**Paper CS07**

# Using the WIN32 API from SAS

Edward Foster, Oxford Pharmaceutical Sciences, UK

## ABSTRACT

The WIN32 API is a collection DLL's (Dynamic Link Libraries) that contain functions that Windows uses to control the actions of the operating system. Using the SAS functions MODULEN and MODULEC it is possible for SAS to access the functions in the DLLs. Having access to these functions allows the programmer to integrate any SAS-based application with the Windows operating system.

## INTRODUCTION

The core DLLs that make up the WIN32API are KERNAL.DLL, USER32.DLL and GDI32.DLL. These DLLs most contain the functions that the SAS programmer will be interested in accessing such as basic file management functions or process execution functions. As new operating systems are created by Microsoft they add new DLLs rather than replace the old ones to promote backward compatibility of their products.

From the SAS programmer's point of view the main reasons for using the WIN32 API are;

- Seamless integration with Windows – By using the WIN32 API the programmer can dispense with the use of the 'X' or SYSTASK statements to execute shell commands. This means no more ugly black box popping up each time we want to do some file management task like creating a directory
- Increased speed – because the WIN32 API is effectively part of the operating system using these low level functions should increase the speed of the tasks.
- It gives us something else to play with!

There are a number of components that the SAS programmer needs to set up in order to use the WIN32 API. Firstly SAS needs to understand how to access the WIN32 functions so in order to do this you must create an 'Attribute Table' which is essentially a set of metadata about the function. Secondly SAS needs to reference this attribute table through a special filename called SASCBTBL. Now SAS is ready to access the WIN32 API functions using the MODULEN and MODULEC. This may, at first, put you off using these functions but it's not really that bad! In fact once set up these functions can be wrapped nicely into SAS macros that others can use.

This paper will outline how to set these elements up as well as how to use a few of the functions.

## SETTING UP SAS TO USE THE WIN32 API

As previously mentioned SAS needs a few elements to be set up before it can access the functions within the WIN32 API. The first and perhaps most important of these is the Attribute table.

#### ATTRIBUTE TABLE

An attribute table is simply a definition of a WIN 32 function. The reason that this table needs to be created is because SAS has no knowledge of the commands within the operating system. So, for SAS to understand how to use these functions we need to provide a set of metadata that describes the name and location of the functions as well as the number, type and purpose of the arguments that each function accepts as parameters. The great thing, however, is that an attribute table is little more than a set of string arguments that can be stored in any ASCII file (.TXT etc) or even a SAS catalog. Moreover, more than one attribute table can be stored in such a file making management and maintenance easier.

Typically this file is called WIN32API.txt and can contain multiple attribute tables. Each attribute table is made up of a number of lines of text that define the function in question. The standard structure of an attribute table is shown below.

```
ROUTINE functionalias
    MODULE = DLL NAME
    MINARG = n
    MAXARG = n
    STACKPOP = CALLED/CALLER
    RETURNS = type;

ARG n action type addressing format;
```

A

B

The key part of section A is to name and locate the WIN32 function. There is only ever one of these per attribute table.

- The ROUTINE parameter equates to the name of the WIN32 function that you want to create the attribute table for and is what is referenced by MODULEN and MODULEC when you want to call the function.

- The MODULE parameter identifies the DLL that contains the function we want to use.

- The MINARG and MAXARG parameters are integer values that define the minimum and maximum number of arguments that will be passed to the function when it is called.

- STACKPOP can have the value of either CALLER or CALLED. It refers to the memory management that is undertaken by Windows when the function is called. This value is typically set to CALLED as it allows Windows to return to SAS after the function has executed.

- The RETURNS parameter defines the return value of the function. This is a C/C++ type, for example an INT, BOOL or BYTE, but these are generally returned to SAS as a numeric value. Quite often this value is important as it tells you something about the execution of the function. For example the MESSAGEBOXA function allows SAS to popup a message box. The return value tells you what button was pressed by the user.

