ABSTRACT

The SAS® System for the VMS™ environment supports user-written functions written in PL/I, MACRO, and C. This presentation provides an introduction to SAS user-written functions, with an example, written in the C programming language, that extracts Job Process Information (JPI) without exiting the SAS System. JPI includes characteristics of a user process such as user name, account name, privileges, limits and quotas for system resources, and priority. This function serves as a sample to other users interested in writing user-written functions. It is not intended as a complete implementation of the GETJPI service.

INTRODUCTION

A SAS user-written function, like a SAS function, is a DATA step statement and is invoked the same way. SAS user-written functions provide access to frequently used, complicated computations with a single SAS statement, or you can make calls to an operating system and retrieve information. It is up to you, with your imagination as the only limitation.

User-written function implementations are operating system specific. Under the VMS environment, a SAS user-written function is stored in executable form in the directory pointed to by the logical names SAS$SASLIB and SAS$USERLIB. When SAS sees a function reference that is not in the SAS library of known functions, it first looks for a user-written function in SAS$USERLIB and then in SAS$SASLIB. When the SAS user-written function is found, it is then dynamically loaded and executed.

STRUCTURE OF A USER-WRITTEN FUNCTION

There are two parts to a user-written function in the VMS SAS System, the initialization procedure and the function procedure. When the SAS compiler encounters a user-written function, the initialization procedure is called and fills in a structure that describes the user-written function to the SAS System. Each time the function is invoked, the function procedure performs the work of the function. It is passed two parameters: the number of arguments and a pointer to an array of structures containing the length and address of each parameter.

THE INITIALIZATION PROCEDURE

A sample initialization procedure is included with the SAS System in SAS$SASroot.UWFUNCS. It is made up of a #include statement, func.h, and code that you write to fill in elements of a structure. The initialization procedure must be the first procedure in the code. A listing of func.h follows and can be found in SAS Technical Report P-169, User-Written Functions, Formats, and Informats for the Version 5 SAS System under VMS.

```c
typedef struct FUNC_DCL *FUNC_DCL_PTR;
/* Tools for building the parameter types descriptor use */
#define CHAR_VAR 1
#define NULVAR 2
#define VARTYPE (t,X) (t«{2*(x-1)})
define CHAR Var 1
#define NV 1
#define VARTYPE (t,X) (t«{2*(x-1)})
define TYP e (CHAR VAR, 1)
define NV 1
```

The FUNC_DCL structure is needed by the SAS System to parse the function reference and call your function code. Your code fills in all the fields of the structure except the FUNC_MBZn fields.

An example of code for filling in the structure follows:

```c
/* This code fills in the FUNC_DCL structure */
strncpy (p->func_name, "JPIFNC.", 8);
strncpy (p->func_alias, "JPIFNC.", 8);
p->FUNC_ADDR=jpiFnc;
p->VARS=3;
p->FUNC_MIN_ARG=2;
p->FUNC_MAX_ARG=2;
p->FUNC_ARG_TYPE=CV(1) | NV(2);
/* This passes in the arguments */
p->FUNC_PARAMETER_LENGTH=0;
p->FUNC_P特定_CASE=0;
p->FUNC_SPECIAL_CASE=0;
```

A detailed description of each field of the code follows:

FUNC_NAME

This is the name of the function. It must follow the rules for a SAS variable name. If the name is less than eight characters long it must be padded to eight with trailing blanks.

FUNC_ADDR

This is a pointer to the function procedure. In C it is a pointer to a function returning long.

FUNC_MBZn

This field is reserved and may not be altered by your code. It is always initialized to 0 by the SAS System.

FUNC_ALIAS

This must contain the same name as the FUNC_NAME field. A user-written function is not allowed to have an alias name. It must also be padded to eight characters.
FUNC_RT_SPEC_CASE

This is the runtime special case flag. This field is normally 0. You set it to 1 if missing values and uninitialized variables are allowed as arguments to the function.

