Paper 248-30

SAS® with the Windows API

David H. Johnson, DKV-J Consultancies Ltd, UK & Australia

ABSTRACT

The core of the SAS® language is written in C, which permits SAS to be readily ported to many operating systems. This is done by placing core SAS functionality over a layer that is specific to the Operating System (OS).

Since the first release of SAS version 6, this layer has become progressively more transparent. More functionality at the OS level has become available to common SAS language and functions. However, there is a wealth of other OS functionality that can be accessed from SAS language and procedures with a little preparation on the part of the SAS programmer.

This paper will examine some of the additional Windows functionality that we can access from within SAS code. From the techniques we examine, any Windows API should become accessible to us. If we compile our own DLLs for other processes, then these too will be accessible to our SAS programs.

We will research and model the access means for our APIs, using these to perform specific Windows tasks, and debugging that we can set up in the SAS code. The SAS code is relatively simple, and the SAS functionality will all lie within the Base/SAS and SAS/Macro language sets.

INTRODUCTION

The SAS System provides a number of operating system functions in its base language. These include SCL functions in base code that will identify the existence and size of a file. Sometimes however, the information we may want on the file exceeds that available from the available SAS functions. For instance, in Windows operating systems after Windows NT, the file will bear three dates that identify different aspects of the files creation, modification and access. The file attributes also include flags for “Read Only” which is an attribute that has been available since early DOS days, or “Compression” which is a relatively late attribute.

The values of these other attributes may be important to us within a SAS application. We may for instance want to overwrite a file, but if it’s read only flag is set, the operating system will prevent that action. Knowing the flag is set will allow us to code a safe alternative or exit in our program, or simply issue a note to the log indicating the reason for the failure we might expect. (We might also include a process that will remove the flag from within the SAS application.)

To access this extended file information we will use a Windows OS interface. After all, we know the values can be surfaced to a Windows application since we can view these values from within our Windows File explorer. The interface we will use is the common one used by Windows applications, and is called the Windows Application Programming Interface (API).

To access these we will go through similar steps employed by people using Visual Basic, C++, Delphi or other application developer environments. To demonstrate the outcome in this paper we will build a process that surfaces all three file dates and the extended file attributes for each of a series of files in a given directory. Then we will perform some more complex processes.

SPECIFICATION OF OUR PROCESS

To make the process simpler for us to use in a SAS session, we will conceal most of the code and functionality in a SAS macro.

The macro will be called with a single attribute that defines the search parameter. For instance, if we want all the files of type “tmp” that exist in our file directory “C:\Temp”, then our search parameter should be “C:\Temp*.tmp”.

The result will be a SAS table containing the names, attribute dates and attribute parameters of files matching our search parameter.

While we could closely define the required functionality of a robust macro by adding specifications for messaging on “No File Found” or “Directory does not exist”, that exceeds the purpose of this paper. Similarly, while the macro is designed to match certain elements of good design, we will pass over these fairly quickly and focus on the API functionality. For a version of this macro that does include more robust elements, readers are welcome to contact the author.

ACCESSING THE API LAYER WITH A PROTOTYPE

To access the API we need to find the element of the Windows OS that contains the function we want to employ. Web searches, design documents and the Microsoft Developer Network resources will all help in this regard. For our search we need the API that finds files, and our search identifies an API called “FindFirstFile”.

The API takes a single input parameter; the search criterion, and returns a handle on the first file matching the criterion. It also surfaces a set of data that describes the file that has been found. To make this API available to the SAS session, we need to describe the API in terms SAS can understand, so that SAS delivers requests in terms the OS understands, and receive responses from the OS that SAS can map into an appropriate form. Defining a prototype performs this mapping.

Just as in software development, the prototype is a skeleton of the input and output that we want to achieve with the API. To make a prototype available to our SAS session, we build a text file containing the prototype and assign it to our SAS session with the reserved file reference SASCBTBL. (One suggestion the author heard is that this reference is an abbreviation for SAS Control Block TaBLe. This may or may not be true, but it is a good mnemonic to remember when you are debugging applications using the API. One of the most common issues for the ongoing use of APIs is that the prototype is not available to the SAS session. So, if the API call doesn't work, check the SASCBTBL file reference is assigned to the file that you expect.)

FIND FIRST FILE

Our prototype looks like the following:

```
routine FindFirstFileA
        minarg=12
        maxarg=12
        stackpop=called
        module=Kernel32
        returns=long;

arg  1 char input  format=$cstr200.;
arg  2 num  update fdstart  format=pib4.;
arg  3 num  update format=pib8.;
arg  4 num  update format=pib8.;
arg  5 num  update format=pib8.;
arg  6 num  update format=pib4.;
arg  7 num  update format=pib4.;
arg  8 num  output format=pib4.;
arg  9 num  output format=pib4.;
arg 10 char update format=$CSTR200.;
*            MAX_PATH is 260 ;
arg 11 char update format=$CSTR60.;
arg 12 char update format=$CSTR14.;
```

- With the key word "routine", we have instructed the SAS system to associate the subsequent definition to an API called FindFirstFileA. Note the case used for the API name. This is case sensitive, and is one of the places that should be carefully verified. The instructions that follow are to SAS about the prototype, they are not sent to Windows.
- We tell SAS that the fewest number of arguments it will need to make up its required structure for Windows is 12, and that the greatest number is also 12.
- We instruct that Windows is to use a call to control the activity of the stack. The alternative is "calledby" where the API is controlled by another process.
- We identify that the API is part of a compiled module called "Kernel32". In our file system, we can usually find these modules in our "System" or "System32" directories. The modules are a compiled set of instructions called Dynamic Link Libraries, so this API should be in a file called "Kernel32.dll".
- We expect that we will get a large number returned from a successful API call, so we specify it will be a signed long number. Our number could also be a small or "short" number, and if it were unsigned we would precede this key word with "u".
- The "routine" statement is closed with a semi colon.
- The "routine" statement is followed by a series of definitions. Each definition is a single statement terminated with a semi colon. Each statement defines one and only one argument of those required to trigger the API.
- The statement uses the key word "arg" to define it as an argument statement.
- It is followed by an integer that specifies which argument is being defined. All arguments need to be specified, and they need to be specified in ascending order.
- Our argument can be character or numeric in nature, and we specify one or the other with "char" or "num".
- We then specify the manner in which the data will be presented to SAS. If it is an argument we provide that will not be changed, then the specification is "input". If it is an argument that the OS will provide then we specify "output". If we provide a value that may be changed by the OS, then we specify "update".
- Finally we specify the format of the data that will be handled. The format we specify must be a valid SAS format. $Cstr is a special SAS format that retains internal blanks (like "$Char") but also pads out the string to the required value with spaces. Most numeric values tend to be Packed Integer Binary, with short values usually having 4 bytes and long values having 8 bytes.
- In argument 2, the special definition "fdstart" appears. That identifies that this argument and the succeeding ones will be concatenated into a single argument to be passed to the OS. That is because Windows only expects two arguments, the search parameter and the structure of the data it will output or update. Once again, our argument specification is for the benefit of SAS. Windows will see the data translated by SAS.
- After argument 10 there is a comment statement to help in understanding the content of the prototype. It identifies that the actual

file name that can be returned is up to 260 bytes in length. Since this prototype has been in use in the authors programs for more than 8 years, the shorter length was necessary for Version 6 SAS tables, which only allowed character strings to be up to 200 bytes in length. For version 7 and later implementations, arguments 10 and 11 could be combined to 260 bytes, as long as the “minarg” and “maxarg” definitions are also changed.

FIND NEXT FILE

The “FindFirstFile” prototype will find the first file that matches our search criteria, and surface a number that is called the search handle. Since our search might return multiple files, clearly we haven’t finished our prototype definition yet. Unsurprisingly, we need to define another prototype with the following routine.

```
routine FindNextFileA
        minarg=12
        maxarg=12
        stackpop=called
        module=Kernel32
        returns=long;
```

This routine is very similar to the previous one. Again, we define 12 arguments, use the same DLL and expect the same return. There is one difference though, rather than passing it the search parameter again, we need to tell it that a search has already been performed. Otherwise we would be stuck in a loop with just the first file appearing each time. To do this, the first parameter needs to be the handle from the previous search, and the first argument needs to be changed to reflect the different input.

```
* LPCTSTR  hFindFile,  // search handle;
arg  1 num  input  byvalue  format=pib4;
```

This argument introduces another parameter in the definition; “byvalue”. This indicates that the value itself is the key to the API. If it were a pointer to another location where we find the input, then we would use the key word “byaddr” for “by address”. Note that we still need to follow this first argument with the other 11 as they appeared in the previous prototype. Otherwise SAS will not know how to pass and receive data on the file(s) it locates.

