Paper SAS1747-2015

Helping You C What You Can Do with SAS®

Andrew Henrick, Donald Erdman, and Karen Croft, SAS Institute Inc., Cary, NC

ABSTRACT

SAS® users are already familiar with the FCMP procedure and the flexibility it provides them in writing their own functions and subroutines. However, did you know that FCMP also allows you to call functions written in C? Did you know that you can create and populate complex C structures and use C types in FCMP? With the PROTO procedure, you can define function prototypes, structures, enumeration types, and even small bits of C code. This paper gets you started on how to use the PROTO procedure and, in turn, how to call your C functions from within FCMP and SAS®.

INTRODUCTION

The use of functions to reduce the complexity of a program is nothing new to SAS® programmers. The FCMP procedure was introduced in SAS® 9.2 to allow users the ability to define their own functions and subroutines, breaking large problems down into manageable pieces. These functions could be saved and shared as needed, eliminating the need to write code to accomplish the same task over and over again.

However, a lot of programmers already have functions written in other languages that solve complex problems. Converting that code into an FCMP function is not only problematic, but might not be feasible. If those functions are written in C or in code that is callable through a C interface such as C++, the PROTO procedure can be used to make these functions callable directly from SAS®.

A BRIEF OVERVIEW OF FCMP

The FCMP procedure enables you to define your own functions and subroutines. These reuseable blocks of code are supported within many other SAS® procedures and the DATA step. Before we go over the PROTO procedure it is important to note that to take advantage of PROTO and its features within the DATA step, you must first call an FCMP function or subroutine. SAS® procedures that support FCMP functions can call PROTO declared interfaces directly. At this time, the list includes:

- SAS/ETS® procedures: MODEL, SEVERITY, SIMILARITY, VARMAX
- SAS/STAT® procedures: CALIS, GENMOD, GLIMMIX, MCMC, NLIN, NLMIXED, PHREG, SURVEYPHREG, QUANTREG
- SAS/OR® procedures: GA, NLP, OPTMODEL
- SAS® Risk Dimensions® procedures

The following is a subset of the FCMP procedure syntax relevant to this discussion:

PROC FCMP *option(s)*;
ARRAY *array-name*[*dimensions*] </NOSYMBOLS | *variable(s)* | *constant(s)* | (*initial-values)*>;
EXTERNC *double* | *short* | *int* | *long* | *char* <* | **> *variable_name;*
FUNCTION *function-name(argument(s))* <VARARGS> <$> <*length*><LABEL='*string*'>;
STRUCT *structure-name variable*;
SUBROUTINE *subroutine-name (argument(s))* <VARARGS><LABEL='*label*'>;
<expressions>

And here is a simple example that demonstrates defining and saving a function as well as calling it from the DATA step:

```
proc fcmp outlib = work.funcs.test;
  function factorial(x);
    if x < 0 then return(.);
    else if x eq 0 then return(1);
    else return(x * factorial(x - 1));
  endsub;
run;

option CMPLIB = work.funcs;
```

```
data _NULL_;
   y = factorial(5);
   put y=;
run;
```

Discussing the FCMP procedure in detail is outside the scope of this paper. For more information, see the FCMP section of the *Base SAS® Procedures Guide*.

THE PROTO PROCEDURE

The purpose of PROC PROTO is to create an interface to tell FCMP how to call the routines in your C library. Type definitions and prototypes provide FCMP a means to type check function calls and move data between SAS® types and standard C types. The syntax for the PROTO procedure is as follows:

PROC PROTO PACKAGE=*entry* <*options*>;
MAPMISS *type1=value1 type2=value2 ...*;
LINK *load-module* <NOUNLOAD>;
function–prototype–1 <function-prototype-n …;>

At first PROC PROTO will seem familiar to FCMP users in that you're creating a library of function definitions and saving them to a package so that they can be shared and used by others. However, PROC PROTO is a means to define your external function library; it is not possible to run your code without first loading your interface definition into FCMP. PROC PROTO will provide some feedback: for example, if the LINK statement cannot find or load your external library you will get an error message.

