The MODULE function in SAS® 6.11: An OS/2® Experience

Francis J. Kelley
University Computing & Networking Services
The University of Georgia

ABSTRACT

This paper presents an introduction to the use of the MODULE function. An overview of the function, as well as available documentation is provided. A selection of program examples, including the C code used, module definition files, attribute tables and sample SAS® programs are presented and discussed. Later examples build upon those presented earlier; though the sample code used is trivial, it serves to illustrate some uses of this new function.

INTRODUCTION

With release 6.10 of the SAS System for both OS/2® and Windows™, the user community was presented with some new functions and some extended functionality. The (mainframe) COBOLINT function, previously designed to allow SAS programs to call COBOL subroutines, became the vehicle that would now allow SAS programs to call subroutines written in other languages. Renamed MODULE (and actually a family of functions and call routines), the function was officially regarded as experimental in SAS 6.10; in 6.11, it is a fully-supported production feature. The function may be used both in the DATA step and in PROC IML®. The DATA step functions act on individual observations; the IML functions allow entire arrays to be processed. The current paper will focus upon the DATA step versions of this new addition.

To call a subroutine from SAS running under either OS/2 or Windows, it is necessary that the subroutine be stored in a DLL (Dynamic Link Library). It should be possible to generate such subroutines in any of several languages, and SAS Institute documentation provides appropriate formatting information for COBOL, PL/I, FORTRAN, as well as C. In practice, DLL writing seems to be done mostly in C. In part, this may be explained by the fact that a DLL functions as an extension of the operating system; this being so, there can be issues of recursion and reentrancy that might not be so well handled by some languages (FORTRAN, for example.) If an individual wishes to learn how to write a DLL (see bibliography for sources), they will probably find all instructions done using C (and C++) as the programming language. C (and C++) seem to have emerged as the programming language of choice for most application developers in the desktop environment; still, there is every reason to believe that any of the languages above and probably others should be usable for most, though perhaps not all, DLL writing. Having reviewed languages that might be used instead of C to write a DLL, I will now add that all the routines I used in preparing this paper were written in C. Initially, the EMX/GCC (GNU™ C compiler for OS/2) was used; however, all work was later switched to Borland® C++ for OS/2, Ver 2.0.

REFERENCES AND TERMINOLOGY

All references will be included in the bibliography; however, several should be mentioned at the outset. First, and most important, is the actual SAS Institute documentation on the MODULE function: "Accessing External DLLs from the SAS System." This is available from two sources; as an on-line document within a Display Manager session and by ftp from the SAS Institute ftp server as the document "ts322.ps". An additional document, "ts460.ps", provides Windows-specific information and may be of use to OS/2 users, particularly for the information provided on writing and using DLLs, although the specific code shown would not work in OS/2. The methods of obtaining these documents are provided in the bibliography. Although this documentation was written for SAS 6.10, with the exception of a minor problem noted below, it is correct for version 6.11 as well. In the actual writing of the DLLs tested, two texts proved extremely useful: Charles Petzold's OS/2 Presentation Manager Programming (Ziff-Davis Press) and Real-World Programming for OS/2 2.11 (2nd ed) by Derrel Blain, et al. (SAMS Publishing).

When describing C programming, I shall use the terms "function" and "subroutine" interchangeably to refer to a C function (specifically, a user-written one as opposed to a part of the C function library). When describing a DLL, specifically the contents of the DLL (or "module" as it is often called), I shall use the term "routine." When discussing SAS programming, the term "function" will refer to the SAS function library, including MODULEXy and the related PEEK. As is

common with most documentation, including SAS's, "module" will refer to a DLL, while "MODULE" will refer to the SAS function (or family of functions). Thus, the SAS MODULE "function" is used to call a "routine" contained within a DLL; that routine itself is a compiled C "subroutine" (or function). Though perhaps pedantic, this is done in an attempt to minimize possible confusion resulting from the use of identical terms with different meanings.

