SAS Publishing

# SAS/C® 7.50: Changes and Enhancements

*The Power to Know®*

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2004. *SAS/C® 7.50: Changes and Enhancements*. Cary, NC: SAS Institute Inc.

**SAS/C® 7.50: Changes and Enhancements**

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

1st printing, March 2004

# Contents

CHAPTER 1

# SAS/C Compiler Changes in Release 7.50

## Introduction

This chapter provides a complete description of the changes and enhancements to the *SAS/C Compiler and Library User's Guide* for Release 7.50.

## Release 7.50 Enhancements to the SAS/C Compiler

The following enhancements to the SAS/C Compiler have been implemented with Release 7.50:

- IEEE floating-point support and C99 floating-point extensions
- z/Architecture support, include 64-bit addressing
- Compile-time options that target specific S/390 or z/Architecture processors.

## Floating-Point Changes for SAS/C Release 7.50

The SAS/C compiler and library's floating-point support has been extensively changed and improved for Release 7.50. The changes are concentrated in three distinct, but overlapping areas:

- In recent years, the 390 Architecture has been augmented by the addition of support for IEEE standard floating-point. Programs can now be written in SAS/C to exploit this support. Traditional mainframe floating-point continues to be supported. See "IEEE Floating-Point Support" on page 3 for more information.
- The same architectural extensions that introduced IEEE floating-point support also introduced enhancements that can significantly improve the performance of traditional floating-point applications. For instance, the number of floating-point registers has increased. SAS/C now provides a compiler option to generate code targeting the extended floating-point architecture. See "SAS/C IEEE Support" on page 4 and "SAS/C and Mainframe Floating-Point Extensions" on page 4 for more information.
- The ANSI/ISO C99 standard mandates a great deal of floating-point support beyond that required by the 1989 C Standard, including new math functions and new pragmas. SAS/C now supports many of these C99 features, for both IEEE and traditional mainframe floating-point. See "SAS/C C99 Support" on page 5 for more information.

## IEEE Floating-Point Support

The ISO standards document IEC 60559 defines a portable standard for floating-point computation known informally as IEEE floating-point. This standard is implemented by almost all computer systems presently in use. Until recently, the major exception to this statement was the IBM mainframe, which offered a different floating-point implementation defined by IBM in the 1960's. In the last decade, IBM has remedied this situation by offering an implementation of IEEE floating-point parallel to traditional IBM floating-point. Depending on their requirements, mainframe programs can choose to use traditional IBM floating-point, the more portable IEEE floating-point, or, for very advanced applications, both floating-point implementations.

There are advantages and limitations to the use of IEEE floating-point by applications, and the choice between traditional mainframe floating-point and IEEE floating-point is not always obvious.

### Advantages of IEEE Floating-Point

The following list contains a few of the reasons that you might prefer to use IEEE floating-point:

- IEEE floating-point is far more portable. If an application needs to produce exactly the same answers for the same inputs on all platforms, it should use IEEE floating-point.
- A number of the IEEE features make it more flexible for dealing with errors and inaccuracies than IBM traditional floating-point. Examples of such features include not-a-numbers (NaNs), infinities, rounding modes, and the use of status flags rather than trapping for error handling.
- IEEE double precision enables a wider range of exponents than traditional mainframe floating-point. The range of positive IEEE double precision values is approximately from 5E–324 to 2E308. The corresponding range for traditional mainframe floating-point is about 5E-79 to 7E75.
- IEEE floating-point is more uniform in its handling of inaccuracy because it uses base 2 instead of base 16.

### Advantages of the Traditional Mainframe Floating-Point

The following list contains a few of the reasons that you might prefer to use traditional mainframe floating-point:

- On present mainframe models, the traditional floating-point hardware runs significantly faster than the IEEE hardware to perform the same computation. This could change in future hardware.
- For a single double-precision computation, traditional mainframe floating-point is never less accurate than the corresponding IEEE computation (assuming the result is within the range of both), and can be up to 3 bits more accurate. It is not possible to generalize this statement to more complex calculations due to the effects of accumulated rounding.
- Traditional floating-point uses a simpler conceptual model than does IEEE floating-point. This can make analysis of the characteristics of a particular algorithm more manageable.

IBM's traditional floating-point format uses a base-16 representation, that is, the exponent represents a power of sixteen. IEEE floating-point uses a binary representation, with the exponent representing a power of two. For this reason, IBM calls its traditional floating-point format HFP (hexadecimal floating-point), and its IEEE format BFP (binary floating point). HFP and BFP are used as synonyms for traditional mainframe floating-point and IEEE floating-point in the rest of this document.

## SAS/C IEEE Support

When you compile a SAS/C program, you can specify the **`BFP`** or **`NOBFP`** option to specify whether the default floating-point format is BFP or HFP. HFP is the default.

SAS/C enables programs to use both floating-point formats. This is a useful option for programs such as debuggers, and for subroutine libraries whose users want a choice of format. To support the use of both formats, SAS/C provides the **`__binfmt`** and **`__hexfmt`** type modifiers, which can be used to specify the floating-point format independently of whether the **`BFP`** option is used. Similarly, H and B suffixes can be used with floating-point constants to indicate the format.

"Using Both IBM and IEEE Floating-Point Formats" on page 24 contains detailed rules on mixing the two formats, but the basic premise you should keep in mind is: You can use casts to convert between the two formats, but mixed-mode expressions are, in general, not supported. In other words, if you try to add a **`__binfmt double`** to a **`__hexfmt double`**, the compiler cannot determine which kind of addition operation it should perform. You therefore have to code a cast of one operand to the format of the other.

The SAS/C library of mathematical functions is completely supported for both traditional and IEEE floating-point. However, some functions (such as **`isinf`**, which tests to see if its argument is an infinite number) are not meaningful for hexadecimal floating point, and return dummy results of little utility.

SAS/C has added format modifiers to the **`printf`** and **`scanf`** series of functions to enable either format of floating-point number to be read or written. If a format specifier does not have a floating-point type, the corresponding argument is assumed to have the default floating-point format. Use of the extensions is not required for programs which use only one kind of floating-point.

## SAS/C and Mainframe Floating-Point Extensions

The most recent generations of IBM mainframes have included extensions to the floating-point hardware which enable improved performance for traditional floating-point as well as for IEEE support. The SAS/C compiler option **`ARCHLEVEL(C)`** can be specified to cause the compiler to generate code that targets the newer floating-point hardware. If you specify the **`BFP`** option, **`ARCHLEVEL(C)`** is the default ARCHLEVEL setting.

The two main advantages of **ARCHLEVEL(C)** for traditional floating-point code are:

- With the floating-point extensions, processors support sixteen floating-point registers rather than four. This feature enables much more data to be kept in registers. The global optimizer with **ARCHLEVEL(C)** will assign up to eight double or float register variables. Without **ARCHLEVEL(C)**, only two register variables are supported.
- The floating-point extensions offer single instructions to convert between integer types and floating-point types. Without the floating-point extensions, the compiler would have to generate a much longer and more complex sequence of instructions.

## SAS/C C99 Support

SAS/C Release 7.50 supports many elements of the ISO C99 standard relating to floating-point. Some of these elements are oriented specifically to IEEE floating-point, but most apply to both formats.

The most significant SAS/C enhancements in support of C99 are:

- Support for a new hexadecimal exponential format for floating-point constants.
- Support for several C99 standard pragmas to enable a program to control the extent of the optimization of floating-point operations.
- Augmentation of the header file, **float.h**, with further information about the floating-point implementation.
- Support in the **printf** and **scanf** functions for a new **%a** format for hexadecimal input and output of floating-point data.
- Addition of a new header file, **fenv.h**, that defines functions which enable you to test IEEE status flags, specify the rounding mode, raise IEEE exceptions, and so forth. There are dummy implementations of these functions for traditional mainframe floating point. Also, the header file **fenvtrap.h** (which is a SAS/C extension) defines functions which let you control IEEE exception trapping.
- Addition of a number of new math functions defined by C99. Some of these are low-level IEEE functions, such as **isinf** (to test whether a number is infinite) and **nextafter** (which returns the next floating-point number in a particular direction from its argument). Others are more traditional mathematical operations like **remainder** and **log2** (logarithm base 2). All of these functions are supported for BFP and HFP inputs. Functions specific to IEEE such as **isinf** will return degenerate results when called for **__hexfmt** arguments. See *SAS/C Library Reference, Volume 1* for a full list of the supported math functions.
- Adding versions of mathematical functions using the data type float. The C99 standard defines a set of math functions similar to the traditional set, with float arguments and return values. For example, C99 requires the **exp** function to have a float version named **expf**. While most functions, such as **exp**, have a differently named variant function for use with float, a few functions (for instance, the **signbit** function) are defined as macros that can be passed as an argument of any floating-point type.
- Addition of a new header file, **tgmath.h**, that is defined by the C99 standard to provide generic versions of math functions. For instance, **tgmath.h** defines an **exp** macro which can accept any numeric argument, and returns a result of the correct type.