These two prototypes will find the first two matching files for us. If we test the returned value from FindNextFileA, and discover that it is also a handle, then we can call that API again and again until the returned value indicates no file was found. A logic diagram will follow shortly to make that process clearer.

CLOSE THE SEARCH

Having initiated a search, and set a search handle, we need to release that handle before we move on. We do that with an API called FindClose, and its prototype follows.

```
routine FindClose
  minarg=1
  maxarg=1
  stackpop=called
  module=kernel32
  returns=short;

arg 1 num input byvalue format=pib4.;
```

In this case one argument is passed, (the search handle), and the return value will indicate whether the search was closed successfully. If we don’t close the search then a memory allocation is not released. A poorly written application failing to close search and other handle assignments is one of the common causes of memory leak that can eventually crash an OS.

USE THE PROTOTYPE

To use these prototypes we need a data step that will write the data returned through them to an area where we can work with it. In the data step, we need to initialize the data table with an explicit PDV (Program Data Vector). This ensures we have the correct types for the data we need to pass to and accept from the prototype.

```
Data ZZZZFILE( Keep = FILENAME  EXTENSN  CREATESD  ACCESSSD  WRITESD
                      FILEATT   FILESIZE ANAME);
  Attrib FILENAME  Length = $200
                   Label = 'First 200 characters of file name'
         EXTENSN   Length = $32
                   Label = 'Extension for file name'
         CREATESD  Length = 8
                   Label = 'Date time file created'
         ACCESSSD  Length = 8
                   Label = 'Date time file last accessed'
         WRITESD   Length = 8
                   Label = 'Date time file last updated / written to'
```

```
          FILEATT   Length = $8
                    Label = 'File type attributes ( F = Folder)'
          FILESIZE  Format = Comma14.
                    Label = 'File size in bytes'
          ANAME     Length = $14
                    Label = 'Abbreviated file name for DOS';
   Length  NAME PATH $200    BITNAME $60;
   NAME     = " ";   BITNAME  = ' ';    ANAME     = " ";   ATT    = .;
   CRE      = .;     ACC      = .;      WRI       = .;     SLOW   = .;
   SHIGH    = .;     RC       = .;      NUM1      = .;     NUM2   = .;
   LNUM1    = .;     RCODE    = 0;      DWRD0     = 0;     DWRD1  = 0;
   PATH      = "&MDir";
```

In these few statements, we have specified an output table with leading “Z” characters in the table name. This is a convention employed by the author for temporary and system generated tables, which helps to ensure that tables created in macros do not conflict with tables already created in the calling code. We have also specified a subset of the columns we create in the table and used an ATTRIB statement to size, label and if necessary, format the data. (The date time values are not formatted at this point because a separate process creates these. The format is created in that process so that we have a single change point in the event we decide to alter the content of these columns.)

The subsequent “Length” statement initializes the PDV for some character strings that will be created in the data step, but discarded after they are manipulated. A series of assignment statements follow the length statement. These are parameters for the prototype and test values we will use within the data step. They have been listed in a column format to save space here. They are defined to prevent notes appearing in the log indicating “variable …. is uninitialised”. (In our final code version, we should only have one executable statement on each line.)

Finally, we resolve a macro variable called “MDir” into a table variable called “PATH”. The macro variable is the parameter we are passing to the macro call and contains the search parameter we have defined. The variable we thus create becomes the first parameter to our call to the prototype. To call the prototype we are going to use a SAS function called “ModuleN()”. “Module” defines it as a code module external to the SAS system, and the “N” suffix indicates that it has return value that will be numeric. The call looks like this:

```
   HANDLE = Modulen( '*e', 'FindFirstFileA', PATH, ATT, CRE, ACC, WRI,
                      SHIGH, SLOW, DWRD0, DWRD1, NAME, BITNAME, ANAME);
```

“Handle” will be the returned value from the call, which we indicated earlier is a search handle. Ignore the first parameter of the call; it is a special optional control parameter to which we will return later in this paper. The next parameter is the name of the prototype as defined in the “routine” statement. It is followed by 12 parameters that match the 12 parameters we specified in the prototype call. These parameters are:

- PATH is the specification of our search, our example search specification was “c:\Temp*.tmp”.
- ATT is a numeric value that contains binary flags for the file attributes. The file binary values are:
  - FILE_ATTRIBUTE_READONLY 0x00000001 = 1
  - FILE_ATTRIBUTE_HIDDEN 0x00000002 = 2
  - FILE_ATTRIBUTE_SYSTEM 0x00000004 = 4
  - FILE_ATTRIBUTE_DIRECTORY 0x00000010 = 16
  - FILE_ATTRIBUTE_ARCHIVE 0x00000020 = 32
  - FILE_ATTRIBUTE_NORMAL 0x00000080 = 128
  - FILE_ATTRIBUTE_TEMPORARY 0x00000100 = 256
  - FILE_ATTRIBUTE_COMPRESSED 0x00000800 = 2048
  - FILE_ATTRIBUTE_OFFLINE 0x00001000 = 4096
- CRE is the date time when the file was created. It is a very large number that represents the number of milliseconds between 1 January 1601 and the date of creation. We will use a special Windows function to translate this number.
- ACC is the file last access date, it is stored similarly.
- WRI is the date the file was last written to, it is in the same format as CRE and ACC.
- SHIGH and SLOW are two large numbers that represent the high order and low order parts of the file size. As an example, if we were to split a number at 1000 and store the number parts in high and low order numbers; the number 123456 would be recorded as 123 in the high order value, and 456 in the low order value. This method allows us to store very large numbers with a great deal of precision. In the Windows file system the split is at 2^{32}. This allows us to store the correct size of a file that is larger than 4,294,967,295 (approximately 4Gb). The maximum file size we can store in this system variable is around $1.8 * 10^{19}$, which is approximately 1.8 Exabytes. (There are about 1,000,000,000 gigabytes in an Exabyte.)
- DWRD0 and DWRD1 are file attributes reserved for future use. (If the file has the file attribute reparse point set, then DWRD0 holds the reparse tag. The author has not found any need for this data.)
- NAME holds the first 200 bytes of the file name. BITNAME is an additional 60 bytes removed from the end of the file name for consistency with SAS Version 6 data structures.
- ANAME is the alternate name, it is the filename held in the classic 8.3 file name format. For instance, a version 8 SAS table named EnquiryData.Sas7bdat in the file system would have an alternate name like ENQUIR~1.SAS. Note that the ANAME value is

usually only created for file names that don't match the 8.3 name definition and is not created for directories.

For ease of use of the final data, the date values, file size and attribute set are all translated in a process we will review shortly. For ease of use of the variable, NAME is changed to the more descriptive FILENAME. ANAME is retained as it is. Before we review this translation however, let's review where we are in the process. This will help us to understand the next part of the code.

We have found the first file in the File Allocation Table (or NT File System) and returned a search handle from the process. We want to start a process where we can retrieve other matching file names, so we save the data and then use the search handle to initiate a search for subsequent files. Here is the structure of the code that continues that search.

```
If HANDLE >= 1  Then Do;
  FOUND = 1;
  Output;
  Do While( FOUND);
    FOUND = Modulen( '*e', 'FindNextFileA', HANDLE, ATT, CRE, ACC, WRI,
                     SHIGH, SLOW, DWRD0, DWRD1, NAME, BITNAME, ANAME);
    If FOUND   Then   Do;
      Output;
    End;
  End;
End;
```

If our search handle is a positive integer then we have successfully matched our search criteria. So we save the data that was returned in the "FindFirstFile" process. Then we set a flag called FOUND to true and go into a Do While loop.

We reassign FOUND to the result we get from issuing the "FindNextFile" API. The outcome of this search is a set of parameters that match those from FindFirstFile, except that we have a HANDLE from the first search as a parameter. This handle is something like a pointer to a stack, and we are going to step entry by entry down that stack each time we issue the "FindNextFile" API call. Eventually we will reach the end of the stack and the FOUND handle will then be set to 0. When this happens we will drop out of the whole search process.

Let's examine that process with the aid of a diagram to make it simpler. At Figure 1 we can see the process. Note particularly the green line, which describes the looping process while the files in the search stack are processed in turn.