WRITING A DYNAMIC LINK LIBRARY

The methods of writing a DLL will be discussed in the written documentation that accompanies your compiler. Online documentation will also be provided (it may be the only documentation provided), and you should review that information. The information provided here applies specifically to the Borland C++ OS/2 compiler, as mentioned above, but should be sufficiently general that it can be used with other C compilers, such as those from IBM® and Watcom™. Known exceptions will be noted. The programming code and the routines produced are not complex; they are intended to familiarize the reader with the use of this new function. A SAS user unfamiliar with the C programming language should have a C reference at hand; there are many available, at least one specifically tailored to OS/2, and the names of several others are provided in the bibliography. A paper of this sort cannot possibly teach either C or Presentation Manager programming; the references provided should be able to assist in doing both. Nevertheless, some working code can't hurt.

A Cautionary Note

Prominent in the documentation on the MODULE function is this:

Caution: Only experienced programmers should access external DLLs.

> By accessing a function in an external DLL, you transfer processing control to an external function. If done improperly, or if the external function is not reliable, you might lose data or have to reset your computer (or both).

You may be assured this can happen!

Getting Started

In writing a DLL you will typically need to write three files: a file to contain your code, another to contain your function prototypes, and a third to define your module. You will also probably want to write a fourth to serve as your Make file. Let us suppose you wish to write a program to add two integers:

test1.c

```
#include <os2.h>
#include <os2def.h>
#include "test1.h"

long APIENTRY adder(long i, long j)
   {
      return i + j;
   }
```

This trivial piece of code simply takes two integers (i and j) and returns their sum. Here is the header file for this function; it consists simply of the ADDER function prototype:

test1.h

```
long APIENTRY adder(long i, long j);
```

This prototype describes the input (two long integers) and the output (the Adder function returns a long integer). Here is what the module definition file would look like:

test1.def

```
LIBRARY     TEST1       INITINSTANCE

DESCRIPTION   'Just an example'
PROTMODE
DATA        NONSHARED

EXPORTS     adder
```

Before going much further, it would be best to examine what has been written, starting with the preprocessor directives in test1.c. You should examine the two libraries brought in with #include. In particular, os2def.h provides the compiler with the information it needs (at least a good portion of it) to write an "exportable" routine (a routine that is callable from another DLL or an executable module is said to be "exportable"). Both os2.h and os2def.h are essential to this program, but it is in os2def.h that APIENTRY is defined. APIENTRY may be described in several ways, but for our purposes let us just say that it describes the function as exportable. An alternative to APIENTRY is EXPENTRY, and the two will often show up in examples with no explanation of the difference.[1] You do not need to use APIENTRY (or EXPENTRY) to define your function as exportable; other keywords

(such as _Export) will do the same, but they can introduce other complexities that must be resolved. Users of EMX/GCC must resolve them: the emx/gcc compiler does not include os2def.h - in its place is os2emx.h, which specifically removes any special values associated with APIENTRY or EXPENTRY (or FAR)[2]. This is not insurmountable, and many DLLs have been written using EMX/GCC, but the author lacked the time to investigate this. The header file (function prototype) is self-explanatory. The module-definition file is less so, although we see that it EXPORTS the adder routine. For our purposes that is sufficient; for more information on the terms used, consult your compiler documentation, the books by Petzold or Blaine, or other appropriate documentation.