Section B contains the ARG statement. The ARG statement defines each of the parameters that are passed (optional or otherwise) to the function. There can therefore be more than one repeat of section B, dependant on the function. The components that make up the ARG statement are as follows;

- The N component describes the order of the parameters. So if a function had more than one parameter
    - ARG 1…; Would describe the first parameter
    - ARG 2 …; would describe the second parameter.
    - e.t.c …..!

- The ACTION component describes how the function will use the argument that is passed to it. For example if the value of ACTION is INPUT the function will receive the value from SAS for use in its processing. The other value that is used for this function is UPDATE. UPDATE means that the function will pass back a value to this variable and therefore you must specify a SAS variable for these arguments so that the value can be caught.

- TYPE defines the type C/C++ type of the parameter.

- ADDRESSING states how the parameter will be passed to the function. The two options available are BYVALUE or BYADDR. BYVALUE states that the value itself will be passed to the function, whereas BYADDR states that a reference to the memory address will be passed.

- The final component, FORMAT, describes to SAS how the parameter is stored. Valid values are SAS formats but the precise format should be checked.

The ordering of each of the components does not matter in either Section A or B except that the ROUTINE component must come first in section A and the N component must follow the ARG statement declaration in section B.

This may sound a bit daunting now but there is a lot of help out there on the web and in reference books. Any good search engine will retrieve a host of examples and definitions that you can use in your own programs. Richard DiVenezia has a good web site (see Further Reading) with examples that you can use.

The next component that SAS needs to use WIN32 function is the SASCBTBL filename.

#### SASCBTBL FILENAME

This SAS file name is used internally by the MODULEN and MODULEC functions. There are no special requirements for this filename all it needs to do is point to the file that contains the Attribute table definitions. The only important thing is that it is present and valid.

#### MODULEN AND MODULEC

These are the functions used by SAS to reference the WIN32 functions. They are essentially the same accept that MODULEN expects a numeric return value and MODULEC expects a character value. In the vast majority of cases you will use the MODULEN function as most WIN32 functions return a numeric value (BYTE, INT and SHORT are all numeric C/C++ types of varying precision).

## PUTTING IT ALL TOGETHER

The first example below outlines how we can use the WIN32 API to copy a file from one directory to another. This is a relatively simple operation but one which outlines the principle quite well.

Firstly we set up the SASCBTBL filename. This can be set up to reference any plain text file that contains the attribute table we need to execute the WIN32 function. This leads to a couple of strategies when storing your attribute table definitions. Firstly you could store all your definitions in one global text file that all programmers can reference. The problem with this approach is that it does not easily allow programmers to work offline as this file would normally be available through a network share. Another approach is outlined below.

```
filename sascbtbl catalog "work.win32api.copyfile.source";

data _null_;
    file sascbtbl;
    put 'routine CopyFileA';
    put '  module=KERNEL32';
    put '  minarg=3';
    put '  maxarg=3';
    put '  stackpop=called';
    put '  returns=ushort';
    put ';';
    put 'arg 1 input char format=$cstr200.;      * From filename;';
    put 'arg 2 input char format=$cstr200.;      * To   filename;';
    put 'arg 3 input num format=pib4. byvalue;  * 1=dont overwrite an existing file,
0=overwrite; ';
run;
```

This approach creates the attribute table on the fly just before we need it. This approach works well for a couple of reasons. Firstly it gets round the problem of working offline. Secondly because we are creating the table on the fly and storing it in a SAS catalog entry, which itself is stored in the work library, we automatically have a clean up facility when the SAS session ends. The problem with this approach is that it appears that the programmer has to know the attribute table definitions. There is a way round this that I will outline later.

Back to the example! Now we have our attribute table we can actually copy a file. The code below demonstrates the syntax for this. For a full description of the MODULEN function see the SAS help.

```
data _null_;
    rc=modulen('*E','CopyFileA',"c:\Test.log","c:\temp\Test.log",0);
run;
```

As you can see we reference the 'CopyFileA' attribute table in the MODULEN function and copy a file called "Test.log" from "c:\" to "c:\temp" and no 'X' command in sight.

