

%SYSFUNC: Extending the SAS Macro Language

Paul Hamilton
ICOS Corporation

Abstract

This class will introduce the power that SYSFUNC brings to developing SAS macros. Anyone who struggles with developing SAS macros in order to extend the power and ease of use of the SAS system should be aware of the myriad Data step and SCL functions that the SYSFUNC facility makes available.

In the original design of the SAS system, the Data and Proc steps performed useful work, and the macro language provided a way to construct and manipulate various SAS statements. However, the macro language itself was developed with a very limited tool set. With the advent of SYSFUNC, most of the power of the Data step function library, and the more extensive set of functions available in SCL have become available. This allows the macro language itself to express real power in providing solutions to everyday problems faced by virtually all users of the SAS system. Much of the functionality that was previously available only to SAS/AF developers may now be tapped and applied in creating powerful, robust macros.

The Development Process

Some confusion exists in the computer lexicon about the word *developer*. Since the introduction of SAS/AF[®] as a complete solution for creating user-friendly applications, *developer* in the SAS world has come to mean someone who codes using SAS/AF software, writing applications which place the considerable power of the SAS system into the hands of people who do not write SAS (or perhaps any other) code.

This implies that people who work with SAS on a daily basis, who write many (potentially large and complex) production level macros, somehow are not SAS developers, even if the macros become a core asset to a groups' SAS functionality. This viewpoint seems to be rather limited and limiting, and here is an alternate definition:

A SAS developer is anyone who develops SAS code which is re-used by "other people" over time. This may involve interactive screens (as in SAS/AF) or purely through batch code usage (usually in the form of an autocall library). "Other people" may be one or more programmers other than the author of the macro, or just the original author.

This broader definition means that anyone who spends enough time to learn and write permanent macros is indeed a developer. As such, it becomes worthwhile for the developer to spend an appropriate amount of time researching and mastering the tool set available. For most folk, this means learning the SAS macro language

History

The SAS system is a broad and deep suite of software, capable of solving almost any type of problem set one might encounter. Originally consisting of just the Data step and Proc steps, SAS software has grown explosively over the last two and a half decades. The macro language version is a very old part of the system; probably many SAS users believe it to be part of the original implementation. The macro facility (originally available under release 82.3) provides the ability to produce, manipulate, and conditionally execute SAS code. However, compared to the wide variety of functions available in the Data step, the scope of functions incorporated into the macro language is quite sparse.

Introduced during the 6.12¹ release, SAS Institute introduced a new macro function, %SYSFUNC, which allows the use of almost all datastep, many SCL (originally Screen Control Language, now renamed as SAS Component Language) and user-written² functions in the macro environment. It also allows formatting of macro variable values using native SAS formats, or user-specified formats. This course discusses the general syntax of %SYSFUNC as well as some examples of its use.

It should also be noted that many of the SCL functions available to %SYSFUNC are now also available in the data step. Syntax and usage of some of these functions will also be examined.

History of This Paper

Pete Lund and the present author presented papers on this topic to the Pacific Northwest SAS Users Group conference, starting in 1997. Pete's paper was entitled *Fun*

¹ The %SYSFUNC function is available as an unsupported feature in SAS 6.11 TS040.

² Discussion of user-written functions are outside the scope of this course. However, they are available to %SYSFUNC.

with %SYSFUNC(), and the present author's companion piece was *More Fun With %SYSFUNC()*. Both of these papers have been folded into the current course, along with enhancements and additions.

So, What's a Function?

It might be easier to consider what %SYSFUNC is and how it works if we review functions in the SAS System.

A function is essentially a micro-program that another person (in this case, the development staff at the SAS Institute) writes for you, that provides functionality or services. In some respects, the maturity and value of any programming environment may be measured by the richness and depth of the provided function library.

Functions appear in two main places for most people's usage of SAS: in the Data step, and in the macro language. These two sets of functions perform in much the same way, simply in different contexts. In either case, a function is a part of a SAS statement that provides a service to the SAS programmer. This could be very simple (add two numbers):

```
X = sum(1, 2);
```

to fairly complex (calculate internal rate of return)

```
answer = intrr(freq,c0, c1, . . . ,cn)
```

to fairly obscure (calculate the bitwise logical right shift of 2 arguments):

```
x = brshift(@1Cx, 2);
```

Some functions provide nearly identical services in both the Data step and macro environments:

```
/* Data step */
name = upcase(name);
/* Macro code */
%let name = %upcase(&name);
```

How Many Are There?

There are 344 functions in the Data step, 30 (!) in the macro language, and 345 in SCL. When using %SYSFUNC, almost all of the Data step functions are available (exceptions are noted below), and there are 61 SCL functions available specifically for %SYSFUNC.

Functions Available to %SYSFUNC

Almost all of the base SAS dataset functions, as well as a number of SCL functions, are available to %SYSFUNC. There is a complete list of the new SAS 6.12 functions in the Online Help.³

³Searching Version 8.2 Online Help for *SYSFUNC* (without the percent character) leads to a general introduction. Searching for SYSFUNC in the HTML On

A small handful of functions are not available to %SYSFUNC. Considering the lack of a Program Data Vector outside of the Data step, and no requirement for the macro processor to interface with itself, not having these functions available is not problematic. The Data step functions not available using %SYSFUNC include:

Dif	Dim	Hbound	Input	Lag
Lbound	Put	Resolve		Symget

SAS software has, since its inception, been designed to try to hide the complexity of computer architecture from the casual user (e.g. only 2 data types, automatic type conversion, type specification not required, etc.), and to provide a rich variety of built-in tools to perform common tasks (e.g. the myriad Proc steps). This design is evident in the Data step, and SAS/AF, but not the macro language, which as we saw above, is much more spare in its offerings. Fortunately, %SYSFUNC greatly enhances the scope and breadth of the Macro language.

The Basics—Syntax

The general syntax of %SYSFUNC is:

```
%SYSFUNC (function(argument(s))<,format>)
```

where *function* is the Data step or SCL function to execute, followed by its argument list. A SAS System or user-defined format can be applied to change the appearance of the value.

How Does It Look?

One primary difference between calling a function in %SYSFUNC and elsewhere in the SAS System is this last notion of applying a format to the results of a function call. In the Data step, or in SCL code, one must perform an extra step to change the appearance of the output from a function. In the simple Data step below, notice that variable y prints using the default SAS format.

```
data test;
  x = 2;
  y = sum(x, x, x);
  put y= +1 y z4.;
run;

y=6   0006
```

If the (Z.) format was desired for the use of the sum function, then nesting the original function call with a put function would be required:

```
data test;
  x = 2;
  y = put(sum(x, x, x), z4.);
  put y=;
run;
```

Line Doc finds 22 matches, which are useful. In the PDF version, look for a03.pdf in the macro folder.

```
y=0006
```

%SYSFUNC is atypical, because a format may be applied to the output from the called function directly (several examples appear below.)

Why Does %SYSFUNC Exist?