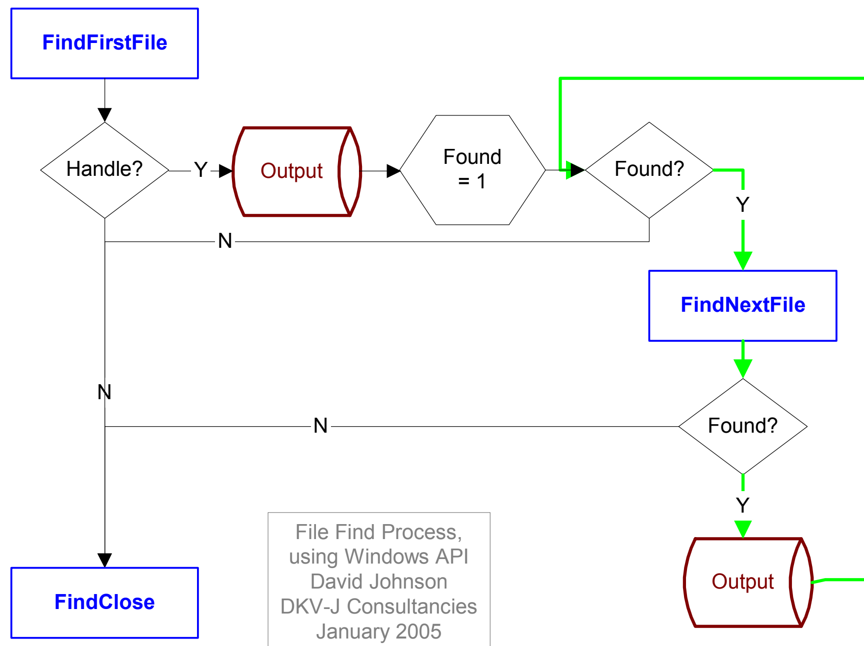


Figure 1

Since the only difference between the "FindFirstFile" and "FindNextFile" searches is the replacement of a search term with a search handle, we will apply the same set of translations to the data returned from the operating system. Rather than hard coding these into the data step, we will code them into another macro so that we call them before each of the output statements. That macro is called "**XGetAttr**" and is explored a little later on.

CLOSE THE SEARCH

We have assigned memory to hold the search handle, so to complete the process we need to release that memory. We do that by calling the FindClose API that we have specified above, and pass it the handle. This line of code follows our last END statement.

```
RC = Modulen( '*e', 'FindClose', HANDLE);
```

RESOLVE THE ATTRIBUTES

This section explores the first part of the "**XGetAttr**" macro mentioned above. Within this macro we want to do the following:

- Translate the attribute block into a format we can more readily understand.

For those with a computer background that extends as far back as DOS, the acronym RASH will be familiar. This acronym reflected the attributes of a file when it was reported in a DOS window. The values stand for **R**ead-only **A**rchive **S**ystem **H**idden. If we replace the attribute number with a similar acronym we will immediately be able to recognise when any of the four of the flags are set. To this the author has added DNTCO to report **D**irectory **N**ormal **T**emporary **C**ompressed **O**ffline. Within the macro then we will define the content of an 8-byte character string. While 9 attributes are described here, and there are also three others that have not been documented, not all combinations can occur for a file. For instance, **N** is mutually exclusive of all other attributes.

- Translate the date time values from milliseconds since 1 January 1601 to seconds since 1 January 1960. In other words, convert them to SAS date time values.
- Translate the two-part file size value to a single value.
- Extract the extension from the file name so we can select files by file type.

Two of these translations can be performed with the following statements.

```
FILESIZE = SHIGH * ( 2**32) + SLOW;

If  ( ATT = '...........1....'b) Then  FILEATT = "D";
If  ( ATT = '....1...........'b) Then  FILEATT = Compress( FILEATT || "C");
If  ( ATT = '.......1........'b) Then  FILEATT = Compress( FILEATT || "T");
If  ( ATT = '...............1'b) Then  FILEATT = Compress( FILEATT || "R");
If  ( ATT = '..........1.....'b) Then  FILEATT = Compress( FILEATT || "A");
If  ( ATT = '.............1..'b) Then  FILEATT = Compress( FILEATT || "S");
If  ( ATT = '..............1.'b) Then  FILEATT = Compress( FILEATT || "H");
If  ( ATT = '...1............'b) Then  FILEATT = Compress( FILEATT || "O");
If  ( ATT = '........1.......'b) Then  FILEATT = Compress( FILEATT || "N");
```

The extraction of the extension was quite straightforward up to Windows 95. It was possible to locate the decimal point that delimited the extension and parse out the extension. Subsequent to Windows 95 it has become possible for a name to contain more than one decimal point, so the parsing process requires a little more sophistication.

```
If Length( Trim( NAME) ) > 2  And Index( NAME, '.')  Then Do;
  POSITION = Length( Trim( NAME) )
             - Index( Left( Reverse( NAME) ), '.')
             + 1;
  FILENAME = Substr( NAME, 1, POSITION - 1);
  EXTENSN  = Substr( NAME, POSITION + 1);
End;

Else Do;
  FILENAME = NAME;
  EXTENSN  = ' ';
End;
```

Reversal of the file name is necessary to ensure we find the last decimal point in the file name. This is the true delimiter of the file extension.

RETRIEVE FILE DATE TIMES

Each file has three date times, and all three are stored the same way. The simplest way to derive all three date time values is to use a macro for each value and pass two parameters to it. The first parameter will be the name of the raw date time value, and the second will give a name for the value resolved as a SAS date time.

To retrieve these values, the most reliable way to perform the conversion is by using a Windows API designed for the purpose. It is called FileTimeToSystemTime. This performs the appropriate calculations to translate the millisecond value into various date and time parts. The prototype to perform the calculation appears like the following.

```
ROUTINE FileTimeToSystemTime
        minarg=9
```

```
        maxarg=9
        stackpop=called
        module=Kernel32
        returns=long;

    * CONST FILETIME lpFileTime,    // pointer to file time to convert ;
  arg  1 num  input          format=pib8.;

    * LPSYSTEMTIME   lpSystemTime   // pointer to structure to receive system time ;
  arg  2 num  output fdstart format=pib2.;   *   WORD wYear ;
  arg  3 num  output         format=pib2.;   *   WORD wMonth ;
  arg  4 num  output         format=pib2.;   *   WORD wDayOfWeek ;
  arg  5 num  output         format=pib2.;   *   WORD wDay ;
  arg  6 num  output         format=pib2.;   *   WORD wHour ;
  arg  7 num  output         format=pib2.;   *   WORD wMinute ;
  arg  8 num  output         format=pib2.;   *   WORD wSecond ;
  arg  9 num  output         format=pib2.;   *   WORD wMilliseconds ;
```

Note that argument 2 includes the specification “fdstart” which indicates it is the beginning of a field definition, which is a concatenation of data elements. All other arguments and elements of the prototype are similar to the previous examples.

A limitation that is sometimes overlooked is that Windows 95 and later Operating Systems store time values in GMT. When the local system clock or a file system date time is displayed, it is offset by the amount defined in the Windows control panel that specifies the time zone in which the computer is operating. For computers in New York this means making a four or five hour adjustment, for computers in Australia this means making a nine or ten hour adjustment. The variation depends on whether the local area is using daylight saving time.

To perform this offset, Windows provides another API that will offset the original file date time according to the time zone and time of year. The API is called FileTimeToLocalFileTime. The prototype is quite simple since it takes a single input value and performs an adjustment according to the time zone setting of the PC.

```
  ROUTINE FileTimeToLocalFileTime
    minarg=2
    maxarg=2
    stackpop=called
    module=kernel32
    returns=long;

  arg 1 input  fdstart num format=pib8.;  * lpFileTime,     // UTC file time to convert;
  arg 2 output fdstart num format=pib8.;  * lpLocalFileTime // converted file time;
```

To make the change we first test that the file handle in our original macro is set true and then apply the time zone translation.

```
    If FOUND >= 1  Then
       RC4 = ModuleN( 'FileTimeToLocalFileTime', &MFTime, LNUM&MFTime.);
```

Notice that an input parameter “MFTime” is passed into the API as a SAS macro. This macro will resolve to a value of ACC, CRE or WRI. The resulting value is written out to a variable called LNUMACC, LNUMCRE or LNUMWRI. Then the adjusted value is conditionally passed into a call to the translation API.

```
    If RC4  Then RC5 = ModuleN( 'FileTimeToSystemTime', LNUM&MFTime., YEAR,
                               MONTH, DAYOFWK, DAY, HOUR, MINUTE,
                               SECONDS, MSECONDS);
```

The author has seen file date times corrupted by a faulty service pack being applied to system software. The corruption caused the file time value on a large number of files to be set to an improbably high value, which also meant all the system time elements were outside the usual values. To protect against the passing of impossible values, they are all tested before being passed into a SAS function to resolve the parts into a SAS date time value.

```
    If  YEAR > Year( Date() )     Or
        MONTH > 12     Or     DAYOFWK > 7   Or    DAY > 31      Or
        HOUR > 24      Or     MINUTE > 60   Or    SECONDS > 60  Or
        MSECONDS > 999  Then Put
          "WARNING: &MDSTime cannot be created "
          "because of faulty date time data" /
          @5  NAME =    YEAR =     MONTH =     DAYOFWK =    DAY =
              HOUR =    MINUTE =   SECONDS =   MSECONDS =;

    Else  &MDSTime =
          Dhms( Mdy( MONTH, DAY, YEAR), HOUR, MINUTE,
                SECONDS + ( MSECONDS / 1000) );
```

A number of points are worth noting about the file times and their true precision.

