Developing IBM® ISPF, DB2™ and SAA Applications with the SAS/C® Compiler

Keefe Hayes, SAS Institute Inc., Cary, NC

ABSTRACT

The SAS/C® compiler provides a productive programming environment for the development of efficient C programs. Together with the IBM® products ISPF and DB2™ the application developer has a powerful combination with which to deliver interactive user dialogs and SQL™ database applications. For the future, SAS/C and Systems Application Architecture™ (SAA) expand the potential of these products by enhancing the portability of applications developed with them.

INTRODUCTION

The following discussion includes:

- an overview of SAS/C language extensions useful for implementing ISPF and DB2 applications
- a description of building blocks for communicating between C programs and ISPF and DB2
- several coding examples
- solutions to some potential problems.

The examples in the paper assume the following software levels: Release 4.00 of the SAS/C compiler, Version 2.3 of ISPF, and Version 1.3 of DB2. The discussion emphasizes SAS/C extensions necessary for using these versions of ISPF and DB2, which do not support the C language. Note that Version 2 of DB2 and Version 3 of ISPF do support C.

SAS/C FEATURES AND LANGUAGE EXTENSIONS

The SAS/C compiler provides several features and extensions that aid in calling ISPF and DB2 from the C environment, including support for

- calling assembler routines from C
- loading modules dynamically
- calling programs in other high level languages from C
- referencing C structures in an assembler routine
- converting assembler structures (DSECTs) to C structures.

Calling assembler routines from C, such as those for ISPF and DB2, involves declaring them as assembler and properly formatting their parameter lists. To declare an assembler function, use the `__asm` keyword. Then, to format its parameter list, use the @ call-by-reference operator to pass argument pointers rather than values. This operator works similarly to the & operator, while providing additional support for passing constants and computed expressions. Of course, when passing an array, C converts its reference to a pointer, so in general do not precede string arguments to assembler programs with the @ operator.

Dynamic loading, using the SAS/C loadm function, insulates your program from future changes to the ISPF and DB2 service routines. This function loads the requested module and initializes the corresponding function pointer. However, declarations using the `__asm` keyword generate a special one word function pointer that is incompatible with `loadm`, but you can initialize a standard C function pointer and then assign its value to the `__asm` pointer.

Calling programs in other high level languages from C is another useful SAS/C extension, for Version 1 of DB2 does not support C as a host language. However, the examples in this paper focus on using assembler as the host language. The advantages of using assembler are that the communication of structures between the C main procedure and assembler routines is straightforward and that a SAS/C tool is available for translating assembler structures, known as DSECTs, to C structures.

Referencing C structures in an assembler routine involves locating and addressing the structures. If you use the `extern` storage class to declare structures of fixed size, such as those for the SQL Communication Area (SQLCA) and for DB2 host variables, an assembler routine can access the structures through V-type address constants. If your program must be reentrant, C environment initialization can allocate the structures in the Pseudo-Register Vector (PRV) if you compile the program with the `RENTExt` option. In that case, the routine can access them through Q-type address constants. If you use the malloc function to allocate areas of arbitrary size, such as the SQL Descriptor Area (SQLDA) and dynamic SQL statements, then the assembler routine can access them through pointers passed in parameters from the C program.

To convert an assembler DSECT to a C structure, use the SAS/C DSECT2C program. For example, to convert the SQLCA and SQLDA, preprocess and assemble the following short DB2 program:

```
         CSECT
         EXEC  SQL  INCLUDE SQLDA
SQLCA    DSECT
         EXEC  SQL  INCLUDE SQLCA
         END
```