Let's examine the syntax a little closer:

PROC PROTO PACKAGE=*entry* <*options*>;
The PROC PROTO statement has one required option, PACKAGE=, where you indicate where your prototypes are to be saved. A three level name is required *(libname.dsname.pkgname)*, allowing for a logical grouping of a family of functions by package name. Unlike FCMP, function names must be unique across all loaded PROTO packages. Procedure options of note are as follows:

- ENCRYPT or HIDE: Any source code (not prototypes) will be encrypted when saved.
- LABEL= : You can provide a text string describing the package of prototypes.
- NOSIGNALS: Tells FCMP that none of the functions provided will produce exceptions, allowing for faster performance at runtime.
- STDCALL: on Windows machines, indicates that the __stdcall calling convention is to be used.

MAPMISS *type1=value1 type2=value2 ...*;
The MAPMISS statement gives the user the flexibility to specify alternative values for MISSING based on type. The options for type are INT, UINT, DOUBLE, LONG, ULONG, SHORT and USHORT. If MISSING values are passed as function arguments and MAPMISS is not used, the function will not be called and any return values from the function will be set to MISSING.

LINK *load-module* <NOUNLOAD>;
The LINK statement specifies the name of your function library and possibly the location. Fully qualified pathnames are recommended but not necessary if system PATH environment variables are setup correctly. More than one LINK statement is supported if functions from more than one library are being described within a single package. The NOUNLOAD option is an optimization allowing you to leave your library loaded between PROC FCMP steps, avoiding the overhead of loading and unloading the module multiple times.

function–prototype–1 <function-prototype-n …;>

This is where you list the prototypes for the functions you want to use from the loaded libraries from the LINK statements. You do not have to list all the functions from each library, just the ones you wish to make available in PROC FCMP. Each function prototype is of the following form:

return-type function-name (*argument-type* <*argument-name*> / <*iotype*><*argument-label*>, ...) <*option(s)*>;

The above is largely what you would expect as a C programmer, but there are a couple of differences to help indicate how this package of functions is to be used. For each argument you can specify the following:

- IOTYPE: For pointer arguments this can be useful to specify whether the value passed will be updated in the called function. This helps the FCMP compiler know it does not have to copy values back into SAS® variables. IOTYPE can be I for input, O for output, or U for update.
- LABEL: You have the option to place a quoted string after the argument name (or the IOTYPE setting) to be used as the argument label.

Lastly for each function you have the option to provide a LABEL describing the function or a KIND to group similar functions together within a package.

CONNECTING THE DOTS

That's enough to revisit our simple FCMP program and expand it to include PROC PROTO. Suppose that you have your own function library which contains a function named *factorial*. Rather than simply return a value, you've implemented the function with an input and output argument. Here's the code for the function for the Microsoft Windows® operating system:

```
__declspec(dllexport) void factorial(double x, double * fact);

void factorial(double x, double * fact)
{
   if (x < 0) {
      *fact = 0;
   }
   else {
      *fact = 1;
      while (x > 0) {
         *fact = *fact * x;
         x = x - 1;
      }
   }
}
```

And here is the PROC PROTO and FCMP code:

```
proc proto package = work.funcs.protopkg;
   link "funcsample";
   mapmiss double = -1;
   void factorial(double x, double * fact / iotype = O);
run;

proc fcmp inlib = work.funcs;
   file log;
   x = 5;
   call factorial(x, y);
   put x= y=;
run;
```

Here is the log from the above code:

Factorial Function

```
28   proc proto package = work.funcs.protopkg;
29      link "funcsample";
30      mapmiss double = -1;
31      void factorial(double x, double * fact / iotype = O);
32   run;

NOTE: 'funcsample' loaded from TKEXTENSION path.
NOTE: Prototypes saved to WORK.FUNCS.PROTOPKG.
NOTE: PROCEDURE PROTO used (Total process time):
      real time           0.14 seconds
      cpu time            0.07 seconds
```

```
33
34   proc fcmp inlib = work.funcs;
NOTE: 'funcsample' loaded from TKEXTENSION path.
35      file log;
36      x = 5;
37      call factorial(x, y);
38      put x= y=;
39   run;

x=5 y=120
NOTE: PROCEDURE FCMP used (Total process time):
      real time           0.25 seconds
      cpu time            0.04 seconds
```