- Not all file systems support the storage of three dates.
- On FAT based systems (as distinct from NTFS), the creation time is accurate to within 10 milliseconds of the system clock, but write time is only accurate to within two minutes. Access time is accurate to one day; in reality only the date part of the date time is correct.
- On NTFS, the update to the last access time may be delayed by up to one hour.
- On XP the retrieved date times are rounded to the correct precision when they are displayed. In NT, the values are often displayed in a misleading precision. E.g., an access time in Windows Explorer on XP will have zeros in the time parts, while on NT those parts will be populated with values.

ERROR HANDLING PROCESSES

We have already dealt with some potential issues in our process when we verified that the date time elements were all in scope. Since we are developing a macro that can be used for a wide variety of programmes, and is expected to function as a "black box", we will anticipate potential errors and attempt to handle them safely. This involves our avoiding SAS errors and warnings and including self-diagnosing code. For instance, our date time test process avoids the errors caused by an illegal parameter to the Mdy() or Dhms() functions and provides a diagnostic message if we have created a missing value.

CHECK RETURN CODES

For the most part, we have already seen how the return codes from the API calls are used to control further processing. Consider these examples.

```
FOUND = Modulen( '*e', 'FindNextFileA', HANDLE, ATT, CRE, ACC, WRI,
                 SHIGH, SLOW, DWRD0, DWRD1, NAME, BITNAME, ANAME);

If FOUND >= 1  Then
   RC4 = ModuleN( 'FileTimeToLocalFileTime', &MFTime, LNUM&MFTime.);

If RC4  Then RC5 = ModuleN( 'FileTimeToSystemTime', LNUM&MFTime., YEAR,
                            MONTH, DAYOFWK, DAY, HOUR, MINUTE,
                            SECONDS, MSECONDS);
```

- The FOUND flag is set true when the FindNextFileA API locates a file in the search stack.
- If the FOUND flag is true, then the FileTimeToLocalFileTime API is called to perform the date time conversion, and returns a non-zero value to RC4.
- If the RC4 return code is true then the date time conversion is performed with FileTimeToSystemTime.

To report if this last API has failed, we could add the following code to our SAS program.

```
If RC5 = 0  Then
  Put "Pgn Wrn: &MDSTime translation failed in the FileTimeToSystemTime API." /;
```

In developing our SAS code to use the Windows APIs, we should take account of the return from the API call and ensure it is either explicitly used, or tested to ensure there are no errors.

SET / GET LAST ERROR

In the statements we reviewed above, we avoided calling a second API if the file had not returned a positive integer as a search handle. If no file were found, this API would return –1. In general, Windows applications use null (missing in SAS), 0 or –1 values to indicate a problem with the API call. These values give us no information on the cause of the error. To test the reason for a Windows process failing, we would look for a system error flag. Just such a flag is available and uses the API GetLastError. Here is the prototype.

```
routine GetLastError
        module=Kernel32
        minarg=0
        maxarg=0
        stackpop=called
        returns=Long;
```

The difference between this API and the others we have seen is that it uses no arguments. The return code from the API is the error status. The bit value returned is a 32-bit value. The 29th bit is reserved for application specific error codes and is never used for Windows system

error codes. It should be noted that not all APIs set a system error code. We need to refer to the API documentation for more information.

The error flags set by the "Find….File" APIs are rather limited. No flag change is performed if the API succeeds, which means that any existing error flag from a previous call may persist. We need to be cautious in designing our SAS code to avoid the problems this may cause. The following statement was inserted into the process adjacent to the "FindFirstFile" call, and passes a parameter for a file that would not be found. The log note demonstrates the result.

```
RC      = ModuleN( '*e', 'GetLastError');

Put "Test FindFirstFile A, file parameter is &MDir, search handle is "
    HANDLE " GetLastError returns " RC;
```

```
Test FindFirstFile A, file parameter is c:\temp\*.xyz, search handle is -1
 GetLastError returns 2
```

Substituting a search parameter that will match existing files, we will retrieve a valid search handle. Since "FindFirstFile" doesn't update the system error when it succeeds, it seems the existing error will persist (in GetLastError). The following log entry demonstrates this result.

```
Test FindFirstFile A, file parameter is c:\temp\*.tmp, search handle is 151531960
 GetLastError returns 2
```

Let's perform the same test on the "FindNextFile" API. Since this is conditionally called, we will first look at the error returns for a search that yields two files. "FindFirstFile" will find the first, and we expect results from the second search as well as the search after the process has reached the bottom of the stack.

```
Test FindFirstFile A, file parameter is c:\temp\*.bmp, search handle is 150392088
 GetLastError returns 2
Test FindNextFile A, file parameter is c:\temp\*.bmp, found status is 1  GetLastError returns 2
Test FindNextFile A, file parameter is c:\temp\*.bmp, found status is 0  GetLastError returns 18
```

Notice that the error status 2 seems to persist from previous processes, and is not reset by either "FindFirstFile" or "FindNextFile". However, we have a valid search handle on the first file, and a valid status on the second file. At the point where we loop again looking for more files, the "FOUND" status returned is 0, and the error status is finally changed and returns 18. This is the "end of stack" error, although in this case it doesn't tell us anything we didn't already know from the "FOUND" status.

In this case, the best way to manage the error trap is to conditionally call GetLastError if the API doesn't return a valid response. Let's look at the code we developed earlier after we have added some of the extra pieces we discussed and see how it all fits together.

```
PATH    = "&MDir";
HANDLE  = Modulen( '*e', 'FindFirstFileA', PATH, ATT, CRE, ACC, WRI,
                    SHIGH, SLOW, DWRD0, DWRD1, NAME, BITNAME, ANAME);
If HANDLE >= 1  Then Do;
  FOUND = 1;
  %XGetAttr;
  Output;
  OUTRECS ++ 1;
  Do While( FOUND);
    FILEATT = ' ';
    FOUND = ModuleN( '*e', 'FindNextFileA', HANDLE, ATT, CRE, ACC, WRI,
                     SHIGH, SLOW, DWRD0, DWRD1, NAME, BITNAME, ANAME);
    RC      = ModuleN( '*e', 'GetLastError');
    If FOUND   Then   Do;
      %XGetAttr;
      Output;
      OUTRECS ++ 1;
    End;
    Else If RC = 18  Then Put "PNB: End of search with " OUTRECS "file names.";
    Else Put 'Pgm Wrn: There is a problem with your '
             'FindNextFileA API call. ' RC =;
  End;
  RC = Modulen( '*e', 'FindClose', HANDLE);
End;
Else Do;
  RC      = ModuleN( '*e', 'GetLastError');
  If HANDLE = -1  Then Put "PNB: No files found matching the criteria &MDir..";
  If RC Ne 2  Then Put
       'Pgm Wrn: There is a problem with your FindFirstFileA API call. ' RC =;
End;
```

- If the search handle is invalid then we step to the last "Do" block where we retrieve the Last Error in the red coloured statement.
- We use the "HANDLE" to report the absence of matching files, but don't use the Last Error response unless the error code is other than 2. ('2' indicates no file was found, and having two log lines saying the same thing is pointless.)
- If the search handle is valid, then immediately after we submit the "FindNextFile" API we retrieve the Last Error value.
- However, we don't process it unless the API fails to return a true status on the "FOUND" flag. We know that a value of 18 relates to the end of the search stack, so we use that value to report the end of the search along with a status message. That code is coloured blue, as well as the accumulator that holds the count of files found.
- We only output the last error value if it is other than the value of 18 that we expect.
- Notice too that we are conditionally calling a macro called XGetAttr, which is coloured **olive** in the text. This macro collects the attribute interpretation code we discussed earlier.

The SetLastError API that follows here is offered in an incomplete state. It may be that there is an incompatibility with SAS that is causing some of the unexplained behaviour of this API. If the issues are resolved, the author will publish an update on the web site. In the meantime, some behaviour of the API is as expected, and the format of the prototype is different enough from previous examples that discussing it is worthwhile.