Direct the assembler listing to DSECT2C, requesting that it process the SQLCA, the SQLDA, and the SQLDA sub-structure SQLVARN. Then, enhance the DSECT2C output to parallel the C structure definitions found in the DB2 Version 2 Application Programming Guide. The edited results for the SQLCA look like this:

```
#if !defined(_CL8)
#define _CL8
typedef char CL8[8];
#endif

#if !defined(_CL70)
#define _CL70
typedef char CL70[70];
#endif

typedef struct
  {
  CL8    sqlcaid;
  int    sqlcabc;
  int    sqlcode;
  short  sqlerrml;
  CL70   sqlerrm;
  CL8    sqlerrp;
  int    sqlerrd[6];
  CL8    sqlwarn;
  CL8    sqlext;
  } SQLCA;
```

```
#if !defined(SQLCACSE)
#define SQLCACSE sqlca. /*component selection expression*/
#endif

#define SQLCODE                         SQLCACSE sqlcode
#define SQLWARN0                        SQLCACSE sqlwarn[0]
#define SQLWARN1                        SQLCACSE sqlwarn[1]
#define SQLWARN2                        SQLCACSE sqlwarn[2]
#define SQLWARN3                        SQLCACSE sqlwarn[3]
#define SQLWARN4                        SQLCACSE sqlwarn[4]
#define SQLWARN5                        SQLCACSE sqlwarn[5]
#define SQLWARN6                        SQLCACSE sqlwarn[6]
#define SQLWARN7                        SQLCACSE sqlwarn[7]
```

The results for the SQLDA look like this:

```
#if !defined(_CL8)
#define _CL8
typedef char CL8[8];
#endif

#if !defined(_CL30)
#define _CL30
typedef char CL30[30];
#endif

typedef struct
  {
  CL8    sqldaid;
  int    sqldabc;
  short  sqln;
  short  sqld;
  struct SQLVAR
    {
    short  sqltype;
    short  sqllen;
    char   *sqldata;
    short  *sqlind;
    struct { short length; CL30 data; } sqlname;
    } sqlvar[0];
  } SQLDA;

#define SQLDASIZE(n)  \
   (sizeof(SQLDA) + (n)*sizeof(struct SQLVAR))
```

BASIC BUILDING BLOCKS

This section develops SAS/C and assembler building blocks for

- mapping C program variables into the ISPF function pool
- providing a VDEFINE exit
- requesting service of DB2
- supporting WHENEVER statements in your C program
- dynamically loading the ISPF and DB2 service routines.

Mapping C program variables into the ISPF function pool uses the ISPF VDEFINE service. **Table 1** describes the correspondence of DB2 and C data types to ISPF formats. Note that for Version 2 of DB2, support for embedding SQL statements in C requires that date, time, and timestamp fields end in a null terminator. This is also true for character strings, for which DB2 defines a new type.

DB2 data type		host data type		ISPF format	
code	name	non-C	C	non-C	C
384/385	DATE	char[10]	char[11]	CHAR	USER
388/389	TIME	char[8]	char[9]	CHAR	USER
392/393	TIMESTAMP	char[26]	char[27]	CHAR	USER
448/449	VARCHAR	struct { short; char[n]; }		USER	USER
452/453	CHAR	char[n]	char	CHAR	CHAR
460/461	CHAR	-	char[n+1]		USER
480/481	FLOAT	double	double	USER	USER
484/485	DECIMAL	packed	-	USER	USER
496/497	INTEGER	long	long	FIXED	FIXED
500/501	SMALLINT	short	short	FIXED	FIXED

Table 1 Correspondence of DB2 and host data types to ISPF formats

The decimal data type is a special case, for C does not support it. However, the SAS/C Interlanguage Communication Feature defines two macros, pdval and pdset, for converting between decimal and floating point data. Alternatively, you can use the in-line machine code interface to perform operations directly on decimal data fields.

From the table, some useful examples of #defines and typedefs relating the various DB2 data types to C include:

```
#define TYPECODE(code) ( type == code || type == code+1 )
#define type_DATE       TYPECODE(384)
#define type_TIME       TYPECODE(388)
#define type_TIMESTAMP  TYPECODE(392)
#define type_VARCHAR    TYPECODE(448)
#define type_CHAR       TYPECODE(452)
#define type_NULCHAR    TYPECODE(460)
#define type_FLOAT      TYPECODE(480)
#define type_DECIMAL    TYPECODE(484)
#define type_INTEGER    TYPECODE(496)
#define type_SMALLINT   TYPECODE(500)

#define ispf_VARCHAR  2  /* length field overhead    */
#define ispf_NULCHAR  1  /* null terminator overhead */
#define ispf_FLOAT   24  /* ispf size for float      */
#define ispf_INTEGER 11  /* ispf size for long       */
#define ispf_SMALLINT 5  /* ispf size for unsigned short */

typedef short NULLIND, TYPE;
typedef char DATE[10], TIME[8], TIMESTAMP[26];
typedef __noalignmem struct \
   { short len; char text[0]; } VARCHAR;
typedef double FLOAT;
typedef long INTEGER;
typedef unsigned short SMALLINT;
```

Providing a VDEFINE exit is necessary for data types with the ISPF format of "USER" in the table. In general, ISPF accesses service exits, such as that for VDEFINE, using an address you pass in a parameter when calling the service. By compiling the source module with the `INDep` option, that address can point directly to the exit routine. The `INDep` option ensures that when ISPF calls the exit, the C environment is available.