THE MAKEFILE

Using a Makefile to produce the DLL is highly recommended; Petzold (pp804-805) provided some useful CMD files to set up the environment variables used by MAKE in both an IBM and a Borland environment. Neither CMD file is reproduced here; use of the appropriate macros within the file are sufficient.

test1.mak

```
BORCC=bcc -c -Id:\bcos2\include -sd -o
BORINCL=d:\bcos2\lib d:\bcos2\lib\c02d.obj
BORLINK=tlink -c -x -v -L$(BORINCL)
BORLIB=c2 + os2

test1.dll: test1.obj  test1.def
  $(BORLINK) test1, test1.dll, NUL, $(BORLIB), \
   test1.def
test1.obj: test1.c  test1.h
  $(BORCC)test1.obj  test1.c
```

The macro variables may be assigned with set statements in a CMD file. In practice, I found that method to be preferable; it reduced clutter in makefiles and CONFIG.SYS. The actual method of compiling and linking your DLL will depend upon the compiler used; you should carefully review available documentation, paying close attention to switches used and libraries accessed. The Makefile above will produce our TEST1.DLL, containing the adder routine.

USING THE DLL ROUTINE

Assume we have written, compiled, and linked our DLL routine adder. We have it stored in TEST1.DLL and this, in turn, is in a directory that has been named in the path/libpath statement of our CONFIG.SYS. To use this in a SAS program, we must provide SAS with an interface to our routine, the "Attribute Table." This table has two parts; the first describes the routine we are calling, the second provides information on the individual arguments being passed.

Three parts are required for the ROUTINE portion of the table: the routine name itself (or ordinal), the maximum, and minimum number of arguments in the call (MAXARG, MINARG). These may optionally be included: the module the routine is in, the "calling sequence" of the routine call, the data type of the returned value, as well as several other sometimes-necessary things. In some instances defaults are provided in other cases the information may be provided elsewhere, for example, the module may be specified within the SAS program as part of the call. The Argument statement consists of the word ARG followed by the "argument number" followed by the type of the argument (NUM or CHAR). Optionally, you may specify whether the argument is: Input, Output, or Update; how the argument is being passed (related to the "Calling Sequence" mentioned above), the presence of a structure, and (very important) the FORMAT of the argument. The FORMAT specification can be very important because C is a strongly typed language with a full set of data types: signed and unsigned, long and short integers, floating and double (precision) floating numbers, characters, and pointers. SAS has two data types, numeric and character. All SAS numbers are used (and by default stored) internally as double-precision floating point numbers, so the integer 2 is stored as though it were the double 2.00. In a SAS program, misusing a floating point number as an integer can yield perplexing results; in a C program, it will produce an error. In my own work, I found careful attention had to be paid to the Attribute Table. Here is what the table for our adder routine would look like:

test1.txt

```
routine    adder
   minarg=2
   maxarg=2
   callseq=byvalue
   module=test1
   returns=long    ;
arg  1  num    input     format=ib4.  ;
arg  2  num    input     format=ib4.  ;
```

Recall that adder is a routine in TEST1.DLL ; note that the routine is identified (as required), as is the module it is in. By default, CALLSEQ is BYADDR (by address), however, it is conventional in C functions to pass arguments by value. For those unfamiliar with this, it simply means that a copy of the values is

passed to the function; the function may not alter the actual values, only the copy it has. A Fortran programmer might find such calls all but useless. How does one invert a matrix using this method? The answer is that for something that requires the data be altered, C does not pass a copy of the value, instead it passes the location of the value - that is, the value's address. This address is handled by a pointer. In the case of a matrix - or array - it would be an array of pointers.

Returning to our Attribute Table, we see that both MAXARG and MINARG are the same; both equal 2, it is not necessary they be the same but this will generally be the case. We also see the routine RETURNS a "LONG" - this is a long integer; it will be a signed 32 bit (= 4 bytes) number (corresponding to the Fortran INTEGER*4). Several other data types may also be returned; note this corresponds to the type declared by our function: long APIENTRY adder(long i, long j);

Our ARG statements are identical, except for their numbering; both ARG 1 and ARG 2 are numeric and used as INPUT to the routine. The FORMAT specified (IB4.) corresponds to the data types specified for the variables I and J as shown above. It is used to translate the floating point valuea that SAS uses into the integer types that our C program expects.