So, the primary purpose of %SYSFUNC is make additional functions and formats available to the macro processor. In the following simple example, the macro variable TheDate would be assigned the current system date (in SAS date format):

```
%let TheDate = %sysfunc(Date());
```

You can also add an appropriate format following the function call to change the value to the formatted value. If the above example had set TheDate = 15225, the following:

```
%let TheDate=%sysfunc(Date(),worddate18.);
```

would yield September 7, 2001. Notice that this code can be submitted outside of a Data step, and outside of a macro. This leads to a discussion of the various frameworks from which one can make %SYSFUNC calls.

Where %SYSFUNC May be Called

For the SAS developer, there are several hierarchical layers of the system:

1. The “base layer”: Data and Proc steps,
2. The simple “meta” layer, or open Macro code
3. The complex “meta” layer, or Macro routines and functions
4. The “application” layer, or a SAS/AF application.

Suppose that a Data step is required which will subset the data from an existing SAS data set in the WORK library. The variable used to subset the data is StartDt, the start date of some point of interest. Assume that this code will be part of a long-lasting production job, but that the reference data will be changed each time the job is submitted.

Using a simple Data step, the task is quite simple:

```
/* Layers of SAS Code-Example 1 */
data recent;
  set sasdata;
  where (startdt ge '07sep2001'd);
run;
```

The code that needs to be modified from run to run appears in red, and underlined. It is of course perfectly simple to put whatever date is desired into this location. However, this small Data step is shown alone, with no context. Perhaps it occurs on page 27 of a 300 page SAS program! Replacing the value is not problematic—simply

finding the correct place to make the change could be. Also, consider the possibility that the same change may be required in multiple places.

The second way to write this code is to use the most simple type of macro coding, called macro substitution. Using this technique, the parts of the program that need to be changed from run to run and sequestered together at the top of the program, where comment blocks can make clear what changes need to be made from time to time.

```
/* Layers of SAS Code-Example 2 */
/* Replace <theDate> below with desired value */
```

```
%let theDate = 07Sep2001;
```

```
data recent;
  set sasdata;
  where (startdt ge "&theDate"d);
run;
```

Using this technique, every place in the program that needs to reference the desired cutoff date will have the same value available to it, with only one (easily found) place to change the code.

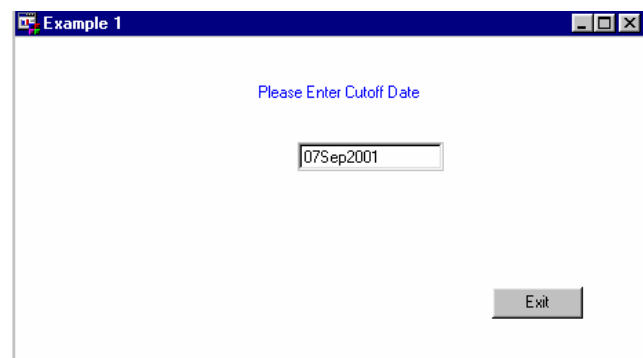
The third technique is to create a full-fledged macro routine, where the critical date becomes a calling parameter. Using this very simple example, there is no compelling reason to do so, but it is the next logical step.

```
/* Layers of SAS Code-Example 3 */
%macro example1(theDate);
```

```
data recent;
  set sasdata;
  where (startdt ge "&theDate"d);
run;
%mend example1;
```

```
%example1(07Sep2001)
```

The last step is to wrap the code in a full-screen SAS/AF application. Again, there is no particular reason to do so given this small code segment, but it does complete the illustration of the different layers of application development.



As various examples are provided throughout the course, they will be identified as being appropriate for Layers 1, 2, or 3 (layer 4 is not considered.)

Changing the Look of a Macro Variable

Another handy feature of %SYSFUNC is the ability to apply a format (native SAS format, or user defined) to the output from a function call. You may have noticed that the Put and Input functions were marked as not being available in %SYSFUNC. Actually, this is a bit of misdirection, since their close cousins from SCL are available: PutC, PutN, InputC, and InputN.

For example, assume the macro variable &Salary has a value of 12343435. If you wished to display it using the built-in dollar format, the following (layer 2) code would serve:

```
%put %sysfunc(PutN(&Salary, dollar12.));
```

would produce the following in the log:

```
$12, 343, 435
```

This illustrates one of the important features to understand about naming conventions of SCL functions: they are organized in families, using a common base text string (e.g. Put) followed by character(s) that indicate variations on the family theme. In this case, the original Data step Put functions evolved into PutN and PutC, one for numeric and one for character variables. When we examine the large scope of SCL functions available, realizing this will help keep the extensive number of functions more manageable.⁴

Does Anyone Know What Time It Is...

After one has used the SAS system for a while, it becomes apparent that not all things are what they first seem to be. A good example is the date and time that can be placed in the title section of output. Two of the more commonly used automatic macro variables are &SysDate and &SysTime. These are often used in titles and footnotes to note run date and run time. However, the date and time

⁴ As a side note, the author has adopted an unusual style of using capitalization after years of writing SCL code. Most SAS programmers probably leave function names either all upper or all lower case. Because SCL functions often have this characteristic of (possibly multiple) suffix endings that represent (important) whole words, this course follows the author's personal convention of capitalizing certain letters of function calls, as seen above. Feel free to adopt, criticize, or ignore this convention when writing your own code.

represent the time the SAS session began, and not the current date and time when the Proc (or Data) step produced output.

%SYSFUNC has solved this problem. Using the Time() function, which returns the current system time (assuming your clock is set right), one can now replace the following (layer 1) footnote:

```
footnote "Start: &SysTime";
```

```
Start: 10:04:41 PM
```

with

```
footnote "Start: %sysfunc(time(), timeampm.)";
```

```
Start: 11:13:27 PM
```

Now, not only does the output reflect the actual time that the procedure was run, but it can be formatted any way desired.

Dates are available as well as times:

```
footnote "Start: %sysfunc(today(), date9.);
```

```
Start: 30AUG2001
```

In fact, they can be combined into a date-time stamp:

```
footnote "Start: %sysfunc(datetime(), datetime.);
```

```
Start: 30AUG01:22:06:17
```

Go Ahead and Use Your Own

You are not limited to using native SAS formats in %SYSFUNC. User-defined formats, created in PROC FORMAT, are supported as well. Suppose I created the following categories to group income:

```
proc format;
  value income
    0-500000 = 'a minor leaguer'
    500001-2500000 = 'a utility infielder'
    2500001-10000000 = 'maybe a starter'
    10000001-high = 'probably Ichiro';
run;
```

Assuming the macro variable &Salary has a value of 343435, the (layer 2) statements:

```
%put With a salary of
      %sysfunc(PutN(&salary, dollar12.));
%put You are
      %sysfunc(PutN(&salary, income.));
```

would produce the following log:

```
With a salary of      $343,435
You are a minor leaguer
```

Or, using the value from our earlier example (12343435):

```
With a salary of $12,343,435
You are probably Ichiro
```

Nesting %SYSFUNC Calls

In base SAS programming there are often times when it is convenient to nest function calls. Unfortunately, functions cannot be nested in %SYSFUNC, but %SYSFUNC calls themselves can be nested. For example, two functions which are commonly nested are Trim and Left. This forces a character value to be left justified and removes any trailing spaces. The following (layer 2) statement would produce an error:

```
%let trimmed = %sysfunc(trim(left(&myvar)));
```

However, nesting each function in a separate %SYSFUNC call is valid:

```
%let trimmed = %sysfunc(trim
(%sysfunc(left(&myvar))));
```

There are numerous situations where nested functions can streamline your code. The following example (layer 1) solves a common problem in open macro code.

