Paper 280-26

Coding across the boundaries

David Johnson, DKV-J Consultancies, Holmeswood, England

ABSTRACT

One of the challenges facing a consultant in the SAS System® is the need to deliver a 'fast start' on a new contract. One of the ways to achieve this, is to start an assignment with a toolkit of common utilities.

If a utility is developed for the Windows platform to find and store the characteristics of files, a similar macro can be developed for the Unix or MVS environments. If the same name and set of parameters is used for the macro on each platform, then programs developed around the macro for one platform can be quickly deployed to a different platform.

This allows the consultant to more quickly focus their attention on the main variable between assignments: the nature and structure of the business data.

INTRODUCTION

Sometimes a consultant's assignment requires development of Production code for a mainframe platform. Access to the platform may be limited, or writing of the program involve development and testing of an extensive set of data cleansing rules.

Early development may be better suited to a Windows or other Graphical User Interface (GUI) platform. In this case, the availability of operating system interface macros with common names and parameters will allow the code to be developed on one system and ported to another.

This paper explores some of the problems and solutions involved in developing macros that function on multiple operating systems. We explore some of the development techniques used for Windows macros using the Windows Application Programming Interface (API).

We look at the analysis performed to ensure the tool is 'general' enough for versions on multiple platforms. We also analyse a number of macros for the Windows, Unix and MVS platforms that perform similar functions in widely different circumstances.

IT WASN'T DEVELOPED HERE

Sometimes, for resource, choice or availability reasons, code may need to be developed on one platform and run on another. The change freeze for Production platforms around year 2000 was a classic occasion where project development may not be stopped, but any change at all in the core reporting systems was an unacceptable risk.

It was this restriction that moved a clients' substantial data cleansing effort from the mainframe to the Warehouse' Unix platform. Access to the platform was limited and used rudimentary tools, so most code development occurred on the Windows platform and was run on Unix.

In this situation, development of the code to load our new data warehouse was prototyped and tested on a Windows platform, but was run on Unix. Since programs were transferred between the platforms and run locally, rather than through SAS/Connect, macros called within the program needed to exist and work on both platforms.

Many of the macros used calls to the Operating System. Those developed for Windows used the Windows API almost exclusively. Those run on Unix used Operating System commands submitted and captured through 'PIPEd' filename allocations.

FINDING FILES IN WINDOWS

The Windows API to find files involves a number of smaller pieces, and is dealt with in detail at the end of this paper. In summary though, once a file that matched a search criterion was located, a call to a series of APIs would surface details on the file parameters. These included:

- File modification date
- File attribute flags - all systems: R (Read Only), A (Archived), S (System) and H (Hidden)
- File attribute flags - Windows NT: Compressed.
- File creation date (not available Windows 95)
- File access date (not available Windows 95)

It was clear very early in the design process that not all issues occurred across platforms. If a Windows macro was to be effective as a generic tool, then it must also deal with a number of different versions of the Windows Operating System.

FINDING FILES IN UNIX

In Unix, there was no API interface readily available. So the device type 'Pipe' was invoked on the Filename allocation. This allowed an Operating System command to be submitted, and its results surfaced to the SAS session. The Unix command 'ls -al' allowed adequate data to be made available on Unix files, but the parsing and validation of the text returned was too complex to be undertaken on an ad-hoc basis. A macro was needed to deal with the data reading and validation on a consistent and robust basis.

On comparison with the Windows return values, a number of issues arose. The file dates available related solely to the creation date of the file, and the file attributes were substantially different. Within Unix, a file has three possible actions. They are 'R' (Read), 'W' (Write) and 'X' (eXecute).

However, they are defined for three different types of user. The first relates to the file owner, the second to the group to which the file belongs and the third to the global environment. In this way a file may, for example, be updated by the owner, read by other members of the owners group and hidden from all other users on the machine.

BRINGING FILE INFORMATION TOGETHER

You can see that this presents some real problems to the user. If one distils the information surfaced by each environment (Windows 95, Windows NT and Unix) to the lowest common denominator, then valuable information is lost from each.

The solution was to find the common points of agreement, and name the variables for each Operating System similarly. Then additional variables were made available on the data set created to allow for the extra attributes. It was also essential that each macro was comprehensively documented, recording the points of comparison across Operating Systems, and providing warnings

for those areas where differences would cause cross platform problems.