We have already established that an error may or may not be set by an API call, so it can be difficult to know if the error is an artefact of an earlier process or an outcome of the API that was just called. Understanding the valid return values from the API and careful design of the surrounding process are still the best approach for using the API. In theory, it should be possible to set the error status to some expected value and then test it subsequent to the API call for an altered state.

The SetLastError API is defined as follows in one of the Help files used by the author for API research. The file is discussed later on in this paper.

```
The SetLastError function sets the last-error code for the calling thread.

VOID SetLastError(
    DWORD dwErrCode        // per-thread error code
   );

Parameters

dwErrCode
Specifies the last-error code for the thread.

Remarks

Error codes are 32-bit values (bit 31 is the most significant bit). Bit 29 is reserved for
application-defined error codes; no Win32 API error code has this bit set. If you are defining an
error code for your application, set this bit to indicate that the error code has been defined by your
application and to ensure that your error code does not conflict with any system-defined error codes.
This function is intended primarily for dynamic-link libraries (DLLs). Calling this function after an
error occurs lets the DLL emulate the behavior of a Win32 function.

Most Win32 functions call SetLastError when they fail. Function failure is typically indicated by a
return value error code such as FALSE, NULL, 0xFFFFFFFF, or -1. Some functions call SetLastError under
conditions of success; those cases are noted in each function's reference page.

Applications can retrieve the value saved by this function by using the GetLastError function.
```

SetLastError does not have a return value, so the prototype will be coded as follows.

```
routine SetLastError
        module=Kernel32
        minarg=1
        maxarg=1
        stackpop=CALLED;

arg 1 input byvalue num format=pib4.;   * dwErrorCode   // per-thread error code ;
```

Note from our SAS documentation that the format "pib4." Is equivalent to a long (32 bit) number. Note too that there is no "returns" value specified.

Because there is no return specified, we cannot use the "ModuleN" function because this expects a numeric return value. Instead, we will use a direct call to a "Module" function, which will branch synchronously to the API, but will not attempt to set any value on its return.
The following piece of test code demonstrates the means of calling the API in conjunction with the GetLastError API to test the change of error state. It will also show the persistence of the previous error as the loop iterates in the data step.

```
Data TEST;
  Do LOOP = 1  To 5  By 1;
    /* Test the error value before we change it */
    RC1      = ModuleN( '*e', 'GetLastError');
    Call Module( '*e', 'SetLastError', LOOP);

    /* Test the error value has been set correctly */
    RC2      = ModuleN( '*e', 'GetLastError');

    Put "PNB: Error values: previous loop " RC1 " this loop " RC2;
  End;
Run;

PNB: Error values: previous loop 2  this loop 1
PNB: Error values: previous loop 1  this loop 2
PNB: Error values: previous loop 2  this loop 3
PNB: Error values: previous loop 3  this loop 4
PNB: Error values: previous loop 4  this loop 5
```

The API certainly appears to be functioning properly, but there is a persistent problem when it is used in conjunction with some other APIs. As remarked earlier, the author has observed "FindFirstFile" always returns an error code of 2. This error should indicate "No File Found", yet testing of the search handle and observation of the data delivered shows that this was not the case. An attempt to set the Error code prior to the API did not seem to work unless the following sequence was followed:

- GetLastError (usually returns 2)
- SetLastError
- GetLastError (returns value set in previous statement)
- Call "FindFirstFile" API
- GetLastError (usually returns 2).

We know the GetLastError API is functioning correctly because it correctly returns 18 when the end of the search stack is reached in "FindNextFile". In the previous test as well, the value set on one iteration of the loop was correctly detected at the end of the loop, and persisted into the next iteration, as we would expect. So there remains an unexplained behaviour with the errant value of 2.

For those wishing to set their own error values, the author uses the value 20000000x, which sets the 29th bit of the error code. (The documentation above reports that bit 29 is not set by any OS processes.) The persistence of this value should mean we could be confident that no system process has tried to set the error flag.

MODULE TEST PARAMETERS

SAS too provides debugging information in this process. You'll recall that there was a special parameter in "FindFirstFile" API call. The parameter is highlighted in the code immediately above. This first parameter can be dropped without affecting the normal performance of the ModuleN call. However, by using one of the following values, it is possible to make development simpler and production use more robust.

- *h calls for help data to be output on the function ModuleN(). This allows developers to check syntax of all aspects of the Module call.
- *i calls for an information output on the Module. The information is quite detailed and includes hexadecimal data and the argument pointers to be printed to the log. The detail provided is of greatest use during development and the parameter would usually be dropped once development is complete.
- *e calls for error data only to be output. The material is less detailed than that provided by the information output. It is best suited to production use since output is suppressed if the Module is processed without error. Note that error data appears with the information data if an error occurs while the *i parameter is set.
- *Sx changes the separator character between field definitions in the call from the default comma to the character specified in 'x'. This allows one or more commas to be embedded in parameters passed to the API, perhaps for messaging purposes in the messaging APIs.
- *t list attribute information.

Here is the log entry generated by the use of the '*i' parameter to the FindClose API. In the case of more complex API calls, it is possible to track the parameter values passed to the call, and the values returned after completion of the call.

```
---PARM LIST FOR MODULEN ROUTINE---
CHR PARM 1 0521BC4C 2A69 (*i)
CHR PARM 2 0521BC43 46696E64436C6F7365 (FindClose)
NUM PARM 3 0521B720 000000309AEDA141
---ROUTINE FindClose LOADED AT ADDRESS 7C80EFD7  (PARMLIST AT 0B331808)---
PARM 1 18CDF608     <CALL-BY-VALUE>
---VALUES UPON RETURN FROM FindClose ROUTINE---
PARM 1 18CDF608     <CALL-BY-VALUE>
---VALUES UPON RETURN FROM MODULEN ROUTINE---
NUM PARM 3 0521B720 000000309AEDA141
```

In the following log entry, a missing argument forces the notification of a problem with the call. The same report is delivered whether the API is called with the parameter '*e' or '*i'. If the parameter '*i' is used, then the detailed parameter list reported above is suppressed since the Module does not complete the process of passing the call to the OS. We need to observe that the problem with the API call is reported with a SAS note. <u>This reinforces the sound advice that the log of a SAS session should be checked carefully at the end of the run</u>.

```
NOTE: Module FindClose was not given its minimum argument count of 1.
NOTE: Invalid argument to function MODULEN at line 11 column 168.
FILENAME=iana_x_s EXTENSN=bmp CREATESD=21JUL2004:06:14:40.76 ACCESSSD=05NOV2004:00:00:00.00
WRITESD=21JUL2004:06:14:42.00 FILEATT=  FILESIZE=41,878 ANAME=  NAME=iana_x_s.bmp
PATH=c:\temp\*.bmp BITNAME=  ATT=32 CRE=1.2734824E17 ACC=1.2744047E17 WRI=1.2734824E17
SLOW=41878 SHIGH=0 RC=. NUM1=. NUM2=. LNUM1=. RCODE=0 DWRD0=41878 DWRD1=41878 HANDLE=149736568
FOUND=0 POSITION=9 YEAR=2004 MONTH=7 DAYOFWK=3 DAY=21 HOUR=6 MINUTE=14 SECONDS=42 MSECONDS=0
LNUMCRE=1.2734864E17 RC4=1 RC5=1 LNUMACC=1.2744086E17 LNUMWRI=1.2734864E17 OUTRECS=2 _ERROR_=1
_N_=1
NOTE: Mathematical operations could not be performed at the following places. The results of
      the operations have been set to missing values.
      Each place is given by: (Number of times) at (Line):(Column).
      1 at 11:168
```

RESET FILE ATTRIBUTES

Windows APIs allow us to perform changes through the OS, which can be even more useful than reading the values of files and system settings. The author found one of the most useful functions has the ability to change attributes of a file.

I had to do this for a large set of files when an update to a networked antivirus program caused the date time values for all the files to be changed to an impossibly high value. With a SAS project in the late development phase, it was impractical to go back to the latest set of full network backups, which would have cost us three days work.

Among other things, we found that because the date time values were invalid, our local file backup process wasn't working. It was a SAS based process that compared the date time stamp of a file with the date time stamp of a zip file. If the file bore a later date then it was copied to a zipped archive. In that way the system developers had their own set of local backups to use for recovery or roll back of versions of our SAS data sets, code or catalogues.

Recovering valid dates would restore our ability to verify the change process in our system-testing plan, and get our support processes working again.

