section describes the file format understood by ptjmc, which is intended to be a compatible subset of the file format understood by the microsoft message compiler.

A message file consists of optional message type and category definitions, followed by message definitions. There may be comments interspersed with the definitions.

Comments begin with a semi-colon (;) and continue to the end of the line.

The message type definition allows you to define symbolic names for the message type (which is called severity here). The format is

```
SeverityNames=(NoneName=0 InfoName=3 WarnName=2 ErrorName=1)
```

Where *NoneName et al* are to be replaced by the names you want to use. If you prefer, the numeric values can be given in hexadecimal notation, preceded by 0x, but the maximum value is 3 so this simply results in more typing. No space is permitted around the equals signs. Any amount of white space can separate the definitions. 'None', 'Success', 'Warning', and 'Error' always work unless you redefine them.

The category definition allows you to define symbolic names for the category (which is called facility here). If you use this feature, you should define message ids with severity Success and no facility for each facility name and register the library as a category message file. The facility names should be numbered sequentially starting at 1.

Message definitions have this format:

```
MessageId=<decimal number>
Severity=<message type>
Facility=<category>
SymbolicName=<valid C identifier>
Language=English
<Message text
which may take more than one line>
.
```

Severity, Facility, and SymbolicName are all optional. The value of Language is ignored, however the language entry must be present as it introduces the message text. Only one language can be entered per file. If you need to create a multi-lingual message file, you either need to maintain the messages for each language in separate files and combine them, or invest in a better development environment.

The severity name must be one of the entries in the SeverityNames definition, or one of the default names mentioned above. The default is 'information'. The facility name must be one of the entries in the FacilityNames definition. The default is 0.

The SymbolicName is used in the C header file mentioned below, it must follow the rules for C symbolic names if you want to use this header file.

The message text is mostly just text, terminated by a period on a line by itself. Any newlines in the text are converted to spaces. Within the text, % is treated as a special character. It and the characters following it are replaced when the message is formatted according to the following rules:

Text	*Replaced by*
%%	Per-cent sign (%)
%.	Period (.). This is to allow the output to show a period on a line by itself.
%n	Line break.
%!	Exclamation point (!). This is to allow an exclamation point following a parameter substitution.
%*n*	The *n*th string parameter logged to the event log.
%*n*!*fmt*!	The *n*th string parameter logged to the event log, formatted according to *fmt*. *fmt* is a printf format specifier. Useful values are s (equivalent to leaving off the format specifier), –*l*s (left-justify the parameter within a space *l* characters wide), and 0*l*s (right-justify the parameter within a space *l* characters wide, and pad with zeros).
%0	Ends the format string. Without this, there is always a trailing space (or new-line if you use microsoft's message compiler). This is a good idea after category names.

5.19.2 Output Files

Given an input file *file*.mc, ptjmc generates three output files. None of them are needed once the message dll has been built.

file.h is a C-language include file which defines symbolic constants for each message having a SymbolicName entry. This simplifies the process of logging events from C programs, since the message id passed to the API is an amalgam of the severity, category, and message id. It's not needed for rexx programming.

file.rc is the resource compiler input file. It defines a default language and defines binary resource 1 to be the third output file.

file.bin is the compiled message file.

6 OLE Automation

OLE Automation is a mechanism which allows applications to be scripted using any scripting language. That is, an application which supports automation can execute macros written in Rexx, even if the original application programmer has never heard of Rexx.

To use the automation functions, you need to know what objects and methods are supported by the application you want to automate. Sometimes, the application's author will provide you with documentation which explains what objects, methods, and attributes are available to you.

The automation interface may be enhanced in the future to provide more information about the objects supported by applications.

6.1 List of Automation Functions

w32createobject	(ProgramId) → handle or 0
w32releaseobject	(handle[, handle, ...]) → 0 (success) or 1 (failure)
w32olecleanup	( ) → nothing
w32getobject	([FileName], [ProgramId]) → handle or 0
w32olegetid	(handle, name[, name2, ...]) → dispid [dispid2 ...]
w32callfunc	(handle, name[, typelist, parm1, ...]) → value
w32callproc	(handle, name[, typelist, parm1, ...]) → 0 or 1
w32getproperty	(handle, name[,typelist, parm1, ...])) → value
w32getsubobj	(handle, name[, typelist, parm1, ...]) → handle or 0
w32putproperty	(object, name, typelist, value) → 0 or 1
w32olegetarray	(handle, name, stemname) → 0 or 1
w32oleputarray	(handle, name, stemname) → 0 or 1 [not implemented]
w32olenext	(object[, skipcount) → handle or 0
w32olegeterror	( ) → string

6.2 w32createobject

```
w32createobject(ProgramId) -> handle or 0
```

Given a registered object name, open an object of that type, and return its handle. Returns 0 on failure. You find out this object name in the same place you find out the method names and so forth. If you feel like hacking (what could go wrong?), look at the sub-keys of the CLASSES_ROOT registry key. *programId* can be given as a name (as in 'DXImageTransform.Microsoft.Chroma') or as a CLSID ('421516C1-3CF8-11D2-952A-00C04FA34F05').

6.3 w32releaseobject

```
w32releaseobject(handle[,handle, ...]) -> 0 (success) or 1 (failure)
```

Releases one or more handles acquired with w32createobject, w32getobject, or w32getsubobj.

It's very important that you release every object that you get a handle to. If you don't, you won't be able to close the applications that implement the objects, and it will cause problems. w32olecleanup( ) can be called to release all handles to OLE objects. w32releaseobject( ) is useful for portability with releases prior to 1.3.0 and for releasing objects as soon as they are no longer needed.

The ability to specify more than one handle in a single call is a new convenience feature in version 1.3.1.

Returns 0 on success or 1 on failure.

6.4 w32olecleanup

```
w32olecleanup()
```

Releases all handles acquired with w32createobject, w32getobject, or w32getsubobj. It's very important that you release every object that you get a handle to. If you don't, you won't be able to close the applications that implements the objects, and it will cause problems. Regina itself may fail to exit if it holds references to OLE objects. Calling w32olecleanup at the end of each program ensures that this will not happen.

There is no return code from w32olecleanup. The function first appeared in version 1.3.0, so should not be used if compatibility with earlier releases is required.

6.5 w32getobject

```
w32getobject([FileName], [ProgramId]) -> handle or 0
```

Given an existing file, or an ID of a running program, opens the relevant application and returns a handle. See w32createobject, section 6.2 for a discussion of *programID*. Note that *FileName* is generally speaking the name of a data file associated with an application. For instance, it might be the name of a spreadsheet data file, not the name of the spreadsheet executable.

```
wrd = w32GetObject('mydocument.doc')
excl = w32GetObject(, 'Excel.Application')
```

6.6 w32olegetid

```
w32olegetid(handle, name[, name2, ...]) -> dispid [dispid2 ...]
```

Given an object handle returned by one of the functions that returns object handles, return the dispid for one or more property or method names belonging to that object. The dispid can be used in place of the name when calling w32GetProperty and the other functions that take a name parameter.

Automation uses numeric identifiers to distinguish between different methods and properties. When an automation function, for instance, is called using its name, the library has to first look up the numeric identifier associated with the name and then call the function. Using identifiers instead measurably improves the performance of programs which use a lot of automation function calls.

More than one name can be passed to w32OleGetID, in which case more than one dispid is returned. These can be sorted out, for instance, using a parse template. There is no real performance penalty for making separate calls to w32OleGetID for each name.

The format of a dispid is @:[four binary bytes]. The binary bytes give the numeric identifier for the name, which will be the same every time for the same version of the same application. Nice application

vendors will have the identifier be the same for different versions of the same application. It may be beneficial to save the dispids in one run of an application, and then 'hard code' them in subsequent runs. For instance,

```
/* test run */
do until val = 0
  val = w32OleNext(hnd)
  id = w32OleGetID(val, 'MyProp')
  call lineout 'x.rex', 'id = "' || c2x(id) ||'"x'
  call w32callfunc val, id
  end

/* production run */
id = "403a27000000"x    /* good for v1.2 of myapp */

do until val = 0
  val = w32OleNext(hnd)
  call w32callfunc val, id
  end
```

6.7 w32callfunc

```
w32callfunc(handle, name[, typelist, parm1, ...]) -> value
```

Given a valid handle to an object, the name of a method of this object which returns a value, and some parameters, executes the method and gives its return code. Sets RC to 1 on failure.

As of version 1.5.1, *name* can be a dot-delimited path to a method of a sub-object. This can be helpful in situations where a deep tree of objects needs to be traversed to access an isolated method. For instance, rather than

```
  app = w32CreateObject('Example.ptjm')
  alpha = w32GetSubObj(app, 'Alpha')
  bravo = w32GetSubObj(alpha, 'Bravo')
  charlie = w32GetSubObj(bravo, 'Charlie')
  delta = w32CallFunc(charlie, 'Delta',, 'Echo', 'Foxtrot')
  call w32ReleaseObject alpha, bravo, charlie
```

one could have

```
  app = w32CreateObject('Example.ptjm')
  delta = w32CallFunc(charlie, 'Alpha.Bravo.Charlie.Delta',, 'Echo', 'Foxtrot')
```

In addition to being more compact, this approach will be faster in circumstances where you need only a single method or property of the sub-object.

typelist is a string in which each character corresponds to a parameter of the method, and it gives the type of the parameter. If no typelist is given, all parameters are assumed to be strings. *Typelist* is only necessary in situations where the type of the argument is ambiguous or is an array. For instance, if the argument can be either a string (say, a document name) or an integer (say, an index into an array of documents), there's no way to tell which is intended. The types are:

a array result. This must be the first argument to a function that returns an array of values. The corresponding argument must be the name of a stem, into which the array values returned by the OLE function will be placed. Note that this functionality is modified from some contributed code and is untested in its current form in version 1.5.1. I would appreciate some test cases;

A array argument. The corresponding argument must be the name of a stem, from which the array values will be taken. This functionality is currently unimplemented. I would appreciate some test cases;

b boolean (VT_BOOL);

c currency (VT_CY);

d date (VT_DATE);

i short integer (VT_I2);

I integer (VT_I4);

o object to dispatch (VT_DISPATCH) (the argument should have been returned from w32GetSubObj);

r float (VT_R4);

R double (VT_R8);

s string.