HOW DO THE MACROS COMPARE?

Looking more closely at the macros developed for Windows and Unix, we find that they have **common names and parameters**. This was the first and most important rule of development. To support this naming, all operating system macros were named with 'X' as the first character.

Where one Operating System requires an additional parameter not provided on another, the extra parameter was provided on both macros. It was assigned a default value on the macro that didn't populate it meaningfully. In this way, a macro that had been extensively used previously could be extended without threatening existing applications.

The data set built within the macro for file finding had a common name on both macros. To prevent issues of contention however, the **name of the data set always met a certain naming standard**. All began with the letters ZZ, and the site programming standards specified that no data set named with ZZ was to be created in any application. This was also most important, and one of the few times I have seen a SAS programming standard rigidly enforced.

Aside from the 'File Find' macros, many others were created, some providing a single value within a macro token. The third standard to be applied defined that all such macro tokens would begin with 'Z' and no code outside a System macro was to create any macro similarly named. This was the third standard to be rigidly applied.

A SIMPLE EXAMPLE

Since the application was being developed as a Production system, responsibility for reports and output generated from the system could change on a regular basis. The identification of the user running the SAS session that produced the output, would be the person responsible for following up issues. Differing approaches needed to be adopted depending on the platform. Let us look at how a solution for mainframe Production reporting jobs was applied to two other platforms.

REPORT NAMING - MVS PROGRAMS

For mainframe jobs, the identity of the person running the jobs was held in the name on the JCL job card. The standard SAS macro token ***SysJobId*** would surface this value. By retrieving this value within a simple macro, additional functionality to compare the value against a database could be used to report the name of the user. This database could be populated on a regular basis from extracts generated from Access Control Files (ACF, RACF et al).

For full production runs, the standards at most sites mean the job has a production name that bears no resemblance to the author. In this case, the value retrieved would be the same as the name of the SAS program file. However, since most mainframe production sites separate the JCL and SAS code into separate members of different libraries, the JCL would use a SYSIN card to retrieve the SAS code.

The name of the 'included code file' would then be stored in, and retrievable from, the ***SYSIN*** SAS system macro token. If these two values agreed, then the job was apparently a Production reporting job. In this case the name of the person responsible ('Duty Analyst') could be retrieved from a file updated as different people took responsibility for the batch processes… perhaps monthly. The process diagram looks something like that in Figure 1.

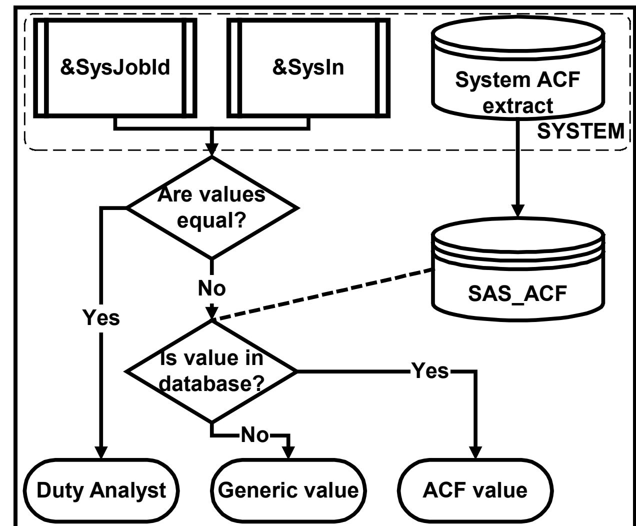

Figure 1

The approach proved to be robust, and in the best traditions of Mainframe 'shops', the monthly reports were produced with footnote entries that identified the name of the person responsible, and at one site, their telephone number. All of the code was embedded in a macro called ***XGetUser*** which was distributed in the 'SASAUTOS' directory assigned to the SAS session at start-up. This macro also created the footnote for the report, which included the name of the SAS source code library entry (***SysIn***). In later releases, the macro was further extended to identify the email recipient for critical messaging.