Suppose a macro variable exists, &ThisMnth, in YYMM format (i.e., 9711). Further suppose that you need to create another variable, &LastMnth, which has a value of one month earlier (i.e., 9710). Obviously, you can't simply subtract 1, using the %Eval function, because when I get to January (i.e., 9801) my last month value will be slightly invalid (i.e., 9800). Now that nearly all the Data step functions are available, I can simply (using layer 2 techniques):

1. convert the macro variable to a SAS date, with the InputN function
2. decrement it one month with the Data step intnx function,
3. and return the new value in YYMM format.

Let's look at each of those pieces separately. First, I want to convert &ThisMnth to a SAS date. By adding a day value (the constant 01) to the end of &ThisMnth, I can use the YYMMDD6. informat to convert 971101 to a SAS date:

```
%let sd=%sysfunc(InputN(&Mnth.01, yymmdd6.))
```

&sd now equals 13819, the SAS date value for November 1, 1997. Now we can use the intnx function to back this date up one month. Also, remember that a format may be added to the %SYSFUNC call, so in one call the new value can be formatted in YYMM format.⁵

⁵ Notice that the time interval argument (month) to INTNX is not quoted as it would be if used in a data step. I have

```
%let LastMnth = %sysfunc
(intnx(month, &sd, -1), yymmdd4.);
```

&LM now has a value of 9710. The two %SYSFUNC calls can be nested to create the new variable with one statement:

```
%let LastMnth = %sysfunc
(intnx(month, %sysfunc
(InputN(&ThisMnth.01, yymmdd6.)),
-1), yymmdd4.);
```

Types of Work

All the examples so far have shown calls to %SYSFUNC that accomplish a certain task, but without any particular feedback from the function. This is typical of function in the Data step: if a function is called with an inappropriate argument (e.g. sqrt(-3)), the function itself provides no feedback. Instead, the feedback comes from the Program Data Vector (PDV), using the automatic variable _ERROR_, along with an informative message (NOTE:) in the SAS Log. However, this is the exception to the rule in the land of SCL (and therefore much of the macro language) where no PDV exists as a "container" for the processing.

When coding SCL to support SAS/AF applications, it turns out that there are several types of work that must be done:

1. The actual SAS code to accomplish a task (e.g. Data steps, Proc Freq, Proc Sort, etc.).
2. Possible macro code invocations,
3. SCL and visible screen widgets to control the user interface, and
4. Code that self-corrects from user input errors, and provides feedback to the interactive user.

You might notice the parallel to the four layers of software introduced earlier. This is not a coincidence! The heartbreaking fact is that a developer does more work than a typical SAS programmer. You have to do the work of getting information about what the work is, check the work information, stop or correct if necessary, perform the actual work, then provide feedback on how the work went.

This is a way of thinking that should be adopted by anyone who codes a production level macro. Often, we feel lucky to find a macro that has simply been well documented, so that the calling sequence and usage of all input parameters are clearly defined. We are luckier yet if the macro has internal documentation, describing what is going on during the execution of the code. However, at the highest level, a

found a few places where arguments are not passed exactly in %SYSFUNC as in data step code. As far as I know, they are not documented and it is a trial and error process to work around them.

macro (just like a SAS/AF application) can carefully check the conditions of the input parameters provided, verify that all necessary components exist, carry out the original work intended by the design, and provide full feedback for unexpected results that occur. This takes some of the “burden of correctness” from the end user, and places it squarely on the developer. How this can occur is explained after a short example.

You are Feeling Sleepy...

There is one SAS data step function that is almost always coded by itself in a `_NULL_` Data step: the `sleep` function.

```
data _null_;
  x = sleep(30);
run;
```

Code like the above, which would cause the SAS session to pause (sleep) for 30 seconds, is often used to delay execution while something external to your SAS program is initiated, such as starting Excel before establishing a DDE connection. There is nothing wrong with this code, it just adds a little clutter to the log. It can now be replaced with the (layer 2) statement which performs the same function:

```
%let rc = %sysfunc(sleep(30));
```

The function will return the argument value, 30 in this case, so you need to use a macro assignment statement to contain the value. Actually, try the following (level 1) code for fun. What do you think of a `Title` statement that executes code?

```
title "We sat still for %sysfunc(sleep(5))
seconds";
```

What Returned?

One of the major differences in writing `%SYSFUNC` code vs. standard macro code is the need to pay close attention to the aftermath of a function call. In the Data step, we have the SAS Supervisor looking out for us, and providing (hopefully) helpful message in the SAS Log, and setting the `_ERROR_` variable in the Program Data Vector to 1 to provide some feedback. However, it is all too easy to ignore such messages, especially when they are expressed as Notes or Warnings. In standard macro language, things either work, or they do not, and debugging macros has never been described as a good time.

The good news about calling SCL and Data step functions in `%SYSFUNC` is that the function will typically provide extensive feedback on how things went during the execution. The bad news is that you, the developer, need to attend to that feedback. The worse news is that the feedback is provided in the form of *return codes*, which vary tremendously from function to function. One SCL

function might return a zero if all went well, whereas in a other function a zero return code indicates that something very bad indeed has occurred. One needs to refer to the documentation frequently to keep this all sorted out.

Note that there are certain naming conventions that tend to be adopted by SCL programmers. They are merely conventions and not sacred, but some are used in this course. For instance, `RC` is often used to hold the value of a Return Code, while `DSID` is often used to point to files.

Fooling Around with Files

It is very often necessary for your program to detect whether an external file, dataset or catalog exists. SAS provides many tools to obtain this information: `Proc Datasets`, `Proc Contents`, `Dictionary` tables and `SASHELP` views. All provide useful information but, like the `Sleep` function in the `_NULL_` Data step, require executing a Data or Proc step.

There are a number of SCL functions available to `%SYSFUNC` which can provide file information without having to run a step. Two functions exist that can be used to check for the existence of an external file: `FileExist` and `FExist`. (Warning: Return Codes ahead!)

The `FileExist` function accepts as its argument the name of an external file and returns a 1 if the file exists, and 0 if not.

```
%let exist =
%sysfunc(FileExist(c:\delme.dat));
```

Notice the construction of the function call above. This is very typical of SCL function calls, and is something you should become comfortable with as a `%SYSFUNC` developer as well. An SCL function is called in the context of a `%Let` statement, which in ordinary macro language usage we associate only with a value assignment, not performing some action. Perhaps it will be helpful to think of the verb (`FileExist` in this case) doing something, and then returning a value which is captured by the `%Let` portion of the statement.

If you find this construction somewhat confusing (say, backward), it might help to remember that the Data step is arranged in somewhat the same way. The `Data` statement typically builds data set(s) which are then filled with data depending on the statements that follow (e.g. `Infile/Input`, `Set`, `Merge`, etc.).