6.8 w32callproc

```
w32callproc(handle, name[, typelist, parm1, ...]) -> 0 or 1
```

Given a valid handle to an object, the name of a method of this object which does not return a value, and some parameters, executes the method. Sets RC to 1 and returns 1 on failure, 0 on success.

See w32callfunc, section 6.7, for a discussion of *name* and *typelist*.

6.9 w32getproperty

```
w32getproperty(handle, name[, typelist, parm1, ...]) -> value
```

Given a valid handle to an object and the name of a property of this object, returns the value of the property. Sets RC to 1 and returns 1 on failure, 0 on success.

Some properties allow non-default values to be retrieved by specifying parameters. As of version 1.3.0, it's possible to get at these values by specifying a *typelist* (see section 6.7) and parameters. See the same section for a discussion of *name*.

When a property is an object, use w32getsubobj to retrieve it.

6.10 w32getsubobj

```
w32getsubobj(handle, name[, typelist, parm1, ...]) -> handle or 0
```

Given a valid handle to an object, the name of a method or property of this object which returns another object, and some (possibly optional) parameters, executes the method (or evaluates the property) and returns a handle to the resulting object. Sets RC to 1 on failure.

See w32callfunc, section 6.7, for a discussion of typelist.

6.11 w32putproperty

```
w32putproperty(object, name, typelist, value) -> 0 or 1
```

Given a valid handle to an object, the name of a property of this object, a typelist, and a value, sets the value of the property. *Typelist* is not needed if the property is string-valued. See w32callfunc, section 6.7 for discussions of the *name* and *typelist* arguments. Returns 0 on success and 1 on failure, and sets RC to the same value.

6.12 w32olegetarray

```
w32olegetarray(object, name, stemname) -> 0 or 1
```

Note that this function is modified from code contributed by Alan Insley, and has not been tested in its current state. If you need to use it, please let me know if it works, and send me a test case.

Given a valid handle to an object, the name of a property which returns an array, and the name of a stem variable, sets the stem to match the array. If the array is one-dimensional, the stem follows the numeric index convention as discussed in section 1.4. If the array has more than one dimension, the stem follows a natural extension of the numeric index convention. *stem*.0 holds the cardinality of the first dimension, *stem*.0.0 holds the cardinality of the second dimension, and so forth. For a three-dimensional array, *stem.i.j.k* holds the element indexed by *i*, *j*, and *k*.

The only way to determine the dimension of one of these stems is to keep adding .0 to the stem name until it's no longer a number. I feel this is not a problem, since one ordinarily needs to know the cardinality of an array before one can use it.

Returns 0 on success and 1 on failure, and sets RC to the same value.

6.13 w32oleputarray

```
w32oleputarray(object, name, stemname) -> 0 or 1
```

This function has not yet been implemented. I'd appreciate examples of array properties so that I can test it out.

Given a valid handle to an object, the name of an array property of this object, and the name of the stem, sets the property to match the stem. See the previous section for a discussion of the format of the array.

6.14 w32olenext

```
w32olenext(object[, skipcount) -> handle or 0
```

Given a handle to a collection object, return the next object in the collection. If *skipcount* is 'Reset' (only the first character is signficant), the collection starts over from the beginning. If *skipcount* is a positive integer, that many entries in the collection are skipped.

Collection objects usually have a property called 'Count' and a method called 'Item', which takes an integer argument and returns the corresponding item in the collection. w32olenext( ) is an alternative way to iterate through these collections. In some cases, the nefarious cretins implementing the collection object don't bother providing 'Count' and 'Item', and w32olenext( ) is the only way to iterate through the collection. It corresponds to Visual Basic's 'foreach' operator.

You should iterate through the collection completely (*i.e.*, until w32olenext( ) returns 0), to ensure that all the objects referenced along the way are released.

6.15 w32olegeterror

```
w32olegeterror() -> string
```

w32olegeterror( ) returns error text related to the last failure of the OLE functions. This can sometimes be helpful in debugging applications. I expect the format of this error string to change in the future.

6.16 getoleclass Type Library 'Browser'

Applications which provide Automation interfaces sometimes document them using a set of OLE interfaces called a type library. A type library can be exported from a physical file, perhaps with the extension .tlb, or the extension .olb, or even the extension .dll, or by the Automation dispatch interface itself.

Getoleclass is a sort-of browser for type libraries. It takes as arguments either the program id of an Automation interface or the name of a type library file, and it writes information about the classes defined by that interface to the standard output. For instance, the classes defined by Microsoft Word can be written to a file with the command

```
getoleclass \msoffice\office\msword8.olb > word.classes
```

or with the command

```
getoleclass Word.Application > word.classes
```

More than one type library name can be put on the command line at a time, and wildcards can be used when dealing with file names, so you could have

```
getoleclass \winnt\system32\*.dll > fishing.classes
```

which would list all the classes defined in .dll files in the system32 directory.

The output format should be self-explanatory for the most part, but I suppose it helps to understand it if you wrote the program in the first place. I find it useful as a quick-reference listing of methods and properties. The remainder of this section gives a brief description of the output.

Each interface starts with two equal signs on a line by themselves, followed by the name of the type library. If the library was extracted from a file, the name is given as 'library *filename*', while if it was extracted from an automation interface, the name is given as 'class *progid*'.

Following the interface name is the definition of each class defined in the type library. Each definition starts with 'Type *typename*', and is followed by a list of the constants defined by that type, under the heading 'Constants:', a list of the properties which can be accessed without arguments, under the heading 'Properties:', and a list of the properties which require arguments and the methods, all under the heading 'Methods:'.

Each constant definition consists of the name assigned to a particular value, followed by an equals sign, followed by the numeric value. For instance, if you want to set a property to 'msoConnectorElbow', because the documentation says that this will give you the effect you desire, you can copy the line 'msoConnectorElbow = 2' from the getoleclass output into your program.

Each property definition consists of the name of the property, followed by (in parentheses), the word 'read' to indicate that the property can be read, the word 'write' to indicate that the property can be set, or 'read/write' to indicate that the property can be both read and set, and the type of the property.

Each method definition consists of the name of the method, followed by, in parentheses, a comma-delimited list of arguments, with the format '*name type*', followed by the type returned by the method. Optional arguments are in brackets.

The types which can be returned are 'number', meaning a number of some sort, 'date', meaning a date in the format given by the 'short' date and time formats in your nationalisation settings, 'string', meaning a character string, 'object', meaning an unspecified object (to pass this as an argument, you must have a handle created using w32GetSubObj), 'boolean', meaning –1 for 'true' or 0 for 'false', 'many', meaning more than one type is acceptable, 'unknown', which usually means some unspecified object type, or one of the types defined in the type library. Finally, some methods are procedures which return no value, so they are followed by the text 'no value'.

Watch out for property, method, and type names which start with an underscore. They generally indicate that I'm not handling something properly. The specific situations of which I'm aware are _NewEnum, which can show up as either a method or a property returning 'unknown'. You can't use _NewEnum directly – it indicates that the type is a collection which can be enumerated using w32OleNext. The return

type will generally be the same as the return type of the 'Item' method. Type names which start with an underscore seem to always be invisible subsidiaries of some other class. For instance, in Microsoft Word, there's a class called 'Document' which has two subsidiariy classes, 'DocumentEvents' and '_Document'. '_Document' gives the real methods and properties for the document class, but getoleclass doesn't handle this correctly. Sometimes, the definition of a subsidiary type is nowhere near the definition of its parent type, so if you need the definition of a type which apparently has no body, look for one with the same name, but starting with an underscore.

Properties and methods are really just different kinds of function calls, and sometimes properties are defined as taking arguments. When this happens, getoleclass lists them as methods. Sometimes it will list them twice, with different sets of arguments. I generally find that properties which are defined this way return an object of some sort, so I use w32GetSubObj to retrieve them, and the distinction between properties and methods doesn't matter. For other cases, all I can say is that the application-supplied documentation is probably more useful than the output of getoleclass.

Here are two of the five-hundred or so classes defined in msword8.olb.

```
Type ConnectorFormat
    Properties:
        Application (read Application)
        Creator (read number)
        BeginConnected (read MsoTriState)
        BeginConnectedShape (read Shape)
        BeginConnectionSite (read number)
        EndConnected (read MsoTriState)
        EndConnectedShape (read Shape)
        EndConnectionSite (read number)
        Parent (read object)
        Type (read/write MsoConnectorType)
    Methods:
        BeginConnect(ConnectedShape Shape, ConnectionSite number) no value
        BeginDisconnect() no value
        EndConnect(ConnectedShape Shape, ConnectionSite number) no value
        EndDisconnect() no value
Type MsoConnectorType
    Constants:
        msoConnectorTypeMixed = -2
        msoConnectorStraight = 1
        msoConnectorElbow = 2
        msoConnectorCurve = 3
```

What this says is type type 'ConnectorFormat' has 10 properties and 4 methods. The property 'Application' can be read, and is of type 'Application', which is defined elsewhere in the output of getoleclass. 'Creator' can be read, and is a number. 'Type' can be both read and written, and is of type 'MsoConnectorType', which is an enumeration with four valid values. You would use w32GetProperty to retrieve 'Creator', 'BeginConnected', 'BeginConnectionSite', 'EndConnected', 'EndConnectionSite', and 'Type', since these all return scalar values. You would use w32GetSubObj to retrieve the other properties, since they have object type (you'll have to trust me that 'Application' and 'Shape' refer to objects, while 'MsoTriState', like 'MsoConnectorType', is a scalar).