However, you cannot use the `INDep` option if your program must be reentrant. In that case, another mechanism is necessary to restore the C environment prior to entering the exit. In the example below, an assembler bridge provides this mechanism by loading the C Runtime Anchor Block (CRAB) pointer from the data address parameter passed in the initial call to the ISPF service.

```
         ENTRY VDFEXITB
VDFEXITB DS    0F
         B     BRIDGE-*(,R15)
         DC    F'0'            data address parm offset
         DC    V(VDFEXIT)      C subroutine exit address
BRIDGE   DS    0H
         SAVE  (12)            save register 12
         L     R12,4(,R15)     load data address offset
         L     R12,0(R12,R1)   load data address
         L     R12,0(,R12)     load CRAB address
         ST    R14,CRABUSR1    save return address
         L     R15,8(,R15)     load exit routine address
         BALR  R14,R15         branch to exit routine
```

```
         L      R14,CRABUSR1      restore register 14
         RETURN (12)              restore reg 12 and return
```

The C mainline then initializes the VDEFINE exit data structure to use the bridge

```
#include <code.h>                  /* for _stregs() */
extern __asm int vdfexitb();
struct { __asm int (*vdfexitb)(); void *crab; } vdfexitd;
  .
  .
  .
vdfexitd.vdfexitb = vdfexitb;
vdfexitd.crab = (void *)_stregs(R12);
```

Requesting service of DB2 through SQL statements contrasts with ISPF's function call mechanism. The example below illustrates how to embed SQL in an assembler program. First, a header identifies the program, associates base registers with the control and data structures, and declares those structures. Then, for each SQL statement, entry code links it to the C caller and loads the base registers necessary to provide addressability to the appropriate structures. After the statement, control returns to C.

```
EXAMPLE  CSECT                      Identify the program section
         SQLSECT ,                  Save the section identifier
         CREGS USING                Associate base regs w/ C
         USING SQLSTMTD,R5           and DB2 control structures
         USING SQLDSECT,R3
         USING SQLCA,R2
         COPY  DSA                  Include C control structures
         COPY  CRAB
SQLCA    DSECT                      Definitions for SQLCA
SQLDA    DSECT                       and SQLDA in C mainline
SQLSTMTD DSECT                      Dynamic sql statement
SQLSTMT  DS    HL2,CL1
         SQLSECT RESTORE            Resume program section
    .
    .
    .
SQLPREP  CENTRY LASTREG=R6,BASE=R6  Enter assembler routine

         LM    R2,R3,=Q(SQLCA,SQLDSECT)  PRV offsets
         AL    R2,CRABPRV           Address the SQLCA
         AL    R3,CRABPRV           Address the SQLDSECT
         L     R4,0(,R1)            Address the SQLDA
         USING SQLDA,R4             Associate base reg w/ SQLDA
         L     R5,4(,R1)            Address the SQL statement

         EXEC  SQL PREPARE SQLSTMT INTO SQLDA FROM SQLSTMT

         CEXIT LASTREG=R6,RC=(R15)  Return to caller
```

To call the above assembler routine to execute the SQL PREPARE, issue the following statement:

```
sqlprep ( sqlda, sqlstmt );
```

To support WHENEVER statements in your C program, declare these data fields and `#define` these SQL keywords:

```
#include <setjmp.h>
jmp_buf env[3];
char sqlcond[3] = { 0, 0, 0 };
int  whentype;

#define EXEC
#define SQL
#define WHENEVER   whentype =
#define FOUND
#define TO
#define GO         goto
#define GOTO       goto
#define SQLERROR   0; sqlcond[0] = 1;                    \
                      if ( setjmp(env[0]) < 0 )
#define NOT        1; sqlcond[1] = 1;                    \
                      if ( setjmp(env[1]) == 100 )
#define SQLWARNING 2; sqlcond[2] = 1;                    \
                      if ( setjmp(env[2]) > 0 || \
                         SQLWARN0 == 'W' )
#define CONTINUE    ; sqlcond[whentype] = 0;
```