USING OUR ROUTINE

Here is a sample program that uses the adder routine:

test1.sas

```
filename  sascbtbl  'd:\sugi21\test1.txt' ;

data _null_ ;
 do i1 = 1 to 3 ;
   do i2 = 4 to 6 ;
      sum = modulen("e','adder',i1,i2) ;
       put i1= i2= sum= ;
   end;
 end;
run;
```

The LOG will have something like this:

```
I1=1  I2=4  SUM=5
I1=1  I2=5  SUM=6
  ... and so on ...
```

The Fileref used, SASCBTBL, is a required name used to identify the attribute table (supposedly, this name is related to the original function name and refers to the CoBol attribute TaBLe).

THE MODULExy FAMILY

It has been mentioned that MODULE is actually a family of functions; the specific name used depends upon both what is being returned and whether it is being used within a DATA step or within PROC IML. The IML versions may pass entire arrays and will have an "I" in their name.

Here are the function names:

```
module      call module(argument-list);
modulec     ch = modulec(argument-list);
modulen     no = modulen(argument-list);

modulei     call modulei(argument-list);
moduleic    ch = moduleic(argument-list);
modulein    no = modulein(argument-list);
```

The data type (if any) returned by the underlying C function determines the MODULE function used. You will recall that in the TEST1.C example above the function was defined as "long" and consisted of the statement "return i + j;". Clearly, this was a numeric (integer) type and our Attribute Table specified "returns=long." In the SAS program, the call was made thusly:

```
sum = modulen('adder',i1,i2);
```

The documentation has several warnings about using the appropriate version of the function; only those routines which do not return a value (void) should use the CALL MODULE form. If your routine returns a number you should use MODULEN. If your routine returns a character you should use MODULEC.

The description of the "argument-list" used in the call has been postponed until now, partly because sections of it needed more explanation. The argument-list consists of three parts: an optional control string, the module name/routine name, and the actual list of arguments used. The control string is a quoted string containing an asterisk(*) and one or more special characters.

The only control string I used for most of my testing was "I'. This prints somewhat detailed hexadecimal representations of the arguments to and from the called routine. An alternative is "E', which prints error messages (without either E or I, there are no error messages of real use).

You should know that '*l' can produce quite a good deal of output. Following the control string is the module-name/routine name. If the module (the DLL) is named in the Attribute table, it may be omitted but the Routine (or its ordinal) must be specified. Some routines may be specified only by their ordinal, and the documentation provides two examples using routines in the DOSCALLS.DLL.[3] In the case of TEST1.SAS, our call could have been made as sum = modulen('TEST1,adder', ...) or sum = modulen('adder',...). *A note*: DLLs are not case-sensitive but routine names are. The argument list has already been discussed.

With this, we can examine some additional programs.

mydll1.c

```
/* Code for 3 routines within a single DLL   */

#include <os2.h>
#include <os2def.h>
#include <stdarg.h>
#include "mydll1.h"

/* add two integers   */
long      APIENTRY  adder(long x, long y)
     {
        return  x + y ;
      }

/* multiply two floats   */
double   APIENTRY  tmul(double x, double y)
     {
        double  ans;
        ans = x * y ;
        return ans ;
      }

/*  swap the values of two integers */
void      APIENTRY  tswap(int *x, int *y)

      {
         int temploc ;

         temploc = *x;
         *x = *y ;
         *y = temploc;
       }
```

The header file for this (mydll1.h) will simply consist of the function prototypes (declarations) and is not included here.

Here is the module definition file:

mydll1.def

```
LIBRARY       MYDLL1      INITINSTANCE

DESCRIPTION  'Three sample routines'
PROTMODE
DATA          NONSHARED

EXPORTS       adder
              tmul
              tswap
```

This example expands upon the TEST1 program earlier but not in any dramatic way. The TMUL function will return a real number (type is double) and TSWAP returns no value (type is void), but does use pointers.