FUNC_MIN_ARGS

FUNC_MAX_ARGS

These two integers tell the DATA step compiler how many arguments are permitted or required for the function. They both must range from 1-15 and the minimum number of arguments must be less than or equal to the maximum number of arguments.

FUNC_ARG_TYPES

This is mapped as a set of 16 bit pairs that describe the types of the arguments and the return value. In C this is done with a set of macros that perform the bit mapping.

To declare a value that describes the parameters, use the following format:

#define _ARGTYPE_ (argtype)

where p points to a structure of type FUNC_DCL. The first type is the type of the function's return value, followed by the parameters in order. The example indicates that the function return type is numeric (NV), the first argument is numeric, and the second argument is character (CV).

FUNC_RETURN_LENGTH

This is the length of the return value. Set the field to 0 for a numeric argument, or to a number in the range 1 to 200 for a function that returns a character value.

The initialization procedure is called once and fills in the fields of the FUNC_DCL structure returning 0 as a result, indicating success. The initialization routine is passed, by reference, a pointer to the FUNC_DCL structure.

THE FUNCTION PROCEDURE

The function procedure is called each time the function is invoked in the DATA step. It should be declared as follows:

long myfunc (argsptr, parmptr);
long * nargsptr;
PARMS_PTR *parmptr;
{
  <code>
}

The nargsptr argument is a pointer to the number of parameters passed in the DATA step. PL/I passes this value by reference and C expects parameters to be passed by value; therefore you must dereference it.

The parmptr argument is a pointer to the list of parameters to be passed to the user-written function from the DATA step. These will be either double precision numbers or character arrays. It is treated as an array that indexes each parameter.

PARMS_PTR is the pointer to the PARMS structure. This is the structure that contains the function parameters. It is an array in which each element contains the length and pointer to the data for each parameter. The PARMS structure is declared as follows:

struct PARMS {
  long PARNMER;
  char *PARMSPT;
};
typedef struct PARMS PARMS_PTR;

Each time the function is executed, the SAS System passes the name of the information wanted as a parameter. A binary search is performed on the structure that contains the JPI name and codes. When a match is found, LIB$GETJPI uses the returned code to retrieve the wanted system call. This parameter is passed back to the SAS System. The function procedure must return either zero to signal success or any nonzero value to signal failure. Failure will stop the DATA step.

COMPILING AND LINKING THE FUNCTION

A few simple steps are necessary to link the source code to the SAS System. The image is dynamically loaded and must be linked with SHARE so that it will generate an .address fix-up section. The code must be compiled and linked giving the executable image an extension of .FUN. The executable image must be placed in either SAS$SASLIB or SAS$USERLIB. The following code will compile the included example function, link it, and place the executable file in SAS$USERLIB:

$ CC JPIFNC
$ LINK/SHARE=SAS$USERLIB:JPIFNC.FUN JPIFNC,
  SYS$SHARE:VAXCRTL/SHARE
<ctrl-z>

Executing a user-written function in a SAS DATA step is the same as executing a SAS function in a SAS DATA step. The user-written function is located and loaded automatically by the SAS System. To use the included example, three parameters are passed to the function. The first parameter is your SAS variable name. The second parameter is the quoted character name of the Job Process Information that you want to extract. This has to be in the form that LIB$GETJPI looks for since that is what we are accessing. The third parameter is the process id number for the process you want. For example,

data ....null;
  length account $ 8;
  account=jpfnc("account",20204322x);
  put account ;
run;

causes the account name of process 20204322 to be printed on the SAS log.

CONCLUSION

A SAS user-written function is a tool that makes frequently used computations and processes accessible. It gives you the ability to do exactly what you want, how you want it, when you want it with one simple SAS statement. A SAS user-written function gives you the means to give the SAS System any and all capabilities for your particular environment.