Then, write a macro that calls an SQL routine and tests the sqlcode.

```
#define EXECSQL(sqlfunc) ( sqlfunc,  \
  ((sqlcond[0] && SQLCODE < 0) ||    \
   (sqlcond[1] && SQLCODE == 100) || \
   (sqlcond[2] && (SQLCODE > 0 || SQLWARN0 == 'W'))) ?    \
    longjmp(env[((sqlcond[0] && SQLCODE < 0)    ? 0 :     \
                 (sqlcond[1] && SQLCODE == 100) ? 1 : 2) \
                ], SQLCODE) : (void)0 )
```

Finally, issue WHENEVER statements in the program to identify the branch-to location for handling a condition.

```
do {
   EXEC SQL WHENEVER SQLERROR GOTO label; break;
   label:
      /* prevent recursion if sql routine here fails */
      EXEC SQL WHENEVER SQLERROR CONTINUE;
      /* put sql error handling code here */
   } while(1);
```

Dynamically loading the ISPF and DB2 service routines provides an alternative to linking them with your program. To perform the loading, declare each routine with a function pointer.

```
#define ISPEXEC(buffer) \
   ( (*ispexec) ( astrlen ( buffer ), buffer ) )
#define ISPLINK  (*isplink)
#define DSNALI   (*dsnali)
#define DSNHLI   (*dsnhli2)
#define DSNTIAR  (*dsntiar)
extern __asm int (*ispexec)();
extern __asm int ISPLINK();
extern __asm int DSNALI();
extern __asm int DSNHLI();
extern __asm int DSNTIAR();
```

Then call `loadm` to initialize the pointers.

```
#include <dynam.h>
int (*fp[5])(); /* one element per loadm'ed module */

    .
    .
    .
loadm ( "ISPEXEC", &fp[0] );
ispexec = **(__asm int (***)())&fp[0];
```

By using the in-line machine code interface, you can even issue the load directly.

```
#include <svc.h>
#define LOAD(name) (_ldregs(R0+R1,name,0), _ossvc(8), \
   (__asm int (*)())_stregs(R0))
#define DELETE(name) (_ldregs(R0,name), _ossvc(9), NULL)

    .
    .
    .
ispexec = LOAD ( "ISPEXEC " );
```

To use the dynamically loaded DB2 service routine involves a little trick. Since the DB2 preprocessor expansion of an executable SQL statement assumes that a routine named DSNHLI is link-edited with the caller, dynamic linking fails. However, by including a bridge with the name DSNHLI, the statement calls it instead. The bridge can then locate the dynamically loaded DSNHLI and branch to it. For example, if the C mainline initializes the (*dsnhli2)() function pointer by loading DSNHLI, you can use the following simple assembler bridge:

```
         ENTRY DSNHLI
DSNHLI   DS    0H
         USING *,R15
         L     R15,=Q(DSNHLI2) load PRV offset of dsnhli2
         AL    R15,CRABPRV     add PRV pointer from CRAB
         L     R15,0(,R15)     load dsnhli pointer from PRV
         BR    R15             branch to dsnhli
```

```
DSNHLI2  DXD   A                  __asm func ptr declared in C
```

Of course, a complex bridge involving the call attachment facility is also possible.

```
#include <code.h>                 /* for _stregs() */
#define NOCONNECT 0x00C10205

int dsnhli ( void *sqlplist )
{
int rc = 0;

if ( dsnali == NULL )
   {
   dsnali  = LOAD ( "DSNALI  " );
   dsnhli2 = LOAD ( "DSNHLI2 " );
   if ( rc = DSNALI ( "OPEN          ","DB2 ","PLAN     " ) )
      {
      sqlplist = NULL;
      DSNALI ( "TRANSLATE    ", &sqlca );
      if ( _stregs(R0) == NOCONNECT ) rc = NOCONNECT;
      }
   }
if ( sqlplist != NULL )
   EXECSQL ( rc = DSNHLI ( sqlplist ) );
else
   {
   if ( rc == 0 ) DSNALI ( "CLOSE        ", "SYNC" );
   dsnhli2 = DELETE ( "DSNHLI2 " );
   dsnali  = DELETE ( "DSNALI  " );
   if ( rc == NOCONNECT )
      {
      printf ( "unrecognized db2 connection failure" );
      exit ( 12 );
      }
   }
return ( rc );
}
```