The further benefit of this approach was that development programs were also written with a call to the macro. Since the site standard specified that certain macro values were 'reserved' for local processing rules, and that the first footnote would be provided by the macro, all output generated was automatically identified with the analyst responsible. We'll come back once more to the 'reserved' values after we look at the same approach for Unix and Windows.

REPORT NAMING - UNIX PROGRAMS

The problem with the same approach for SAS on the Unix platform can be found in the 'SAS Companion for Unix environments'. It states: " (***SYSJOBID***) lists the PID of the process that is executing the SAS System, for example, 0024". The Process Identifier (PID) is derived from the Unix master list of processes running on the machine at the time of the start of the process. It bears no relation to the user who started the job.

To retrieve the name of the user, a different approach was needed. The approach involved sending a command to the Operating System, and then retrieving the result. The method used was to issue a SAS 'Filename' statement with the device option set to 'PIPE'. Then the command was embedded in the external file reference. Referencing the file reference in a data step would cause the command to be issued, and surface the return value to the data step. Then this could be parsed and a value retrieved. The SAS code looks like this:

```
Filename  SYSREQST  Pipe  'env';
Data _NULL_;
  Retain USERNAME 'unknown username';
  Infile SYSREQST  TruncOver;
  Input @1 READSTR $char80.;
  If READSTR Eq: "LOGNAME" Then  USERNAME =
     Trim( Substr( READSTR, Index(
       READSTR, '=')+1 ) );
```

```
Run;
```

This is an abbreviated version of the actual macro that was deployed. The final macro used the environment (env) command to also surface data on the current working directory, timezone, path assignment, mail path and terminal type for other system integrated functionality.

The user set-up for Unix sites may change from place to place, but this was modelled at a site where the following rules were applied:

- The production username for SAS applications was 'SASADMIN'.
- All production SAS data was owned by SASADMIN.
- The SASADMIN and all individual users of SAS applications were members of the user group 'SASUSERS'.
- All data created by SASADMIN and all members of SASUSERS was readable by the group SASUSERS, and hidden from the general user population.

Consequently, the 'Owner' permissions were a proxy for the effective file permissions. We also found that all SAS data was shared and secured. This meant that Production batch runs were identified as initiated by SASADMIN, and reports were footnoted with the name of the 'Duty Analyst' or the general project help line as appropriate.

REPORT NAMING - WINDOWS PROGRAMS

If Unix was difficult, Windows was even more complex. The 'SAS Companion for the Microsoft Windows Environment' tells us "(***SYSJOBID***) returns a number that uniquely identifies the SAS task under Windows". This is similar to the Unix process number, and equally uninformative. While some sites store user name information in Operating System tables or Windows environment variables, this is not done consistently.

The first point of consistency across most sites however, is that the Windows machine is usually connected to a Local Area Network. A LAN identity may then provide a unique user identifier. To use this value, we need to surface the LAN ID through a call to the Windows API.

Issuing an executable command through an External DLL uses the Windows API. These Dynamic Link Libraries are files that contain many commands or utilities that may be used by many applications. The process for using external DLLs is treated in some detail in a number of additional resources made available by SAS. These are listed at the end of the article and are very good supplementary resources.

The principle of using external DLLs is threefold:

- Create a 'prototype' for the command that contains the DLL name, the function name, and the attributes that need to be passed to the DLL.
- Save this prototype in a text file and make it available to the SAS session by using the reserved file name 'SASCBTBL'.
- Call the command by using the 'ModuleN' function with the appropriate number of parameters required.

Here is a prototype for an API that will find the Network logon identity for a user when Windows networking is employed:

```
routine WNetGetUserA
  minarg   = 3
  maxarg   = 3
  stackpop = called
  returns  = long
  module   = mpr;
arg 1 char  update  format = $cstr20.;
arg 2 char  output  format = $cstr20.;
arg 3 num   update  format = pib4.;
```

Retrieving the value involves submitting code as we did in the following SAS session:

```
165  Data _NULL_;
166    Length DUMMY  USER $20.;
167    DUMMY = " ";
168    USER  = " ";
169    RC = ModuleN( "WNetGetUserA", DUMMY,
USER, 255);
170    If RC >= 1 Then Put @5
171     "Network & username not available.";
172    Else If RC < 0 Then Put @5
173     "Network not available or not
responding.";
174    Else Put 'Lan User name found : '
USER;
175  Run;

Lan User name found : Administrator
NOTE: The DATA statement used 0.34 seconds.
```

