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 environ­
ment 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 poten­
tial 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 3 of ISPF do support
the SAS/C extension for
calling assembler routines
from C. However, the exten­
sions 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 fmtting
their parameter lists. To declare an assembler function, use the
__asm keyword. Then, to format its parameter list, use the
__attribute 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 argu­
ments 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 rou­
tines. 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
option 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 environ­
ment initialization can allocate the structures in the Pseudo-
Register Vector (PRV) if you compile the program with the
__declspec option. In that case, the routine can access them
through O-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:

```
EXEC SQL INCLUDE SQLCA SQLDA
```

Direct the assembler listing to DSECT2C, requesting that it pro­
cess 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:

```
sqlca;
typedef char BlS[8];

typedef char Bl[76];
```

```c
typedef struct
{ char sqleid;
  int sqlabc;
  int sqlcode;
  short sqlerrml;
  int sqllim;
  int sqlerrd[4];
  int sqllim;
  int sqlselat;
} SQLCA;
```
The results for the SQLDA look like this:

```
if (defined(SQLCACSE))
  define SQLCODE SQLCA sos.*
endif

define SQLNAMES SQLCA sos[0]
define SQLNAMES SQLCA sos[1]
define SQLNAMES SQLCA sos[2]
define SQLNAMES SQLCA sos[3]
define SQLNAMES SQLCA sos[4]
define SQLNAMES SQLCA sos[5]
define SQLNAMES SQLCA sos[6]
define SQLNAMES SQLCA sos[7]
```

The results for the SQLDA look like this:

```
if (defined(SQLCA))
  define _CL130
typedef char CA[64];
endif

define _CL130
typedef char CA[128];
endif

typedef struct
  |
  CA SQLCA;
  int sqlpdb;
  short sqln;
  short sqlh;
  struct SQVAR
  |
  short sqltype;
  short sqllen;
  char sqldata;
  short sqlflag;
  struct short length; C10 data; | sqlname;
  | sqlname;
  SQLVAR;
#define SQLDASIZE(n) \( (sizeof(SQLVAR) + (n)\times(sizeof(struct SQVAR))) \)
```

### 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

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.

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

```
define SQLDATE TYPECODE(34)
define SQLTIME TYPECODE(35)
define SQLTIMESTAMP TYPECODE(36)
define SQLVARCHAR TYPECODE(43)
define SQLWCHAR TYPECODE(44)
define SQLDECIMAL TYPECODE(54)
define SQLINTEGER TYPECODE(63)
define SQLFLOAT TYPECODE(67)
define SQLREAL TYPECODE(68)
define SQLDOUBLE TYPECODE(69)
define SQLSHORT TYPECODE(96)
define SQLLONG TYPECODE(97)
define SQLSHORTINT TYPECODE(106)
define SQLLONGINT TYPECODE(107)
define SQLSHORT reallen strlen
```

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.

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

<table>
<thead>
<tr>
<th>DB2 data type</th>
<th>host data type</th>
<th>ISPF format</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME</td>
<td>char[8]</td>
<td>char(8)</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>char[26]</td>
<td>char(26)</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>struct(1, short; char[8];</td>
<td>C, USER</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>char[9]</td>
<td>char(9)</td>
</tr>
<tr>
<td></td>
<td>char[12]</td>
<td>char(12)</td>
</tr>
<tr>
<td></td>
<td>char[8]</td>
<td>char(8)</td>
</tr>
<tr>
<td></td>
<td>long</td>
<td>long</td>
</tr>
<tr>
<td></td>
<td>short</td>
<td>short</td>
</tr>
</tbody>
</table>

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 inline machine code interface to perform operations directly on decimal data fields.
The C mainline then initializes the VODERNE exit data structure to use the bridge.

```c
#include <code.h> /* for _strrev() */

extern volatile const struct _...asw int (*vdfexit[]); void *vdr; J vdfexit;

vdfexit.vfexit = vdfexit;

The C mainline initializes the VODERNE exit data structure to use the bridge.
```

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: CSEC1, CENTRY, CEXIT**

```c
EXEC SQL SELECT ...
```

**EXEC SQL SELECT** Identify the program section

**CSEC1** Save the section identifier

**CENTRY** Associate base reg w/C

**EXEC SQL USING** and DB control structures

**USING SQLCA, R1** Include C control structures

**USING SQLDA, R2** Include base reg w/SQL
defs...

**CENTRY LASTREG=R6, RIAS=R6** Enter assembler routine

**EXEC SQL PREPARE SQLSTMT INTO SQLDA FROM SQLSTM** Return to caller

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

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

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

```c
#include <syssem.h>

char sqlcondition[3] = { 0, 0, 0 }
int whentype;

#define TYPE EXEC
#define SQL
#define WHENEVER whentype
#define FOUND
#define TO
#define GOTO goto
define SQUAIR 0; sqlcondition[0] = 1;

#define NOT 1; sqlcondition[0] = 1;

#define SQRLORING 2; sqlcondition[0] = 1;
```

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

```c
#define EXECSQL(sqlfunc) (sqlfunc, 
[(sqlcondition[0] <= SQLC0DE < 6)] ? 
[(sqlcondition[0] <= SQLC0DE == 100)] 
[(sqlcondition[0] <= SQLC0DE > 0) || SQLCODE == 'W']) ? 

EXEC SQL WHENEVER SOLERROR CONTINUE;
/* prevent return if sql routine here fails */
EXEC SQL WHENEVER SQLERROR CONTINUE;
/* put sql error handling code here */
while(1);
```

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

```c
EXEC SQL WHENEVER SQLERROR GOTO label; break;
EXEC SQL WHENEVER SOLERROR 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.