#### CREATING WIN32 FUNCTION MACROS

You are probably thinking that this is a lot of work just to copy a file and in some ways it is. However, these functions are perfect candidates for wrapping in macros. Wrapping all this functionality as a utility macro means that the user (i.e. the programmer) no longer has to remember all the attribute table details in order to use the macro. There can be unseen benefits too, For example have you ever had to create an 'X' command or a CALL SYSTEM where the path you need to reference has a space in it. If you have you will know the fun you have arranging all your quotes to get the path to resolve correctly. With this

copy example you don't need to worry about quotes again. Below is an example of a macro we could create for this operation.

```
%MACRO CopyFileA(inFile         /*INPUT FILENAME*/
                ,outFile        /*OUTPUT FILENAME*/
                ,overwrite=1    /*0=OVERWITE FILE (IF EXISTS) 1=DONT OVERWRITE*/
                        );

    %put ** COPYFILEA FUNCTION STARTING **;

    ** DO ERROR CHECKING **;
    %if %sysfunc(fileexist(&inFile))=0 %then %do;
           %put ** THE INPUT FILE (&inFile) DOES NOT EXIST **;
           %goto QUIT;
    %end;
    ** GET OUTPUT DIRECTORY **;
    %let reverse=%sysfunc(reverse(&outFile));
    %let outDir=%sysfunc(reverse( %substr(&reverse,%index(&reverse,%str(\))) ));
    %if %sysfunc(fileexist(&outDir))=0 %then %do;
           %put ** THE OUTPUT FILE CANNOT BE WRITTEN TO THE DIRECTORY &outdir **;
           %goto QUIT;
    %end;
    %if &overwrite^=1 AND &overwrite^=0 %then %do;
           %put ** THE VALUE &overwrite FOR OVERWRITE IS NOT VALID. VALID VALUES ARE 0 OR 1
**;
           %goto QUIT;
    %end;

    ** ASSIGN SASCBTBL FILENAME **;
    filename sascbtbl catalog "work.win32api.copyfile.source";

    ** CREATE ATTRIBUTE TABLE IN CATALOG ENTRY **;
    data _null_;
           file sascbtbl;
           put 'routine CopyFileA';
           put '  module=KERNEL32';
           put '  minarg=3';
           put '  maxarg=3';
           put '  stackpop=called';
           put '  returns=ushort';
           put ';';
           put 'arg 1 input char format=$cstr200.;      * From filename;';
           put 'arg 2 input char format=$cstr200.;      * To   filename;';
           put 'arg 3 input num format=pib4. byvalue;   * 1=dont overwrite an existing file,
0=overwrite; ';
    run;

    ** EXECUTE MODULEN FUNCTION **;
    data _null_;
           rc=modulen('*E','CopyFileA',"&inFile","&outFile",&overwrite);
    run;

    ** CLEAN UP **;
    filename sascbtbl;
    proc datasets mt=CATALOG nolist nodetails nowarn;
           delete win32API;
    quit;

    %QUIT:

    %put ** COPYFILEA COMPLETED **;

%MEND CopyFileA;
```

```
%CopyFileA(c:\test.log, C:\temp\test.log, overwrite=0);
```

As you can see other than a bit of error checking and some tidying up at the end the code above is similar to the non-macro code outlined earlier. The great advantage of this however is the fact that all the programmer now has to deal with is a simple macro call (see bottom of example). I prefer this approach to creating these macros as it is important that you are careful when dealing with the WIN32 functions. By creating the macros in this way you get a single versionable unit that can easily be maintained with no links to external attribute files.

## OTHER EXAMPLES

There are loads of possible functions but here are a few more which you could create your own macros for.

```
filename sascbtbl catalog "work.win32api.delfile.source";

data _null_;
    file sascbtbl;
    put 'routine DeleteFileA';
    put '   module=KERNEL32';
    put '   minarg=1';
    put '   maxarg=1';
    put '   stackpop=called';
    put '   returns=long';
    put ';';
    put 'arg 1 input char format=$cstr200.;   * filename ;';
run;

data _null_;
    rc=modulen('*E','DeleteFileA',"c:\temp\test.log");
run;
```