# 64-Bit Support

Release 7.50 of the SAS/C Compiler and Library includes support for execution in 64-bit addressing mode under Release 1.2 or higher of z/OS.

This support is intended to allow SAS/C customers to gain experience with 64-bit support, and to write applications which allocate and access memory *above the bar*. However, note that SAS/C 7.50 does not actively exploit 64-bit addressing. While it is possible to allocate 64-bit addressable memory via the SAS/C multiple heap support, the SAS/C library continues to allocate all its other storage in 31-bit addressable memory. A limited number of library functions have been modified to accept 64-bit pointers, and the remainder of the library will never be executed in AMODE 64. For Release 7.50 of SAS/C, the design permits AMODE 31 and AMODE 64 code to coexist within the same application, provided that care is taken in linkages.

A design goal for a future release of the SAS/C library will be to exploit the capabilities of 64-bit addressing. In order to exploit 64-bit addressing SAS/C library control blocks will need to be changed to reflect the larger pointer size, which will prevent previously compiled SAS/C code from working in a future release. In other words, in a future version of SAS/C, it will be impossible to combine old AMODE 31 compiled code with new AMODE 64 compiled code, and old applications will require recompilation if they are to be used in a 64-bit addressing environment. Because Release 7.50 does allow old and new code to be mixed, it may be a useful tool for migrating existing applications gradually to 64-bit addressing, rather than having to modify all components of an application simultaneously.

## Compile-Time Support for 64-bit Addressing

SAS/C support for 64-bit addressing is enabled by use of the **HUGEPTRS** option. When this option is specified, the generated code targets the z/Architecture, and uses 64-bit pointers for all memory references. As one would expect, the **HUGEPTRS** option implies the **ARCHLEVEL(D)** option, which informs the compiler that z/Architecture instructions may be generated, and that 64-bit general purpose registers exist.

Implementations of the C language on systems with 64-bit pointers have made various choices about the sizes of integers. One popular choice is LLP64, in which the integer type **long** remains a 32-bit type, like the type **int**, and the only 64-bit integer type is **long long**. Another popular choice is LP64, in which the type **long** has 64 bits, like **long long**. There are advantages to either choice. SAS/C has chosen to adopt the LP64 model, in which a **long** is a 64-bit type. Thus, use of the **HUGEPTRS** option will not only enable 64-bit pointers, but will also change the size of **long** from 4 bytes to 8.

When you compile with **HUGEPTRS**, pointers are by default 8 bytes in size. You can declare a pointer as a 32-bit (fullword) pointer by using the type modifier **__near**. You can also use the modifier **__huge** to explicitly declare a 64-bit pointer. The **__near** and **__huge** keywords follow roughly the same placement rules as the **const** and **volatile** keywords. For instance, the declaration

```
int __near * __huge *myaddr;
```

declares the variable **myaddr** as being a huge (8-byte) pointer to a near (4-byte) pointer to an int.

When you use **__near** pointers, you need to be aware of an important difference between 31-bit addressing and 64-bit addressing. In 31-bit addressing, the high-order bit of a pointer is ignored. For this reason, the high-order bit of pointers is frequently used in assembler applications as a flag. This usage is prevalent in older system control blocks. In 64-bit addressing mode, the high order bit of a **__near** pointers is not ignored. Attempting to derefence a **__near** pointer with the high-order bit set will cause a paging exception and an 0C4 ABEND. If your application processes such pointers, before dereferencing them, you must clear the high-order bit. The bit can be cleared either by casting to an integral type and using the "and" operator, or by casting the **__near** pointer to a **__huge** pointer, which also clears the bit. The following code example illustrates both techniques:

```
__near char *impure;
__near char *purified;
__huge char *purehuge;
purified = (__near char *)(((unsigned int) impure) & ~0x80000000);
purehuge = (__huge char *) impure;
```

The change in pointer size for 64-bit support also means that certain standard types defined by C header files will change. The two most important such types are **`size_t`** and **`ptrdiff_t`**. Without the **`HUGEPTRS`** option, **`size_t`** continues to be defined as **`unsigned int`**, and **`ptrdiff_t`** as **`int`**. When **`HUGEPTRS`** is specified, **`size_t`** is **`unsigned long`** and **`ptrdiff_t`** is **`signed long`**, both of which are 64-bit types.

The table below shows the effect of **`HUGEPTRS`** on the sizes of objects of various types. Types which are not derived from one of these types can be assumed to be the same regardless of the **`HUGEPTRS`** setting. For instance, the sizes of function pointers and **`enums`** are not changed.

| Type | NOHUGEPTRS Size | HUGEPTRS Size |
|---|---|---|
| **`long`** | 4 | 8 |
| **`size_t`** | 4 | 8 |
| **`ptrdiff_t`** | 4 | 8 |
| **`unqualified`** pointer | 4 | 8 |

You can use the **`__huge`** keyword in a program which you compile without the **`HUGEPTRS option`**, but the generated code will be unable to dereference or return a **`__huge`** pointer. If you do not compile with the **`HUGEPTRS`** option, uses of 8-byte pointers should be limited to use as unreferenced structure elements, or as arguments which are passed directly to other functions. This limited support is sufficient to allow AMODE 31 functions to manipulate data structures which include 64-bit pointers, so long as no attempt is made to dereference them.

It is possible to call AMODE 64 functions from AMODE 31 functions and vice versa. In SAS/C Release 7.50 all function calls are made in AMODE 31. A function compiled with **`HUGEPTRS`** will switch into AMODE 64 using the SAM64 instruction after the function prolog has executed, and will switch back to AMODE 31 using the SAM31 instruction before calling any other function, and before returning to its caller. This allows AMODE 31 and AMODE 64 functions to be arbitrarily mixed in the calling chain, while assuring that the prolog need not execute additional instructions to avoid use of the high-order parts of the general purpose registers. This technique does introduce a performance penalty which would not be present in an environment which was completely 64-bit enabled.

When developing an application in which some code is compiled with **`HUGEPTRS`** to execute in AMODE 64, and some is compiled without **`HUGEPTRS`** to execute in AMODE 31, there is significant risk associated with shared data. Any type and data declarations used by both AMODE 31 and AMODE 64 code must be carefully constructed to have the same size and mapping in both AMODEs. Following are some guidelines which may be helpful here. Do not declare data items as **`long`** or **`unsigned long`**, or as **`size_t`** or **`ptrdiff_t`**. Declare them as **`signed int`** or **`unsigned int`** if they should be 32-bit integers, or as **`signed long long`** or **`unsigned long long`** if they should be 64-bit integers. Alternatively, use types defined by the header file **`<stdint.h>`** such as **`intfast32_t`**, which do not change when **`HUGEPTRS`** is specified. Similarly, pointers should be explicitly declared as **`__near`** or **`__huge`**, as appropriate. In particularly tricky or complex cases, you may need to code a **`#if _O_HUGEPTRS == 1`** test to define distinct data mappings for 31-bit and 64-bit components.

The above considerations also apply to function prototypes. Functions which can be called from both **`HUGEPTRS`** and **`NOHUGEPTRS`** callers should avoid the use of types like **`long`**, **`size_t`** and **`ptrdiff_t`**, and define any pointer arguments as explicitly **`__huge`** or **`__near`**.

*Note:* In Release 7.50, function pointers continue to be 4-byte pointers, regardless of whether **`HUGEPTRS`** is specified. △

## 64-bit Support and Access Register Mode

With the Systems Programming Environment (SPE), you can specify both the **`HUGEPTRS`** option and the **`ARMODE`** option. In this case, the code will execute in both access-register mode and AMODE 64. With Release 7.50, **`__far`** pointers continue to be 8 bytes in size, containing a four-byte address and a four-byte ALET value.

In 31-bit addressing, the high-order bit of the second word of a **`__far`** pointer is not significant. In 64-bit addressing, this bit is significant. If the bit is set, a dereference of the pointer will result in an 0C4 ABEND. If your 64-bit application has to process **`__far`** pointers which may have this bit set, you can clear the flag using code similar to that below:

```
__far char *impure;
 __far char *purified;
purified = (__far char *)(((unsigned long long) impure) &
              ~0x80000000ULL);
```

## 64-bit Support and Inline Machine Code

The SAS/C inline machine code feature is supported for programs compiled with **`HUGEPTRS`**. Inline machine code has access to all 64 bits of the general purpose registers. When the **`_ossvc`**, **`_osarmsvc`**, or **`_ospc`** function is used to perform a system call, the compiler generates code to leave AMODE 64 before the system call and to restore it afterwards. See Chapter 13, "Inline Machine Code Interface," of the *SAS/C Compiler and Library User's Guide* and "IEEE and 64-Bit Updates to the Inline Machine Code Interface" on page 17 in this document for more information on the inline machine code facility.