Here is what the attribute table would look like:

mydll1.txt

```
routine        adder
 minarg=2   maxarg=2
 callseq=byvalue
 module=mydll1
 returns=long ;
arg 1 num   input     format=ib4. ;
arg 2 num   input     format=ib4. ;

routine        tmul
 minarg=2   maxarg=2
 module=mydll1
 returns=double ;

arg 1  num input  byvalue  format=rb8. ;
arg 2  num input  byvalue  format=rb8. ;

routine        tswap
 minarg=2  maxarg=2
 callseq=byaddr
 module=mydll1 ;

arg 1  num  update  format=ib4. ;
arg 2  num  update  format=ib4. ;
```

We have already seen the Adder routine. Tmul simply multiplies two doubles and returns a double.

Note: there is an error in the current documentation; using the format RB8. for a double works just as the "Formats" section of the documentation specifies. Often in C functions, floating-point numbers are assigned to the type "real". The documentation

incorrectly defines this as being the SAS Format RB4. Use FLOAT4. when your argument list includes real-type variables. In practice, I found I used "double" almost all the time - at least partly because that is how SAS uses numbers.

"BYVALUE" was specified on the ARG statements more as an example. Tswap returns void, so should be used as a CALL; additionally, it uses pointers. In the sample C program run to test this function, the call was made: "tswap(&a,&b)," where both a and b had been defined as int. The "&" address operator allows this call to pass the addresses of a and b. The arguments are used for both input and output so "update" is appropriate. The routine itself merely swaps the values of two integers and is really a small introduction to one way pointers, and may be used in this function. Here is a SAS program that will use these routines:

mydll1.sas

```
filename  sascbtbl  'd:\sugi21\mydll1.txt' ;
data test;
 do i = 1 to 4 ;  * generate some numbers;
    j = i - 3;      * - and some more      ;
     i1 = i; j1 = j; * copy original values   ;
     r1 = i1 + ranuni(0); * make them look;
     r2 = j1 + ranuni(0); * like reals       ;
     sum = modulen('e','adder',i1,j1);
     prod = modulen('e','tmul',r1,r2);
     call module('e','tswap',i1,j1);
     output;
 end;
proc print;
run;
```

This little program will add the integers I1 and J1, multiply the real valued R1 and R2, then swap the values stored for I1 and J1 (the original values of I and J are preserved). The Tswap routine is of some interest because it uses (dereferenced) pointers to perform the swap. In the output from PROC PRINT we will find:

for I=1 and J=-2;

I1	J1	R1	R2	SUM	Prod
-2	1	1.35743	-1.73668	-1	-2.35741

and for I=2 and J=-1;

I1	J1	R1	R2	SUM	Prod
-1	-2	2.59601	-0.45564	1	-1.18284

and so on.

Now let's write another DLL:

mydll2.c

```
#include <os2.h>
#include <os2def.h>
#include <stdarg.h>
#include <string.h>
#include "mydll1.h"
#include "mydll2.h"

/*    subtract two integers        */

long        APIENTRY   suber(long x, long y)
      {
          return  x - y ;
      }

/*    modify two integers          */

long         APIENTRY   tmod(long x, long y)
      {
           long   ans ;

           ans = adder(x,y) * suber(x,y) ;

           return ans ;
      }

/*    multiply two doubles -- pointer example    */

double*      APIENTRY    pmul(double x, double y,
double *z)
      {
           static double   ans;
           double   *p_ans ;

           p_ans = &ans ;

           ans = x * y ;

           *z=ans;

           return  p_ans ;
      }
/*    swap two character values        */

void      APIENTRY   cswap(char *a, char *b)
      {
          char  temploc[200] ;
          char  *p_temp ;

          p_temp = temploc ;

          (Void)strcpy(p_temp,a) ;
          (Void)strcpy(a,b) ;
           (Void)strcpy(b,p_temp) ;
      }
```

As before, the header file (mydll2.h) contains only the function prototypes and is not included here. Note that the header file for MYDLL1 is also included; this is because the Tmod routine will call another routine (Adder) in our previous DLL, and the function declaration is needed. In fact, Suber - the reverse of Adder in our previous example - is added to allow Tmod to call both. This requires a bit more as we shall see, but not in our program file. The next routine is Pmul. This is similar to the Tmul of MYDLL1, but returns a pointer to the result instead of the result itself; a (dereferenced) pointer is also included as a return value in the argument list. Finally, to complement the Tswap of MYDLL1, there is Cswap. This routine swaps the values of character variables. Once again, pointers must be used. I included string.h to allow use of the strcpy function.