The above example deletes the “c:\temp\test.log file” we created earlier. The example below creates a new directory called ‘Test’ inside ‘c:\temp\’.

```
filename sascbtbl catalog "work.win32api.created.source";

data _null_;
    file sascbtbl;
    put 'routine CreateDirectoryA';
    put '   module=KERNEL32';
    put '   minarg=2';
    put '   maxarg=2';
    put '   stackpop=called';
    put '   returns=short';
    put ';';
    put 'arg 1 input char format=$cstr200.;   * directory to create (cannot create
multiple levels in one call);';
    put 'arg 2 input num format=pib4. byvalue;* security attributes, use 0;';
run;

data _null_;
    rc=modulen('*E','CreateDirectoryA',"c:\temp\test",0);
run;
```

The thing to note about the above function is that ‘CreateDirectoryA’ can only create one directory level at a time. You may need to place this function in a recursive macro to create a whole directory tree structure (or find a WIN32 function that will do it for you!). Note also that you will nearly always set the second parameter to ‘0’. The reason for this is that the second parameter sets the security settings for the new directory and a value of ‘0’ propagates the settings from the parent directory.

The following 2 functions I have found useful in the past because they allow you to programmatically set the attributes of a file. These attributes include the read-only flag for a file so using these functions can help you protect (in a small way) against accidental editing or deletion of files. I have found this most useful to use at the end of derived dataset or table program where I can set the files to read-only. **N.B. note that I have put both attribute tables into the SASCBTBL file at the same time.**

**You of course to not have to do this**

I have included the valid values for the attributes in the comments in the program.

```
filename sascbtbl catalog "work.win32api.attrib.source";

data _null_;
    file sascbtbl;

    put 'routine GetFileAttributesA';
    put '   module=KERNEL32';
    put '   minarg=2';
    put '   maxarg=2';
    put '   stackpop=called';
    put '   returns=long';
    put ';';
    put 'arg 1 input char format=$cstr200.;      * lpFileName;';
    put 'arg 2 output num format=pib4. byvalue;  * dwFileAttributes;';

    put 'routine SetFileAttributesA';
    put '   module=KERNEL32';
    put '   minarg=2';
    put '   maxarg=2';
    put '   stackpop=called';
    put '   returns=long';
    put ';';
    put 'arg 1 input char format=$cstr200.;     * lpFileName;';
    put 'arg 2 input num format=pib4. byvalue;  * dwFileAttributes;';

* attribute flags, can be ORd;
* READONLY     01x ;
* HIDDEN       02x ;
* SYSTEM       04x ;
* DIRECTORY    10x ;
* ARCHIVE      20x ;
* NORMAL       80x ;
* TEMPORARY   100x ;
* COMPRESSED  800x ;
* OFFLINE    1000x ;

run;

data _null_;
    rc=modulen('*E','SetFileAttributesA',"c:\temp\test.log",01x);
run;
data _null_;
    rc=modulen('*E','SetFileAttributesA',"c:\temp\test.log",80x);
run;
```

Note that the first MODULEN sets the file to READ-ONLY and the second sets it back to a normal file.

## SPECIAL MENTION FOR WINEXEC

At a first glance the WINEXEC Win32 function is not that fancy. In fact it essentially does the same job as the SAS ‘X’ command or CALL SYSTEM by allowing you to issue shell commands yourself. So why would you use it. Well firstly you get rid of the command box popping up when you execute a shell command. This is not annoying when it is just the once, but if you need to execute a range of commands then having the command box bounce around the screen can give you a headache! The second advantage is down to the second parameter for the WINEXEC function. The second parameter allows you to control the state of the window of any application you open when you execute a command. This includes minimising the window or even hiding it. The example below opens Excel and minimises it (though the path may differ for you). I have included valid values for the window status in the code.