The `FExist` function accepts as its argument a previously assigned `FileRef`, and also returns 1 if the file exists and 0 if not.

```
filename myfile 'c:\delme.dat';
%let exist = %sysfunc(FExist(myfile));
```

The following macro uses the `FileExist` function to check for the existence of a directory. If the directory does not exist, one is created. A `FileRef` is then established for the directory.

```
options noxwait xmin;
%macro ChkDir(dname, fref);
  %if not %sysfunc(FileExist("&dname"))
    %then %let rc = %sysfunc(
      system(md "&dname"));
  %let rc=%sysfunc(filename(&fref, &dname));
%mend;
%ChkDir(c:\JuneData, Prod);
```

In the above example, the macro checks to see if the directory, *c:\Production*, exists and, if not, creates it. It then assigns the `FileRef`, *Prod*, to that directory using the `FileName` SCL function. This functionality of taking correction action is typical of well designed %SYSFUNC macros.

The `FileName` function works much like a typical SAS filename statement. It takes a file reference and a file or directory name as its arguments. When called via %SYSFUNC, the file reference argument must be a macro variable reference without an ampersand.

You can also delete an external file with the `FDelete` function. It takes as its argument a `FileRef`, and returns a 0 if successful and non-0 if not.

```
filename x 'c:\delme.dat';
%let rc = %sysfunc(FDelete(x));
```

In much the same manner as checking for the existence of an external file, you can check for the existence of a SAS dataset with the `Exist` function. It also returns 1 if the dataset exists and 0 if not. One or two level dataset names are allowed.

```
%let exist = %sysfunc(exist(my_ds));
%let exist = %sysfunc(exist(my_lib.my_ds));
```

There is one other function for checking the existence of a "file", in this case a file structure internal to the SAS System.. The `CExist` function (Catalog Exist) checks for the existence of a catalog or catalog entry. It accepts as its arguments the name of the catalog or entry and an optional flag (U) to test whether the catalog is available for updating.

```
%let rc=%sysfunc
  (CExist(sashelp.flags.canada.image))
and
%let rc=%sysfunc
  (CExist(sashelp.flags.canada.image,
    U));
```

both return 1. Note that the update flag, U, checks if the catalog can be opened for update. In the above case, the image entry cannot be opened for update, but the catalog itself can. Also note that a one- or two-level is assumed to

be a catalog name and a three- or four-level name is assumed to be a catalog entry name.

More External Files

Aside from the examples above, %SYSFUNC also supports a family of functions that exist to read directory level information (from non-SAS files). Some functions define attributes for other functions to use, while others perform more obviously useful work. The results of these external file functions are highly platform dependent. We will examine three different types of information provided by SCL functions through %SYSFUNC.

Directory Information

The first example of these functions obtains general information about a specified directory. First, the developer needs to identify the directory location with the `FileName` function (note that the dot indicates the current directory, or *here*). Of course, this could also be accomplished with the more standard `FileName` command in base SAS, but this is a course in %SYSFUNC, so...

```
%let rc = %sysfunc(filename(theDir, .));
```

Next, the macro needs to open the directory. We will see other, related ways to open other types of structures in later examples. For this example, the macro will open a directory, so the SCL function name is `DOpen` (Directory Open). This is accomplished in the context of a %let statement, and something critical happens that is different from other functions: the `DOpen` function returns a number that is used as a file handle, in this case named `&did` (Directory ID). This local macro variable will be used in the next several throughout the macro as a way of referencing the structure.

```
%macro d1;
%let did = %sysfunc(DOpen(&theDir));
```

Next, the program needs to obtain the number of options that are available for directory information. This will vary across operating systems, and perhaps versions of SAS.

```
%let OptNum = %sysfunc(DOptNum(&did));
```

`&OptNum` now contains the number of options that are available for retrieval. Once that is known, a `DO` loop can retrieve all the values of Directory Option Names (`DOptName`) and option values (`DInfo`: Directory Information). The values retrieved are in this case placed into single-value macro variables and immediately printed. Should it be necessary to save them for later processing, the program would need to make them global macro variables, and use the `&&var&counter` technique for naming and storing them. If this is too complicated, they could be stored in a standard SAS data set instead.

```
%do i = 1 %to &OptNum;
  %let OptName = %sysfunc
    (DOptName(&did, &i));
  %let OptValue = %sysfunc
    (DInfo(&did, &OptName));
  %put Option # &i (&OptName) is &OptValue;
%end;
```

As always, after processing with %SYSFUNC, we need to clean up after execution:

```
%let rc = %sysfunc(DClose(&did));
%mend d1;
```

The results from running this macro vary considerably by operating system, as one would expect:

Windows

```
Option # 1 (Directory) is C:\
```

Sun OS

```
Option # 1 (Directory Name) is /home/hamilton
Option # 2 (Owner Name) is hamilton
Option # 3 (Group Name) is staff
Option # 4 (Access permission) is rwx-----
```

Mac OS

```
Option # 1 (Directory Name) is MPFC:WUSS
```

File Information

The information retrieved and displayed in the example above worked with the directory as a whole. There are a corresponding set of functions that work with file level information, contained in the directory.

In this example, we will create a macro to read all of the actual files in a directory (using Windows as a platform). First, the macro needs to know the location of the directory (root of the C:\ drive):

```
%macro d2;
  %let rc= %sysfunc(filename(thedir, C:\));
```

Next, we need to open the directory (DOpen) and extract a file handle (&DId). The file handle will be used in several statements below.

```
%let did = %sysfunc(DOpen(&thedir));
```

Now that the directory is identified, extract the number of members that exist in the directory (DNum). This is returned to a local macro variable &DNum. The name of the return value need not match the name of the function call, but it is somewhat natural (as in Set ... NOBS = NOBS; in the Data step.)

```
%let dnum = %sysfunc(DNum(&did));
```

Now the macro has all the information it needs to begin the real work. First, execute a %DO loop to process all of

the files. The SCL function to read the file information from the directory is DRead (Directory Read: please don't dread it!) A simple %Put statement writes out the information in this example. After processing has completed, close the directory with the DClose (Directory Close) function.

```
%do j = 1 %to &DNum;
  %let fname = %sysfunc(dread(&did, &j));
  %put File # &j is &fname;
%end;
%let rc = %sysfunc(dclose(&did));
%mend d2;

%d2;
```

The results are not terribly interesting, but they are what we were after:

```
File # 1 is I386
File # 4 is WINNT
File # 5 is RMSP.BAT
<major snip occurs>
File # 59 is readme.001
File # 60 is Leame.001
File # 61 is Leggimi.001
File # 62 is Lisezmoi.001
File # 63 is Info.001
File # 64 is Leiname.001
```

File Information, Yet Again

Our third example dealing with external files will extract information about individual files. First, assign a FileRef with the FileName function. In this example, a FileRef of theFile is assigned: this is completely arbitrary designation.

```
%macro d3;
  %let rc = %sysfunc(filename(theFile,
    C:\UG\WUSS2001\SysFunc Slides\sysfunc.ppt));
```