## 64-bit Support and Communication with Other Languages

In general, the SAS/C features for communication with assembler language or with other high-level languages should be used with care by programs compiled with the **`HUGEPTRS`** option. Specific caveats are as follows:

- All calls, including calls to assembler routines, are performed in 31-bit mode. If a called function uses the high-order four bytes of the general registers, it must save and restore them. An assembler function may enter AMODE 64 if it wishes, but it must restore 31-bit mode before returning. Also, if assembler code calls any C functions or C library functions, it must be in 31-bit mode at the time.
- If a function is called from assembler in situations where a **`__near`** pointer argument may have the high-order bit set (as to indicate the end of a VL argument list), you cannot dereference the pointer without clearing the bit. See "Compile-Time Support for 64-bit Addressing" on page 6 for further information.

- The **__asm** keyword may be used in code compiled with **HUGEPTRS**. However, note that if the last argument in an **__asm** argument list is a **__huge** pointer, the high-order bit will not be set, and the called function will have no indication of the end of the argument list.
- A call to a function declared using the **__ref** keyword always passes **__near** pointers in the argument list. If a **__huge** pointer argument is specified, it will be converted to a **__near** pointer, which is passed in its place. Similarly, if an argument is accessible only via a **__huge** pointer, its address will be truncated to 31 bits in the argument list.
- The **linkage(OS)** pragma and the **__ibmos** keyword are not supported for programs compiled with **HUGEPTRS**.
- Use of the SAS/C ILC feature in programs compiled with **HUGEPTRS** is strongly discouraged. The effect of attempting to pass a **__huge** pointer, or data addressable only using a **__huge** pointer, is both unpredictable and futile because no IBM language implementation other than assembler presently supports 64-bit addressing.

## 64-Bit Support and the SAS/C Library

SAS/C Release 7.50 provides interim 64-bit addressing support. A significant subset of the SAS/C library supports **__huge** pointer arguments and execution in 64-bit mode, but many important areas still have no such support. For example, there is no way in this release to read data from a socket into a storage area allocated *above the bar*.

This is similar to the operating system support (z/OS 1.3) at the time this level of the product was released. While it is possible in z/OS 1.3 to allocate 64-bit addressable memory, z/OS does not yet support reading or writing directly from such memory. In the long run, such support is necessary, but we are able to provide meaningful 64-bit support earlier by delaying such features until a later release of the operating system.

SAS/C library support for AMODE 64 and huge pointers in this release can be summarized as follows:

- The standard string functions are supported with **__huge** arguments, and lengths greater than 2 gigabytes.
- The **malloc** function, the **palloc** function, and the new multiple heap allocation functions can be used to allocate memory requiring a **__huge** pointer for addressibility. Also, the header file **<iarv64.h>** provides a low-level interface to the z/OS IARV64 service, which is used to allocate memory “above the bar”.
- The **printf** family of functions has been extended to format **__huge** pointer values, and strings accessible via **__huge** pointers. Similarly, the **scanf** family can read values into areas addressed via a **__huge** pointer. New format modifiers allow the program to completely control such formatting.

The following table contains a list of the library functions that accept **__huge** pointer arguments. Attempts to pass **__huge** pointers to library functions not in the table will result in their conversion to **__near** pointers, with possible data loss. A warning message will be generated by the compiler whenever this occurs.

| | | | | |
|---|---|---|---|---|
| **afread** | **afreadh** | **afread0** | **afwrite** | **afwriteh** |
| **afwrite0** | **atoi** | **atol** | **atoll** | **bsearch** |
| **calloc** | **fgets** | **format** | **fprintf** | **fputs** |
| **fread** | **fscanf** | **fwrite** | **gets** | **hpalloc** |
| **hpcalloc** | **hpfree** | **hppoolcreate** | **hprealloc** | **IARV64** |

| | | | | |
|---|---|---|---|---|
| **malloc** | **memcasecmp** | **memchr** | **memcmp** | **memcpy** |
| **memfil** | **memlwr** | **memmove** | **memscan** | **memscntb** |
| **memset** | **memupr** | **memxlt** | **nan** | **nanf** |
| **nanl** | **nanl** | **pdel** | **pfree** | **pool** |
| **printf** | **puts** | **qsort** | **realloc** | **scanf** |
| **snprintf** | **sprintf** | **sscanf** | **stcpm** | **stcpma** |
| **strcasecmp** | **strcat** | **strchr** | **strcmp** | **strcpy** |
| **strcspn** | **strlen** | **strlwr** | **strncasecmp** | **strncat** |
| **strncmp** | **strncpy** | **strpbrk** | **strrchr** | **strrcspn** |
| **strrpbrk** | **strrspn** | **strscan** | **strscntb** | **strspn** |
| **strstr** | **strtod** | **strtof** | **strtok** | **strtol** |
| **strtold** | **strtoll** | **strtoull** | **strupr** | **strxlt** |
| **TPUT** | **TPUT_ASID** | **TPUT_USERID** | **TGET** | **vformat** |
| **vsnprintf** | **vsprintf** | **vsscanf** | **WTO** | **WTOR** |
| **WTP** | | | | |

### 64-Bit Support and ANSI/ISO Conformance

When the **HUGEPTRS** compiler option is specified, the SAS/C Compiler and Library are not completely conformant to the ANSI/ISO standard. The primary reason for this has to do with data types. For example, the C standard specifies that the second argument to the **fseek** function should be a **long**. When the **HUGEPTRS** option is specified, a **long** is a 64-bit integer. However, the actual implementation of **fseek** in SAS/C Release 7.50 continues to expect a 32-bit integer. The prototype for **fseek** therefore specifies **int** for the second argument, which does not conform to the standard. Similar problems can arise when functions accept arguments of type **size_t** or **ptrdiff_t**. For example, the second argument to the **fread** function is defined as having type **size_t**. When **HUGEPTRS** is specified, **size_t** is defined as **unsigned long**, which is a 64-bit type. Nevertheless, the actual implementation of **fread** continues to expect this argument to be a 32-bit integer, requiring a non-standard prototype for **fread**.

One implication of the above is that functions compiled with **HUGEPTRS** should not provide their own prototypes for library functions. Such functions should always include the appropriate header files to obtain the SAS/C prototypes to avoid passing an incorrect argument list.

Note that the SAS/C product remains fully ANSI/ISO compliant for applications which execute in 31-bit mode and do not use the **HUGEPTRS** compiler option.

## New Options

### New Compiler Options

The SAS/C Compiler accepts a number of options enabling you to alter the way code is generated, the way listing files appear, as well as other aspects of the compilation.

This section describes the new compiler options available with Release 7.50 and how they are implemented in various IBM operating systems.

## New Options Summary

The following table contains the long and short forms of the new compiler options. This table is an extension of the table labeled, “Compiler Option Equivalents,” in Chapter 5, “Compiling C Programs.”

**Table 1.1** Compiler Option Equivalents

| Long Form | Short Form |
|---|---|
| **archlevel**(*l**) | **-ar***l** |
| **asynsig** | **-asy** |
| **bfp** | **-mi** |
| **coverage** | **-ec** |
| **c99subset** | **-c9** |
| **hugeptrs** | **-mh** |
| **mpsafe** | **-mp** |
| **stkbelow** | **-mb** |

* Where *l* is one of the letters **a**, **b**, **c**, or **d**.

The following table summarizes all the new compiler options. This table is an extension of the table labeled, “Compiler Options” in Chapter 6, “Compiler Options.”

The first column lists the new options in long form for the IBM 370. Capital letters indicate the abbreviation for the option. The second column indicates the default for each option. For the default value of the option, you are referred to the description of the option later in the chapter. The third column indicates how the option is specified from the UNIX System Services (USS) shell. The fourth column indicates whether the option can be negated. An exclamation point (!) means the option can be negated, and a plus sign (+) means it cannot be negated. The description of the digraph option identifies the negated form of the option. The next three columns represent the environments for which an option is implemented. An asterisk (*) indicates the option affects this environment. The Affects Process column names the process that is affected by the option. The C in the Affects Process column indicates that compilation is affected by the option. An asterisk in the Sys column warns that the form or meaning of the option may differ depending on the environment in which the compiler is running.

*Note:* If you specify contradictory options under MVS batch, the OpenEdition shell, and CMS, the option specified last is used. △

**Table 1.2** Compiler Options

| Option Name | Default | USS | Negation | OS/390 Batch | TSO | CMS | Affects Process | Sys |
|---|---|---|---|---|---|---|---|---|
| **ARChlevel** | **see description** | **-Karchlevel=let** | + | * | * | * | C | |
| **ASYnsig** | **ASYnsig** | **-Kasynsig** | ! | * | * | * | C | |
| **BFp** | **NOBFp** | **-Kbfp** | ! | * | * | * | C | |
| **COVerage** | **NOCOVerage** | **-Kcoverage** | ! | * | * | * | C | |

| Option Name | Default | USS | Negation | OS/390 Batch | TSO | CMS | Affects Process | Sys |
|---|---|---|---|---|---|---|---|---|
| **C99Subset** | **NOC99Subset** | **-Kc99subset** | ! | * | * | * | C | |
| **HUgeptrs** | **NOHUgeptrs** | **-Khugeptrs** | ! | * | * | * | C | |
| **MPsafe** | **MPsafe** | **-Kmpsafe** | ! | * | * | * | C | |
| **STKbelow** | **NOSTKbelow** | **-Kstkbelow** | ! | * | * | * | C | |
| **STMap** | **STMap** | **-Kstmap** | ! | * | * | | * | |

## New Options

**archlevel** (**-Karchlevel=let** under USS)
allows you to request code generation for a specific level of the 390 architecture. By specifying this option, you can exploit newer features of recent processors, but you should be aware that the generated code will fail if you run it on a processor that does not have the indicated feature.