The flaw with this approach is that a network connection is required, and it must be either Windows networking or certain other Network types. It has not worked on some networks tested. As a backup to this, we can retrieve the name of the person who logged on to Windows. This is done with the GetUserNameA routine. The prototype for this routine follows.

```
routine GetUserNameA
  minarg   = 2
  maxarg   = 2
  stackpop = called
  module   = advapi32
  returns  = long;
arg 1 char  update  format = $cstr20.;
arg 2 num   update  format = pib4.;
```

By combining the calls to the two APIs, we can test for the second value if the first does not exist. The commented statement at line 255 below will prevent resetting the value if the WNetGetUserA call is successful.

```
244  Data _NULL_;
245    Length DUMMY  USER $20.;
246    DUMMY = " ";
247    USER  = " ";
248    RC = ModuleN( "WNetGetUserA", DUMMY,
USER, 255);
249    If RC >= 1 Then Put @5
250     "Network & username not available.";
251    Else If RC < 0 Then Put @5
252       "Network not available or not
responding.";
253    Else Put @5 'Lan User name found : '
USER;
254    Put 'SUGI debug WNetGetUserA ' RC=;
255    /* If RC = 1  Then Stop; */
256    RC = ModuleN("GetUserNameA",USER,255);
257    If RC = 1 Then Put @5
258     "Windows logon user " user "found.";
259    Else Do;
260      If RC > 1 Then Put @5
261         "Error in retrieving Windows
logon.";
262      Else Put @5
263         "Windows logon responding, no
username found.";
264    End;
265    Put 'SUGI debug GetUserNameA ' RC=;
266  Run;
```

```
     Lan User name found : Administrator
SUGI debug WNetGetUserA RC=0
     Windows logon user Administrator found.
SUGI debug GetUserNameA RC=1
NOTE: The DATA statement used 0.58 seconds.
```

The log entries marked 'SUGI Debug' were included to highlight one of the potential pitfalls in using APIs. The return code from each successful call may be different. This means developing a Windows API call from scratch will require some careful research into the API details.

I know of, and use three electronic resources to assist with the definition of the prototype in the attribute file, and the returns expected from each call. They are:

- the Windows Software Development Kit,
- the support area of the Microsoft web site and
- the WIN32.HLP help file packaged with Borland Delphi.

This last file was made available from Microsoft, and as such may not reflect recent changes to your operating system. However, for anyone else who has the product licensed, it is a very good resource. Here in the UK, Borland have generously made a number of releases of Delphi available on magazine cover disks, so the help file is fairly commonly available. I should also add that there are quite a few good books available dealing with the subject of Windows APIs. If you'd like some recommendations, feel free to write to the author.

The only significant question with the API approach is whether the user is logged on to the machine. In the case of Windows 3.1 and Windows 95, it is possible to start up the Operating System without logging on. In this case, the Windows macro needs to take account of the version of Windows. This can be retrieved with a Windows API call, and is also available through the automatic global macro tokens defined within the SAS session. The following is part of the entry put to a SAS log when the command '*%Put _all_;*' is submitted to a SAS session:

```
AUTOMATIC SYSSCPL WIN_NT
```

By reading the value of the ***SysSCpl*** token, we can include branches in our ***GetUser*** macro that accommodate different Windows platforms.

EMAILING RESULTS TO WHOM?

One of the more common requirements that emerged with production applications was the need to deliver information more quickly. The author has seen the increasing use of email as an information delivery tool in production applications. Aside from the clear benefits of delivering business information more quickly to the information users, there was also a strong benefit in delivering progress and failure reports quickly.

The ***GetUser*** macros were easily extended to provide an email address that could be used to report process continuation, completion or failure. Since each program on a given platform used a common macro, a single change to a macro could surface an email address to the program. Then those programs requiring email functionality could be extended to generate the appropriate messages.

ARE ALL WINDOWS APIS SO EASY TO SET UP?