Next we'll open the file itself. Since we opening an external file (as opposed to a SAS structure such a data set or catalog), we need to use the FOpen function (File Open). As usual, this will return a pointer to the structure of interest, in this case stored in the local macro variable &fid.

```
%let fid = %sysfunc(FOpen(&theFile));
```

Since these functions operate differently across operating system, we need to extract the number of options available on the current OS. Again, we are extracting the File Option Number, so the function name called is FOptNum. In this example, the information will be captured into the local macro variable &OptNum.

```
%let OptNum = %sysfunc(FOptNum(&fid));
```

Now that the macro has sufficient information, we can execute a %DO loop across all of the available options. For

each option, we will extract the File Option Name of the option (using function FOptName), as well as the value of the option (File Information: FInfo). The file information is displayed using a %Put statement. After processing has completed, close the file with the function FClose (File Close).

```
%do i = 1 %to &OptNum;
  %let OptName = %sysfunc(FOptName(&fid,
    &i));
  %let OptValue = %sysfunc(FInfo(&fid,
    &OptName));
  %put Option # &i (&OptName) is &OptValue;
%end;
%let rc = %sysfunc(FClose(&fid));
%mend d3;

%d3;
```

The results are:

```
Option # 1 (File Name) is
C:\UG\WUSS2001\SysFunc Slides\sysfunc.ppt
Option # 2 (RECFM) is V
Option # 3 (LRECL) is 256
```

Computing with %SYSFUNC

Mike Rhoads contributed an interesting illustration of a %SYSFUNC utility to the SAS-L:

```
%put Base date for SAS software is
%left(%sysfunc(putn('01jan1960'd, best.)));
```

results in

```
Base date for SAS software is 0
```

Further illustrating the computational power of (layer 1) macro language:

```
%let begdate = '01Jan1995'd;
%let enddate = '31Dec1995'd;
%put Number of days from BEGDATE to ENDDATE
is %left(%sysfunc(putn(&enddate-&begdate+1,
best.)));
```

results in:

```
Number of days from BEGDATE to ENDDATE is
365
```

Put Your Code to the Test

Many functions called from %SYSFUNC return a value indicating the status of the function. As noted above, very often it is simply a 1 if the function was successful and 0 if not. Others, however, have a more extensive set of return codes. You can test and/or print the return code and associated message to control the flow of subsequent processing. Assuming the file "c:\delme.dat" does not exist, submitting the following:

```
filename x 'c:\delme.dat';
%let rc = %sysfunc(fdelete(x));
```

```
%put &rc -- %sysfunc(sysmsg());
```

would produce the following in the log:

```
20006 -- ERROR: Physical file does not
exist, c:\delme.dat.
```

A quick note about the SCL functions SysMsg and SysRC. These functions return the error messages and error codes related to the execution of the most recent SCL function. The above %Put statement could be replaced with

```
%put %sysfunc(sysrc())--%sysfunc(sysmsg());
```

with the same results.

Note: the error message returned by SysMsg is reset to blank when the function is called. However, SysRC is not reset, and will always contain the error condition of the last SCL function call. Thus, invoking the above statement twice in succession

```
%put %sysfunc(sysrc())--%sysfunc(sysmsg());
%put %sysfunc(sysrc())--%sysfunc(sysmsg());
```

would generate the following in the log:

```
20006 -- ERROR: Physical file does not
exist, c:\delme.dat.
20006 --
```

The significance of this is that you'll need to check SysRC and SysMsg immediately after (each and every!) function call. That way you're always sure to which function call the error values are related.

Of course, providing feedback to an end user with error codes (20006) that wouldn't mean anything to most SAS programmers may be less than useless. It is the art of capturing and dealing gracefully with such feedback that is the pinnacle of the developer's art.

Get to Know Your Stuff

We've probably all used Proc Contents or the Dictionary.Tables (a.k.a. SASHELP Views) in a Data step or Proc SQL to get information about a dataset. With SCL functions at our fingertips, another option exists. There are two SCL functions which can tell you almost everything you'd want to know about a dataset. They two functions, AttrN and AttrC, differ only in the type of information they return. AttrN returns numeric attributes; flags, counts and dates. AttrC returns character attributes, such as names, labels and types.

In order to use these functions, we'll need to "open" the dataset, call the appropriate function(s), and then close the dataset. This actually recapitulates much of the functionality that the Data step provides for us behind the scenes, and which many SAS programmers happily ignore.

The **Open** function returns a dataset id which is used to identify the dataset to the **ATTRX** functions. The other argument to the **ATTRX** functions is the name of the attribute to be extracted. An important note: an open dataset is locked to other processing and must be closed, using the dataset id, prior to further use. Always close the dataset when you have completed your work with it.

The following example (all layer 2) opens the **CRIMES** dataset in the **SASUSER** library. It then retrieves the following attributes:

- The number of observations (**NOBS**)
- The number of variables (**NVARS**)
- The creation date of the dataset (**CRDTE**)
- The variables the dataset is sorted by (**SORTEDBY**)

```
%let dsid=%sysfunc(open(sasuser.crime));
%let NumObs= %sysfunc(AttrN(&dsid,NObs));
%let NumVars=%sysfunc(AttrN(&dsid,NVars));
%let CreateDt=%sysfunc(AttrN(&dsid,Crdte));
%let SortBy=%sysfunc
  (AttrC(&dsid,SortedBy));
%let rc = %sysfunc(close(&dsid));
```

Notice that all the calls to **AttrN** and **AttrC** use the dataset ID (**&Dsid**) which was returned by the **Open** function. This ID is also used in the **Close** function to release the dataset.

All in the Macro Family

SAS Institute has created a set of macros which correspond to most of the functions supported by **%SYSFUNC**. One example is the macro **%REVERSE**:

```
%MACRO REVERSE(ARG1);
  %SYSFUNC(REVERSE(&ARG1))
%MEND;
```

Both of the following statements:

```
%put %sysfunc(reverse(REVERSE ME));
%put %reverse(REVERSE ME);
```

would produce:

EM ESREVER

If you find yourself using the same **%SYSFUNC** statement again and again, consider enclosing it in your own macro. Earlier, we used the **Exist** function to check for the existence of a dataset. We can create a macro called **Exist**:

```
%macro Exist(DSname)
  %sysfunc(exist(&DSname));
%mend;
```

Invoking this macro as follows:

```
%Exist(sasuser.crime)
```

will produce a 1 (assuming the dataset exists). Note that this does not violate the suggestion to always provide full feedback and error handling code: this macro is intended to be called by other macros in the developer's library, and not necessarily by the end user directly.

More fully, the **Exist** macro will often be called as:

```
%if %exist(sasuser.crime) %then ...
```

You might recall the embedded **%SYSFUNC** calls we used to calculate the value of the previous month when passed a date in **YYMM** format. Enclosing this is a macro greatly simplifies further usage:

```
%macro LastMnth(ThisMnth);
  %sysfunc(intnx(month,
    %sysfunc(inputn(&ThisMnth.01, yymmdd6.)),
    -1),
    yymmdd4.);
%mend;
```

Invoking this macro, as follows:

```
%LastMnth(9701)
%LastMnth(9812)
```

produces 9612 and 9811, respectively. Much easier to look at than repeating the function call.