'BeginConnect' and 'EndConnect' both take an object of type 'Shape' and a number. We need to look at the documentation that comes with the application to find out what to do with them, though. Because these methods are procedures, they should be called using w32Callproc. If they returned a scalar value, they would be called using w32CallFunc, and if they returned an object value, they would be called using w32GetSubObj.

7 Service Control Manager

The service control manager is used to drive back-ground applications, such as databases and networking software. The service control manager interface can be used to add and remove service control manager entries, and to start and stop services. This can be useful for writing installation scripts, which is why the functions are here.

There is currently no support for implementing services in Rexx, although it seems like a good idea.

7.1 List of Service Control Functions

w32svcstart (name) → 0 or error code: starts the named service;

w32svcstop (name) → 0 or error code: stops the named service;

w32svcremove (name) → 0 or error code: removes the named service from the list of services;

w32svcinstall (servicename, displayname, programpath[, autostart, user, password]) → 0 or error code: installs a service.

7.2 w32svcstart

```
w32svcstart(name) -> 0 or error code
```

Starts the named service. Name can be either the registry key name (the first argument to w32svcinstall) or the display name (which is what shows up in the services applet). Returns 0 for success, 1 if the service was already running, -1 if the process could not attach to the service control manager, -2 if the service could not be opened with sufficent rights to query the status and start it, -3 if the service's status could not be queried, and -4 if the service did not start.

7.3 w32svcstop

```
w32svcstop(name) -> 0 or error code
```

Stops the named service. Name can be either the registry key name (the first argument to w32svcinstall) or the display name (which is what shows up in the services applet). Returns 0 for success, 1 if the service was not already running, -1 if the process could not attach to the service control manager, -2 if the service could not be opened with sufficent rights to query the status and stop it, -3 if the service's status could not be queried, and -4 if the service did not stop.

7.4 w32svcremove

```
w32svcremove(name) -> 0 or error code
```

Uninstalls the named service. Name can be either the registry key name (the first argument to w32svcinstall) or the display name (which is what shows up in the services applet). Returns 0 for success, -1 if the process could not attach to the service control manager, -2 if the service could not be opened with all possible rights, -3 if the service's status could not be queried, -4 if the service was running and did not stop, and -5 if the service could not be deleted.

7.5 w32svcinstall

```
w32svcinstall(servicename, displayname, programpath[, autostart,
              user, password]) -> 0 or error code
```

Installs the named service. servicename is the registry key for the service. displayname is the name that will display in the service control manager. programpath is the full path to the application, which must be written to support the service control manager. autostart is 1 if the service should start automatically at boot time, or 0 otherwise (default is not to start automatically). User is the userid that should be used to run the service, and password is the user's password (default is to use the local system account).

Returns 0 for success, -1 if the process could not attach to the service control manager or -2 if the service could not be created.

8 Shell

The shell is the user interface, and it controls the start menu, desktop, and probably other things I don't know about. Currently, the shell interface can be used to create and modify shortcuts. Since I mostly want to do this when setting up the start menu, I call the functions w32menu*x*.

8.1 List of Shell Functions

w32menuadditem ([menu], [submenu], display, path, [workingdir], [arguments], [icon number], [icon file], [hot key]) → 0 or error code: creates or replaces a shortcut;

w32menuedititem ([menu], [submenu], display, [path], [workingdir], [arguments], [icon number], [icon file], [hot key]) → 0 or error code: modifies a shortcut;

w32menuremoveitem ([menu], [submenu], display) → 0 or error code: removes a shortcut;

w32menumoveitem ([menu], [submenu], oldname, newname) → 0 or error code: renames or moves a shortcut;

w32menumove ([menu], submenu, [newmenu], [newsubmenu]) → 0 or error code: renames or moves a folder.

8.2 w32menuadditem

```
w32menuadditem([menu], [submenu], display, path, [workingdir],
               [arguments], [icon number], [icon file],
       [hot key]) -> 0 or error code
```

Adds an item to the start-up menu, creating any sub-menus that are needed along the way.
If *menu* is given, it can be

'all' put the item on the start menu for all users (the default)

'current' put the item on the start menu for the current user

'all desktop' put the item on the desktop for all users

'desktop' put the item on the desktop for the current user anything else treat as a directory name, which must exist.

If *sub-menu* is given, it must be a valid directory name. If the directory doesn't exist, it will be created. This name will show up on the menu as a sub-menu (or on the desktop or other directory as a folder). To create many nested sub-menus, enter the submenu as `'menu1\menu2'`. All intervening directories will be created.

If *sub-menu* is not given, the item will be put on the menu indicated by *menu*.

display is the name of the item that will appear on the sub-menu.

path is the full path to the program being put on the menu.

workingdir is the directory in which to start the program.

args are the program's arguments.

iconnumber is the number of the icon within the icon resource file which should be displayed on the menu item, starting at 0. The default is 0.

iconfile is the icon resource file which should be used. The default is the file given in *path*.

hotkey is a key-sequence which should bring up the program. The format is something like `'Alt-J'`. This isn't implemented yet.

If you call this function on an existing item, the item will be altered.

8.3 w32menuedititem

```
w32menuedititem([menu], [submenu], display, [path], [workingdir],
                [arguments], [icon number], [icon file],
[hot key]) -> 0 or error code
```

Changes the characteristics of a start-up menu item. The item is identified by its display name, given in the parameter *display*, and the menu and sub-menu on which it is found. See w32menuadditem, section 8.2, for a description of the parameters. Any parameter which is not specified is left unchanged by this function.

If you call this function on an item which does not exist, it will do nothing.

8.4 w32menuremoveitem

```
w32menuremoveitem [menu], [submenu], display
```

Removes the item from the submenu. See w32menuadditem for the meanings of the arguments.

8.5 w32menumoveitem

```
w32menumoveitem [menu], [submenu], oldname, newname
```

Renames a menu item. See w32menuadditem for the meanings of the arguments.

8.6 w32menumove

```
w32menumove [menu], submenu, [newmenu], [newsubmenu]
```

Renames or moves a menu. If *menu* is given, it is the old menu on which the item sits. *submenu* is the menu item. If *newmenu* is given, it is the new *menu* on which the item should sit (otherwise, the original menu is used). *newsubmenu* is the new submenu name. If it's not given, the menu item is moved but not renamed.

9 Program Execution

Programs can generally be executed by passing them to a rexx environment. The functions in this section use the function ShellExecuteEx( ) to start programs. This can change the way programs are executed in subtle ways, for instance by taking an alternative PATH from the registry.

The other functions in this section can be used to register a function which was not written specifically to support rexx with the rexx intepreter.

These functions were introduced in version 1.4.0.

9.1 List of Program Execution Functions

w32execute (file) → 0 or error code: executes a program;

w32executestem (file., result.) → 0: executes one or more programs and returns their return codes in the result stem;

w32funcadd (function, library, rexxname[, argtypes] [,returntype]) → 0 or 1: registers a function from a DLL;

w32funcdrop ([rexxname]) → 0 or 1: unregisters one or all functions registered using w32funcadd;

w32funcquery (rexxname) → 0 or 1: determines whether the named function has been registered.

9.2 w32execute

```
w32execute(file) -> 0 or error code
```

Executes the program *file*, or the program which is registered to process *file*, if *file* is not executable itself.

9.3 w32executestem

```
w32executestem(file., result.) -> 0
```

Executes the programs listed in the stem variable *file.*, and writes their return codes to the stem variable *result.*. *file.* must follow the numeric index convention, with the count of entries *n* assigned to *file.*0, and the entries themselves assigned to *file.*1, *file.*2, . . . , *file.n*.

9.4 w32funcadd

```
w32funcadd(function, library, rexxname[, argtypes]
           [,returntype]) -> 0 or 1
```

Registers the API function *function* from the shared library *library* as name *rexxname*. Once the function has been registered, it can be used as any other function. This is analagous to rxfuncadd, except that the functions do not need to be designed for use with rexx.

The routine works by putting a standard wrapper function around the API function. It's likely that performance will be worse than a dedicated rexx library (for instance, I would expect the Rexx util SysMoveObject to provide better performance than registering MoveFileExA with this function), but not every product provides a Rexx API with their interface library.

There are restrictions on the functions which can be called using this mechanism, and the whole approach is rather fiddly and not very well tested, but the worst thing that can happen is you can crash your machine and erase all the data, so as long as you keep regular back-ups, what can go wrong? Seriously, this function requires you to know the calling convention and data types used by the functions you want to call, and incorrect calls will be likely to crash your application.

Returns 0 on success or 1 on failure.

Data Types Many useful functions take only strings as arguments and return an integer. Such functions do not require the use of the *argtypes* and *returntype* arguments. Unfortunately, most functions are more complicated than that, and there's no standard, widely-supported way for the library to find out what the arguments are, so they must be specified.