Output 1. Log Output from PROC PROTO and FCMP

Notice that using a void function with an output argument essentially makes our external routine a CALL routine, so FCMP requires the use of 'call'. Also notice that *factorial* already looks for negative input values, so the MAPMISS statement in PROC PROTO simply passes any missing arguments as a -1. The INLIB= option is used to load the PROTO information directly, but the global option CMPLIB= can be used as you would for FCMP function libraries.

A TYPE IS A TYPE IS A TYPE

So exactly what types are supported in PROC PROTO? For return arguments you get the following:

Function Prototype	SAS Variable Type
short	numeric
short *	numeric, array
int	numeric
int *	numeric, array
long	numeric
long *	numeric, array
double	numeric
doube *	numeric, array
char *	character
struct *	structure
void	

Table 1. Supported C Return Types

Void as you saw before is comparable to a CALL routine in SAS syntax. With pointers you can assign the return value of the function to a scalar or an array. Structures are an essential addition and we will get to that in just a bit, but unions are not supported.

Function arguments support all of the above types, plus the following:

Function Prototype	SAS Variable Type
short **	array
int **	array
long **	array
double **	array
char **	character
struct **	structure

Table 2. Supported C Argument Types

The use of pointers to pointers requires that you pass an array to the function, at least for numeric variables. Both arguments and return types can be specified as unsigned. Please note that single precision floating point variables (float) are not supported directly. All values must be converted to a double to be passed to or from FCMP. Pointers and alternative numeric types can be declared within your user defined functions but require the use of the EXTERNC keyword.

```
proc fcmp;
   externc short ** sptrptr;
   externc short *  sptr;
   externc short    s;

   file log;
   s = 123.456;
   sptr = s;
   sptrptr = sptr;
   put s= sptr= sptrptr=;
run;
```

Here is the log output from the above code:


```
s=123 sptr=123 sptrptr=123
NOTE: PROCEDURE FCMP used (Total process time):
      real time           0.02 seconds
      cpu time            0.01 seconds
```

Output 2. Log Output from PROC FCMP

ADDING SOME STRUCTURE TO YOUR DATA

Structures are immensely useful, allowing you to group related variables together. Many C functions rely on passing pointers to structures. Within SAS®, you can only declare a structure definition within PROC PROTO but you can use that declared structure within PROC FCMP. Here's an example demonstrating defining a structure with members of various types:

```
proc proto package = work.funcs.protopkg;
   struct range {
      double high;
      int    mid;
      long * low;
   };
run;
```

Once defined, the structure type can be used and referenced in your PROC FCMP functions:

```
proc fcmp inlib = work.funcs;
   struct range   price;
   externc long * lptr;
   externc long   low;

   file log;
   put price=;

   low = 100;
   lptr = low;
   price.high = 1000.00;
   price.mid  = 500;
   if (isnull(price.low)) then price.low = lptr;
   put price=;

   call setnull(price.low);
   put price=;
run;
```

Here's the log output for this code:


```
price {high=0, mid=0, low=NULL}
price {high=1000, mid=500, low=100}
price {high=1000, mid=500, low=NULL}
NOTE: PROCEDURE FCMP used (Total process time):
      real time           0.16 seconds
      cpu time            0.03 seconds
```

Output 3. Log Output from PROC FCMP

Notice that all members of the structure variable are initialized to 0 (NULL). C style dot notation is used to access and set the structure members. However pointer members can be tested for NULL and set to NULL using the FCMP helper functions ISNULL and SETNULL. Simply pass the pointer variable to each function, ISNULL will return 1 or 0 based on the pointer's current settings and SETNULL will reset the pointer to NULL. For these two functions it is not required that the pointer variables be members of a structure: for example, we could have used SETNULL to reset LPTR.