CHANGING FILE DATE TIMES

Resetting the values involved reading the appropriate date values from the internal headers of SAS tables, or entry dates for SAS catalogs and then applying these dates to the Read, Write and Access dates for the files. To set a date time on a file we used the API "SetFileTime". Here is the prototype for that API.

```
ROUTINE SetFileTime
  minarg=4
  maxarg=4
  stackpop=called
  module=kernel32
  returns=long;
arg 1 num update byvalue format=pib4.;  * hFile            // handle to file ;
arg 2 num update format=pib8.;          * lpCreationTime   // creation time ;
arg 3 num update format=pib8.;          * lpLastAccessTime // last-access time ;
arg 4 num update format=pib8.;          * lpLastWriteTime  // last-write time ;
```

Bear in mind that when we retrieved the file time in our earlier searches, we performed two translations. The first: "FileTimeToLocalFileTime"

translated the value from GMT to the local clock, and the second: “FileTimeToSystemTime” parsed the file time value to its date time parts. The values we apply to this API, which you will note can change all three values in one pass, must have been converted from a date time value to a Windows File time value, and then offset from the local clock to GMT.

To perform these changes we need to add APIs for “SystemTimeToFileTime” and “LocalFileTimeToFileTime”. In a later section of this paper, we will develop these API prototypes as an exercise. This will give us experience of developing our own prototypes.

To get the handle for the file, it is possible to use the FindFirstFileA API with a parameter that specifies only that file. Then use the SetFileTime API to update the file times. If updates are only going to be applied to one or two dates, the use of a missing value on the remaining file time(s) will translate to a Null value at the OS level. The file time structure(s) that are passed a null value will remain unchanged.

CHANGING FILE ATTRIBUTES

The binary values associated with the attributes of a file are detailed above. If we wish to alter the attributes of a given file, we can calculate the required attribute number and then use the API SetFileAttributesA. Here is the prototype for that API.

```
routine SetFileAttributesA
        module = Kernel32
        minarg = 2
        maxarg = 2
        stackpop = called
        returns = short;

        arg 1 char byaddr  format = cstr200.;
        arg 2 num  byvalue format = pib4.;
```

Certain aspects of this prototype differ from others we have seen. The first argument is, as usual, required to identify the target of the API function. Unlike the SetFileTime and some other APIs, this API uses a string search on the file name to locate the target file. Since we use a name search to identify the file, we don’t yet have a handle on the file. So the first parameter uses an address to identify the file. That’s why we have a “byaddr” attribute on the argument. The second argument is the new file attribute value, which is supplied as a numeric value. Therefore the argument is specified to be applied “byvalue”.

Note that not all attributes can be set with the SetFileAttributes API.

- Compressed; the compression attribute needs to be set through the device I/O control.
- Directory; files cannot be converted to directories.
- Normal; setting the normal attribute on a file overrides all other attributes, and setting any attribute other than normal releases the normal flag.

In a sense, this is probably one of the simplest APIs to use, since it only requires a single ModuleN call, and is passed a file name parameter in a format we readily understand and use.

FIND API DEFINITION

While the APIs detailed above are important, there is often a need to find the data necessary for us to create new API prototypes. These were the starting point for the author’s collection of API based macros, and the process needed to research and implement new prototypes will be explored next. A caution should be given first though.

API calls work more directly with the operating system and memory than SAS does, and errors may cause your SAS session to hang, the SAS session to abruptly terminate, the OS to hang or the OS to go to the dreaded “blue screen”. The author has observed this rather too often on Windows 95, NT, 2000 and XP home platforms. Indeed, in the attempt to force an API error message, a SAS session was terminated during the preparation of this paper. Since that was the allegedly more stable Windows XP, no OS should be considered robust enough to be immune from such problems.

When developing APIs, it is best if only the SAS session and help files or an Internet browser are open, and that code is saved before every test and submission. Starting the SAS session with the “–altlog” will ensure the log is saved to a file in case the session terminates. This option should also be used with the undocumented option “–unbuflog” which commits log data to disk as soon as it is generated, rather than waiting for a full data page before committing it to disk. From these we may be able to recover lost code and identify issues with changes we have made.

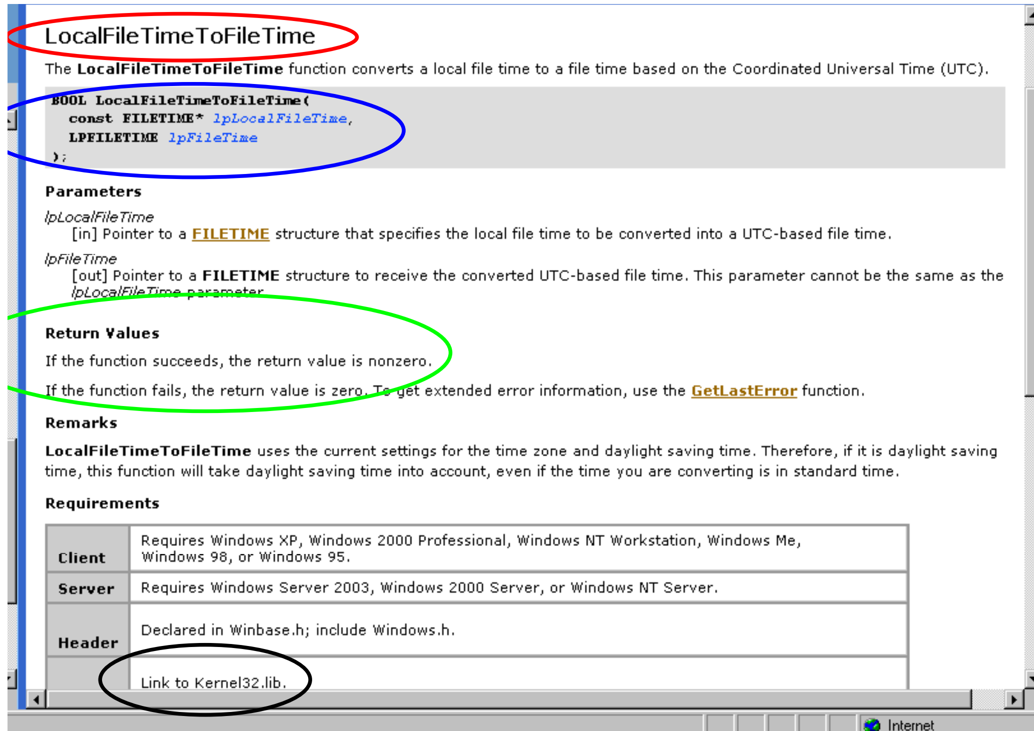
BUILDING A PROTOTYPE FROM MSDN

In the previous section, two APIs were identified for which no prototypes were defined. They were SystemTimeToFileTime and LocalFileTimeToFileTime. They are a simple place to start because we already have two similar prototypes to guide us. Given the risks involved in using incorrect API prototypes, it is important to try to get them right first time. That means that we shouldn’t try to develop an API from first principles if we have a similar structure to take us part way there. To demonstrate the process, we’ll develop LocalFileTimeToFileTime by first reviewing the structure used for FileTimeToLocalFileTime.

```
ROUTINE FileTimeToLocalFileTime
  minarg=2
  maxarg=2
  stackpop=called
  module=kernel32
  returns=ulong;

arg 1 input  fdstart num format=pib8.;  * lpFileTime,     // UTC file time to convert;
arg 2 output fdstart num format=pib8.;  * lpLocalFileTime // converted file time;
```

From an enquiry on a search engine using the key word “LocalFileTimeToFileTime” the following link was surfaced: http://msdn.microsoft.com/library/en-us/sysinfo/base/localfiletimetofiletime.asp. The author used to keep a link to the root of the MSDN API library, but it is changed reasonably often and a web search is more reliable to find the documentation. At figure 2 is a partial image of the page returned.


LocalFileTimeToFileTime

The **LocalFileTimeToFileTime** function converts a local file time to a file time based on the Coordinated Universal Time (UTC).

```
BOOL LocalFileTimeToFileTime(
  const FILETIME* lpLocalFileTime,
  LPFILETIME lpFileTime
);
```

Parameters

lpLocalFileTime
[in] Pointer to a **FILETIME** structure that specifies the local file time to be converted into a UTC-based file time.

lpFileTime
[out] Pointer to a **FILETIME** structure to receive the converted UTC-based file time. This parameter cannot be the same as the *lpLocalFileTime* parameter.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, use the **GetLastError** function.

Remarks

LocalFileTimeToFileTime uses the current settings for the time zone and daylight saving time. Therefore, if it is daylight saving time, this function will take daylight saving time into account, even if the time you are converting is in standard time.