The simple answer is 'No'. All the authors of API related information below warn against casual API experimentation. An error in an API can crash a SAS session or a Windows session resulting in loss of data. One of the more involved macros undertaken by the author is the FileFind macro. It comprises one master macro, and two subsidiary ones. It also uses six API prototypes, which follow. (I have edited them for space reasons. The full versions are available on the author's web site.)

```
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.;


routine FindNextFileA
        minarg=12
        maxarg=12
        stackpop=called
        module=Kernel32
        returns=long;
* LPCTSTR  hFindFile,  // search handle;
arg  1 num  input  byvalue  format=pib4;

* LPWIN32_FIND_DATA  lpFindFileData  //
address of returned information ;
*   DWORD dwFileAttributes ;
   < arg 2 - 12 of FindFirstFileA routine >

routine FindClose
  minarg=1
  maxarg=1
  stackpop=called
  module=kernel32
  returns=short;
arg 1 num input byvalue format=pib4.;


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


routine FileTimeToLocalFileTime
  minarg=2
  maxarg=2
  stackpop=called
  module=kernel32
  returns=ulong;
* lpFileTime,  // UTC file time to convert;
arg 1 input  fdstart num format=pib8.;
* lpLocalFileTime // converted file time;
```

```
arg 2 output fdstart num format=pib8.;


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 ;
*   WORD wYear ;
arg  2 num  output fdstart 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 ;
arg  9 num  output         format=pib2.;
```

The above attributes are referenced as needed in the following macro. This finds the first file matching the search criteria, and then uses that successful search to find each other file that also matches.

```
%Macro XFileFnd( MDir = ) /
   Des="SYSTEM: List MDIR to ZZZZFILE";

  Data ZZZZFILE( Keep = FILENAME  EXTENSN
                 CREATESD  ACCESSSD  WRITESD
                 FILEATT  FILESIZE);
    Length FILENAME NAME PATH $200
        EXTENSN $32  FILEATT $8  BITNAME $60
        ANAME $14  CREATESD ACCESSSD WRITESD
        FILESIZE 8;
    NAME = " ";  BITNAME = ' ';  ANAME = " ";
    ATT  = .;    CRE     = .;    ACC   = .;
    WRI  = .;    SLOW    = .;    SHIGH = .;
    RC   = .;    NUM1    = .;    NUM2  = .;
    LNUM1= .;    RCODE   = 0;
    PATH = "&MDir";
    RC   = ModuleN( 'SetLastError',0);
    HANDLE = ModuleN( 'FindFirstFileA',
         PATH, ATT, CRE, ACC, WRI, SHIGH,
         SLOW, 0, 0, NAME, BITNAME, ANAME);
    RC     = ModuleN( 'GetLastError');
    If RC Ne 2  Then Put 'WARNING: There is
         a problem with your API call.';
    If HANDLE >= 1  Then Do;
      %XGetAttr;
      Output;
      FOUND = 1;
      Do While( FOUND);
        FILEATT = ' ';
        FOUND = ModuleN( 'FindNextFileA',
           HANDLE, ATT, CRE, ACC, WRI, SHIGH,
           SLOW, 0, 0,NAME, BITNAME, ANAME);
        If FOUND   Then   Do;
          %XGetAttr;
          Output;
        End;
      End;
      RC = ModuleN( 'FindClose', HANDLE);
    End;
  Run;

%Mend XFileFnd;
```

For each located file, a call is made to the ***XGetAttr*** macro, which reads, interprets and validates the attributes of the file. The following is the ***XGetAttr*** macro.

```
%Macro XGetAttr /
    Des = 'Get file attributes';

  ** Reverse the string in case multiple
periods appear in the file name;
  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;

  %XGetTime( MFTime=CRE, MDSTime=CREATESD);
  %XGetTime( MFTime=ACC, MDSTime=ACCESSSD);
  %XGetTime( MFTime=WRI, MDSTime=WRITESD);

  FILESIZE = SHIGH * ( 2**32) + SLOW;
  If Input( Substr( Put( ATT, Binary8.),
            1, 1), 1.) Then ATT = 0;
  If Input( Substr( Put( ATT, Binary8.),
            4, 1), 1.) Then FILEATT = 'F';
  If Input( Substr( Put( ATT, Binary8.),
            8, 1), 1.) Then FILEATT =
        Compress( FILEATT || "R");
  If Input( Substr( Put( ATT, Binary8.),
            3, 1), 1.) Then FILEATT =
        Compress( FILEATT || "A");
  If Input( Substr( Put( ATT, Binary8.),
            6, 1), 1.) Then FILEATT =
        Compress( FILEATT || "S");
  If Input( Substr( Put( ATT, Binary8.),
            7, 1), 1.)  Then FILEATT =
        Compress( FILEATT || "H");

%Mend XGetAttr;
```