We do this using an argument list which is similar to, but different from the type list used by w32callfunc (section 6.7). *argtypes* is a case-insensitive string of letters, most of which map directly to a corresponding argument (for instance, the first type in the string is usually assigned the value of the first argument passed to the function at run time). The types are:

i 32-bit integer. The corresponding argument is converted to an integer and passed to the API function;

h 16-bit (half) integer. The corresponding argument is converted to a short and passed to the API function;

f float. The corresponding argument is converted to a single-precision floating point value and passed to the API function;

d double. The corresponding argument is converted to a double-precision floating point value and passed to the API function;

s string. The corresponding argument is null-terminated and passed to the API function. This can be used for any function argument which takes a pointer for an input-only parameter, whether or not the pointer is treated as a character string. A null pointer can be passed by leaving out the argument;

r rexx string. The corresponding argument is passed to the API function exactly as it was passed from the interpreter (as a pointer to RXSTRING);

b buffer. This should be used when the API function expects a fixed-size, pre-allocated buffer. It can be followed by a number in brackets (*e.g.*, '[1000]'), in which case a buffer of that size will be allocated and passed to the API function. The default length is 255. The corresponding argument in the function call must be the name of a rexx variable. The variable will be evaluated before the function call and its contents placed in the buffer (the buffer will be expanded to the size of the variable value if required). After the call, the rexx variable will be set to the contents of the buffer and the buffer will be released;

p pointer. This must be followed by one of i, h, f, d, or s, indicating the type we're pointing to. The corresponding argument must be the name of a rexx variable. The variable will be evaluated before the function call, its value converted to the appropriate type, and a pointer to this value passed to the API function. Note that ps is handled differently. An argument of type ps is treated as a pure pointer. The rexx variable must evaluate to a 4-byte pointer value, and any pointer returned by the API function will be assigned to this variable as a pointer.

The function does not support structures or arrays. If you really need to call a function which takes a structure argument and you really can't write your own C wrapper function, then must construct a rexx variable that looks like the contents of this structure and use the b or s type.

Return Types The API currently supports only three return types: 32-bit integer, null-terminated string, and pointer. These are specified by passing 'i', 's', or 'p', respectively, as the *returntype* argument.

Examples The win32 API includes a function called GetUserName, which takes a pointer to a buffer and a pointer to an integer, and writes the current user name into the buffer and the length of the user name into the integer. The real name of the function we want to call is GetUserNameA, which is distinguished from the unicode version, which we don't want to call because we don't support unicode yet. The integer must be initialised to the buffer size, and will be changed to the length of the user name, including a terminating null. This would be registered like this:

```
call w32funcadd 'GetUserNameA', 'advapi32.dll', 'GetUserName', 'bpi'
```

and called like this. Note that *uname* is set to the full size of the buffer (255 since I've used the default value), and we must use *namel* to restrict it to the correct size.

```
namel = 255
call GetUserName 'uname', 'namel'
namel = namel - 1
say left(uname, namel) namel length(uname)
```

There's another function called MessageBox, which takes a window handle (which can be NULL), two strings and an integer, displays a message box, and returns an integer indicating which button was pressed to make the message box go away:

```
call w32funcadd 'MessageBoxA', 'user32.dll', 'MessageBox', 'sssi'

button = MessageBox( ,'Text', 'Title', 0)
```

Finally, MoveFile takes two strings, renames the first to the second, and returns an integer:

```
call w32funcadd 'MoveFileA', 'kernel32.dll', 'MoveFile'

call MoveFile file1, file2
```

9.5 w32funcdrop

```
w32funcdrop([rexxname]) -> 0 or 1
```

Unregisters the function which was registered under name *rexxname* using w32funcadd. If no name is passed, unregisters all functions. This is analagous to rxfuncdrop, except that this library performs reference counting, so a function which is added twice and unregistered once will still be registered.

If all functions from a particular library are dropped, the library will be unloaded from memory.

Returns 0 on success or 1 on failure.

9.6 w32funcquery

```
w32funcquery(rexxname) -> 0 or 1
```

Returns 1 if the named function is *not* registered or 0 if it is.

10 Common Dialogs

The win32 API windows includes functions which present dialogs for performing common tasks, notably selecting colours, files, and printers. As of version 1.4.1, I provide interfaces to some of these functions.

10.1 List of Common Dialog Functions

w32dlgopenfile (file.[, deffile][, filter.][, directory][, title][, flags]) → 0 or error: presents a 'file open' dialog;

w32dlgsavefile (file.[, deffile][, filter.][, directory][, title][, flags]) → 0 or error: presents a 'file save' dialog;

w32dlgchoosecolor ([red, green, blue]) → red green blue: presents a colour selection dialog;

w32dlgchoosecolour ([red, green, blue]) → red green blue: presents a colour selection dialog;

10.2 w32dlgopenfile

```
w32dlgopenfile(file.[, deffile][, filter.][, directory]
               [, title][, flags]) -> 0 or error
```

Presents a standard dialog for file selection. The user is able to navigate the system's directory structure, and the current directory can be set to the last directory selected.

file. is the name of a stem variable which is used to return the selected file or files, using the numeric index convention. If the 'Read only' box is ticked, the compound variable *file*.readonly is set to 1. Otherwise, it is set to 0. *deffile* is used as the default value for the 'File Name' edit box.

filter. is a stem containing a list of filters to be used for file selection. The *filter*.0 is the count of filters, as in the numeric index convention. For each *i* between 1 and *filter*.0, inclusive, *filter*.display.*i* is text which will be displayed in the 'Files of Type' list box and *filter*.wildcard.*i* is the wildcard which will be applied to file selection when the filter is selected. *filter*.default is the index of the filter which should be selected by default. If *filter*.default is not supplied, 1 is used. If *filter* is not supplied, or *filter*.0 is 0, the 'Files of Type' list box is left empty, and all files are displayed.

directory is the initial directory for file selection. If it is not specified, the current directory is used.

title is the title for the dialog box. If it is not specified, the default title is 'Open' (most likely in the user's default language).

flags is a set of space-delimited flags which controls the behaviour of the dialog box. Only the characters shown in bold face need to be specified (and are checked). The full names can be specified for readability, or made up names with the same prefix can be specified just for the fun of it. Any combination of flags can be specified, although some of combinations don't make sense.

AllowMultiSelect Allow more than one file to be selected;

CreatePrompt Prompt for confirmation if the user enters a file which does not exist;

FileMustExist Don't allow the user to enter a file which does not exist;

HideReadOnly Don't display the 'Read Only' check-box;

NoChangeDir Don't change the current directory based on the user's selection;

NoNetworkButton Don't display the network button;

PathMustExist Display a warning message box if the user types a file path where a directory does not exist;

ReadOnly Check the read-only check-box.

The function returns 0 if one or more files were selected, and sets the *file.* stem to contain the file names. It returns 1 if the user cancelled. If an error occurs, it returns, it returns a different non-zero value, which you should report as a bug.

10.3 w32dlgsavefile

```
w32dlgsavefile(file.[, deffile][, filter.][, directory]
               [, title][, flags]) -> 0 or error
```

Presents a standard dialog for file saving. This is identical to W32DlgOpenFile, except the default title is 'Save As', there is a 'Save' button, rather than an 'Open' button, the 'Read Only' parameters do not apply, and is one additional flag:

OverwritePrompt Present a message box indicating that the selected file already exists.

See section 10.2 for a description of the parameters.

10.4 w32dlgchoosecolor

```
w32dlgchoosecolor([red, green, blue]) -> red green blue
```

w32DlgChooseColor presents a standard colour selection dialog. The user is able to select a colour from either a pre-defined list or a kind-of lava lamp thing. The user can define custom custom colours which are preserved across calls.

If arguments are specified, they must be decimal integers between 0 and 255, giving the *red*, *green*, and *blue* components of a colour in the additive colour system.

The return code is the *red*, *green*, and *blue* components of the selected colour. If no colour was selected, *red* is set to –1. If an error occured, *red* is set to a negative error code, which should be reported since it should never happen.

10.5 w32dlgchoosecolour

w32DlgChooseColour is an equivalent function to w32DlgChooseColor for the convenience of people who spell 'colour' correctly.

11 System Parameters

The system parameters functions provide access to a variety of information abut the running system, and allow some of it to be changed. I've generally avoided adding functions which duplicate functionality in the RexxUtil library, and I've tried to stick to functionality which works with all versions from windows '95 up. These functions were introduced in version 1.3.0.

11.1 List of System Parameter Functions

w32sysgetusername ( ) → value: returns the current logged-in user's name;

w32sysgetcomputername ( ) → value: returns the (NetBIOS) name of the machine;

w32syssetcomputername (newname) → 0 or 1: sets the (NetBIOS) name of the machine;

32sysgethardwareprofilename ( ) → value: returns the name of the active hardware profile;

w32sysshutdown ([how][, force]) → 0 or 1: shuts down the system, or logs off the current user;

w32syssetpowerstate ([how][, force]) → 0 or 1: suspends operation of the machine;

w32sysgetpowerstatus ( ) → line battery batterypct batterytime batterylife: returns information about the system's power source;

w32sysgetosversion ( ) → major '.' minor build platform CSDVersion: returns information about the operating system version;

w32sysgetcolours (stem) → 0 or 1: returns the shell colours;

w32syssetcolours (stem) → 0 or 1: sets the shell colours;

w32sysgetparameter (parm) → value(s): returns the value of various system parameters;

w32syssetparameter (parm, permanent, value[, value2, ...]) → 0 or 1: sets the value of various system parameters;

w32sysshortfilename (filename [, filename2, . . .]) -¿ shortname [shortname2 . . .]: returns the short (8.3 with tildes) name associated with each real name passed as an argument;

w32syslongfilename (filename [, filename2, . . .]) -¿ longname [longname2 . . .]: the inverse of sysshortfilename.

11.2 w32sysgetusername

```
w32sysgetusername() -> value
```

Returns the name of the currently logged-in user. This is the short name (pmcphee), rather than the human name (Patrick McPhee).

11.3 w32sysgetcomputername

```
w32sysgetcomputername() -> value
```

Returns the name of the current machine. This is the NetBIOS name, rather than the TCP/IP host name, but the two are generally the same, although the host name sometimes includes a domain name.