Requirements

Client	Requires Windows XP, Windows 2000 Professional, Windows NT Workstation, Windows Me, Windows 98, or Windows 95.
Server	Requires Windows Server 2003, Windows 2000 Server, or Windows NT Server.
Header	Declared in Winbase.h; include Windows.h.
	Link to Kernel32.lib.

Internet

Figure 2

In building the new prototype we need to verify each element in turn.

- The name of the prototype appears at the top of the screen and is circled in red.
- There are two arguments specified (circled in blue), and it makes sense that we need both arguments to perform the function. So our minarg and maxarg values will be 2.
- Our memory management (stackpop) will usually be “called”.
- In the requirements section (circled in black) is the first mention of the library containing the API. This indicates that the required module will be Kernel32.
- Return values are likely to be the same and will usually be a long integer.
- Note that the documentation for FileTimeToLocalFileTime included comments against the arguments, which specified the Windows variable names. By comparing these names to the current entry we can see they are the same, and write the arguments reversing the order of the arguments.

Here is the prototype we would develop.

```
ROUTINE LocalFileTimeToFileTime
  minarg=2
  maxarg=2
  stackpop=called
  module=kernel32
```

```
   returns=long;

 arg 1 output fdstart num format=pib8.;  * lpLocalFileTime // converted file time;
 arg 2 input  fdstart num format=pib8.;  * lpFileTime,     // UTC file time to convert;
```

From this simple example, we can now look at the specification for SystemTimeToFileTime and compare it to the FileTimeToSystemTime prototype that we have. Here is the original prototype we used for our file find process, and at figure 3 an image of the SystemTimeToFileTime API information from MSDN.

```
ROUTINE FileTimeToSystemTime
        minarg=9
        maxarg=9
        stackpop=called
        module=Kernel32
        returns=long;

  * CONST FILETIME lpFileTime,    // pointer to file time to convert ;
arg  1 num  input          format=pib8.;

  * LPSYSTEMTIME   lpSystemTime   // pointer to structure to receive system time ;
arg  2 num  output fdstart format=pib2.;   *   WORD wYear ;
arg  3 num  output         format=pib2.;   *   WORD wMonth ;
arg  4 num  output         format=pib2.;   *   WORD wDayOfWeek ;
arg  5 num  output         format=pib2.;   *   WORD wDay ;
arg  6 num  output         format=pib2.;   *   WORD wHour ;
arg  7 num  output         format=pib2.;   *   WORD wMinute ;
arg  8 num  output         format=pib2.;   *   WORD wSecond ;
arg  9 num  output         format=pib2.;   *   WORD wMilliseconds ;
```

SystemTimeToFileTime

The SystemTimeToFileTime function converts a system time to a file time.

```
BOOL SystemTimeToFileTime(
  const SYSTEMTIME* lpSystemTime,
  LPFILETIME lpFileTime
);
```

Parameters

lpSystemTime
[in] Pointer to a SYSTEMTIME structure that contains the time to be converted.

The **wDayOfWeek** member of the **SYSTEMTIME** structure is ignored.

lpFileTime
[out] Pointer to a FILETIME structure to receive the converted system time.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Example Code

For an example, see Changing a File Time to the Current Time.

Requirements

Client	Requires Windows XP, Windows 2000 Professional, Windows NT Workstation, Windows Me, Windows 98, or Windows 95.
Server	Requires Windows Server 2003, Windows 2000 Server, or Windows NT Server.
Header	Declared in Winbase.h; include Windows.h.
Library	Link to Kernel32.lib.

Internet

Figure 3

Once again, we can verify the name and types of arguments for the API. We note that a FILETIME structure is referred to in the documentation, a structure we used in the FileTimeToSystemTime by declaring the first argument of the structure with the special key word "fdstart". So we can define the new API prototype as follows.

```
ROUTINE SystemTimeToFileTime
        minarg=9
```

```
        maxarg=9
        stackpop=called
        module=Kernel32
        returns=long;


  * LPSYSTEMTIME   lpSystemTime   // pointer to structure to receive system time ;
 arg  1 num  output fdstart format=pib2.;   *   WORD wYear ;
 arg  2 num  output         format=pib2.;   *   WORD wMonth ;
 arg  3 num  output         format=pib2.;   *   WORD wDayOfWeek ;
 arg  4 num  output         format=pib2.;   *   WORD wDay ;
 arg  5 num  output         format=pib2.;   *   WORD wHour ;
 arg  6 num  output         format=pib2.;   *   WORD wMinute ;
 arg  7 num  output         format=pib2.;   *   WORD wSecond ;
 arg  8 num  output         format=pib2.;   *   WORD wMilliseconds ;

  * CONST FILETIME lpFileTime,    // pointer to file time to convert ;
 arg  9 num  input          format=pib8.;
```

This method makes good use of the resources published to the Web by the MicroSoft Developers Network, and has the advantage that other relevant information is likely to be indexed in the search results and can be called upon if difficulties are encountered.

Before moving on, we'll review a process used by the author to fix a handful of files. Here is the core of a macro that processed a handful of SAS tables corrupted by a file system problem.

```
Proc Sql  _Method STimer;

  Create Table SETTIME As
    Select CRDATE, MODATE
    From Dictionary.Tables
    Where LIBNAME = 'WEBLOGS'  And
          MEMNAME = "&MFile";

Quit;
```

The creation and modification date for a table are read from the Dictionary.Tables SQL view, and stored to a table.   The table is selected by name matching the Macro parameter "MFile".

```
Data SETTIME( Keep = CRDATE CREATED ACCESSED MODATE WRITTEN);
  Set SETTIME;

  *** Creation date is set to data set CRDATE from DS header;
  CREATED = .;
  WRITTEN = .;
  ACCESSED = .;
  MSECONDS = 0;
  SECONDS = Second( CRDATE);
  MINUTE = Minute( CRDATE);
  HOUR = Hour( CRDATE);
  DAY = DAY( DatePart( CRDATE) );
  DAYOFWK = WeekDay( DatePart( CRDATE) );
  MONTH = MONTH( DatePart( CRDATE) );
  YEAR = YEAR( DatePart( CRDATE) );
  RC = ModuleN( 'SystemTimeToFileTime', YEAR, MONTH, DAYOFWK,
                DAY, HOUR, MINUTE, SECONDS, MSECONDS, CREATED);

  *** Modification date is set to data set MODATE from DS header;
  SECONDS = Second( MODATE);
/* Repeated code that is similar to the above derivations removed for clarity */
  RC = ModuleN( 'SystemTimeToFileTime', YEAR, MONTH, DAYOFWK,
                DAY, HOUR, MINUTE, SECONDS, MSECONDS, WRITTEN);

  *** Last accessed date is set to current date and time;
  SECONDS = Second( DATE() );
/* Repeated code that is similar to the above derivations removed for clarity */
  RC = ModuleN( 'SystemTimeToFileTime', YEAR, MONTH, DAYOFWK,
                DAY, HOUR, MINUTE, SECONDS, MSECONDS, ACCESSED);
  Format CREATED ACCESSED WRITTEN 32.;
Run;
```

Using the CRDATE value, the MODATE value and the current date, new date times called CREATED, WRITTEN and ACCESSED were created. The process was run during winter 2003 in England. The local date time was the same as GMT, so there were no LocalFileTime conversions required.

```
Data ZZZZFILE( Keep = PATH STATE CREATEDS UPDATEDS ACCESSDS);
  Length  PATH $200;
  Set SETTIME( Keep = CREATED ACCESSED WRITTEN);
  CRE    = .;
  ACC    = .;
  WRI    = .;
  NAME = PATH;
  LNUM1  = .;
  PATH = "&MLanRoot.DJ\&MFile..sd2";
  STATE = 'Get time values';

  RC     = ModuleN( 'SetLastError', 0);

  HANDLE = ModuleN( 'CreateFileA', PATH, 2**30, 0, 0, 3, 0, 0);
  RC = ModuleN( 'GetFileTime', HANDLE, CRE, ACC, WRI);

  RC     = ModuleN( 'GetLastError');
  If RC Ne 2  Then Put 'ERROR: There is a problem with your API call.';

  If HANDLE >= 1  Then Do;
    %XGetTime( MFTime = CRE, MDSTime = CREATEDS);
    %XGetTime( MFTime = WRI, MDSTime = UPDATEDS);
    %XGetTime( MFTime = ACC, MDSTime = ACCESSDS);
    Output;

    RC = ModuleN( 'SetFileTime', HANDLE, CREATED, ACCESSED, WRITTEN);

    RC = ModuleN( 'GetFileTime', HANDLE, CRE, ACC, WRI);
    %XGetTime( MFTime = CRE, MDSTime = CREATEDS);
    %XGetTime( MFTime = WRI, MDSTime = UPDATEDS);
    %XGetTime( MFTime = ACC, MDSTime = ACCESSDS);
    STATE = 'Set time values';
    Output;

    RC = Modulen( 'CloseHandle', HANDLE);
  End;
Run;
```

An audit trail of the process was retained. We store the date time values against a “state” discriminator, which identifies the point at which the date time was read. Current date time values are first stored against a state of “Get time values”, the new date time values were applied, and then read back from the file data and stored with a state of “Set time values”.