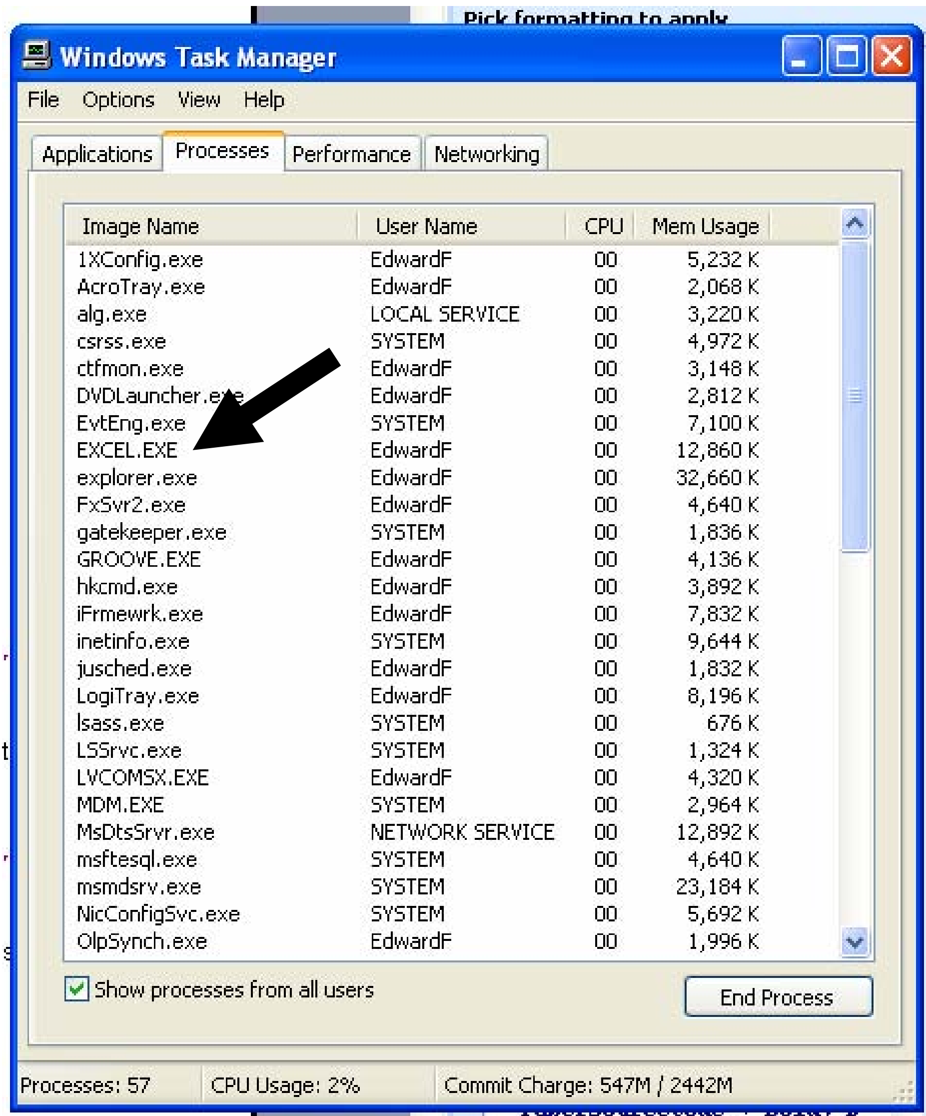
```
filename sascbtbl catalog "work.win32api.winexec.source";
```

```
data _null_;
    file sascbtbl;
    put 'ROUTINE WinExec';
    put '         minarg=2';
    put '         maxarg=2';
    put '         stackpop=called';
    put '         returns=ushort';
    put '         module=Kernel32;';
    put '         arg 1 char input byaddr  format=$CSTR200.;  * LPCSTR lpCmdLine,      //
address of command line ;';
    put '         arg 2 num  input byvalue format=pib4.;      * UINT   uCmdShow      //
window style for new application ;';

* SW_HIDE                0   ;
* SW_SHOWNORMAL          1   ;
* SW_NORMAL              1   ;
* SW_SHOWMINIMIZED       2   ;
* SW_SHOWMAXIMIZED       3   ;
* SW_MAXIMIZE            3   ;
* SW_SHOWNOACTIVATE      4   ;
* SW_SHOW                5   ;
* SW_MINIMIZE            6   ;
* SW_SHOWMINNOACTIVE     7   ;
* SW_SHOWNA              8   ;
* SW_RESTORE             9   ;
* SW_SHOWDEFAULT         10  ;
* SW_MAX                 10  ;
run;

data _null_;
    rc=modulen('*E','WinExec',"C:\Program Files\Microsoft Office\OFFICE11\excel.exe",2);
run;
```