11.4 w32syssetcomputername

```
w32syssetcomputername(name) -> 0 or 1
```

Changes the name of the current machine. You must have administrative privilige on the machine to do this, and you might need to make changes on the domain controller to connect to the network afterwards. Returns 0 if successful, or 1 otherwise.

11.5 w32sysgethardwareprofilename

```
w32sysgethardwareprofilename() -> value
```

Returns the name of the active hardware profile. Hardware profiles are collections of system settings which can be selected at boot time. They're used to select different system services, depending on how the user was feeling at the time the machine was booted. The most common use is to either start or not start networking software depending on whether the machine is hooked up to a network. This function could be used to prevent parts of a script from running if the system configuration is not appropriate.

Returns a zero-length string on error.

11.6 w32sysshutdown

```
w32sysshutdown([how][,force])
```

Shuts down the system, or ends the current login session.

how indicates how the system should be shut down. Only the first letter is significant. Possible values are:

reboot Shut down the system and reboot it;

poweroff Shut down the system and switch the power off. If there's no support for switching the power off, reboot instead;

logoff Log off the current session;

shutdown Shut down the system and wait for the system to be switched off manually. This is the default.

force indicates how to deal with applications. Only the first letter is significant. Possible values are:

force Don't give applications a chance to shut down cleanly. Use this as a last resort;

noforce Send applications a message and wait for them to shut down before taking down the system. This is the default.

11.7 w32syssetpowerstate

```
w32syssetpowerstate([how][,force])
```

Puts the system into suspended state, or takes it out of suspended state. *how* is either 'suspend' (the default) or 'nosuspend'. *force* is either 'force' or 'noforce'. Only the first character of each option is significant. Forced operation means that the system does not ask applications whether they're too busy to suspend operations just now.

The system must have support for suspended operation installed.

11.8 w32sysgetpowerstatus

```
w32sysgetpowerstatus() -> line battery pct time life
```

Returns the state of the system power supply.

line is 'Offline', indicating that it is running on battery power, 'Online', indicating that it's plugged in and running off the mains, or 'Backup', which is not documented but I'm guessing means that the machine is plugged in but running off a UPS.

battery gives an idea of how much charge there is in the battery. It can be 'High', 'Low', 'Critical', 'Charging', or 'NoBattery'.

pct gives an estimate of the battery charge remaining, as a percentage. It is 255 if there is no estimate.

time gives an estimate of the battery charge remaining, in seconds, or –1 if there is no estimate.

life gives an estimate of the life of a fully charged battery, in seconds, or –1 if there is no estimate.

w32sysgetpowerstatus returns a 0-length string when there is an error. It is not supported at all under Windows NT, and most likely needs some optional part of the system to be installed on the other win32 platforms.

11.9 w32sysgetosversion

```
w32sysgetosversion() -> major '.' minor build platform CSDVersion
```

Returns detailed information about the OS version. For portability, you should use the RexxUtil function SysVersion( ), however if you need more detailed information, w32SysGetOSVersion( ) gives all the information the system will disclose.

major and *minor* are the major and minor version numbers, respectively. *build* is the build number. *platform* is a numeric identifier indicating the platform. The values are 0, 1, and 2, meaning win32s, windows 95/98/ME, and windows NT/2000, respectively.

CSDVersion gives additional information about service packs, or who knows what else?

11.10 w32sysgetcolours

```
w32sysgetcolours(stem) -> 0
```

Populates the named stem variable with the values of the shell colours currently in effect. The colours are represented by strings with three space-separated numbers, giving the red, green, and blue indices. The colour indices range from 0 to 255. w32sysgetcolors is a synonym for this function name.

The colour names match the values in the CURRENT_USER\Control Panel\Colors registry key. They are:

ACTIVEBORDER the border of the active window;

ACTIVETITLE Tthe title bar of the active window;

APPWORKSPACE the background for multiple-document-interface windows;

BACKGROUND	the desktop colour;
BUTTONDKSHADOW	dark shadow for 3-dimensional elements;
BUTTONFACE	the face of buttons;
BUTTONHIGHLIGHT	synonym for BUTTONHILIGHT because I can't bring myself to spell high without the gh;
BUTTONHILIGHT	highlight colour for 3-dimensional elements;
BUTTONLIGHT	light colour for 3-dimensional elements;
BUTTONSHADOW	shadow colour for 3-dimensional elements;
BUTTONTEXT	the text on buttons;
GRAYTEXT	greyed-out text;
GREYTEXT	a synonym for 'GRAYTEXT';
HIGHLIGHT	a synonym for 'HILIGHT';
HIGHLIGHTTEXT	a synonym for 'HILIGHTTEXT';
HILIGHT	back-ground of selected items in a control;
HILIGHTTEXT	text of selected items in a control;
INACTIVEBORDER	the border of inactive windows;
INACTIVETITLE	the title bar of inactive windows;
INACTIVETITLETEXT	the title text of inactive windows;
INFOTEXT	the text of tool tips;
INFOWINDOW	the back-ground of tool tips;
MENU	the back-ground of menus;
MENUTEXT	the text of menus;
SCROLLBAR	the background of scroll bars;
TITLETEXT	the title text of the active window;
WINDOW	the background of single-document-interface windows;
WINDOWFRAME	the window frame;
WINDOWTEXT	default colour of text in windows.

Returns 0 on success and failure (windows doesn't provide a way to distinguish the two).

11.11 w32syssetcolours

```
w32syssetcolors(stem) -> 0 or 1
```

For each index of *stem* which corresponds to one of the colour names given in section 11.10, sets the corresponding system colour to the given value. If a colour doesn't appear in the stem, it is left alone. Colours are represented by strings with three space-separated numbers, giving the red, green, and blue indices. The colour indices range from 0 to 255. w32syssetcolors is a synonym for this function name.

w32syssetcolors does not change the colours permanently. To do that, you must set the appropriate values in the CURRENT_USER\Control Panel\Colors registry key.

11.12 w32sysgetparameter

```
w32sysgetparameter(parm) -> value
```

Retrieves the current value of the named system parameter. The table in section 11.13 gives the possible parameter names. Most of the system parameters have a single numeric value, but some of them have several – these are returned as a space-separated list. The table indicates how many parameters to expect.

Returns a zero-length string on error.

11.13 w32syssetparameter

```
w32syssetparameter(parm, options, [value[, value2[, ...]]]) -> 0 or 1
```

Sets the value of the system parameter named by *parm*. *options* is a set of flags, currently 'p', meaning that the change should be made permanently, and 'n', meaning that applications should be notified of the change. The number of values required depends on the parameter. In general, they should be integers, but there is one exception.

Some parameters can be set, but not retrieved, and vice-versa. The table indicates the parameter name, its function, the number of values it takes, and whether it can be get, set, or both. Parameters can be omitted, in which case they default to 0. The parameters are:

AccessTimeOut (Both, 2) Sets a time-out period after which the access parameters will be disabled. I suppose this would be useful if you don't like someone who needs the features, and you want to taunt them. The second argument is the time-out period in milliseconds. I don't have documentation, so I've no idea what to set the first value to. It seems like 1 turns the timeout on, while the flag value in the SDK for 2 is called 'onofffeedback';

FilterKeys (Both, 5) Turns on the filterkeys option. The first value is a set of flags. 1 means to turn the feature on, and other things can be added to that. I will read up on this some time and report back. The other options are timeouts in milliseconds: time to hold a key before it's accepted, time to wait before repeating, time to wait between each repetition, and time to wait between keystrokes before a duplicate is accepted;

MouseKeys (Both, 6) Turns on the mousekeys option, which allows the pointer to be controlled using the numeric keypad. The first value is flags. 1 means to turn the feature on, and again there are other values that could be added in. The next three options are the maximum cursor speed, the time in which to reach the maximum cursor speed, and another speed which doesn't seem to correspond to anything in the control panel. The remaining two parameters are not used currently;

ScreenReader (Both, 1) 1 if there is a screen reader active, or 0 otherwise;

ShowSounds (Both, 1) 1 if the showsounds option is on, 0 otherwise;

StickyKeys (Both, 1) A set of flags indicating the status of the sticky keys feature. 1 means that it's on, and there are others that I don't have documentation for;

ToggleKeys (Both, 1) A set of flags indicating the status of the toggle keys feature. As before, 1 means that it's on, and other values are also possible;

Cursors (Set, 0) Causes the system cursors to be reloaded;

DeskPattern (Set, 0) Causes the system to change the desktop pattern by reading the registry;

DeskWallPaper (Set, 1) Sets the name of the wallpaper file. The value is a string giving the full path to the file. If wallpaper is not currently in use, the options must be set to 'np' for it to take effect immediately, otherwise options doesn't need to be set (unless the change should be permanent, of course). To revert to the permanently-set wallpaper, omit the value. To remove the wallpaper, set the value to a zero-length string;

FontSmoothing (Both, 1) 0 if font anti-aliasing is off, or 1 if it's on. Anti-aliasing sets pixels to grey values to give the appearance of higher resolution text;

GridGranularity (Both, 1) I have documentation on this, but I still don't know what it is;

IconHorizontalSpacing (Both, 1) The width in pixels of an icon, for use in arranging icons in the 'large icon' view;

Icons (Set, 0) Reloads the system icons;

IconTitleWrap (Both, 1) 1 if icon text can be wrapped, or 0 otherwise;

IconVerticalSpacing (Both, 1) The height in pixels of an icon, for use in arranging icons in the 'large icon' view;

Beep (Both, 1) 0 if the warning beep is off, 1 if it's on;

KeyboardDelay (Both, 1) The amount of time to hold down a key before it starts repeating. The value ranges from 0 (250ms) to 31 (1s);