ACKNOWLEDGMENTS

I would like to thank Brad Chisholm, part-time Analyst, Quality Assurance, SAS Institute Inc., Tony Fisher, VMS Host Manager, SAS Institute Inc., and Mike Hecht, Systems Developer, VMS Host Group, SAS Institute Inc., for all their time, patience, and help while this paper was being written.
EXAMPLE CODE

```c
#include "func.h"
#include <ctype.h>
#include <jplib.h>
#include <stsde.h>

/* Macro for setting up descriptor for system call */
#define SDESCRIPTOR(dsc, buf, bin)
do { struct DSC { short len; short typ; char *ptr; } d;
   d = struct DSC [(dsc)];
   d->len = (bin);
   d->typ = DSC$ILTYPLT+(DSC$ILCLASSLs<0);
   d->ptr = (buf);
}while(0 )
#define FAILURE -1

int _func_p;
#include <func.h>

FUNCLPTR *func_p;
int jfunc_p();

/* This fills in the code for func.h */
strncpy (p->FUNC_NAME, "JPIFNC",8);
strncpy (p->FUNC_ALIAS, "JPIFNC",8);
p->FUNC_ADDR=jfunc_p;
p->FUNC_MAX=0;
p->FUNC_MIN=0;
p->FUNC_ARG=0;
p->FUNC_RET=0;
p->FUNC_ARG=0;
p->FUNC_ARG=0;
p->FUNC_ARG=0;
p->FUNC_ARG=0;
p->FUNC_ARG=0;
p->FUNC_ARG=0;
p->FUNC_ARG=0;
p->FUNC_ARG=0;
p->FUNC_ARG=0;
p->FUNC_ARG=0;

int jfunc_p(argc_ptr, parl_lst)
long argc_ptr;
PARAMS PTR *parl_lst;

/* Variable declarations */

PARAMS PTR *parl_lst;
int argc, pid, code, len, lookup();
int itemlen, rc;
struct dsc dsc$result;

/* Accessing the parameters */

PARAMS PTR *parl_lst;
char *s, item[13], *temp;
long argc, pid, code, len, lookup();
int itemlen, rc;
struct dsc dsc$result;

/* Make sure the desired function is in upper case */
for (s=item;*s != ' ' & *s != '"' & *s != '\'); s++
   *s = toupper(*s);
   *s = ' ';

/* Call function to look it up in the table */
code=lookup(item, itemlen);

/* The function was not found, return to SAS */
if (code == FAILURE) return(FAILURE);

/* Initialize descriptor for LIB$GETJPI */
SDESCRIPTOR(dsc$result, (char*)parms[2].PARAMS PTR,8);

/* Make the call to LIB$GETJPI */
ret=lib$getjpi(code,pid,0,dsc$result,item);

if (IVM$STATUS_SUCCESS(ret))
   return FAILURE;

else {
   parms[2].PARAMS PTR="len";
   return(0);
}

/* Table for LIB$GETJPI */
long lookup(s, max)
char *s;
int max;

static struct ITEMTBL{
   char *itmnam;
   short itemcod;
} itmtbl[] = [ [ "ACCOUNT", JPIE.ACCOUNT ],
   [ "APTCNT", JPIE. APTCNT ],
   [ "ASTACT", JPIE.ASTACT ],
   [ "WPAREA", JPIE.UWAREA ],
   [ "WISIZE", JPIE.WISIZE ],
   [ "WSSIZE", JPIE.WSSIZE ] ];

/* Binary search of Table */
int first, last, mid;
short i;

for (;;)
   mid=(first + last) / 2;
   if (strcmp(s, itmtbl[mid].itmnam, max) < 0)
      first=mid;
   else
      last=mid;
   if (last < first)
      return(FAILURE);
   /* did not find it */
   return((long)itmtbl[mid].itemcod);
}

SAS is a registered trademark of SAS Institute Inc., Cary, NC.
VMS is a trademark of Digital Equipment Corp.
```