Opening a Data Set

A fundamental concept in SCL is that before code can read from, add to, or manipulate an object such as a SAS data set, a directory, or a file, it first must be identified by being "opened" with an SCL function call. In the case of a SAS data set, this is accomplished in **%SYSFUNC** macros by using the corresponding **Open** and **Close** functions. The **Open** function call returns a numeric value, which is captured and utilized by the programmer in many subsequent function calls. The following simple example executes in open macro code:

```
%let dsid = %sysfunc
  (Open(wuss.example1,i));
%let dsname = %sysfunc(DsName(&dsid));
%put The data set opened was &dsname;
%let rc = %sysfunc(close(&dsid));
```

that results in the following string being written to the **SASLOG**:

```
The data set opened was WUSS.EXAMPLE1.DATA
```

Note the importance of always closing open data sets after processing has completed. Also, it is highly desirable to perform error checking before assuming the **Open** function was successful. This necessitates the creation of a macro (layer 3), since open macro code cannot process **%If...%Then** logic.

Using the technique started above, create a macro **ODSet** (Open Data Set) which attempts to open a SAS data set; if

it exists, the macro extracts the name of the data set using the `DsName` function, writes the name to the SASLOG, and closes the data set. If the data set is not found, the macro retrieves the resulting error number and error message through the use of the `SysRC` and `SysMsg` functions.

```
%macro odset(dsname);
%let dsid = %sysfunc(Open(&dsname, i));
%if &dsid le 0 %then %put %sysfunc(SysRC())
-- %sysfunc(SysMsg());
%else %do;
    %let dsname = %sysfunc(DsName(&dsid));
    %put The data set opened was &dsname;
    %let rc = %sysfunc(Close(&dsid));
%end;
%mend odset;

%odset(wuss.example);
%* Note incorrect name;
```

Invoking the macro above with the data set name misspelled results in the following message in the SAS LOG:

```
70009 -- ERROR: File WUSS.EXAMPLE.DATA does
not exist
```

Note the creation and repeated reference to `&dsid`, the data set id. This numeric value will be used through SCL and %SYSFUNC code to refer to previously opened data sets. *Dsid* is commonly used as the data set identifier throughout SAS documentation and examples, but any valid SAS name may be utilized.

How Big is the Data?

There are many occasions when it is useful to know how many observations a particular data set contains. This is a good opportunity to develop a useful addition for your macro toolbox.

Let us assume that this macro will be called by other macros, so the feedback will be in the form of simple return codes, and not English text messages. The macro will have just one parameter, `&DSN` (Data Set Name). This should include a library reference if the data set is not in the Work library.

1. Define the return code (RC) to be local to the macro environment. The information from the macro (data set exists or not) will not be returned into a global macro variable in this case.
2. If the input parameter is missing, assign it the value of the system options `_last_`. This defaults to whatever the last created data set, much as when a Proc is called without a data set reference (e.g. Proc xxxx data = ...).

3. Use the `Exist` function to determine if the data set even exists. If it does not, skip all other processing and simply return a `<-1>` value.
4. If the data set does exist, use the `Open` function to open it and assign a file handle (`&dsid`).
5. Use the `AttrN` function to retrieve the number of observations, stored in the local macro variable `&NObs`.
6. Close the data set per good housekeeping rules.
7. Here is the most interesting part: the last executable statement is simply the invocation of the local macro variable `&NObs`. This would seem to have no useful effect—clearly, this resolves to some numeric value which is then dropped out of the macro.

```
%macro GetObs(dsn);
%local nobs rc;
%if &dsn= %then %let dsn=_last_;

%* Verify that dataset exists at all;
%if %sysfunc(exist(&dsn)) %then %do;
%* Open the dataset, get a file handle;
%let dsid = %sysfunc(open(&dsn, I));
%* Get the number of Observations;
%let nobs=%sysfunc(AttrN(&dsid, NOBS));
%let rc = %sysfunc(close(&dsid));
%end;
%* Dataset does not exist--return error
code;
%Else %let nobs=-1;
&nobs
%mend GetObs;
```

This can be captured (even in layer 2 code) and displayed as follows:

```
%* Data set five has 5 observations;
%let NumObs = %GetObs(five);
%put &NumObs;
%* Writes value <5> to the Log;
```

One caveat about the `GetObs` macro is that it is somewhat dependent on your SAS environment. It is intended to be used only on traditional SAS data sets, and not such special cases as:

- Data Step view
- SQL View
- Compressed SAS data set
- Data sets containing records marked for deletion, but not yet purged.

Retrieving Data Into a Title

If data from a SAS data set need s to be placed into a title, a `data _null_;` step could retrieve the relevant information, call `Symput` to create a global macro, then reference the macro variable in a `Title` statement. %SYSFUNC macros permit this action without the overhead of a DATA step. The following macro will open

a SAS data set, read through it searching for a particular city name, and then retrieve a numeric livability index assigned to that observation.

Begin the macro definition. The macro has one parameter, city (name).

1. Define the macro variable *index* to have global scope, since it will be referenced outside of the current macro environment.
2. Use the SCL function **Open** to open the data set in input mode.
3. Use the **AttrN** (**A**tttribute **N**umeric) function to retrieve the number of observations in the data set. This allows a macro do loop to know how many times to cycle through the data set, retrieving one observation at a time.

```
%macro ret(city);
  %global index;
  %let dsid=%sysfunc
    (open(wuss.example4,i));
  %let nobs=%sysfunc(attrn(&dsid, nobs));
```

4. Begin a do loop to process all observations. The **Fetch** (just what it sounds like!) function retrieves the next observation from the data set, somewhat like the **SET** statement in the DATA step. However, (as used here) it does not load any values from the observation into memory (unlike the **Set** statement in the Data step).
5. Extract the value of *city* from the data set using the **GetVarC** (**G**et **V**ariable **C**haracter) function. The function needs to know which data set and which variable in the data set to retrieve. The data set is identified by *&dsid*, while the variable number is extracted by utilizing the **VarNum** (**V**ariable **N**umber) function.
6. If the city just retrieved is the same as the one specified in the macro invocation, proceed as in step 7 to retrieve the variable *index*. The only difference here is the use of **GetVarN**, (**G**et **V**ariable **N**umeric) since *index* is a numeric not a character variable.
7. As always, close data sets once processing has completed.

```
%do i=1 %to &nobs;
  %let rc=%sysfunc(fetch(&dsid));
  %let thisCity=%sysfunc(GetVarC(&dsid,
    %sysfunc(VarNum(&dsid, city))));
  %if &thisCity = &city %then
    %let index=%sysfunc(GetVarN(&dsid,
      %sysfunc(VarNum(&dsid, index))));
  %end;
  %let rc = %sysfunc(close(&dsid));
%mend ret;
```

Once the macro has been compiled, invoke it along with the name of the desired city. The result is placed in a global macro variable that can be used in a following title statement. Note the use of double quotes to surround the title text, allowing the macro processor to detect and resolve the enclosed macro reference with its value.

```
%ret(ISSAQUAH);
proc print data = wuss.example4 noobs;
  title1 "Issaquah has a livability index of
&index";
run;
```

Issaquah has a livability index of 13.6

CITY	INDEX
LOS MESAS	9.2
LE GRAND	11.6
HOUSTON	1.2
ISSAQUAH	13.6
SEATTLE	12.5
BEDFORD	12.1

Yet Another View of Contents

The SAS developer already has several methods available for obtaining information about the contents of a SAS data set: **Proc Contents**, **Proc Datasets**, the **Access** window, the **Var** window, and the **SasHELP.Vcolumn** view. The %SYSFUNC macros offer yet another way to get at this information.