DoubleClickTime (Set, 1) The amount of time, in milliseconds, which can go by between mouse clicks for them to be considered part of the same double-click;

DoubleClkHeight (Set, 1) The vertical distance, in pixels, which the pointer can be moved between clicks of a double-click;

DoubleClkWidth (Set, 1) The horizontal distance, in pixels, which the pointer can be moved between clicks of a double-click;

KeyboardPref (Both, 1)

KeyboardSpeed (Both, 1) The spead at which characters are repeated when a key is held down. The value ranges from 0 (2.5/s) to 31 (30/s);

LangToggle (Set, 0) Forces the system to re-read the registry to get the language toggle value. The registry setting is current_user\keyboard layout\toggle, and the values are 1 (alt-shift), 2 (ctrl-shift), and 3 (none);

MouseButtonSwap (Set, 1) if the value is 1, sets the mouse buttons to have the opposite meaning from the original. Otherwise, restores the original meaning;

MouseHoverHeight (Both, 1) The height of a rectangle over which the pointer must be left for the system to consider it to be hovering;

MouseHoverTime (Both, 1) The amount of time the pointer has to be left over some rectangle on the screen for the system to consider it to be hovering;

MouseHoverWidth (Both, 1) The width of a rectangle over which the pointer must be left for the system to consider it to be hovering;

MouseTrails (Both, 1) Indicates the number of cursors to draw to enable the mouse trails feature. The value can be any whole number;

SnapToDefButton (Both, 1) 1 if the pointer should be set to the default button when a dialog is launched, or 0 otherwise;

WheelScrollLines (Both, 1) The number of lines to scroll when the mouse wheel is rotated;

LowPowerActive (Both, 1) 1 if low power screen saving is in effect, 0 otherwise. This does not work consistently in all windows versions;

LowPowerTimeout (Both, 1) timeout for low power screen saving (in seconds?). This does not work consistently in all windows versions;

PowerOffActive (Both, 1) 1 if power-off screen saving is in effect, 0 otherwise. This does not work consistently in all windows versions;

PowerOffTimeout (Both, 1) timeout for power-off screen saving (in seconds?). This does not work consistently in all windows versions;

ScreenSaveActive (Both, 1) 1 if screen saving is in effect, 0 otherwise;

ScreenSaverRunning (Both, 1) 1 if the screen saver is actually saving the screen. This does not work consistently in all windows versions;

ScreenSaveTimeout (Both, 1) timeout for screen saving, in seconds;

Animation (Both, 1) 1 if minimize and restore animiation are on, 0 otherwise;

Border (Both, 1) 'Border multiplier factor' which is used to indicate the size of the sizing border;

DragFullWindows (Both, 1) 1 if the entire window should be moved as a window is dragged from one location to another, or 0 if just the frame should be moved;

DragHeight (Set, 1)

DragWidth (Set, 1)

MinimizedMetrics (Both, 4) how minimized windows should be arranged, the first three values are numbers of pixels specifying, respectively, the width, horizontal separation, and vertical separation. The fourth value indicates where the arrangement should start and in which direction it should flow. The values for starting location are 0, 1, 2, and 3, meaning bottom-left, bottom-right, top-left, and top-right, respectively, while the values for flow are 0, 4, and 8, meaning horizontal, vertical, or hidden. The values can be combined, meaning a value of 6 would arrange icons starting in the top-left and flowing down;

PenWindows (Set, 1) 0 to unload pen windows, or 0 to load it;

WindowsExtension (Get, 1) returns 0 if Windows Plus! is not installed, or 1 if it is. This works only for Windows 95.

Returns 1 on error, or 0 on success.

11.14 w32sysshortfilename

```
w32sysshortfilename(fname1[, fname2[, ...][) -> shortname1 [shortname2 ...]
```

For each file name specified as an argument, w32sysshortfilename returns the 'short' version of the file name. This is a requirement for some old applications, some new applications, and parts of the operating system. It can also be useful with some applications that don't like spaces in file names (although the easier solution is to not put spaces in filenames).

```
parse value w32SysShortFileName(data1, data2, data3) with sdata1 ,
            sdata2 sdata3
'oldapp' sdata1 sdata2 sdata3
```

11.15 w32syslongfilename

```
w32syslongfilename(fname1[, fname2[, ...][) -> longname1 [longname2 ...]
```

For each file name specified as an argument, w32syslongfilename returns the 'long' version of the file name. This is useful when some application has returned a 'short' file name and you want a more useful one. Note that if you are the sort of person who puts spaces in file names, you should only pass one name as an argument (otherwise it could be difficult or impossible to parse out the results, which are space-delimited).

11.16 w32sysfullfilename

```
w32sysfullfilename(fname1[, fname2[, ...][) -> fullname1 [fullname2 ...]
```

For each file name specified as an argument, w32sysfullfilename returns the full path to a file, including the drive and directory. This does not change short file names into long names. Note that if you are the sort of person who puts spaces in file names, you should only pass one name as an argument.

12 Clipboard Functions

The clipboard is a buffer used to share information between applications. It's typically used to implement cut and paste operations in interactive applications. The clipboard can be useful for retrieving data from or sending data to an application which doesn't support automation, or in cases where human interaction is required to determine which data should be processed.

Only one piece of information can be stored in the clipboard at one time, but the information can be stored in many formats. There are several pre-defined clipboard formats, and user applications can register their own formats. Rexx applications can store or retrieve data in any format, however the application is required to understand and process the data itself in most cases. In particular, the length of the data returned by w32ClipGet is equal to the length of the buffer which was allocated to hold it. This is often longer than the the actual length of the user data, leaving random garbage at the end of the buffer. The rexx application should be programmed to detect the end of the data it's processing.

The clipboard is intended for interactive use, and it's not a good idea to use it as a general inter-process-communications mechanism. For one thing, the clipboard could be cleared and filled with new data between the time one process sets it and another tries to use it. For another, it is extremely annoying for a process to clear the clipboard without a user requesting it. For instance, a large-scale cut-and-paste between two applications can be completely ruined by rogue applications mucking up the clipboard between the two operations, and this can lead to frustration or violence on the part of the user.

The clipboard functions were introduced in version 1.5.0.

12.1 List of Clipboard Functions

w32clipopen ( ) → 0 or 1: locks the the clipboard open for use;

w32clipclose ( ) → 0 or 1: closes the clipboard;

w32clipgetstem (stemname) → 0 or 1: retrieves text and writes it to a stem;

w32clipget ([format]) → value: retrieves the data in the specified format;

w32clipsetstem (stemname) → 0 or 1: writes the contents of the stem as text;

w32clipset (value[, format]) → 0 or 1: writes the specified value in the specified format;

w32clipregisterformat (format) → value: registers a format name and returns its numeric identifier;

w32clipenumformat (stem) → 0 or 1: lists the numeric identifiers of formats which currently have data;

w32clipformatname (formatid) → value: returns the name associated with a numeric identifier;

w32cliptestformat (stemname) → id, 0, or –1: returns first available format from list;

w32clipempty ( ) → 0 or 1: empties the clipboard.

12.2 w32clipopen

```
w32clipopen() -> 0 or 1
```

Use W32ClipOpen to lock the clipboard for use by more than one other call. For instance, when setting data, you should open the clipboard, clear the clipboard, set the data in as many formats as you need, then close it.

It's not necessary to open the clipboard when making only one clipboard call, since the individual w32clip calls will open and close the clipboard as necessary. For instance, to retrieve the current text into a stem, it is sufficient to call w32ClipGetStem without explicitly opening or closing the clipboard.

12.3 w32clipclose

```
w32clipclose() -> 0 or 1
```

Closes the clipboard after a call to w32ClipOpen. Calls to w32ClipOpen and w32ClipClose are reference-counted, meaning that the clipboard will still be open if there are two calls to w32ClipOpen but only one to w32ClipClose.

One should close the clipboard as soon as possible after opening it, since other applications are not able to use the clipboard while it is open.

12.4 w32clipgetstem

```
w32clipgetstem(stemname) -> 0 or 1
```

Retrieves the current text from the clipboard and writes it into a stem named *stem*, using the numeric index convention with one line per index. Most applications write a text version of their data to the clipboard, and the operating system can perform automatic conversions from some other formats.

w32ClipGetStem assumes every line is terminated with CR (decimal 13) and LF (decimal 10). Some applications may terminate lines with either CR or LF by themselves, in which case the routine will not have the expected results.

There may be problems with character set conversion, especially on machines where the windows character set and OEM character set (the one used in DOS boxes) are set differently. Do write if you have trouble, but I strongly recommend setting the OEM character set to be the same as the windows characer set (search the registry for OEMCP and ACP).

12.5 w32clipget

```
w32clipget([format]) -> value
```

Returns the current data in the specified format. *format* can be either the numeric identifier of the format, or the format's name. The names are not case-sensitive. Use w32ClipRegisterFomat to retrieve the numeric identifier for a user-defined format.

w32ClipGet makes no effort to process the data it retrieves. In particular, the length of the return string is the size of the memory block used to hold the data, which in general will be larger than the

data itself. In some cases, this will not matter, but in others, the application has to process the data to determine the correct length.

I may elect to fix this situation for some or all of the standard formats in a future release.

There are some standard clipboard formats, and others which are specified by applications. You can find the names of the formats used by applications of interest to you by calling w32ClipEnumFormat and w32ClipFormatName. There are several standard formats, which do not have official names, but which have fixed numeric identifiers. For the purpose of this library, names have been assigned based on the manifest constants specified in the windows toolkit, and are listed below.