The ***XGetAttr*** macro has up to three date-time values to interpret. These three values represent the Creation date, the Modification date and the Last accessed date for the file. (Not all are available in all versions of Windows.) The three pointers returned by the file find processes are passed in turn into the ***XGetTime*** macro and valid dates resolved.

```
%Macro XGetTime( MFTime=, MDSTime=);

  &MDSTime = .;  YEAR = .;    MONTH = .;
  DAYOFWK = .;   DAY = .;     HOUR = .;
  MINUTE = .;    SECONDS = .; MSECONDS = .;
  If HANDLE >= 1  Then RC4 =
      ModuleN( 'FileTimeToLocalFileTime',
              &MFTime, LNUM1);
  If RC4  Then RC5 =
      ModuleN( 'FileTimeToSystemTime',
```

```
          LNUM1, YEAR, MONTH, DAYOFWK, DAY,
          HOUR, MINUTE, SECONDS, MSECONDS);
   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) );
   Format &MDSTime DateTime21.2;

 %Mend XGetTime;
```

The date time components are all independently validated. The author has seen the date time stamps of a large group of files corrupted by a malfunctioning application.

For those interested, there is a comparable macro for MVS. It uses a utility for retrieving information from the Data set Management System (DMS). The utility is usually found in the 'System Proc' libraries on IBM mainframes.

The macro to interface with this utility will be published on the author's web site shortly.

CONCLUSION

When we review the development process we undertook for our 'cross platform macros', the following guidelines emerge:

- Devise any new macro as generically as possible, it may become necessary on another platform
- Research the macro call carefully. If you are parsing the return from a system call, then look at what else may be available and either add that to the macro return, or document it in the macro for future use.
- Set up a standard naming convention for system and generic macros and their parameters, and never mix these with the names used for application macros.
- Set aside a block of data set names, library and filename references. Use these always and only with your cross platform macros.
- Document the macros thoroughly and carefully. In a year from now, you may be the person trying to understand the documentation to fix a problem, or extend functionality.

TRADEMARK CITATION

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 registered trademarks or trademarks of their respective companies.

REFERENCES

SAS Institute Inc (1995), *TS460: Accessing External DLLs with SAS 6.1x for Win32s*, Cary, NC, SAS Institute Inc.

Barron, David L & Hemedinger, Chris (1996), "Accessing Dynamic Link Library Routines with the SAS System for Windows", *Observations: The Technical Journal for SAS Software Users*, First Quarter 1996.

SAS Institute Inc (1996), *Microsoft Windows Environment: Changes and Enhancements to the SAS System, Release 6.11*, Cary, NC, SAS Institute Inc.

Johnson, David H (1997), "DLLs, APIs and SAS", SAS Melbourne User Group.

Johnson, David H (2000), "Have SAS, will travel", SAS European User Group International, Dublin, SAS Institute Inc.

ACKNOWLEDGEMENTS

My clients who have asked for the robustness and functionality that required this development to take place.

The SUGI committee and especially the Systems Architecture section chair, Deb Cassidy, for supporting the production of this paper.

The SAS Institute Inc for their documentation, and the 'Observations' publications which provide a forum for sharing ideas and experiences.

The helpful people who post ideas and assistance to SAS/L, the global SAS discussion list you can access at http://www.listserv.uga.edu.

Last but not least, my family for their support and patience.

CONTACT INFORMATION

Your comments, suggestions and questions are valued and encouraged. Please contact the author:

David Johnson
DKV-J Consultancies
C/- 'Bonds Cottage,
Holmeswood Rd
Holmeswood nr Rufford
Lancashire England L40 1UA
Business Phone: +44 (0)7080 81 8399
Fax: +44 (0)7092 25 9556
Email: sugi280@dkvj-cons.com
Web: http://www.dkvj-cons.com