ABSTRACT
SAS/C Compiler products provide a powerful and flexible environment for the development of C programs. Extensions to the ANSI C language described in this paper make it easy to develop applications that take full advantage of IBM products such as GDDM and ISPF.

INTRODUCTION
This paper describes some extensions to the C language provided by the SAS/C Compiler that enable you to develop applications that interact with IBM products. Most of the examples presented here interact with the Graphical Data Display Manager (GDDM) and the Interactive System Productivity Facility (ISPF). These products were chosen because they are widely used and have interfaces that can be used to illustrate all of the necessary techniques for communicating with other IBM products.

This paper assumes that you are familiar with the C language. You do not need to have any knowledge of the SAS/C Compiler products, GDDM, or ISPF.

CALLING NON-C PROGRAMS FROM THE C LANGUAGE
Many IBM products provide an interface that enables application developers to write a program that interacts with the product by calling the functions in the interface. Since most IBM products are not written in C and the interfaces were designed to be called from a variety of languages, some care must be taken to ensure that the SAS/C Compiler produces code that can be used to call these non-C interface functions from C programs. The subsequent section describes the correspondence between the C data types and the arguments expected by IBM products.

Calling Conventions
When a function call occurs in a C program, the calling function creates a parameter list to pass to the called function. This parameter list consists of a copy of the current values of the parameters.

In the following example, the call to the function abc creates a parameter list consisting of a 4-byte integer, an 8-byte double, and a pointer to a string. This form of parameter passing is normally called pass-by-value.

```c
int i;
double x;
abc(i, x, "String");
```

Most IBM products, however, expect to receive parameter lists that consist of a list of pointers to the parameters rather than the actual parameter values. This form of parameter passing is called pass-by-reference. The previous example can be changed easily to produce a parameter list consisting of a list of pointers by using the ampersand (&) operator to obtain a pointer to each of the non-pointer arguments:

```c
int i;
abc(&i, x, "String");
```

The & operator must not be used for the third argument because a constant character string in C is already treated as a pointer.

This technique produces a parameter list consisting of a list of pointers. However, the & operator can be used only on arguments that are variables or array elements. This can be inconvenient if you need to write many calls whose arguments are constants or expressions. To provide a more convenient way to pass such arguments, the SAS/C Compiler supports the at-sign (@) operator. The @ operator returns a pointer to its operand just like the & operator. However, the & operator can be applied to any variable, constant, or expression. (For constants and expressions, the @ operator returns the pointer to a temporary copy of the value).

For example, if you want to call the abc function used previously with an integer constant 1 for the first argument, the value of 2*x as the second argument, and the same constant string as the third argument, the statement is

```c
abc(1, (2*x), "String");
```

Many non-C functions accept a variable number of arguments. The standard way to mark the end of a parameter list is to set the high-order bit of the last argument pointer to 1. (The high-order bit of a word is not part of the pointer, so modifying its value does not change the value of the pointer). You can request that the SAS/C Compiler set the high-order bit of the last argument pointer by adding the prefix _asm to the call statement or declaring the function with the _asm keyword. (For the prefix, there are two underscores before the _asm keyword, for the keyword, there are two underscores before the asm). The prefix and the keyword perform exactly the same function. The prefix can be used only in a function call and is attached in front of the name with no spaces. The keyword is used in a declaration of the function. You can use whichever one is more convenient. For example, the following statement calls the function abc and passes it a parameter list consisting of three pointers with the end-of-list flag set on the last pointer:

```c
_asm abc(i1, i2, "String");
```

This technique requires that every call to abc have the _asm prefix. The same effect can be obtained by declaring abc as an_asm function once and then writing normal call statements such as

```c
_asm void abc();
```

For a real example, the GDDM function _ascput fills a specified field on the screen with a string of characters. The first argument is the field number, the second argument is the string length, and the third argument is the string. A typical call is

```c
char *msg = "This is the string to write in the field";
_asm_ascput(i99, strlen(msg), msg);
```

You can simplify the appearance of the calls by using #define statements to supply the _asm prefix and the appropriate @ operators. An example of the #define statement for the ascput function is

```c
#define ascput(field_id, length, string) \ _asm_ascput(id, length, string)
```

752
ARGUMENT TYPES

Some care must be taken to ensure that the types of arguments in the C program correspond to the types expected by the non-C functions. The following sections describe each of the C data types and their corresponding representation in most IBM products.