The **archlevel** option specifies an architectural level by a single-letter code. Four codes are supported currently, with the following meanings.

| | |
|---|---|
| **a** | The processor supports the **logical string assist** facility. This facility allows the compiler to generate better code for **string.h** built-in functions such as **strlen**, **strcpy**, and **strcmp**. |
| **b** | The processor supports the immediate and relative instruction set, as well as the compare and move extended facility. This allows the compiler to generate improved code in many areas. |
| **c** | The processor supports the floating-point extensions feature. This feature allows the compiler to generate improved code for floating-point computations, and is required to use IEEE floating-point. Note that use of **archlevel(c)** is advantageous for programs which use the traditional 390 hex format for floating-point as well as for IEEE applications. |
| **d** | The processor supports the z/Architecture. This feature allows the compiler to exploit 64-bit registers for programs which use the long long data type, and is a prerequisite for 64-bit addressing support. |

*Note:* The codes are cumulative so that, for instance, specifying an architecture level of **c** indicates presence of all features defined for levels **a** and **b**,as well as **c**. △

If no architecture level is specified, the compiler assumes that none of the above architectural features can be used. However, if the **bfp** option is specified, an **archlevel** of **c** is assumed by default, and if the **hugeptrs** option is specified, the **archlevel** is assumed to be **d**.

In TSO and under MVS batch, the archlevel option is specified as follows:

```
archlevel(let)
```

For example, the following indicates that the compiler can assume the presence of the floating-point extensions:

```
archlevel(c)
```

**asynsig** (**-Kasynsig** under USS)
specifies that extra code should be generated when necessary to detect asynchronous signals on exit from a function. If the program does not use any

asynchronous signals (such as SIGINT, SIGALRM, or any POSIX signal), you can improve performance slightly by specfying **noasynsig**.

If **noasynsig** is specified but the program responds to asynchronous signals, detection of these signals by the program may be delayed, which causes the handler to be called later than otherwise would be expected.

**bfp** (**-Kbfp** under USS)
specifies that the default floating point format is binary (IEEE). Note that the **BFP** option implies the **ARCHLEVEL(C)** option.

**coverage** (**-Kcoverage** under USS)
activates the COVERAGE feature of the compiler, which provides information on which lines of code written in C were executed at runtime. Use of the coverage option increases code size and execution time in order to support tracking of the executed code. Note that the effective use of the **coverage** option requires the user to provide a **__cvgtrm** routine to output the accumulated coverage data.

**c99subset** (**-Kc99subset** under USS)
enables the following new features of the ISC C99 standard:

- Variable declarations may occur anywhere within a compound block and in the first clause of a FOR statement.
- The treatment of large unsuffixed decimal constants is C99 compliant in the determination of type, for example, **4000000000** is a 64-bit signed integer instead of a 32-bit unsigned integer.
- The predefined identifier **__func__** is made available for each function. When it is referenced, **__func__** is treated as if it were declared at the beginning of the function, for example:

  ```
  static const char __func__[] = "function-name";
  ```

  If the identifier is not used, the declaration will be deleted and no space will be wasted.
- Preprocessor macros may have a variable number of arguments, for example:

  ```
  #define LOGIT(...) fprintf(logfile, __VA_ARGS__)
  LOGIT("x was %d, but y was %d", x, y);
  ```
- **inline** is a keyword. This is equivalent to the SAS/C **__inline** keyword except for the following:
  - Only one definition may occur in a compilation unit.
  - An actual external definition will be created if the function is declared with external linkage.
- **restrict** is a keyword. The optimizer does not use the information provided by the use of this qualifier; consequently, its use will not result in better code generation. The keyword is enabled as a convenience for porting code written for C99 to SAS/C.

**hugeptrs** (**-Khugeptrs** under USS)
specifies that the object code is intended to execute in 64-bit addressing mode. When **hugeptrs** is specified, the default pointer type is **__huge**, and the size of signed and unsigned long data is 8 bytes. Note that the **hugeptrs** option implies the **ARCHLEVEL(D)** option.

Though the **hugeptrs** option is valid under CMS, the object code generated with this option cannot be executed under CMS because of operating system limitations.

**mpsafe** (**-Kmpsafe** under USS)
specifies that extra code should be generated to assure correct behavior when a SAS/C asynchronous signal is detected on a different processor in an MP

configuration than the one executing the SAS/C program. An example would be a user-added asynchronous signal which is generated by a subtask of the SAS/C program. **mpsafe** causes a slight performance penalty in the function epilog, so it should be used only when the object code may be used in the presence of such asynchronous signals.

If **nompsafe** is specified or defaulted, an asynchronous signal generated on another processor may be ignored or may cause an ABEND if it occurs while the function is returning.

**stkbelow** (**-Kstkbelow** under USS)
causes the stack frame for functions to be allocated below the 16M line. If the **_stkabv** external variable has been set to indicate the the program wants to have the stack above the line, but certain functions cannot tolerate this (for example, stack variables will be passed to system services that run only AMODE=24), then **stkbelow** can be specified to force the **auto** variables of such functions to be allocated below the line. For assembler routines, the **STKBELOW=YES** option of the CENTRY macro will accomplish the same result. For best results, use **stkbelow** to compile only those functions that require a stack below the line.

**STMap** (**-Kstmap** under USS
requests that a map of structure elements and their offsets be generated in the cross-reference for each structure tag enclosed. Specifying the **-Kstmap** option implies the **-Kxref** option.

## New Preprocessor Symbols

The following table contains the new option names, symbols, and corresponding values for those new options that are assigned preprocessor symbols by the compiler. This table is an extension of the table labeled "Preprocessor Symbols" in Chapter 6, "Compiler Options."

**Table 1.3** Preprocessor Symbols

| Option | Symbol | Value |
|---|---|---|
| **archlevel unspecified** | **_O_ARCHLEVEL** | 0 |
| **archlevel(a)** | **_O_ARCHLEVEL** | 1 |
| **archlevel(b)** | **_O_ARCHLEVEL** | 2 |
| **archlevel(c)** | **_O_ARCHLEVEL** | 3 |
| **archlevel(d)** | **_O_ARCHLEVEL** | 4 |
| **bfp** | **_O_BFP** | 1 |
| **nobfp** | **_O_BFP** | 0 |
| **c99subset** | **_O_C99SUBSET** | 1 |
| **noc99subset** | **_O_C99SUBSET** | 0 |

| Option | Symbol | Value |
|---|---|---|
| **hugeptrs** | **_O_HUGEPTRS** | 1 |
| **nohugeptrs** | **_O_HUGEPTRS** | 0 |

## New COOL Options

### COOL Options Summary

The following table contains the long and short forms of the new compiler options. This table is an extension of the table labeled, "COOL Options Equivalents," in Chapter 7, "Linking C Programs."

**Table 1.4** COOL Options Equivalents

| Long Form | Short Form |
|---|---|
| **coverage** | none |
| **severe** | **-we** |

The following table lists the new options available for the COOL utility and the systems to which these options apply. This table is an extension of the table labeled, "COOL Options," in Chapter 7, "Linking C Programs."

**Table 1.5** Compiler Options

| Option Name | TSO | CMS | OS/390 Batch | USS |
|---|---|---|---|---|
| **coverage** | X | X | X | X |
| **severe** | X | X | X | X |

### COOL Options

**coverage** (**-Kcoverage** under USS)
activates the COVERAGE feature of the compiler, which provides information on which lines of code written in C were executed at runtime. Use of the coverage option increases code size and execution time in order to support tracking of the executed code. Note that the effective use of the **coverage** option requires the user to provide a **__cvgtrm** routine to output the accumulated coverage data.

**severe** (**-Asevere** under USS)
causes COOL to assign the same level of importance to warnings as it does to errors. If COOL returns a warning, the COOL return code will be the same as if COOL had returned an error; however, the message the user receives for a warning will remain the same as before. It will indicate only that COOL has returned a warning, not an error.

## New DSECT2C Options

In Appendix 1, "The DSECT2C Utility," add the following options to the table labeled, "DSECT2C Options."