Here is the module definition file; note the new instruction:

mydll2.def

```
LIBRARY        MYDLL2        INITINSTANCE

DESCRIPTION  ' four more routines
PROTMODE
DATA           NONSHARED

EXPORTS        suber
               tmod
               pmul
               cswap
IMPORTS        MYDLL1.adder
```

This time, because a routine is called (imported), both it and the module it is in are specified in the IMPORTS section. Now let's look at the Attribute table:

mydll2.txt

```
routine        suber
 maxarg=2    minarg=2
 callseq=byvalue
 module=mydll2
 returns=long  ;
arg 1     input      format=ib4. ;
arg 2     input      format=ib4. ;

routine        tmod
 maxarg=2    minarg=2
 callseq=byvalue
 module=mydll2
 returns=long   ;
arg 1   input      format=ib4.  ;
arg 2   input      format=ib4.  ;

routine        pmul
 maxarg=3    minarg=3
 module=mydll2
 returns=dbiptr  ;
arg 1 num  input   byvalue  format=rb8. ;
arg 2 num  input   byvalue  format=rb8. ;
arg 3 num  update  byaddr   format=rb8. ;

routine        cswap
 maxarg=2    minarg=2
 module=mydll2
 callseq=byaddr   ;
arg 1       update      format=$cstr199.  ;
arg 2       update      format=$cstr199.  ;
```

Suber and Tmod are much as we've seen before; Pmul and Cswap are a bit different. Both of these routines made use of pointers; each writes its results directly back to the location referenced by the pointer (BYADDR is used in the call). Note that the formats used (RB8. and $CSTR199.) are not the data types SAS would use for a pointer (SAS has no pointer type, you should specify PIB4. when a pointer is passed - see the SAS documentation for examples using the PEEK function to view the contents being pointed to). In this case, the actual values will be displayed upon completion of the call. Here is a small SAS program making use of these calls:

mydll2.sas

```
filename sascbtbl  'd:\sugi21\mydll2.txt' ;

data test2 ;
 length  ch1 ch2 xch1 xch2  $ 30 ;
 input  i   j   ch1 $   ch2 $ ;
 xch1 = ch1;  xch2 = ch2  ;
 ri = i + ranuni(0) ;
 rj = j + ranuni(0) ;
 sub  = modulen('suber',i,j);
 tst  = modulen('tmod',i,j) ;
 prod = modulen('pmul',r1,r2,rp) ;
 call module('cswap',ch1,ch2) ;
cards;
1  2  this  is
3  4  a  test
5  6  akindalongword  shortword
7  8  tiny  realyrealyrealylongword
;
proc print; run;
```

With the input values I=1, J=2, CH1=this, and CH2=is the following was produced:

RI	RJ	SUB	TST	PROD	RP
1.73173	2.37617	-1	-3	4.1149	4.1149

CH1	CH2
is	this

CONCLUSION

The MODULE function provides SAS users with considerable capacity. In addition to user-written functions, system DLLs may also be accessed. The rules for using this are somewhat rigid, but not unreasonable, considering what is being done. All the routines were tested using C driver programs, and that is strongly recommended wherever feasible. The Attribute Table can handle some error checking regarding the data types passed, but error recovery from a bad routine was not extensively tested at this writing. It seems likely, given various warnings (and experiences) that this ability is minimal at best. Nevertheless, those who need custom-tailored routines have the means of designing them. This is a very powerful feature.