Name	*Identifier*	*Format*
TEXT	1	text, null-terminated
BITMAP	2	a bit-map resource
METAFILEPICT	3	windows meta-file
SYLK	4	symbolic link
DIF	5	Data Interchange Format
TIFF	6	tagged image file format
OEMTEXT	7	text in the OEM (DOS box) code page, null-terminated
DIB	8	Device-independent bitmap (a BMP without the first 14 bytes or so)
PALETTE	9	colour palette
PENDATA	10	input from a pen palette
RIFF	11	riff audio format
WAVE	12	wave audio format
UNICODETEXT	13	text in unicode, double-null-terminated
ENHMETAFILE	14	enhanced windows meta-file
HDROP	15	drag-and-drop handle
LOCALE	16	handle to a locale identifier
OWNERDISPLAY	128	data will be returned via call-back (don't use this)
DSPTEXT	129	text representation of private data
DSPBITMAP	130	bitmap representation of private data
DSPMETAFILEPICT	131	metafile representation of private data
DSPENHMETAFILE	142	enhanced metafile representation of private data

12.6 w32clipsetstem

```
w32clipsetstem(stemname) -> 0 or 1
```

Sets the clipboard text data to the contents of the stem *stemname*. The stem must use the numeric index convention. CR and LF are inserted between each stem value. The data is null-terminated.

12.7 w32clipset

```
w32clipset(value[, format]) -> 0 or 1
```

Sets the specified clipboard data format to the specified value. *format* can be a name or numeric identifier. The name can be one of the standard names listed in section 12.5, or any name which has been registerd using w32ClipRegisterFormat. The format name may be registered as a side-effect of this call.

value should be data in the format expected by other applications which understand format *format*. If the format is private to your applications, you should have some mechanism for determining the length of the data, since it's impossible to do so from the clipboard itself. w32ClipSet writes two nulls after the end of the data. This ensures that the data is correctly delimited when writing text- or unicode-format data.

The following example shows how the clipboard is typically set. It uses w32ClipOpen and w32ClipClose to lock the clipboard open, preventing other applications from slipping in and setting their own data. The

clipboard functions perform implicit opens and closes if necessary, so the w32ClipOpen and w32ClipClose calls can be left out in cases when only one call is being made.

```
/* lock the clipboard so the other operations happen 'atomically' */
if \w32ClipOpen() then do
   say 'Failed to open the clipboard'
   exit 1
   end

/* clear existing data */
call w32ClipEmpty

/* put a description for applications which don't process images */
call w32ClipSet 'A picture of a bug', 'text'
/* put put the image data as a device-independent bitmap, which is
 * everything after the first 14 (usually) bytes of a bmp file */
call w32ClipSet bugimage, 'dib'

/* close the clipboard to allow other applications to process it */
call w32ClipClose
```

12.8 w32clipregisterformat

```
w32clipregisterformat(format) -> value
```

w32ClipRegisterFormat makes a clipboard format available for use by an application. *format* is a name which is understood by one or more applications. For instance, 'Rich Text Format' is a name understood by MS Word, which refers to the word processing format of the same name. The function returns the numeric identifier by which the format is known to the clipboard. This identifier can be used only for the current session of windows.

w32ClipSet and w32ClipGet accept either the numeric identifier or the name, and the name will be registered by those functions if it is not already registered, so w32ClipRegisterFormat doesn't need to be called for many purposes. Its return code can be helpful either in interpreting the output of w32ClipEnumFormat or in slightly speeding up set and get operations.

12.9 w32clipenumformat

```
w32clipenumformat(stem) -> 0 or 1
```

Retrieves a list of formats currently available on the clipboard. This allows an application which understands more than one format to select the format best suited to its purposes. *stem* is the name of a stem, which will be set following the numeric index convention.

The example shows how this function can be useful, and calls into question the way the stem is set:

```
call w32clipopen

text = 1 /* from the table above */
rtf = w32ClipRegisterFormat('Rich Text Format')

call w32clipEnumFormat 'fmts.'
hasfmt. = 0
do i = 1 to fmts.0
   x = fmts.i
```

```
   hasfmt.x = 1
   end

if hasfmt.rtf then
  call rtf_processing w32ClipGet(rtf)
else if hasfmt.text then
  call text_processing w32ClipGet()
else
  call nodata_processing
call w32ClipClose
```

12.10 w32clipformatname

```
w32clipformatname(formatid) -> value
```

Returns then name associated with the numeric identifier *formatid*. If *formatid* does not refer to a registered clipboard format, returns the empty string.

12.11 w32cliptestformat

```
w32cliptestformat(stemname) -> id
```

Given *stemname*, an ordered list of registered clipboard formats, w32ClipTestFormat returns the numeric identifier of the first format in the list for which data is available on the clipboard. *Stemname* follows the numeric index convention.

If the clipboard is empty, w32ClipTestFormat returns 0. If none of the listed formats is available, w32ClipTestFormat returns –1.

w32ClipEnumFormat returns all the available clipboard formats, while w32ClipTestFormat reports the first format from the ones that you specify. This might be slightly more efficient if you have a lot of clipboard formats registered, but this function doesn't seem to add much, which is probably why it took me 9 months to notice that I'd left it out of the documentation. The example from section 12.9 could be written like this:

```
text = 1 /* from the table above */
rtf = w32ClipRegisterFormat('Rich Text Format')

fmt.0 = 2
fmt.1 = 'Rich Text Format'  /* preferred format */
fmt.2 = 'Text'              /* this will do */

gf = w32ClipTestFormat('fmt.')

if gf = 0 then say 'clipboard is empty'
else if gf = -1 then say 'no text data'
else if gf = text then call text_processing w32ClipGet()
else call rtf_processing w32ClipGet(rtf)
```

12.12 w32clipempty

```
w32clipempty() -> 0 or 1
```

W32ClipEmpty removes all data from the clipboard. One normally does this before writing any data, to ensure that all data formats in the clipboard refer to the same data. See w32ClipSet, section 12.7 for an example.

13 Example Programs

The example programs are intended to demonstrate the use of the different API functions. Some of them started life as test programs for new functions, but some are solutions to real problems. This section describes the intent and design of some of the examples.

13.1 regregina.rex

regregina sets up an association between the file extension .rex and the regina executable. After running it, files with extension .rex can be used from the command prompt as if they were ordinary executables (assuming the path to the regina executable is `d:\ptjm\bin\regina.exe`, which is not true on a surprising number of machines).

It was written to demonstrate the registry functions. The approach is to create a registry key called `HKEY_CLASSES_ROOT\.rex\Shell\Open\Command` and make its default value be a template which can be used to execute rexx scripts. See the documentation for NT's ftype command for a description, albeit an incomplete one, of the format for this template. In this case, the template is 'regina.exe %1 %*', which passes the name of the script and all of its parameters to regina.exe.

regregina starts by setting variables to hold the name of the rexx interpreter and the extension of rexx command files. It then checks to see if the necessary registry key exists by trying to open it. There's a weak attempt to allow the user to overwrite an existing association, followed by a series of w32RegCreateKey calls which create the necessary registry entries. The heart of the routine is

```
if w32RegSetValue(crsockey, '', "REG_SZ", regexe '%1 %*') then do
```

which sets the default value of the 'Command' key to the execution template.

Once the association is set up, the script adds '.rex' to the PATHEXT environment variable. Curiously, it retrieves the existing PATHEXT from the system environment key, modifies it, then gives the user the option of writing the result to either the system or user environment. It really ought to check for an existing PATHEXT in the user environment before overwriting it.

13.2 eventlog.rex

eventlog was written to demonstrate the event log reading functions. Usage is

```
eventlog [machine][/logname]
```

where *logname* is one of 'Application', 'Security', or 'System', and *machine* is the name of the machine for which the log should be retrieved.

The most important thing to know when traversing the event log is that it doesn't necessarily start at event 1. It's important to call w32getoldesteventlogrecord (see section 5.5) to get the range of event numbers.

13.3 evterr.rex

evterr was written because I sometimes get a message on my screen saying 'not all services started – check the event log'. evterr searches all the event logs on the local machine looking for error messages which were generated today, and prints them on the screen.

It loops backwards, so it prints the events in reverse order. I had originally intended to stop at the time of the last boot, rather than using the date as a boundary, but I got lazy, and it works for me.

13.4 findservice.rex

findservice lists services which run under a given user id. The usage is

```
findservice domain\user
```

This is useful in environments which force password changes. Apart from the fact that services can fail to run after a password change, if you generate enough failed logins from services, your account can be locked, which is quite irritating.

All the program does is traverse the 'Services' registry key, searching for services whose 'Object-Name' value is set to the specified user name, and prints the service's 'DisplayName'. It demonstrates the registry query functions, w32RegOpenKey, w32RegEnumKey, w32RegQueryValue, and w32Reg-CloseKey.

I use the services applet to change the password for all the listed services.

13.5 word.rex

word.rex is a simple example of automation support which was part of the original Ataman port of Regina. It uses the MS Word Basic interface, which doesn't seem to be documented anywhere, to create a new file, insert some text, and save it. See pdfword.rex in section 13.10 for a more complete example of interaction with MS Word.

13.6 menu.rex

menu.rex was written to demonstrate the function w32menuadditem. It creates a folder on the user's programs menu, and inserts links to the README, FIXES, and manual for the w32funcs distribution. the README and FIXES files are opened using notepad, and each take icons from notepad. The manual will be opened using file association.

13.7 vss.rex

vss converts a visual source safe project into a series of RCS files, preserving comments, labels, user names and check-in dates. This started life as a program to retrieve a source safe project based on a label, but I decided I would feel much happier if all of my source safe projects were backed up in a reliable version control system. It was written using a white paper on source safe automation which I retrieved from a Microsoft web site.