**Table 1.6** DSECT2C Options

| Option | Explanation |
|---|---|
| **-mh** | DSECT2C generates output that explicitly specifies the **__near** qualifier when declaring object pointer types. This allows such types to be compiled correctly without depending on the setting of the **HUGEPTRS** compiler option. The **__huge** keyword is always used for 64-bit pointer types. |
| **-mi** | DSECT2C generates output that explicitly specifies the **__hexfmt** keyword on floating point types. This allows such types to be compiled correctly when the BFP compile option is specified. The **__binfmt** keyword is always used for binary floating-point types. |

## New Run-Time Option

The run-time option, **=rsntrace** is new for Release 7.50. Add the following information for **=rsntrace** to Chapter 9, "Run-Time Argument Processing."

Add the following entry to the table labeled, "General Run-Time Options:"

| option | negation | int _options | int _negopts |
|---|---|---|---|
| **=rsntrace** | **=norsntrace** | **_RSNTRACE** | **_NORSNTRACE** |

Add the following entry to the list of options that follows the table labeled, "General Run-Time Options:"

**=rsntrace**
**=r**
requests that library diagnostics include infomation about failing operating system calls. If **=rsntrace** is specified, whenever a library diagnostic message is printed, the library's system macro information (see the *SAS/C Library Reference, Volume 1*) is checked to see whether a system macro failure has been recorded. If so, information about the failure is added to the message. It is possible that the failure is not related to the contents of the message; however, in many cases it will be, and the additional information may be helpful in problem determination. In particular, this option can be useful in debugging applications such as socket programs, which are often sensitive to configuration problems that manifest themselves as unusual or misleading failure conditions.

In the section titled, "Program specification," add the following entry to the list of currently implemented options:

| | |
|---|---|
| **_RSNTRACE** | annotates library messages with system macro information. |

In the section titled, "Program specification," add the following entry to the list of currently implemented options for **_negopts**:

| | |
|---|---|
| **_NORSNTRACE** | does not annotate library messages with system macro information. |

# Release 7.50 Changes to the SAS/C Compiler and Library User's Guide

## IEEE and 64-Bit Updates to the Inline Machine Code Interface

The SAS/C support for IEEE Floating-Point computation and 64-Bit processing requires several changes to the documentation for the inline machine code interface. The following sections contain details about these changes. All of the following section names refer to sections in Chapter 13, "Inline Machine Code Interface."

### Update to Built-in functions

In the section titled "Overview," add the following entry to the table under "Built-in functions:"

| | |
|---|---|
| **`_ospc`** | generate an OS/390 PC instruction |

### Updates to _diag, _cms202 and _ossvc

- The section title should read "_diag, _cms202, _ossvc, and _ospc."
- The first sentence in this section should read:

  Several built-in functions issue specific supervisor call instructions: **`_diag`** , **`_cms202`** , **`_ossvc`**, and **`_ospc`**.

### Updates to Code Macros

- In the section titled, "Code Macros," add the following item to the list of header files:
  - **`<gen164.h>`**
- The first paragraph after the list of header files should read:

  These header files provide appropriate macros for all IBM 370 machine instructions except UPT, SIE, and PC. These three instructions cannot be supported because of conflicts in register use between the instructions and the compiled C code. Note that most uses of the PC instruction in OS/390 can be generated using the **`_ospc`** function.

### Updates to Inline Machine Code Usage Notes

- The second Caution should read:

***CAUTION:***
**Do not use general-purpose registers 4 through 13 with the _ldregs function.** The compiler assigns general-purpose registers 6 through 11 and floating-point registers 4 and 6 to register variables. If your use of registers in **`_ldregs`** conflicts with the compiler's assignment of registers to register variables, the generated code might be less than optimal. General-purpose registers 0 through 3 and 14 through 15 can be used freely, as well as floating-point registers 0 and 2. If the compiler option **`ARCHLEVEL(C)`** is specified, you can use floating-point registers 0 through 7 in inline machine code. △

□ The table labeled, “Registers for Use with the _code_stregs, and _idregs Functions” can be simplified to the following:

**Table 1.7** Registers for Use with the _code_stregs and _idregs Functions

| Type of Register | Register Number | Name of Macro | Bit in Mask |
|---|---|---|---|
| general-purpose | 0-15 | R0-R15 | 0-15 |
| floating-point | 0-15 | F0-F15 | 16-31 |

□ The paragraph at the end of the section should read:

The **`_stregs`** function is defined to return **`int`**. This allows the **`_stregs`** function to be used to store an integer or a pointer from a single register. Three alternate forms of **`_stregs`** are available to store other types of data. The function **`_stfregs`** returns double, the function **`_stpregs`** returns **`__far void *`** and the function **`_stllregs`** returns **`long long`**. Except for the difference in return type, these functions behave the same as **`_stregs`**.

*Note:*

- □ The **`_stfregs`** function stores its return value in the highest floating register in the mask.
- □ The **`_stllregs`** function is only available if the **`ARCHLEVEL(D)`** option has been specified.
- □ The **`_stllregs`** function, through use of a cast, can be used to store a **`__huge`** pointer value as well as a **`long long`**.

△

## Updates to the _bbwd Function

□ The following changes should be made to the **`_bbwd`** function description:

- □ The code example in the “Synopsis” section should be:

```
#include <code.h>
   void _bbwd(unsigned short op, unsigned short target,...);
```

- □ The first paragraph in the “Description” section should read:

  **`_bbwd`** causes the compiler to generate a branch instruction whose target is the previously defined label whose **`_label`** number is specified by **`target`**. The **`op`** argument specifies the first halfword of the instruction to generate. Both arguments to **`_bbwd`** must be compile-time constants. If **`_bbwd`** is used to generate a 6-byte branch instruction (BCTG, BXHG or BXLEG), a third unsigned short argument must be passed to **`_bbwd`** specifying the last 2 bytes of the instruction. For other branch operations, this argument must be omitted.

- □ In the “Description” section, add the following arguments to the list of supported arguments:

  **BCTG**

  **BXHG**

  **BXLEG**

- □ The last paragraph in the “Description” section should read:

  Optimizations, such as branch folding, may cause the instructions generated as a result of **`_bbwd`** to differ from those expected. However, any such optimizations will not change the effects of the instructions. Also, if **ARCHLEVEL(B)** or higher has been specified, the compiler may generate a relative branch instruction equivalent to the branch specified.

## Updates to the _bfwd Function

The following changes should be made to the **_bfwd** function description:

□ The code example in the "Synopsis" section should be:

```
#include <code.h>
   void _bfwd(unsigned short op, unsigned short target,...);
```

□ The first paragraph in the "Description" section should read:

**_bfwd** causes the compiler to generate a branch instruction whose target is the previously defined label whose **_label** number is specified by **target**. The **op** argument specifies the first halfword of the instruction to generate. Both arguments to **_bfwd** must be compile-time constants. If **_bfwd** is used to generate a 6-byte branch instruction (BCTG, BXHG or BXLEG), a third unsigned short argument must be passed to **_bfwd** specifying the last 2 bytes of the instruction. For other branch operations, this argument must be omitted.

□ In the "Description" section, add the following arguments to the list of supported arguments:

**BCTG**

**BXHG**

**BXLEG**

□ The last paragraph in the "Description" section should read:

Optimizations, such as branch folding, may cause the instructions generated as a result of **_bfwd** to differ from those expected. However, any such optimizations will not change the effects of the instructions. Also, if **ARCHEVEL(B)** or higher has been specified, the compiler may generate a relative branch instruction equivalent to the branch specified.

## Updates to the _branch Function

The following changes should be made to the **_branch** function description:

□ The code example in the "Synopsis" section should be:

```
#include <code.h>
void _branch(unsigned mask, unsigned short op, unsigned short target,...);
```

□ Add the following text after the list in the "Description" section:

Note that if **_branch** is used to generate a 6-byte branch instruction (operation code BCTG, BXLEG or BXHG), a third unsigned short argument must be specified to complete the instruction.

## Updates to the _ldregs Function

□ The second paragraph of the "Descriptions" section should read:

The first argument, MASK, is a 32-bit mask. This argument must be a compile-time constant because the compiler has to know which registers are needed. General purpose registers can be specified using the macros R0 through R15, and floating point registers 0 through 15 can be specified using the macros F0 through F15. (Only F0, F2, F4 and F6 can be used if the **ARCHLEVEL** option is not **C** or greater.) Multiple registers can be specified by adding (or logically ORing) the macros, for example, R0+R1 (or R0|R1). Refer to the “Bit Masks for Using Registers” section in the “Inline Machine Code Interface” chapter of the *SAS/C Compiler and Library User’s Guide* for more information on the MASK argument.

□ The third paragraph of the “Descriptions” section should read:

Remaining arguments specify the values to be placed in the registers (low to high) that are specified by MASK. Any C expression that has an integer, pointer, or floating-point type may be used. The number of arguments, excluding MASK, must be equal to the number of one-bits in MASK; that is, you must supply an expression for each register. The type of each expression must be valid for the register in which the value is to be loaded. For example, the effect of attempting to load a pointer into a floating-point register is unpredictable. If the **ARCHLEVEL(D)** option has been specified, you can load an 8-byte value (**long**, **long long**, or **__huge pointer**) into a 64-bit register. If the **ARMODE** option has been specified, you can load a **__far** pointer into a 32-bit general register and the corresponding access register. All other loads of a general register will load only the low-order 32 bits.