1. The **Stmt** option on the macro invocation means that the macro can be called from the command line of any window.
2. The **Exist** function is then called to determine if the requested data set exists; if not, line 16 reports that condition and the macro ends.
3. If the data set does exist, line 3 uses the **Open** function to open the data set in *input* mode.
4. Use the **AttrN** function to determine how many variables *&dsn* contains. Have a **%DO . . . %END** loop execute once for each variable in the data set.
5. In line 6, the variable name of the current *ith* variable is extracted using **VarName**.
6. The *varname* just extracted is then used to determine the variable number (*varnum*) on the data set, using **VarNum**. SAS developers usually do not concern themselves with the variable number in a data set, but the %SYSFUNC functions depend on it heavily.
7. Lines 8-11 then use the *varnum* to extract the variable's label, type (expressed as C or N), format, and informat.

8. Finally, the %Put writes out the information collected in a concise report. Naturally this layout could be altered to suit individual preference.

```
%macro GetInfo(dsn) / stmt;
%if %sysfunc(exist(&dsn)) %then %do;
  %let dsid = %sysfunc(open(&dsn, I));
  %let numvars=%sysfunc(attrn(&dsid,NVARS));
  %do i = 1 %to &numvars;
    %let varname=%sysfunc(varname(&dsid,&i));
    %let varnum=
      %sysfunc(varnum(&dsid,&varname));
    %let varlabel=%sysfunc(varlabel(&dsid,
      &varnum));
    %let vartype=%sysfunc(vartype(&dsid,
      &varnum));
    %let varfmt=
      %sysfunc(varfmt(&dsid,&varnum));
    %let varinfmt=%sysfunc(varinfmt(&dsid,
      &varnum));
    %put Varname= &varname (&varlabel)
      &vartype fmt &varfmt infmt
&varinfmt;
  %end;
  %let rc = %sysfunc(close(&dsid));
%end;
%else %put &dsn does not exist;
%mend GetInfo;
```

Invoking this macro from a command line:

```
%getinfo(wuss.example1)
```

results in the following information being written to the SAS Log:

```
Varname=NAME(Name of person) C fmt $8. infmt $8.
Varname=HIREDATE (Hire Date) N fmt DATE9.
infmt MMDDYY8.
Varname= GENDER (Gender) C fmt $GENDER.
infmt $1.
```

Invoking the macro with an nonexistent data set results in:

```
%getinfo(wuss.example)
wuss.example does not exist
```

Displaying Settings

Aside from saving and restoring settings, it may be useful for you to have a utility macro that displays the current values of critical system option settings. In version 6.12 and previous, it was easy to display system option values by invoking the Options window (one of the more obscure DMS functions!). This window still exists under version 7 and 8, but is now more organizing, displaying the information in a series of different tabs. While this is more organized than the single window available previously, it does require that you learn which tabs hold the information of interest. I suspect this will be a poor use of

memorization energy for most SAS users, especially when there is an easy %SYSFUNC way.

We can use the `GetOption` function mentioned above to facilitate easy surfacing of this type of information. If your environment involved frequent changing of the Line and Page sizes in your SAS session, you could write a utility to display them directly:

```
%macro ShowSize;
  %let LineSize = %scan
    (%sysfunc(GetOption
      (linesize, keyword)), 2,=);
  %let pagesize =
    %scan(%sysfunc(GetOption(pagesize,
      keyword)), 2,=);

  %put The page size is &PageSize, while the
line size is &LineSize.;
%mend ShowSize;
```

On my system, this displays as:

```
The page size is 60, while the line size is
180.
```

Saving and Restoring Settings

When developing interactions, care should be taken to ensure the application is “well behaved”. Should the application need to set system options to particular values in order to function properly, it should first record and save the values, set the options as needed for itself, then restore the system to the original state.

This utility may designed in (at least) 2 ways:

1. Create two macros: one to retrieve and save the desired option, and another to restore it, or
2. Create one macro with 2 code sections inside.

For no particularly good reason, let’s choose option 2. This means that the first parameter needs to be the type of action take: to retrieve and store the parameter, or later to restore the parameter. Since we’re looking at how to be a good developer, first the macro should perform error checking and provide appropriate feedback to the user:

```
%macro optRstore(action,optname,macsname);
  %global &macsname;

  %let action = %upcase(&action);
  %if &action ne STORE and
    &action ne RESTORE %then %do;
    %put ERROR: Macro optRstore called with
Action set to &action;
    %put ERROR: Valid choices are: [STORE
RESTORE];
    %goto finish;
  %end;
```

```
%Finish:
```

```
%mend optRstore;
```

Calling this macro with an inappropriate Action setting results in useful feedback to the end user. The feedback indicates the name of the macro being called, the erroneous entry provided, and a list of correct values to choose among when correcting the code.

```
%optRstore(store1);
ERROR: Macro optRstore called with Action
set to STORE1
ERROR: Valid choices are: [STORE RESTORE]
```

The second and third parameters the macro needs are the name of the option to store/restore, and the name of a global macro variable to store it in.

This last parameter raises an interesting issue: for this version of the macro, we are placing the burden of knowing where each option setting is stored in the global macro symbol table upon the shoulders of the end user (who calls the macro). Undoubtedly, this functionality could be assumed by the macro itself. One might be tempted to use the name of the option itself as the macro variable for storing its value. Unfortunately, option names can be as long as 19 characters, while macro names are still limited to 8 characters. The macro could easily truncate the option name to eight characters, but this raises the possibility that future versions of the SAS System could have different options whose first 8 characters are the same. You will have to make your own judgment of the risks of this occurring vs. the burden of asking your macro users to come up with a third parameter.

The completed macro looks like:

```
%macro optRstore(action, optname, macsname);
  %let action = %upcase(&action);
  %if &action ne STORE and
    &action ne RESTORE %then %do;
    %put ERROR: Macro optRstore called with
    Action set to &action;
    %put ERROR: Valid choices are: [STORE
    RESTORE];
    %goto finish;
  %end;

  %let optname = %upcase(&optname);
  %let msname = %substr(&optname, 1, 8);
  %global &macsname;

  %if &action = STORE %then %do;
    %let optvalue = %scan
    (%sysfunc(getOption
    (&optname, keyword)), 2,=);
    %let &macsname = &optvalue;
  %end;
```

```
%else %if &action = RESTORE %then %do;
  options
  &optname = &&macsname;
%end;
%Finish;
%mend optRstore;
```

Invoking the macro in a way as to test it results in the following output to the SAS Log:

```
Options pagesize = 40;
%ShowSize;
The page size is 40, while the line size is
124.
%optRstore(store, pagesize, mypage);
options pagesize = 60;
%ShowSize;
page size is 60, while the line size is 124.
%optRstore(restore, pagesize, mypage);
%ShowSize;
The page size is 40, while the line size is
124.
```

Other Useful Functions

While not part of the functionality included in %SYSFUNC, the following tidbits were introduced at the same time, and may have been overlooked by the intrepid macro coder. They are worthy of at least a quick mention.