This behaviour allows us to control the application we start from SAS. The following code does the same accept that it hides Excel from view.

```
data _null_;
    rc=modulen('*E','WinExec',"C:\Program Files\Microsoft Office\OFFICE11\excel.exe",0);
run;
```

You can tell that Excel has started if you open the task manager and click on the ‘Processes’ tab. In the list you will see an entry that says ‘EXCEL.EXE’




You must remember to close Excel down either by right-clicking on this item and selecting 'End Process' or by executing the following code, otherwise it will hang round in the background.

```
filename cmds dde 'Excel|system';
data _null_;
    file cmds;
    put '[error(false)]';
    put '[quit()]';
run;
```

**WINEXEC AND DDE**

DDE stands for Dynamic Data Exchange. It is an old data transfer technology but works very well with SAS and Office products like Excel. One of the problems with DDE is that in order to access the data in a file a version of the application must be running on the machine running the DDE code. For example if you wish to import an Excel spreadsheet you need to first open and Excel, then the spreadsheet you wish to import then you can read the sheet. While this method of import is relatively fast if you need to read a number of spreadsheets it can get annoying if you have Excel bouncing round the screen as the DDE code processes. If this is applied in an application setting then it can stop the application form feeling as seamless as it might be.

The good news is that while the application (like Excel) needs to be open it doesn't actually need to be visible. Can you see where I am going?! If we hook up WINEXEC it becomes possible to open Excel invisibly. You can interact with Excel from you

code in the same way you always have you just cannot see it. If you find this a bit scary then at least you now have the option to open Excel as minimised straight away so at least it does not obscure the user. The following code creates a spreadsheet in 'c:\temp' called 'PhUSE Example.xls' it will contain the contents of the SASHELP.CLASS dataset (you will need to have assigned the SASCBTL filename and created the attribute table first).

```
** OPEN A HIDDEN EXCEL SESSION **;
data _null_;
    rc=modulen('*E','WinExec',"C:\Program Files\Microsoft Office\OFFICE11\excel.exe",0);
run;

** CREATE THE FILE WE WANT **;
filename cmds dde 'Excel|system';
data _null_;
    file cmds;
    put '[error(false)]';
    put '[save.as("c:\temp\PhUSE Example.xls")]';
run;

** EXPORT THE DATA **;
filename data dde "Excel|[PhUSE Example.xls]Sheet1!R1C1:R50C5" notab;
data _null_;
    set sashelp.class;
    file data;
    put name '09'x sex '09'x age '09'x height '09'x weight;
run;

** SAVE THEN CLOSE **;
data _null_;
    file cmds;
    put '[save()]';
    put '[quit()]';
run;
```

## CONCLUSION

The WIN32 API is the core of the Windows operating systems. By utilising the API functions from SAS using MODULEN and MODULEC real benefits can be realised in terms of efficiency, application/operating system integration and user experience. At first glance, using these functions can appear unwieldy but once they are encapsulated into macros they become much easier for other programmers to use. Some functions provide additional functionality that is just not available from traditional shell command execution (WINEXEC is a good example).

Care must be taken when testing these functions because they act at a very low level in the operating system. Because of this they can cause programs to hang or even crash if not defined properly. However, if their use is tested first and the programmer understands what the functions actually do using the WIN32 API can be a useful addition to the SAS programmer's skill and tool set.

## RECOMMENDED READING

As I mentioned earlier Richard DeVenizia has a good site (www.devenevia.com) with lots of resources as well as a whole section dedicated to SAS and the WIN32 API. Other places to look for information are the Microsoft MSDN web site or any search engine.

## CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Edward Foster
Oxford Pharmaceutical Sciences
Lancaster House
Kingston Business Park
Kingston Bagpuize
Oxford / OX13 5FE
Work Phone: +44 (0) 1462 477948
Email: edward.foster@ops-web.com
Web: www.ops-web.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.