What if your structure definition contains an array of structures? Accessing each member of the array requires another helper function that FCMP provides:

CALL STRUCTINDEX(*struct_array*, *index*, *struct_element*);

Let's adapt our PROC PROTO and FCMP code above to define a second structure that contains an array of ranges. For the sake of simplicity, let's remove the long pointer:

```
proc proto package = work.funcs.protopkg;
   struct range {
      double high;
      int    mid;
      long   low;
   };

   struct ranges {
      struct range * r;
   };

   struct range * alloc_ranges(int n);
   externc alloc_ranges;
      struct range * alloc_ranges(int n) {
         struct range * r;
         r = malloc(n * sizeof(*r));
         return(r);
      }
   externcend;
run;
```

Rather than link in a module that defines the function alloc_ranges, it's possible to inline a snippet of C code within PROC PROTO. Only a limited number of C routines are supported -- just malloc, free, and some math routines -- but it is handy. This is also a typical way to create your own simple wrappers to call your external library functions. Let's return to the example:

```
proc fcmp inlib = work.funcs;
  struct ranges  prices;
  struct range   price;

  file log;
  prices.r = alloc_ranges(3);
  do i = 1 to 3;
     call structindex(prices.r, i, price);
     put "Before setting the " i " element: " price=;
     price.high = 1000.00 * i;
     price.mid  = 500 * i;
```

```
      price.low  = 100 * i;
   end;

   do i = 1 to 3;
      call structindex(prices.r, i, price);
      put " After setting the " i " element: " price=;
   end;
run;
```

Each call to STRUCTINDEX gives you the *i*th position in the struct array and you can set values using PRICE like any other structure. Notice that the index passed is 1-based (1,2,3,…*n*) to be consistent with other SAS® arrays. Here is the log output:

Using Arrays of Structures

```
Before setting the  1  element:  price {high=0, mid=0, low=0}
Before setting the  2  element:  price {high=0, mid=0, low=0}
Before setting the  3  element:  price {high=0, mid=0, low=0}
 After setting the  1  element:  price {high=1000, mid=500, low=100}
 After setting the  2  element:  price {high=2000, mid=1000, low=200}
 After setting the  3  element:  price {high=3000, mid=1500, low=300}
NOTE: PROCEDURE FCMP used (Total process time):
      real time           0.03 seconds
      cpu time            0.01 seconds
```

Output 4. Log Output from PROC FCMP

Notice that as before all structure elements are initialized to 0. In this example we made two passes over the array to demonstrate that each position retained its values after the fields of each structure were set. Note also that the PUT statement knows how to print a structure.

BUT WAIT, THAT'S NOT ALL

You can also use enumerated types, typedefs, and #defines in PROC PROTO. Here's an example:

```
proc proto package = work.funcs.protopkg;
   #define TRUE  1;
   #define FALSE 0;

   typedef enum
   {
      January = 1, February, March, April, May, June, July, August, September,
October, November, December
   } Months;

   typedef enum
   {
      Sunday = 1, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
   } Days;

   struct date {
      Months month;
      Days   day;
      int    year;
   };
run;
```

Within PROC PROTO, #define statements must end with a semicolon and be numeric in value. Only simple replacement or unnested expressions are supported. The *enum* keyword cannot be used in a structure definition. That's why we've taken advantage of typedef in the above code. Enumerated values are setup as macro variables within FCMP, so use & to reference a specific value:

```
proc fcmp inlib = work.funcs;
   struct date d;

   file log;
   d.month = &February;
   d.day   = &Saturday;
   d.year  = 2015;
   put d=;
run;
```

The log for the above code shows the following:


```
d {month=2, day=7, year=2015}
NOTE: PROCEDURE FCMP used (Total process time):
      real time           0.03 seconds
      cpu time            0.01 seconds
```