□ In the “Cautions” section, replace the text that reads:

You can safely use the following registers:

| | |
|---|---|
| 0 - 3, 14, 15 | are general-purpose registers. |
| 0, 2 | are floating-point registers. |

With text that reads:

You can safely use the following registers:

| | |
|---|---|
| 0 - 3, 14, 15 | are general-purpose registers. |
| 0, 2 | are floating-point registers, if the **ARCHLEVEL** option is less than **C**. |
| 0 - 7 | are floating-point registers, if the **ARCHLEVEL** option is **C** or greater. |

□ Also in the “Cautions” section, replace the text that reads:

In addition, you can use floating-point registers 4 and 6 if they are not currently assigned to a register variable.

With text that reads:

In addition, if the **ARCHLEVEL** specification is less than **C**, you can use floating-point registers 4 and 6 if they are not currently assigned to a register variable.

## New _ospc Function

Add the new function, **_ospc**, to the “Functions” section.

**_ospc**

Generate an MVS PC Instruction

SYNOPSIS

```
#include <svc.h>
void _ospc(void);
```

DESCRIPTION

**_ospc** generates a program call (PC) instruction, specifically, **PC 0(14)**. This PC instruction is used to link to several MVS services. It is the caller’s responsibility to set up arguments for the program call appropriately, using **_ldregs** or inline code functions.

*Note:* The generated code assumes that the program call will not modify registers other than 14 through 1. This function should not be used to invoke system calls which do not honor this restriction. △

CAUTIONS

If you call **_ospc** incorrectly (for instance, if you pass any arguments), the second pass of the compiler produces an error message, sets the MVS return code to 8, and generates an **EX 0,*** instruction for **_ospc**. The generated instruction causes an execute exception (MVS ABEND code 0C3) if it is executed.

PORTABILITY

**_ospc** is not portable.

IMPLEMENTATION

**_ospc** first stores any values currently in use from general-purpose registers 0, 1, 14 and 15 and clears any information the compiler had about the contents of those registers. It assumes these registers may be altered by the program call. If necessary, **_ospc** then issues a SAM31 instruction to exit 64-bit mode. **_ospc** then saves the address of the PC instruction in the C Run-Time Anchor Block (CRAB) as an aid in traceback production in case the PC causes an ABEND. Next, **_ospc** issues the PC instruction. Finally, if the **HUGEPTRS** option was specified, **_ospc** issues the SAM64 instruction to resume 64-bit mode execution.

EXAMPLE

The following macro is a C macro with approximately the same functionality as the assembler macro:

```
STORAGE OBTAIN,LENGTH=len,ADDR=addr,SP=sp,COND=YES
```

```
#include <code.h>
#include <genl370.h>
#include <svc.h>

#define STORAGE_OBTAIN(len,sp,addr) \
    (_ldregs(R0+R14+R15, (len), 16, (sp) << 8), \
     L(14, 0+b(14)), \
     L(14, 772+b(14)), \
     L(14, 160+b(14)), \
     _ospc(), \
     _stregs(R1+R15, (addr)))
```

RELATED FUNCTIONS

**_ldregs**, **_ossvc**, **_stregs**

## Update to the _osarmsvc Function

The description of **_osarmsvc** should read:

**_osarmsvc** generates a supervisor call (SVC) instruction. **_osarmsvc** takes one argument (*n*) that specifies the number of the SVC to generate; *n* is an execution-time constant in the range of 0 to 255. No code is generated to change the addressing mode. The SVC will be issued in AR mode if the compiler **armode** option has been specified, and otherwise will be issued in primary address space mode. If **_osarmsvc** is issued in a program compiled with the **HUGEPTRS** option, execution is switched to 31-bit addressing before the SVC is issued, and restored to 64-bit addressing after the SVC completes.

## Update to the _ossvc Function

The implementation section of the **_ossvc** function should read:

**_ossvc** first stores any values currently in use from general-purpose registers 0, 1, 14, and 15 and clears any information the compiler had about the contents of those registers. It assumes that these registers are, or may be, altered by the SVC. If needed, **_ossvc** then issues a SAC instruction to cancel access-register mode and/or a SAM31 instruction to cancel 64-bit addressing mode. **_ossvc** then saves the address of the SVC instruction in the C Run-Time Anchor Block (CRAB) as an aid to traceback production in case the SVC causes an ABEND. **_ossvc** then issues the requested SVC instruction. Finally, **_ossvc** issues a either a SAC or a SAM64 instruction, or both, to restore the program's original addressing mode.

## Update to the _stregs Function

Add the following paragraph after the fourth paragraph of the description of the **_stregs** function:

Each argument other than the mask to **_stregs** must have a type of pointer. The pointer must address a data type which is appropriate to the register being stored. For instance, it is erroneous to attempt to store a general purpose register via a pointer to double. In access register mode, you can store both a general register and the corresponding access register if the argument is a pointer to a **__far** pointer. Similarly, if **ARCHLEVEL(D)** has been specified you can store all 64-bits of a general purpose register if the corresponding **_stregs** argument is a pointer to **long**, **long long**, or a **__huge** pointer. It is permitted but not recommended for an argument to **_stregs** to have type **void ***. In this case, the amount of data stored is determined by the size of the register.

## Updates for Floating-Point Support

The following sections contain information about specific changes to the *SAS/C Compiler and Library User's Guide* for floating-point support.

### Updates to Numerical Limits

In Chapter 2, "Source Code Conventions," under the section titled, "Numerical Limits," replace the tabled labeled "Integral Type Sizes" with the following table:

**Table 1.8** Integral Type Sizes

| Type | Length in Bytes | Range |
|---|---|---|
| **char** | 1 | 0 to 255 (EBCDIC character set) |
| **signed char** | 1 | -128 to 127 |
| **short** | 2 | -32768 to 32767 |
| **unsigned short** | 2 | 0 to 65535 |
| **int** | 4 | -2147483648 to 2147483647 |
| **unsigned int** | 4 | 0 to 4294967295 |
| **long (NOHUGEPTRS)** | 4 | -2147483648 to 2147483647 |

| Type | Length in Bytes | Range |
|---|---|---|
| **unsigned long (NOHUGEPTRS)** | 4 | 0 to 4294967295 |
| **long (HUGEPTRS)** | 8 | -9223372038854775808 to 9223372036854775807 |
| **unsigned long (HUGEPTRS)** | 8 | 0 to 18446744073709551615 |
| **long long** | 8 | -9223372038854775808 to 9223372036854775807 |
| **unsigned long long** | 8 | 0 to 18446744073709551615 |

In Chapter 2, “Source Code Conventions,” under the section titled, “Numerical Limits,” replace the tabled labeled “Float and Double Type Sizes” with the following table:

**Table 1.9** Float and Double Type Sizes

| Type | Length in Bytes | Range |
|---|---|---|
| **float** | 4 | +/-5.4E-70 to +/-7.2E75 (__hexfmt)<br>+/-1.4E-45 to +/-3.4E38 (__binfmt) |
| **double** | 8 | +/-5.4E-70 to +/-7.2E75 (__hexfmt)<br>+/-4.9E-324 to +/-1.8E308 (__binfmt) |
| **long double** | 8 | +/-5.4E-70 to +/-7.2E75 (__hexfmt)<br>+/-4.9E-324 to +/-1.8E308 (__binfmt) |

## Updates to Language Extensions

In Chapter 2, “Source Code Conventions,” under the section titled, “Language Extensions,” the section titled, “Specifying floating-point constants in hexadecimal,” should read:

SAS/C now supports the C99 hexadecimal format for floating-point numbers. The syntax is

```
0x<hexdecimal-digits>.<hexdecimal digits>P<binary exponent><suffixes>
```

A simple example of a C99 hexadecimal floating point constant is 1.AP-12. This constant has the value of 1+(10/16) times two to the minus twelfth power. The hexadecimal point is optional in the format, and only one of the two sequences of digits shown is required. For instance, 0xABCP14, 0xABC.P14 and 0x.ABCP14 are all valid hexadecimal floating-point constants. The hexadecimal digits and the letter P can appear in either upper or lower case. A suffix of F or L can follow a constant to indicate a **float** or **long double** constant. As a SAS/C extension, a suffix of B or H can also be used to indicate a **__binfmt** or **__hexfmt constant**. You can use both a C99 suffix and a SAS/C suffix on the same constant, for example, 0x0.C04P0FB.