EXAMPLES

To demonstrate the principles outlined in the preceding sections, the following examples include:

- a VDEFINE exit to illustrate how to use an ISPF exit
- a code fragment to issue a DB2 message
- a set of routines to execute an SQL varying-list select statement and to copy the results to an ISPF table.

A VDEFINE exit depends on several pieces of information: where the program data field is, what data type it is, whether the data is null, and where to position ISPF output. One technique for making this information readily accessible is to preformat it into an area near the data field. For example, by using this area layout

```
struct { short nullind, db2type;
         char cdata[DATASIZE], ispfdata[ISPFSIZE]; };
```

the exit routine listed below identifies the data type and, if necessary, interprets the null indicator by examining the two fields prior to the C data. For numeric types, the exit formats ISPF output after the data field.

```
int vdfexit ( char *udata, long *srvcode, char *namestr,
   long *deflen, char *defarea, long *spfdlen,
   char **spfdatap )
{
int len, type = *(TYPE *)(defarea-sizeof(TYPE));
char *end, fltarea[ispf_FLOAT+1];

if ( *srvcode )   /* write request - copy from ISPF to C */
   {
   if ( (type&1) && *spfdlen == 0 )
      *(NULLIND *)                    /* set null indicator */
         (defarea - sizeof(TYPE) - sizeof(NULLIND)) = -1;
   else
      {
      if (type&1)
         *(NULLIND*) /* set null indicator */
            (defarea - sizeof(type) - sizeof(nullind)) = 0;
      if ( type_VARCHAR )
         {
         len = *deflen < *spfdlen ? *deflen : *spfdlen;
         memcpy ( ((VARCHAR *)defarea)->text, *spfdatap,
            (short)len );
         ((VARCHAR *)defarea)->len = len;
         }
      else if ( type_NULCHAR )
         {
         len = *deflen < *spfdlen ? *deflen : *spfdlen;
         memcpy ( defarea, *spfdatap, (short)len );
         *(defarea+len) = 0;              /* null terminate */
         }
      else if ( type_FLOAT )
         {
         len = ispf_FLOAT < *spfdlen ? ispf_FLOAT : *spfdlen;
         memcpy ( fltarea, *spfdatap, (char)len );
         *(fltarea+len) = 0;              /* null terminate */
         *(FLOAT *)defarea = strtod ( fltarea, &end );
         }

      }
   }
else                /* read request - copy from C to ISPF  */
   {
   if ( (type&1) &&               /* type supports null  */
      *(NULLIND *)                /* test null indicator */
        (defarea - sizeof(TYPE) - sizeof(NULLIND)) < 0 )
      {
      *spfdlen  = 0;
      }
   else if ( type_VARCHAR )
      {
      *spfdatap = ((VARCHAR *)defarea)->text;
      *spfdlen  = ((VARCHAR *)defarea)->len;
      }
   else if ( type_NULCHAR )
      {
      *spfdatap = defarea;
      *spfdlen  = strlen ( defarea );
      }
   else if ( type_FLOAT )
      {
      *spfdatap = defarea + sizeof(FLOAT);
      sprintf ( fltarea, "%+*.*e", ispf_FLOAT,
         ispf_FLOAT-7, *(FLOAT *)defarea );
      memcpy ( *spfdatap, fltarea, ispf_FLOAT );
      *spfdlen = ispf_FLOAT;
      }
   }
return ( 0 );
}
```

Issuing a DB2 message is useful when handling an SQL error. The example below uses the DSNTIAR routine supplied with DB2 to format an error message. By using VDEFINE to map the message area to an ISPF dynamic area variable, the routine directly formats the area for panel display.

```
struct { long width, depth;
         struct { short len; char text[SCRNSIZE]; } area;
       } screen;

       .
       .
       .

screen.area.len = screen.width * screen.depth;
dsntiar = LOAD ( "DSNTIAR " );
DSNTIAR ( &sqlca, &screen.area, &screen.width );
dsntiar = DELETE ( "DSNTIAR " );
```

Executing an SQL varying-list select statement requires the following four steps:

1. Prepare the SQL statement into an SQLDA.
2. Allocate an area for data fields and null indicators.
3. Assign locations in the area to SQLDA pointers.