Integer Arguments

Integral arguments are either full-word or half-word binary integers. These correspond to the C data types int or short int, respectively. You should use the @ operator on one of the arguments that are scalar integers.

When passing a constant or the result of an expression to a function that expects to receive a half-word integer, you must code an explicit cast to force the type to short. The C language definition provides no way to write a short integer constant. In addition, in C all short integers used in expressions are converted to type int before the expression is evaluated. For example, the following statements pass two half-word integers to the function foo:

```
short a = 5, b = 10;
--asm foo();
```

Without the @ operator, both arguments would be passed as full-word integers.

Floating-Point Arguments

Most floating-point arguments to IBM programs are single-precision type. This corresponds to the float type in C. You should use the @ operator to obtain a pointer to all arguments that are scalar floats.

When passing the results of floating-point expressions to non-C functions, you should be careful to ensure that the expression yields the correct type. For example, the following statements actually pass pointers to two temporary variables of type double to the GDDM gsmmove function:

```
float x, y;
--asm gsmmove (2|x*2.5|, 2|y*2.5|);
```

In C, the constant 2.5 has a type of double, forcing conversion of the result of the expression to double. To change the above call to pass pointers to two float temporary variables, you should change it to one of the following statements:

```
--asm gsmmove (2|x*2.5f|, 2|y*2.5f|);
--asm gsmmove (2|x(float)*2.5|, 2|y(float)*2.5|);
```

Character Arguments

In C, character strings normally vary in length, and the end of the string is marked by a null character. With rare exceptions, non-C programs do not use this convention. Fixed-length strings are normally passed as a pointer to the first character. The string is padded with spaces, if necessary, to make it the expected length. Varying-length strings are normally passed as two separate arguments. One argument is a pointer to the first character in the string; the other is a pointer to the integer length of the string.

When calling non-C functions from C, string arguments should be of type char *. Because these arguments are already pointers, you do not use the a or @ operators. If you pass a character constant to a non-C function, it should always be enclosed in double quotes, even if it is only a single character. The following example shows the correct way to pass a character constant to a non-C function:

```
--asm foo("a");
```

The C language definition specifies that a character enclosed in single quotes is of type int. Therefore, the following statement does not pass a pointer to the character 'a' to the function. (It really passes a pointer to a full-word binary integer that contains the character in the last byte).

```
--asm foo("a"); /* Never use this */
```

Many IBM functions take fixed-length strings as arguments. In those cases, the SAS/C function memmove can be useful in converting null-terminated strings to fixed-length strings. It copies a string and adds padding spaces to the end if needed. In the example below, cstr is assumed to point to a null-terminated string. The non-C function xyz expects to be passed a 12-character string padded on the right with spaces. The call to the memmove function copies the string from cstr to fstr and pads it with spaces to make it 12 characters long, if needed.

```
char *cstr;
char fstr[12];
memmove(fstr, cstr, 12, strlen(cstr), ' ');
```

Array and Structure Arguments

Passing arrays to non-C functions is simple. You declare the array with the correct type and size and simply use the array name as the function argument. Because an array name in C is already treated as a pointer, you do not use the a or @ operator in the call. For example, the GDDM function gsmrks draws a series of marker symbols. The first argument is the number of symbols to draw. The other two arguments are arrays of x and y coordinates for the symbols. The following example is a possible call to gsmrks:

```
float x[20], y[20];
--asm gsmrks(20, x, y);
```

Structure arguments are used relatively rarely. In most cases, you can use the C language structure declarations to produce structures that correspond to those expected by the function. You must use the @ or * operator for structure arguments. For example, if you use the reentrant interface to GDDM, the first argument to all functions is a pointer to the Application Anchor Block (aab). The following example is a possible declaration and use of the aab:

```
struct aab
|
short ec; /* GDDM severity code */
short ec; /* GDDM error code */
void *ap; /* GDDM anchor pointer */
aab(0, 0, NULL);
--asm fswin(aab);
```

Use extra care with structures that contain other structures because the rules used by the SAS/C Compiler for alignment of structure members are not the same as those used by the assembler. The
SAS/C Compiler provides the \_\_alignmem and \_\_alignmem keywords to control the amount of alignment padding inserted by the compiler.