The script uses both w32funcs and rexxutil. I have another version which uses 'rexxfile', a library which allows a script to write to the standard input of another program, instead of putting comments in a temporary file. Note that having a working Source Safe installation doesn't guarantee that this script will work. The automation interface must be installed and configured. You must also have RCS executables. RCS seems to be less popular than, for instance, CVS, but I like it because it's simple, reliable, and portable. There's at least one win32 port available, and the RCS parts of the script could be replaced by calls to another version control system.

The script takes a single argument, which is the source safe project name, in the form $/path/to/project. The RCS tree is written to a directory tree under the current directory. The top-level directory of this tree is taken from the lowest-level directory of the VSS project.

The first part of the script is hopefully self-explanatory – it loads the source safe interface by calling w32CreateObject, opens the default source safe database by calling the 'Open' method, then prints a few propreties to give the user something to read while it's processing.

After opening the project using w32GetSubObj, the procedure setupIdentifiers calls w32OleGetObjId for all the method and property names which will be used in the inner loop. Automation uses numeric identifiers to distinguish between different methods and properties. When an automation function, for instance, is called using its name, the library has to first look up the numeric identifier and then call the

function. Using identifiers instead measurably improves the performance of programs which use a lot of automation function calls. One problem with performing look-ups this way is that you need an instance of every object which will be used, and it might be inconvenient or expensive to come up with that list at the start of a run.

The function doProject enumerates all the items in a project and either calls itself recursively, or calls doFile to extract the revisions of a file. The enumeration uses a *collection* called 'Items'. A collection is a sort of array, which typically has a property called 'Count' which gives the number of things in the collection, and a property called 'Item', which takes an integer as an argument and returns one of the things. There is always a method of enumerating all the things in the collection, which is accessible through w32OleNext. In this case, we loop through all the items in numeric order, and check the 'type' property to see if it's a project or a file. If it's a project, doProject calls itself, making this a depth-first operation.

When the item is a file, doFiles is called. It finds the name of the file, initialises the RCS file, turning off file locking for performance reasons, then enumerates all the versions of the file, which happen to be in a collection called 'Versions'. Versions is an odd collection, in that it doesn't provide a 'Count' or 'Item' property. The only way to list the versions is using w32OleNext. This is especially unfortunate, since the order of evaluation is the reverse of the one we need, so all the version handles are put in a stem variable, which is traversed backwards.

For each file revision, the text of the revision is written to a file, the date is normalised, and the file is checked in. The most complicated part of that is date normalisation. The date returned by source safe is in the format given by the short-format for date and time in the system's internationalisation settings. The script assumes this is the 'US' format, month/day/year h:mi:s am/pm, and it needs to be changed to use some other format. This date and time is converted into the rexx 'b' and 's' formats, respectively, the time is changed to greenwich mean time, and the date and time are changed to the canonical y/m/d hh24/mi/ss format. Finally, if the time happens to be earlier than another revision, it is adjusted to be later, and the original time is added to the revision comment (RCS will reject the change, otherwise).

13.8 randcolour.rex

randcolour randomly changes the colour of every UI element every 10 seconds. It demonstrates one possible use of w32SysSetColours. Some people find it a bit irritating when you do this.

13.9 getcolours.rex

getcolours retrieves the current system colour setting, and restores the system colours to the defaults. It is most useful when randcolour has produced an interesting set of colours and you want to know what they are, but it has secondary application when randcolour has produced completely unuseable colours and you want to restore things to the way they were. It demonstrates the use of w32SysGetColours, w32SysSetColours, w32RegOpenKey, w32RegEnumValue, and w32RegQueryValue.

13.10 pdfword.rex

pdfword adds bookmarks and document information to a Microsoft Word document, in preparation for distilling into Adobe's Portable Document Format (PDF), using Ghostscript. It will probably also work with Adobe's pdfwriter, although I haven't bothered to check.

Postscript files can communicate information to PDF conversion utilities using an operator called *pdfmark*. pdfword uses Word's 'print' fields to add pdfmarks which set bookmarks for each heading and which set the title, author, and other information about the document.

The general format for our bookmark pdfmark is

```
[ /Action /Goto /View [/XYZ xpos ypos zoom ] /Title (title)
  /Count count /OUT pdfmark
```

This creates a bookmark whose title is *title*, and which sets the upper left corner of the viewer window to (*xpos, ypos*) of the page containing the pdfmark when it's selected. If *count* is non-zero, the next *count* bookmarks will be displayed as children of this bookmark. The *count* does not include grand-children. *zoom* affects the zoom level after the bookmark has been selected. I set it to null to leave it the same as it was before the bookmark was selected.

What we want to do is add one of those pdfmarks to each heading in the document, so we need to be able to find out, for each paragraph, whether it's a heading, its title, its position on the page, and the number of sub-headings. Each paragraph has a property called 'OutlineLevel', which tells what kind of heading it is. This turns out to be a number from 1 (heading level 1) to 10 (body text). That's an awful lot of outline levels, so we arbitrarily cut it off at 3 – we'll add a pdfmark to each paragraph with outline level 1, 2, or 3. The most obvious way to find the number of children is to count them. The other information can be retrieved from a paragraph property called its range.

To count child headings, we enumerate all the paragraphs, starting at the end of the document. When we encounter a level 3 paragraph, we emit a pdfmark with 0 as the child count, and increment the count of level 3 paragraphs. When we encounter a level 2 paragraph, we emit a pdfmark with the count of level 3 paragraphs, then set that count to 0 and increment the count of level 2 paragraphs. When we encounter a level 1 paragraph, we emit a pdfmark with the sum of the two counts, then set both counts to 0.

The first thing we do in the script is make a connection to Word by calling w32CreateObject, then find the active document by calling w32GetSubObj. The last thing we do before getting on with the task at hand is to find out the size of a page in this document. We need to know this because Word measures pages from the top, but PostScript measures them from the bottom, so we need to perform a conversion.

Word's document class has a property called 'PageSetup', and it has properties called 'BottomMargin', 'TopMargin', and 'PageHeight'. Adding them together gives the offset from which Word gives positions information. PageSetup has another property called 'LeftMargin' which we use to determine the left position for all our /Goto operations. We subtract 4 from the margin to give a 4 point margin after the goto.

The first thing we need to do is get rid of any print fields that might be left over from a previous run of this utility. We do this by enumerating the members of the documents 'Fields' collection, looking for fields which have type wdFieldPrint. wdFieldPrint is a member of an enumeration type called wdFieldType. Its value can be found using gettypeinfo (that's also how I found out about the outline level values). For each print field, we simply call the delete method.

The next little bit of the script is taken up retrieving object ids to speed up all the w32GetSubObj and w32GetProperty calls. For each paragraph, we test its outline level, and if it's a heading, we update the counts as outlined above. Then we retrieve the 'Range' property, and from that we retrieve the 'Text' property, having first stripped its very annoying trailing carriage return. We print the text to the screen to give impatient programmers a feel for where we are.

When we retrieve the range, its start position is at the start of the paragraph, and its end position is at the end. We now set the end position to be at the same spot as the start, and retrieve the vertical position of the range (the value of wdVerticalPositionRelativeToPage was found the same way as wdFieldPrint). The page position needed for the postscript file is obtained by deleting this result from the height of the page. Finally, all that information is put together into the pdfmark, and we create a print field by adding it to the fields collection, and specifying the range (which now is at the start of the paragraph) as the location for the field.

Once all the paragraphs have been set, we want to put some document information into the postscript file, so that it will be available in the PDF. In particular, we want to set the title, subject, author, keywords, creation time, and modification time, and we want to have the document open with the bookmarks displayed.

All that information is retrieved from a document property called BuiltinDocumentProperties, which is a collection indexed by an enumeration called wdBuiltInProperty. Again, I got the values for these indices using gettypeinfo. Once the properties are retrieved, they're written out using another pdfmark, which has the form

```
[ /Author (name) /Title (title) /etc (etc) /DOCINFO pdfmark
```

Each of the document info tags is written only if corresponding properties are set in Word. Finally, we insert the print field containing this pdfmark at the start of the document by getting the range which covers the entire document, setting its end to the start, and adding the field in the spot indicated by the range.

After pdfword has been run, two steps need to be performed manually: the document must be printed to a postscript file, for instance, by selecting print to file with a postscript printer driver, or by defining a new printer whose output port is a local file, and associating it with a postscript printer driver, and then the file needs to be converted to PDF, for instance using Ghostscript's ps2pdf. One common suggestion for generating postscript output from Windows applications is to use the Adobe postscript driver, which can be downloaded from their web site.

I used three resources when working this out. I got the pdfmark reference from Adobe's web site, the Word Visual Basic documentation from the Word installation CD, and a list of all the Word classes from the Word typelib, via gettypeinfo.

13.11 enumword.rex

enumword prints the text of every paragraph in a Microsoft Word document. It was written as a test program while I was trying to figure out why w32OleNext didn't work with Word, but it could be the basis of any number of useful programs, since walking through the contents of a Word document is the sort of thing people do all the time.

All the script does is connect to Word, find the active document, and loop through the 'Paragraphs' collection, and print the text of each one.

13.12 clipex.rex

clipexDemonstrates various things you might want to do with the clipboard functions.

13.13 wrevlog.rex

wrevlog demonstrates writing to the event log. wrevlog.mc contains the text of a series of messages which can be compiled into a message file using the method described in section 5.19. wrevlog.dll is a pre-made version of this message file.

Run without arguments, wrevlog registers the message file as belonging to the 'RexxW32Example' application and writes a series of messages to the event log. The registration step requires administrative access to the local machine. Run with the argument 'remove', wrevlog unregisters the message dll, which also requires administrative access.

wrevlog.mc shows messages of different severities and demonstrates the use of different categories. It's possible to have a catch-all message and pass the details as parameters, but it's better in my opinion to make the messages as specific as possible, to allow automated tools to process message information without examining the message text or parameters.

Index