4. Map to ISPF each data field and null indicator.

The example below performs each of these steps, then fetches the DB2 rows and copies them to an ISPF table.

```
if ( sqlda = prepare ( sqlstmt ) )
   {
   if ( datap = allocate( sqlda ) )
      {
      assign ( sqlda, datap );
      names = map2spf ( sqlda );
      ISPLINK ( "TBCREATE","TABLE "," ",names,"NOWRITE" );
      sqlopen ( sqlda );                          /* open     */
      while ( SQLCODE == 0 )
         {
         sqlftch ( sqlda );                       /* fetch    */
         if ( SQLCODE == 0 )
            ISPEXEC ( "TBADD TABLE" );
         }
      sqlcmit ( );                                /* commit   */

          .
          .
          .
      ISPEXEC ( "TBEND TABLE" );                  /* clean up */
      ISPLINK ( "VDELETE", names );
      free ( names );
      free ( datap );
      }
   free ( sqlda );
   }

static SQLDA *prepare ( VARCHAR *sqlstmt )
{
SQLDA *sqlda, sqlda0;

sqlda0.sqldabc = SQLDASIZE ( sqlda0.sqln = 0 );
sqlprep ( &sqlda0, sqlstmt );

sqlda = malloc ( SQLDASIZE ( sqlda0.sqld ) );
memcpy ( sqlda->sqldaid, "SQLDA   ", 8 );

sqlda->sqldabc = SQLDASIZE ( sqlda->sqln = sqlda0.sqld );
sqlprep ( sqlda, sqlstmt );

return ( sqlda );
}

static char *allocate ( SQLDA *sqlda )
{
int i, type, datal = 0;

for ( i = 0; i < sqlda->sqld; ++i )
   {
   type = sqlda->sqlvar[i].sqltype;
   datal += sqlda->sqlvar[i].sqllen +
            (type&1) * sizeof(NULLIND);

   if      ( type_VARCHAR )
      datal += sizeof(TYPE) + ispf_VARCHAR;
   else if ( type_NULCHAR )
      datal += sizeof(TYPE) + ispf_NULCHAR;
   else if ( type_FLOAT   )
      datal += sizeof(TYPE) + ispf_FLOAT +
         sizeof(FLOAT) - sqlda->sqlvar[i].sqllen;
   }
return ( datal ? malloc ( datal ) : NULL );
}

static void assign ( SQLDA *sqlda, char *datap )
{
int i, type;

for ( i = 0; i < sqlda->sqld; ++i )
   {
   type = sqlda->sqlvar[i].sqltype;
   if ( type&1 )                          /* null indicator */
      {
      sqlda->sqlvar[i].sqlind = (NULLIND *)datap;
      datap += sizeof(NULLIND);
      }
   if ( type_VARCHAR || type_NULCHAR || type_FLOAT )
      {
      *(TYPE *)datap = sqlda->sqlvar[i].sqltype;
      datap += sizeof(TYPE);
      }
   sqlda->sqlvar[i].sqldata = datap;
   if      ( type_VARCHAR ) datap += ispf_VARCHAR;
   else if ( type_NULCHAR ) datap += ispf_NULCHAR;
   else if ( type_FLOAT   )
      {
      memset ( datap, 0, sizeof(FLOAT) );
      datap += ispf_FLOAT +
         sizeof(FLOAT) - sqlda->sqlvar[i].sqllen;
      }
   datap += sqlda->sqlvar[i].sqllen;
   }
}

static char *map2spf ( SQLDA *sqlda )
{
int i, type;
char *name, *names;

names = malloc ( sqlda->sqld*16 + 2 ); /* +2 for '()' */
name  = names;
*name++ = '(';

for ( i = 0; i < sqlda->sqld; ++i )
   {
   type = sqlda->sqlvar[i].sqltype;
   if ( type&1 )                          /* null indicator */
      {
      sprintf ( name, "SQLI%.3d ", i );
      ISPLINK ( "VDEFINE", name, sqlda->sqlvar[i].sqlind,
         "FIXED", &sizeof(NULLIND) );
      name += 8;
      }
   sprintf ( name, "SQLV%.3d ", i );
   ISPLINK ( "VDEFINE", name,
      sqlda->sqlvar[i].sqldata,
      ( ( type_DATE      || type_TIME ||
          type_TIMESTAMP || type_CHAR )      ? "CHAR"  :
        ( type_INTEGER   || type_SMALLINT ) ? "FIXED" :
                                              "USER" ),
      &(long)sqlda->sqlvar[i].sqllen, " ", &vdferitd );
   name += 8;
   }
*name = ')';

return ( names );
}
```