In some cases, particularly for complex structures, IBM supplies structure definitions in the form of assembler language DSECTs. You can use the SAS/C utility program DSECT2C to convert these into the corresponding C structure declarations.

**WRITING EXITS IN THE C LANGUAGE**

Prior sections of this paper discuss techniques for calling IBM products from C programs. Many IBM products also enable you to write exit functions, which are functions called by the IBM product to perform some service. A typical example of this type of exit is the ISPF VDEFINE service. To use the VDEFINE service, the main program written in C calls ISPF and passes it a pointer to an exit routine, also written in C. ISPF then calls the function as needed. This has the effect of putting ISPF (which is not written in C) in the middle of your C program. For this to work, you must perform some extra steps to control execution frameworks and specify the function pointer in the correct format. The following sections discuss each of these topics.

**Execution Frameworks**

In general, successful execution of code written in a high-level language requires the accessibility of an appropriate execution framework. An execution framework (also called an environment) is a collection of data and routines supporting the execution of code. Because each language has its own conventions for access to its framework, it is usually impossible for more than one framework to be accessible at once. Therefore, a call from one language to another must switch frameworks.

When a program compiled using the SAS/C Compiler is run, the main program automatically creates the execution framework and makes it accessible for itself and all C language functions that it calls. If a C program calls a non-C function, the called function saves the machine registers on entry and restores them on exit, thereby saving and restoring the C execution framework. In this case, no special action is necessary on the part of the programmer.

When an exit routine is called by an IBM product, the active execution framework is for the IBM product rather than for SAS/C software. Because a C function, by default, expects the C framework to be active when it is called, it is likely that the exit routine will fail.

The SAS/C INDEP compiler option is designed to enable C functions to be called when other execution frameworks are active. To use the INDEP compiler option, you must follow these steps:

1. Compile the main program and any functions that are called from non-C execution frameworks with the INDEP compiler option. Using the INDEP option to compile a function called from a non-C framework causes the compiler to generate extra code to locate and restore the C execution framework that was saved by the main program.

2. When you link the program, override the default entry point (\_\_main) and change it to \_\_main. Changing the entry point name causes the C library to save the information that is used later by the INDEP-compiled functions.

**Function Pointers**

Passing function pointers to non-C programs also requires special consideration. IBM products expect a function pointer to be a simple pointer, that is, a pointer that points directly to the executable code for the function. However, a normal C function pointer generated by the SAS/C Compiler points indirectly to the code and also contains other information. Adding the \_\_asm keyword to the declaration of the function pointer causes the SAS/C Compiler to generate a pointer that points directly to the executable code.

The following example demonstrates calling the ISPF VDEFINE service to define an exit function:

```c
struct {
    \_\_asm int (*exitptr)(int, char *); /* Exit function pointer */
    char *data; /* User word (not used) */
} udata;

int cuserxt(); /* Exit function */

void main()
{
    char name[41];
    udata.exitptr(\_\_asm int (*)(int))cuserxt;
    \_\_asm .ipltlink ("VDEFINE ", "NAME ", name, "USER ",
    udt, ",", udata);
}

int cuserxt(char *data, int *code,
            char *exitptr,
            int *deflen,
            char *defarea,
            int *spfdlen,
            char *spfdatap)
{
    /* Code for exit function goes here */
}
```

**CONCLUSION**

The support provided by the SAS/C Compiler for calling non-C functions and writing exit routines makes it feasible and desirable to use C for developing powerful applications. The major techniques discussed in this paper are

- the \_\_asm operator and \_\_asm keyword to produce the correct parameter lists for non-C functions
- the DSECT2C utility to convert assembly language structures to C structures
- \_\_asm function pointers and the INDEP compiler option to allow exit functions written in C
- the memcpyp function to convert varying-length strings to fixed-length strings.

**REFERENCES**

IBM Corp. (1984), Graphical Data Display Manager Base Programming Reference, Mechanicsburg, PA.

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