Adding Numbers

Historically, the macro language has not supported non-integer arithmetic expressions. Adding the values 1 + 2 with the SysEval function properly returned 3, but 1.0 + 2.0 would return an error. If the macro programmer needs to evaluation floating point numbers, the new SysEvalF supports this ability.

```
%macro addint(num1, num2);
  %let result = %eval(&num1 + &num2);
  %put &num1 + &num2 is &result;
%end;

%addint(1, 2);
1 + 2 is 3
```

```
%addint(1.0, 2.0);
ERROR: A character operand was found in the
%EVAL function or %IF condition
       where a numeric operand is required.
The condition was: 1.0 + 2.0
ERROR: The macro will stop executing.
```

A new function, SysEvalF, enables non-integer expressions to be evaluated correctly during macro execution:

```
%macro addflt(num1, num2);
  %let result = %sysevalf(&num1 + &num2);
  %put &num1 + &num2 is &result;
%end;

%addflt(1, 2);
1 + 2 is 3

%addflt(1.1, 2.1);
```

```
1.1 + 2.1 is 3.2
```

What's Local?

It has been fairly easy now for some time to show the contents of the global macro symbol table:

```
proc print data = sashelp.vmacro label;
  id name;
  var scope value;
  where (scope eq 'GLOBAL');
run;
```

However, it has been fairly tedious to see all of the contents of a local symbol table—that is, all of the symbols that are defined only within a macro, and only have existence while the macro is running. Since a macro step debugger does not exist (like the Data step debugger or the SCL debugger), the macro developer is forced to place %put statements throughout the macro in an attempt to see what is being created that is causing a problem (Funny, I remember doing that with SAS version 79.6 in 1980! Some things don't change much... or do they?)

The main problem with scattering %put statements throughout your ill-behaved macro is that it will only write out the macro variables you request. The human (not technical) problem is that very often the variable causing problems is not obvious, so is not one that is examined early in the debugging process. A new keyword available for the %put statement is `_local_`. This writes out the entire contents of the local macro symbol table. Careful examination of these values often leads to the desired problem resolution.

```
%macro test;
  %let one = 1;
  %let two = 2;
  %let three = &one.&two;
  %put _local_;
%mend test;

%test;
TEST TWO 2
TEST THREE 12
TEST ONE 1
```

You're Out of Here!

Many macro mavens have devised clever tricks for deleting global macro variables, usually with mixed to no success. Fortunately, this has been resolved in SAS version 8.2. The new macro statement %SymDel (Symbol Delete) makes this a native part of the SAS language.

```
%let globvar = here;

%put &globvar;
here
%SymDel globvar;

%put &globvar;
WARNING: Apparent symbolic reference GLOBVAR
not resolved.
&globvar
```

Resources

You might make the case that SYSFUNC is one of the poorest documented components in the SAS system. There are several generations of books on the general macro language published by the SAS Institute, and several fine Books By Users authors. There have been myriad papers presented at various users conferences about all kinds of macro features, problems, and workarounds for decades. SAS Institute has published several linear feet of documentation about SAS/AF and SCL, along with two BBU volumes.

Web

Despite the fact that SYSFUNC has been available since release 6.12, there are no formal SAS Institute publications, only one SAS Institute authored SUGI paper, and very few presentations by the user community. The principal documentation consists of a PDF file available on the SAS web site (<http://www.sas.com>), called TS544.PDF. This documents the %SYSFUNC is a rather spare way, in the tradition of base SAS documentation. A few examples are provided, but much is left to the imagination and experimentation of the curious and intrepid explorer. Of course, unless one has a clue about what the particular information will provide in terms of benefits, there is no particular reason to explore it!

The SAS-L

The SAS-List server is an invaluable resource for anyone on the planet who wants to further and deepen their knowledge of SAS software. An enormous wealth of knowledge, experience, and sound advice passes through the Internet every day. Fortunately, it is easy to search the archives for a particular string. An appendix of all of the references to %SYSFUNC would make a nice addition to this class, showing real life examples of practical applications. However, this ambition was slowed down when the following result was found by the search engine:

www.listserv.uga.edu

The University of Georgia

[Home](#) | [Browse](#) | [Manage](#) | [Request](#) | [Manuals](#) | [Register](#)

Search results - SAS-L

1528 matches

Instead of all 1,528 examples from the SAS-L, several complete examples from the author's work life are included instead.

Conclusion

Hopefully this post-conference training has supplied the SAS developer with an extensive set of new functions. It will be reasonable to expect some time to go by before this becomes easy and natural to use in your macro programs. There are not only new functions to learn and master, but new ways of thinking about the process of developing macro programs to incorporate into your work flow. As you begin to try new programs using these techniques, or even go back and incorporate new functionality, flexibility, and feedback into existing (unbroken but non-optimal macros), you might begin to feel a sense of freedom and relief. The macro language is one of the oldest, and least updated parts of the SAS System over the last two and a half decades: with one swoop, the SAS development team greatly enhanced the power and flexibility of this vital toolset. Hopefully your macro codes will be more powerful, robust, user-friendly, and straightforward to develop.

Acknowledgments

SAS is a registered trademark of SAS Institute, Inc. in the USA and other countries. ® indicates USA registration.

References

Lund, Peter, "Always There When I Need You: Macros to Create Directories", *Proceedings of the Sixteenth Annual Pacific Northwest SAS Users Group Regional Conference*, 1997.

Lund, Peter, "Fun with %SYSFUNC", *Proceedings of the Sixteenth Annual Pacific Northwest SAS Users Group Regional Conference*, 1997.

Polzin, Jeff and Susan O' Connor, "Macro Facility Enhancements for Release 6.09E and Release 6.11", *Proceedings of the Twenty-First Annual SAS Users Group International Conference*, Cary NC: SAS Institute Inc., 1996.

SAS Institute Inc., *SAS Screen Control Language: Reference, Version 6, Second Edition*, Cary NC: SAS Institute Inc., 1995.

Mike Rhoads, *SAS Tip: Date constants and calculations*. Published with permission from Mike Rhoads, Westat (RhoadsM1@Westat.com).

SAS Institute Inc., *SAS Screen Control Language: Reference, Version 6, Second Edition*, Cary NC: SAS Institute Inc., 1995.

Acknowledgments

SAS is a registered trademark of SAS Institute, Inc. in the USA and other countries. ® indicates USA registration.

Author Contact Information

Paul Hamilton
ICOS Corporation
22021 20th Ave SE
Bothell, WA 98021
voice: (425) 489-2499
e-mail: phamilto@icos.com