In this case, another file search API called CreateFileA was used. If a file exists the API will return a handle on the file, and if it doesn’t then the file will be created with the parameters defined. (For those more familiar with Unix, the “touch” command has similarities.) In this case it has the benefit of only retrieving a single file, and doing that with a simpler call.

```
Proc APPEND Base = MYDATA.CHGEDATE
            Data = ZZZZFILE;
Run;
```

Finally, the change records are written to a master audit trail table for review later. The PATH value at the top of the macro appears to be working on a subsidiary directory to the root LAN data directory. This version is either a test version, or the existing tables were copied off from the root directory for processing and only copied back once the changes had been completed and verified.

BUILDING A PROTOTYPE FROM PUBLISHED REFERENCES

When the author first started developing these prototypes, web information was scant or unavailable, and the best data was published in various books available from the local library or technical bookshop. The following and last example of “rolling your own” prototype follows the path taken to develop an API to delete a file. The API was required because the method using a system call with a DOS command lacked any meaningful return from the system.

The name of the reference is not recorded, but the author subsequently bought a similar reference work called the “WIN32 API superbible.” An image of the front cover appears at figure 4 for those wishing to obtain the latest edition of the work. Various editions are still shown as available from online book retailers.

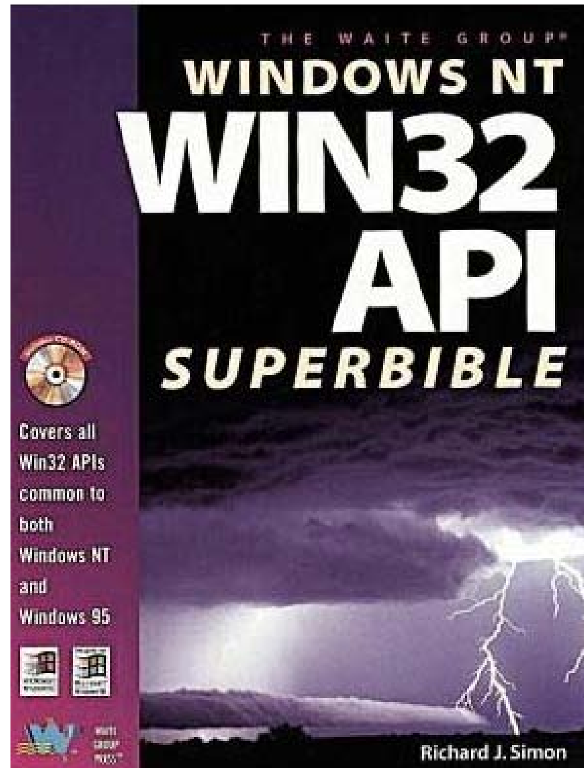


Figure 4

The following API specification is similar to the one retrieved at that time.

```
The DeleteFile function deletes an existing file.

BOOL DeleteFile(
    LPCTSTR lpFileName     // pointer to name of file to delete
   );
```

Parameters

```
lpFileName
Points to a null-terminated string that specifies the file to be deleted.
```

Return Values

```
If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.
```

Remarks

```
If an application attempts to delete a file that does not exist, the DeleteFile function fails.

Windows 95: The DeleteFile function deletes a file even if it is open for normal I/O or as a memory-
mapped file. To prevent loss of data, close files before attempting to delete them.

Windows NT: The DeleteFile function fails if an application attempts to delete a file that is open for
normal I/O or as a memory-mapped file.

To close an open file, use the CloseHandle function.
```

From the above we have a routine name; we know a single parameter is required, and that the parameter will always be required. We know there will be a return code from the process, and we know that the single argument to the prototype will be a null terminated string.

FINDING THE NAME OF THE API

Sharp-eyed readers will have recognised that there were sometimes differences between the name of an API and the name of the routine. That is because the MSDN documentation (and indeed most documentation) deals with the generic version of an API. Some APIs have multiple versions to deal with extended attributes or wide data streams. The addition of “A”, “EX” or “W” will then identify the version of the API that is required. We need to identify which version of the API is available to us. To do that we can examine the DLL that holds the API.

In another part of the documentation we also discover that the API is part of the module “KERNEL32”. We can examine the DLL header data if we use a tool like QuickView, which was available in Windows 95 and 98. Similar DLL Explorer tools are available for more recent OS, and the following is a partial screen shot from one of them examining the “export” set for Kernel32.dll. (The “export” set is the functions that are exported to the OS by the DLL.)

Figure 5

Notice that there are two versions (circled in **red**), DeleteFileA, which is identified in the syntax section as the “ANSI character” version (circled in **green**), and DeleteFileW, which is the wide character version.

Based on the data we have assembled, we can now build the prototype. Note that the name on the routine has been changed from the general API name to the specific module we wish to call in the DLL. Note too that the specification for argument 1 is for a character string up to 260 bytes in length. This is the maximum length for a file name in Windows, but is not compatible with SAS Version 6. If the API is to be with that version then the length needs to be adjusted to 200 bytes.

```
routine DeleteFileA
  minarg=1
  maxarg=1
  stackpop=called
  returns=long
  module=kernel32;

arg  1 char input format=$cstr260.; * LPCTSTR lpFileName // pointer to name of file to delete;
```

In later developments other resources assisted the author, including the book pictured above. At that time too, one of the UK PC magazines released a free version of Borland’s Delphi product on its cover disc. Within the help files for the product was a file called Win32.Hlp, which was a compiled help file, containing details of the parameters of Windows APIs. The Help file was completed before the release of Windows 2000, so some more recent changes and additions are not included. However, most of the core of the Windows OS has changed little and the author is still able to use this resource on a Windows XP machine.

Since it also became evident that Delphi (and C+, VB and other applications) developers faced the same issue of a defined prototype, albeit with a slightly different syntax, the author discovered a significantly wider support group among these developers.

CONCLUSION

We have examined a number of Windows APIs and the SAS functionality around them. We might summarise our observations as follows.

- Windows APIs provide a robust and informative link to Windows OS functions.
- They allow us to read OS and file system settings as well as change them.
- We can use them from within a data step, which gives us substantial power to populate data sets with values from the OS, and use data set values to make changes.
- If we wrap them in a macro then we can port substantial power and functionality to our users, with a simple code interface.
- We need to be careful in development because of the potential impact on our SAS or OS sessions, but if we regularly save our code and divert our log then the worst outcome is a reboot followed by a clean session to finish the task at hand.
- We have a variety of information resources to support our development, and since our interface to the OS is in common with developers in VB, C++, Delphi and other application builders, we can draw on a wide range of expertise for assistance.
- Because of the complexity of some of the functions, we are careful to document the prototypes we develop, and use existing calls as a template for new development. SAS users are used to taking care with syntax and their process flows, so development should not be onerous.

- Our interface to the OS is through a text file containing a set of statements called a “prototype”.
- Our text file must be assigned to our SAS session with the reserved file reference SASCBTBL.
- When we call the OS function we use a statement in the form “`ReturnCode = ModuleN( <control code(s)>, prototype, arg1, … argn );`” for functions with a numeric return value.
- For a function with a character return value, we use the “`ModuleC()`” call.
- If there is no return code, then we use the “`Call Module();`” statement.

REFERENCES

The Win32 API Super Bible (MacMillan computer publishers 1997 ISBN: 1571690891)
Win32 API help file (Borland Delphi and other application development software)
TS-575 Preliminary documentation for CALL MODULE (SAS Institute 2000)
MicroSoft Developer Network

ACKNOWLEDGMENTS

Clients who require robust applications
SAS Technical Support (Cary, UK & Australia)

CONTACT INFORMATION

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

Author Name	David H. Johnson FRSA
Company	DKV-J Consultancies Ltd
Address	4 Bataba St
	Moorabbin Vic
	Australia 3189
Work Phone:	+61 3 9553 0301
Fax:	+44 7005 98 0829
Email:	Sugi30@dkvj.com
Web:	http://www.dkvj.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.