```c
#include <dynam.h>

int *)&p[0]; /* one element per loaded module */

EXEC SQL WHENEVER SQLERROR CONTINUE;
/* put sql error handling code here */
while(1);
```

Then call loads to initialize the pointers.

```c
#include <dynam.h>

EXEC SQL WHENEVER SQLERROR CONTINUE;
/* put sql error handling code here */
while(1);
```

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

```c
#include <syssem.h>

#define DANAME(name)...dbname(name, name, 0), dbenv(0), 
[(sqlcondition[0] <= SQLCODE < 6)] ?

EXEC SQL WHENEVER SQLERROR CONTINUE;
/* put sql error handling code here */
while(1);
```

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 DSNH Li is linked 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 DSNH LI and branch to it. For example, if the C mainline initializes the "(dbname2)" function pointer by loading DSNH LI, you can use the following simple assembler bridge:

**Example: CSEC1, CENTRY, CEXIT**

```c
ENTRY DSNHLI

DSNL i ....
```

**ENTRY DSNHLI**

```c
ENTRY DSNHLI

DSNL1 SU EU
ENTRY X, R15
L R15,ǭ(DSNHLI) load PPF offset of dblhit
AL R15,CRABPRV add PPF pointer from CRAB
L R15,4(R15) load dblhit pointer from PPF
```

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

```c
#include <stdio.h> /* for _strdup() */
define NOCONNECT b00C10025

int dsnt1ar ( void *sqlca, DSNHAR *sqlph, char *sqlca, char *sqlph)
  return ( rc );
```

```c
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

```c
struct { short ftullind, db2type; char tend, flhrealispLFLOAT+1J; int vdfexit if ( [Type] )
```

```c
#include <string.h> /* for _strdup() */
define NOCONNECT b00C10025

define NCONNECT b00C10025

define NOCONNECT b00C10025

define NOCONNECT b00C10025
```

```c
/** for _strdup() */
define NOCONNECT b00C10025

int dsnt1ar ( void *sqlca, DSNHAR *sqlph, char *sqlca, char *sqlph)
  return ( rc );
```

```c
#include <string.h> /* for _strdup() */
define NOCONNECT b00C10025

define NCONNECT b00C10025

define NOCONNECT b00C10025

define NOCONNECT b00C10025
```
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.

```c
if ( sqlda = prepare ( sqlstat ) )
{
    if ( datap = allocate ( sqlda ) )
    {
        assign ( sqlda, datap );
        names = malloc ( sqlda );
        SQLPLAN ( "WHERE", "*", names, "WHERE" );
        sqlda->sqlid = 1; /* open */
        while ( SQLCODE = 0 )
        {
            sqlda = malloc ( sqlda );
            if ( SQLCODE = 0 )
                SQLEXEC ( "FETCH TABLE" );
            sqlda = malloc ( sqlda );
        }
        ASSIGN ( sqlda, datap );
        while ( SQLCODE = 0 )
        {
            SQLPRT ( sqlda );
            if ( SQLCODE = 0 )
                SQLEXEC ( "UPDATE TABLE" );
            sqlda = malloc ( sqlda );
        }
        return ( names );
    }
    for ( i = 1, type = sqlid, sqlda = NULL;
        sqlda = malloc ( sqlda->sqlid + 2 ); /* s2 for '(' */
        sqlda->sqlid = sqlda->sqlid + 1;
        sqlda = malloc ( sqlda );
        free ( sqlda );
    }
}
static SQLDA *prepare ( VARCHAR *sqlinit )
{
    SQLDA *sqlda, sqlda0;
    sqlda0.sqlid = SQLPLAN ( "sqlinit" );
    sqlda = malloc ( sqlda0 );
    SQLEXEC ( "SQLDA" );
    return ( sqlda );
}
static char *allocate ( SQLDA *sqlda )
{
    int i, type, datel = 0;
    for ( i = 0; i < sqlda->sqlid; ++i )
    {
        if ( type = sqlda->sqlid[i].sqltype; datai = sqlda->sqlid[i].sqlid +
            type == SQLCHAR )
            sizeof (WCHAR );
        else if ( type == SQLCHAR )
            datai = sqlda->sqlid[i].sqlid +
            sizeof (WCHAR );
        else if ( type == SQLFLOAT )
            datai = sqlda->sqlid[i].sqlid +
            sizeof (FLOAT );
    }
    return ( datai = sqlda->sqlid[i].sqlid );
}
static void assign ( SQLDA *sqlda, char *datap )
{
    int i, type;
    for ( i = 0; i < sqlda->sqlid; ++i )
    {
        if ( type = sqlda->sqlid[i].sqltype; type == SQLCHAR )
            sizeof (WCHAR );
        else if ( type == SQLCHAR )
            sizeof (WCHAR );
        else if ( type == SQLFLOAT )
            sizeof (FLOAT );
    }
}
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 _aligned 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 CB0 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:


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.

SAS/C is a registered trademark of SAS Institute Inc., Cary, NC, USA.
IBM is a registered trademark of International Business Machines Corporation.
DB2, SQL, and Systems Application Architecture are trademarks of International Business Machines Corporation.