SOLUTIONS TO SOME POTENTIAL PROBLEMS

Several problems may occur as you develop an ISPF application. One involves the "LIST" type VDEFINE that requires program data fields to be contiguous. Since structure element alignment may cause gaps, declare the C data structure and any embedded structures with the `__noalignmem` keyword.

Two types of errors may occur when using ISPF exits. The first type may cause an exit to fail if it is part of a program to which the linkage editor assigned addressing mode (AMODE) is 24. This is because ISPF passes control to exits in 31-bit mode. Therefore, be sure to set the program's AMODE to 31. Another occasionally tricky type of error results from the MVS task structure. If you invoke a program by using the ISPF SELECT CMD service or by using the DSN command processor, ISPF exits run under a separate task from the main procedure. Therefore, the main procedure and the exit cannot share system services that expect to be called under the same task. Examples of when such sharing may cause an error include:

- acquiring storage in an exit that the main procedure frees at termination
- opening a file in an exit that the main procedure closes
- issuing DB2 services in both the main procedure and the exit.

To avoid such errors, invoke the program with the PGM subcommand of the ISPF SELECT service.

CONCLUSION

The SAS/C compiler offers a productive programming environment for developing ISPF and DB2 applications. For the future, the SAA Common Programming Interface leverages the programming investment across a number of environments. You can take several steps now to get ready for SAA. For dialog applications, a major change is in the call interface. SAA uses the ISPCI function, which expects a command string, so that using ISPEXEC now to call dialog services will ease that transition. For database applications, Version 2 of DB2 already supports C as a host language, so that you can easily migrate your C based applications to SAA. Therefore, as you examine your choices for SAA application development languages, consider the potential that C and the SAS/C product offer to your future programming investment.

REFERENCES

The following references offer additional guidance in developing C, ISPF, DB2, and SAA applications:

SAS Institute Inc. (1988), *SAS/C Compiler and Library User's Guide*, Cary, NC.

SAS Institute Inc. (1988), *SAS/C Library Reference Volume 1*, Cary, NC.

SAS Institute Inc. (1988), *SAS/C Library Reference Volume 2*, Cary, NC.

SAS Institute Inc. (1989), SAS Technical Report C-106, *Changes and Enhancements to the SAS/C Compiler, Release 4.00*, Cary, NC.

SAS Institute Inc. (1989), *SAS/C Compiler Interlanguage Communication Feature User's Guide*, Cary, NC.

IBM Corp. (1987), *Interactive System Productivity Facility Version 2 Release 3 for MVS Dialog Management Guide*, SC34-4112, Cary, NC.

IBM Corp. (1987), *Interactive System Productivity Facility Version 2 Release 3 for MVS Dialog Management Services and Examples*, SC34-4113, Cary, NC.

IBM Corp. (1987), *DATABASE 2 Application Programming Guide*, SC26-4293, San Jose, CA.

IBM Corp. (1987), *DATABASE 2 Advanced Application Programming Guide*, SC26-4292, San Jose, CA.

IBM Corp. (1988), *DATABASE 2 Version 2 Application Programming Guide* SC26-4377, San Jose, CA.

IBM Corp. (1987), *Systems Application Architecture Common Programming Interface Dialog Reference*, SC26-4356, San Jose, CA.

IBM Corp. (1987), *Systems Application Architecture Common Programming Interface Database Reference*, SC26-4348, San Jose, CA.

You may also call SAS Institute Technical Support for help with specific questions.

ACKNOWLEDGMENTS

Special thanks to Mark Gass and Bob Patten of SAS Institute for assistance in developing this paper.