*Note:* If you use both suffixes, the SAS/C suffix must appear after the C99 suffix. △

Additionally, SAS/C continues to support its previous format for specifying floating-point numbers in hexadecimal. This format enables specification of the exact bit pattern of a floating-point constant. Unlike the C99 format, this format can be used to describe NaNs and infinities. The syntax is

```
0.x<hexadecimal digits>[.<suffixes>]
```

The indicated digits are stored exactly as specified in memory. If there are fewer than eight digits (for a float) or fewer than sixteen digits (for a double or long double), the remaining digits are considered to be zero. If any suffixes are present, they must be preceded by a period. A suffix of B or H can be used to specify the floating point format, and one of F or L can be used to specify float or long double. As with the C99 format, if both suffixes are used, the format suffix must come last. If there are no suffixes, the constant is assumed to be of type double, and to have the default format. A simple example of a SAS/C-format floating-point constant is 0.x7ff8.B, which represents the binary floating-point default quiet NaN.

## Using Both IBM and IEEE Floating-Point Formats

In Chapter 2, "Source Code Conventions," under the section titled, "Language Extensions," following the section titled, "The #pragma map statement," add a new section titled "Using both IBM and IEEE floating-point formats" that contains the following text:

Most SAS/C programs will use only one format of floating-point number, either the traditional IBM mainframe format or the IEEE format. The choice between formats is made by specifying or omitting the **`BFP`** option. However, advanced programs and libraries might need to use both formats in a single compilation, or to support callers that use either format. This section describes the rules and special considerations required to use both floating-point formats.

The type modifiers **`__binfmt`** and **`__hexfmt`** can be used in floating-point type specifiers to indicate the floating-point format. Floating-point types without an explicit format specification have the default format, as determined by whether the **`BFP`** option was specified.

Floating-point constants can be specified with an H or B suffix to indicate hexadecimal or binary format respectively. Constants without a suffix are assumed to have the default format. A format suffix can appear in combination with a type suffix (F or L), but the format suffix must always come last.

A floating-point value can be converted from one format to another using a cast, assignment, or initialization. Conversion can also occur when an argument is passed to a function with a prototype, or when a function value is returned. Format conversions do not otherwise occur. For instance, adding a **`__binfmt`** double to a **`__hexfmt`** double is invalid. You should use a cast to clarify such expressions.

Conversions from hexadecimal format to binary format are performed according to the current IEEE rounding mode. Conversions from binary to hexadecimal format are exact for doubles, and for floats they are rounded according to the normal hexadecimal format rules. If the conversion cannot be performed, for example, the source is out of range or a NaN, the results are undefined and no signal or exception is raised.

When a function can be used from both **`BFP`** and **`NOBFP`** programs, its prototype should specify **`__binfmt`** or **`__hexfmt`** for each floating-point argument or for a floating-point return type. This will force a conversion to occur when the function is called from a compilation with the incorrect default.

The SAS/C math library is compatible with both formats. When a mathematical function such as **`exp`** is called, the version of the function used is the one specified by the **`BFP`** or **`NOBFP`** option. This is true whether or not **`math.h`** is included. If **`math.h`** is included, each floating-point function argument will be converted to the default format. If **`math.h`** is not included, it is the programmer's responsibility to pass data of the correct floating-point format. For a few low-level functions that are not particularly meaningful for hexadecimal floating point, such as **`isnan`**, if **`math.h`** is not included, the binary version is always called. Because C99 defines these functions as macros, it is technically undefined what happens if you attempt to use them without including the corresponding header file.

The symbols defined by the **`float.h`** header file always reference the default floating-point format, as determined by the presence or absence of the **`BFP`** option.

## Updates to Floating Point (G.3.6)

In Chapter 2, “Source Code Conventions,” under the section titled, “Implementation-defined Behavior,” add the following updates to the section titled, “Floating Point (G.3.6).

- The second item in the list should read:

  When an integer is converted to a floating-point type that cannot accurately represent every value of the source type (for instance, **`int`** to **`float`** or **`long long`** to **`double`**), the result depends on the floating-point format. For binary floating point, the result is rounded according to the current rounding mode. For hexadecimal floating point, the result is rounded. In the latter case, if the value is exactly halfway between two possible results, the result is the one that is larger in magnitude.
- The third item in the list should read:

  When a floating-point number is converted to a narrower number, the result depends on the format. For binary format, the result is rounded according to the current rounding mode. For hexadecimal floating point, the result is rounded, rounding away from 0 when the wider number is equally distant from the two nearest numbers of the narrower format.

## Update to Registers (G.3.8)

In Chapter 2, “Source Code Conventions,” under the section titled, “Implementation-defined Behavior,” the first sentence in the section titled, “Registers (G.3.8)” should read:

There can be up to six integer or pointer register variables, and up to two floating-point register variables (up to eight if the **`ARCHLEVEL(C)`** compiler option is in effect).

## Update to Library Functions (G.3.14)

In Chapter 2, “Source Code Conventions,” under the section titled, “Implementation-defined Behavior,” under the section titled, “Library Functions (G.3.14),” make the following changes to the list:

- Change the first item in the list to read:

  The null pointer constant to which the macro NULL expands is **`0L`** if **`HUGEPTRS`** is specified. Otherwise, it is **`0`**.
- Change the sixth item in the list to read:

  When the second argument of the **`fmod`** function is 0, **`fmod`** returns 0 for hexadecimal floating point, and a NaN for binary floating point.

## Updates to Arithmetic Data Types

In Chapter 3, “Code Generation Conventions,” in the section titled, “Arithmetic Data Types,” change the table labeled, “Data Type Characteristics,” to the following:

**Table 1.10** Data Type Characteristics

| Type | Length | Alignment | Range |
|---|---|---|---|
| **`char`** | 1 | byte | 0 to 255 (EBCDIC character set) |
| **`signed char`** | 1 | byte | - 128 to 127 |
| **`unsigned char`** | 1 | byte | 0 to 255 (EBCDIC character set) |

| Type | Length | Alignment | Range |
|---|---|---|---|
| **`short`** | 2 | halfword | - 32768 to 32767 |
| **`unsigned short`** | 2 | byte | 0 to 65535 |
| **`int`** | 4 | word | - 2147483648 to 2147483647 |
| **`unsigned int`** | 4 | word | 0 to 4294967295 |
| **`long (NOHUGEPTRS)`** | 4 | word | - 2147483648 to 2147483647 |
| **`unsigned long (NOHUGEPTRS)`** | 4 | word | 0 to 4294967295 |
| **`long (HUGEPTRS)`** | 8 | doubleword | -9223372038854775808 to 9223372036854775807 |
| **`unsigned long (HUGEPTRS)`** | 8 | doubleword | 0 to 18446744073709551615 |
| **`long long`** | 8 | doubleword | -9223372038854775808 to 9223372036854775807 |
| **`unsigned long long`** | 8 | doubleword | 0 to 18446744073709551615 |
| **`float`** | 4 | word | =/ - 5.4E - 79 to =/ - 7.2E75 |
| **`double`** | 8 | doubleword | =/ - 5.4E - 79 to =/ - 7.2E75 |
| **`long double`** | 8 | doubleword | =/ - 5.4E - 79 to =/ - 7.2E75 |

| | |
|---|---|
| **`char`**<br>**`unsigned char`** | defines an 8-bit unsigned integer |
| **`signed char`** | defines an 8-bit signed integer |
| **`short`**<br>**`short int`** | defines a 16-bit signed integer |
| **`unsigned short`**<br>**`unsigned short int`** | defines a 16-bit unsigned integer |
| **`int`** | defines a 32-bit signed integer |
| **`long`**<br>**`long int`** | defines a 32-bit signed integer if the **`HUGEPTRS`** option is not specified, or a 64-bit signed integer if **`HUGEPTRS`** is specified. |
| **`unsigned`**<br>**`unsigned int`** | defines a 32-bit unsigned integer |
| **`unsigned long`**<br>**`unsigned long int`** | defines a 32-bit unsigned integer if the **`HUGEPTRS`** option is not specified, or a 64-bit unsigned integer if **`HUGEPTRS`** is specified. |
| **`unsigned unsigned int`** | defines a 32-bit unsigned integer |
| **`long long`** | defines a 64-bit unsigned integer |
| **`unsigned long long`** | defines a 64-bit unsigned integer |

| Type | Length Alignment Range |
|---|---|
| **__hexfmt float** | defines a 32-bit signed floating-point number in the standard 370 representation, that is, a sign bit, a 7-bit biased hexadecimal exponent, and a 24-bit fractional part. The exponent bias is 64. All constants and results generated by compiled code are normalized (except for constants specified in hexadecimal notation). This representation is equivalent to approximately 6 or 7 decimal digits of precision. |
| **__binfmt float** | defines a 32-bit signed floating-point number in IEEE binary format, that is, a sign-bit, an 8-bit biased binary exponent, and a 23-bit fractional part with an implied leading 1. The exponent bias is 127. See the *IBM 390 Principles of Operation* for details on the representation of NaNs, infinities, denormalized numbers, etc. This representation is equivalent to approximately 6 or 7 decimal digits of precision. |
| **__binfmt double** **__binfmt long double** | defines a 64-bit signed floating-point number in IEEE binary format, that is, a sign bit, an 11-bit biased binary exponent, and a 52-bit fractional part with an implied leading 1. The exponent bias is 1023. See the *IBM 390 Principles of Operation* for details on the representation of NaNs, infinities, denormalized numbers, etc. This representation is equivalent to approximately 16 decimal digits of precision. |
| **__hexfmt double** **__hexfmt long double** | defines a 64-bit signed floating-point number in the standard 370 representation, that is, a sign bit, a 7-bit biased hexadecimal exponent, and a 56-bit fractional part. The exponent bias is 64. All constants and results generated by compiled code are normalized (except for constants specified in hexadecimal notation). This representation is equivalent to approximately 16 or 17 decimal digits of precision |

In the same section, the two paragraphs following the table labeled, "Data Type Characteristics," should read:

Note that in contrast to the signed integer representations, negative floating-point numbers are not represented in two's complement notations; positive and negative numbers differ only in the sign bit.