Output 5. Log Output from PROC FCMP

A WORD OR TWO ABOUT SCOPE

The global option CMPLIB is used to load your PROTO definitions just like your function packages from FCMP. When you specify a dataset to be loaded, all packages within that dataset (PROTO or FCMP) will be loaded and available within your current FCMP functions. If you are loading multiple datasets in a list, use the following:

```
option CMPLIB = (work.funcs1 work.funcs2);
```

Packages will be loaded in order, possibly overriding previous definitions (such as with #DEFINE). It is important to ensure that certain elements like functions and enums across all loaded PROTO packages have unique names. Multiple functions with the same name will cause compilation errors.

```
proc proto package = work.funcs1.test1;
   #define VERSION 1;

   int func1(void);
   externc func1;
      int func1(void)
      {
         return VERSION;
      }
   externcend;
run;

proc proto package = work.funcs2.test1;
   #define VERSION 2;

   int func2(void);
   externc func2;
      int func2(void)
      {
         return VERSION;
      }
   externcend;
run;

option CMPLIB = (work.funcs1 work.funcs2);

proc fcmp;
   file log;
   x = func1();
   put "Should be 2: " x=;
run;
```

```
Should be 2:  x=2
NOTE: PROCEDURE FCMP used (Total process time):
      real time           0.06 seconds
      cpu time            0.04 seconds
```

Output 6. Log Output from PROC FCMP

DEALING WITH PESKY ERRORS

There are two types of pesky errors you might encounter while using the PROTO procedure: link errors and compile-time errors.

Link errors occur when the external library you are referencing cannot be loaded. When this happens it is usually one of the following:

1. The path to your library is incorrect. Check environment variables or use fully specified pathnames.
2. A dependent library that your shared library uses is not in the system search path.
3. The dependent library needed is not installed on the system.

The free utility *Dependency Walker* is very helpful when debugging library dependencies on Microsoft Windows®.

Compile-time errors happen when you have invalid C code specified in your PROC PROTO statements or you have multiple definitions of functions or structures in different packages. Compile-time errors are relatively easy to investigate and fix using the global option TKGPARM='dump'. When this option is on, the log from the compilation of the C code will be placed in your current working directory, where you are running SAS®. The log will be called *tkgcmp.log* or *prottkg.log* depending on whether you are running the FCMP procedure or the PROTO procedure.

CONCLUSION

PROC PROTO gives SAS® users a great deal of flexibility and power in defining their FCMP functions. In this overview of PROC PROTO syntax and the options available, we hope to help our users discover another tool available to them.

REFERENCES

- SAS Institute Inc. 2014. *The PROTO Procedure*. Cary, NC: SAS Institute, Inc. Available at http://support.sas.com/documentation/cdl/en/proc/67327/PDF/default/proc.pdf
- SAS Institute Inc. 2014. *The FCMP Procedure*. Cary, NC: SAS Institute, Inc. Available at http://support.sas.com/documentation/cdl/en/proc/67327/PDF/default/proc.pdf
- Eberhardt, Peter. 2009. **"**A Cup of Coffee and Proc FCMP: I Cannot Function Without Them." *Proceedings of the SAS Global Forum 2009 Conference.* Cary, NC: SAS Institute Inc. Available at http://support.sas.com/resources/papers/proceedings09/147-2009.pdf.
- Secosky, Jason. 2007. "User-Written DATA Step Functions." *Proceedings of the SAS Global Forum 2007 Conference*. Cary, NC: SAS Institute Inc. Available at http://www2.sas.com/proceedings/forum2007/008-2007.pdf.

RECOMMENDED READING

- *Base SAS® Procedures Guide, available at* http://support.sas.com/documentation/cdl/en/proc/67327/PDF/default/proc.pdf

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors:

Andrew Henrick
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
Andrew.Henrick@sas.com
http://www.sas.com

Donald Erdman
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
Donald.Erdman@sas.com
http://www.sas.com

Karen Croft
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
Karen.Croft@sas.com
http://www.sas.com