ACKNOWLEDGEMENTS

Thanks to Mark Plaskin and Andrew Walker for reviewing some of the C code used; to Larry Salomon who provided answers when I couldn't find any; and to the many contributors to comp.os.os2.programmer.misc, who open doors just by asking the right questions (and providing answers). Special thanks to Jeannie McElhannon who got this disorganized manuscript in order and to Rick Langston and Lindi Ingold of SAS Institute.

ENDNOTES

1. This is because there is no difference. Prior to the release of OS/2 2.0, there was a difference in the entry point definitions, but OS/2 2.0 introduced the 32-bit flat memory model as opposed to the segmented model of previous versions (and DOS). You may use whichever you prefer; both are defined in os2def.h.

2. An additional note about both os2def.h and os2emx.h: most OS/2 (and Windows) documentation, particularly C programming documentation/examples, will use what may seem at first to be new data types (ULONG, for instance.) These are nothing more than "typedefs" defined in the header files (ULONG mentioned above is just an alias for "unsigned long integer"). Though, for simplicity's sake, I did not use these forms in my examples, such usage is the norm in DLL writing and I would recommend it. There are a number of other such "types."

3. The DOS CALLS.DLL will not be found in OS/2 2.x and above; nevertheless calls made to it still work.

Francis J. Kelley
UCNS Client Services
Computer Services Annex
Athens, GA 30602-1911
jkelley@uga.cc.uga.edu
706-542-5359

REFERENCES

SAS Documentation/MODULE references

SAS Institute, "Accessing External DLLs from the SAS System" SAS Technical Report TS322. This is available from the SAS Institute ftp server (ftp.sas.com) as ts322.ps. It is in techsup/download/technote. It is also available as part of on-line documentation found in the Help pull-down menu of a Display Manager session.

SAS Institute "Accessing External DLLs with SAS 6.1x for Win32s" SAS Technical Report TS460. Available on the SAS ftp server as ts460.ps. See above for directory information. Very interesting information on using system DLL routines.

Langston, Richard D. "Examples Using the MODULE Routine in PC Environments," Proceedings of the Twentieth Annual SAS Users Group International Conference. Cary, NC: SAS Institute, Inc, 1995, pp. 1389-93.

Langston, Richard D. "A Comparison of the MODULE routines with S-PLUS Extensibility," Proceedings of the Twentieth Annual SAS Users Group International Conference. Cary, NC: SAS Institute, Inc, 1995, pp. 1402-1405.

C Programming:

Harbison, Samuel J. and Guy L. Steele. C: A Reference Manual. 4th ed., Englewood Cliffs, NJ: Prentice-Hall, 1995.

Kelley, Al, and Ira Pohl. C by Dissection: the Essentials of C Programming. Redwood City, CA: Benjamin/Cummings Publishing, 1992.

Oualline, Steve. Practical C Programming. Sebastopol, CA: O'Reilly & Associates, Inc, 1993.

Press, William H, et. al. Numerical Recipes in C. New York: Cambridge University Press, 1988. New editions of both this book and the original Fortran version are now available.

Schildt, Herbert. ANSI C Made Easy. Berkeley: McGraw-Hill, 1989.

OS/2 Presentation Manager programming and related:

Blaine, Derrel R., Kurt R. Delimon, and William Jeffrey English. Real-World Programming for OS/2 2.11. 2nd ed. Indianapolis: SAMS Publishing, 1994.

Panov, Kathleen, Larry Salomon, and Arthur Panov. The Art of OS/2 Warp Programming. New York: John Wiley & Sons, 1995.

Petzold, Charles. OS/2 Presentation Manager Programming. Emeryville, CA: Ziff-Davis Press, 1994. This is a very handy book to have.

Stock, Mark. OS/2 Warp Control Program API. New York: John Wiley & Sons, 1995. This is part of a set, and provides information on CP system calls.