In both floating-point formats, there are multiple representations of 0; in hexadecimal floating point, any value with a 0 fractional part is treated as zero, regardless of the exponent. Code that checks float or double objects for 0 by means of type punning (that is, examining the objects as if they were a nonfloating-point type such as **int**) might fail when the value is an unusual representation of 0, such as an IEEE negative zero.

## Updates to Arithmetic Exceptions

In Chapter 3, "Code Generation Conventions," under the section titled, "Arithmetic Exceptions," add the following updates:

- The first sentence of the second paragraph should read:

  Hexadecimal floating-point exceptions produce a program interrupt that causes abnormal program termination (program interruption codes 000D or 000F) if no arithmetic signal handler is defined.

- Add the following paragraph at the end of the section:

  Usually, binary floating-point exceptions do not cause program interrupts. The library function **fesettrapenable** can be used to cause certain binary floating-point exceptions to trap with a program interrupt. Any such trap will indicate program interruption code 0007. See the *SAS/C Library Reference, Volume 1* for more information on **fesettrapenable**. See the *IBM Principles of Operation* manual for more information about binary floating-point exception handling.

## Updates to Register Conventions

In Chapter 3, “Code Generation Conventions,” under the section titled, “Register Conventions,” the second item in the list should read:

If the compiler option **ARCHLEVEL(B)** (or greater) is not specified, register 5 is the base register for the current function. If **ARCHLEVEL(B)** is in effect, register 5 is not reserved by the compiler, except during the prolog, and is assigned usage dynamically.

## Updates to the CENTRY Macro

In Chapter 11, “Communication with Assembler Programs,” add the following information to the section titled, “The CENTRY macro.”

Change the form of a call to the CENTRY macro to:

```
label         CENTRY DSA=dsa-size,
                     BASE=base-reg,
                     FNM=function-name,
                     STATIC=NO/YES,
                     INDEP=NO/YES,
                     LASTREG=last-reg,
                     STKBELOW=YES/NO,
                     BFP=YES/NO,
                     HUGEPTR=YES/NO,
                     AFPSAVE=afp-save-area,
                     AFPLAST=last-afp-reg,
                     HGRSAVE=hgr-save-area,
                     HGRLAST=last-hgr-reg
```

Add the following entries to the list of keywords:

STKBELOW=YES/NO
: specifies whether the assembler routine requires its DSA be allocated below the 16-megabyte line. The default is STKBELOW=NO.

BFP=YES/NO
: indicates whether the assembler routine should be marked as using BFP as the default floating-point format. The setting of the BFP keyword matters only if the assembler routine calls a C library function whose behavior depends on how its caller was compiled, such as **printf** or **sqrt**.

HUGEPTR=YES/NO
: indicates whether the assembler routine should be marked as running in 64-bit addressing mode. The setting of the **HUGEPTR** keyword matters only if the assembler routine calls a C library function whose behavior depends on how its caller was compiled, such as **printf** or **strtol**. Note that CENTRY does not change the program’s addressing mode. If your assembler code requires 64-bit addressing, you should change addressing mode yourself after completion of CENTRY.

AFPSAVE=afp-save-area
: specifies the location of a save area where non-volatile auxiliary floating point registers used by the assembler routine are to be stored. (These are floating point registers 8 through 15.) If your assembler code modifies any of them, you must code the **AFPSAVE** keyword. The save area must be specified by a symbolic name, and must be allocated on a fullword boundary in the first 1024 bytes of the DSA. (If CENTRY specifies DSA=0, the save area must be located within the CRABTAUT work area.) The area must be large enough to save all the registers specified by

the **AFPLAST** keyword. If you modify any of the non-volatile AFP registers, but do not specify the **AFPSAVE** keyword, ABENDs or floating-point errors are likely in any C code which executes after the assembler routine has returned.

AFPLAST=last-afp-reg
: specifies the last non-volatile AFP register used by the assembler routine. The AFP registers are stored in order, starting with register 8. Thus, if you specify AFPLAST=10, floating point registers 8, 9 and 10 will be saved by CENTRY.

HGRSAVE=hgr-save-area
: specifies the location of a save area where the high-order portions of the 64-bit general registers are to be stored. If your assembler code modifies any of them, you must code the **HGRSAVE** keyword. The save area must be specified by a symbolic name, and must be allocated on a fullword boundary in the DSA. (If CENTRY specifies DSA=0, the save area must be located within the CRABTAUT work area.) The area must be large enough to save the high-order portions of all the registers specified by the **HGRLAST** keyword. If you modify the high-order part of any register, but do not specify the **HGRSAVE** keyword, ABENDs are likely in any C code which executes after the assembler routine has returned.

HGRLAST=last-hgr-reg
: specifies the last general register whose high-order portion is modified by the assembler routine. The high-order parts of the general registers are stored in order, starting with register 14. Thus, if you specify **AFPLAST=6**, the high-order portion of registers 14, 15 and 0 through 6 will be saved by CENTRY.

## Updates to the CEXIT Macro

In Chapter 11, "Communication with Assembler Programs," add the following information to the section titled, "The CEXIT macro."

Change the form of a call to the CEXIT macro to:

```
label        CEXIT RC=return-info/(reg),
                   DSA=YES/0,
                   INDEP=NO/YES,
                   LASTREG=lastreg,
                   MPSAFE=NO/YES,
                   AFPSAVE=afp-save-area,
                   AFPLAST=last-afp-reg,
                   HGRSAVE=hgr-save-area,
                   HGRLAST=last-hgr-reg,
                   HUGERC=NO/YES
```

Add the following entries to the list of keywords:

MPSAFE=NO/YES
: specifies whether or not the application requires support for asynchronous signals in a multiprocessor environment. The default is **MPSAFE=NO**. **MPSAFE=YES** guarantees correct results even in the presence of asynchronous signals, but increases the overhead of CEXIT.

AFPSAVE=afp-save-area
: specifies the location of the save area in which the CENTRY macro stored non-volatile AFP registers. See the description of the **AFPSAVE** keyword of CENTRY for more information.

AFPLAST=last-afp-reg
: specifies the last non-volatile AFP register to be restored by CEXIT. The value must be the same as specified by the corresponding CENTRY call. See the CENTRY macro description for further information.

HGRSAVE=hgr-save-area
: specifies the location of the save area in which the CENTRY macro stored the high-order portions of general purpose registers. See the description of the **HGRSAVE** keyword of CENTRY for more information.

HGRLAST=last-hgr-reg
: specifies the last general register whose high-order portion is to be restored by CEXIT. The value must be the same as specified by the corresponding CENTRY call. See the CENTRY macro description for further information.

HUGERC=NO/YES
: indicates whether the RC value is a huge (64-bit) pointer. If **HUGERC=YES** is specified, the **HGRSAVE** and **HGRLAST** keywords must also be specified.

*Note:* The CEXIT macro does not change addressing modes. CEXIT cannot be executed in 64-bit addressing mode. It is the responsibility of the assembler routine to return to 31-bit addressing before issuing the CEXIT macro. Failure to observe this restriction is likely to cause difficult to diagnose ABENDs. △

## Updates to the AR370 Archive Utility

### Updates to AR370 Command Modifiers

In Appendix 2, "The AR370 Archive Utility," in the section titled, "Combinations of command and command modifier charecters," update the table labeled "Command and Command Modifier Combinations" with the **-y** command modifier as indicated below:

**Table 1.11** Command and Command Modifier Combinations

| Command | Accepted Modifiers and Commands |
|---|---|
| **d** | **e, f, j, q, t, v, y** |
| **m** | **e, f, j, q, t, v, y** and **a \| b** |
| **r** | **e, f, j, q, t, v, y** and **a \| b** |
| **t** | **d, e, f, j, m, r, v, x, y** |
| **x** | **e, f, j, t, v, y** |

### Update to the AR370 INCLUDE Statements

In the section titled "AR370 INCLUDE Statements,"add the following caution at the end of the text.

***CAUTION:***
**The new member name may be any name up to a maximum of 18 characters. It is strongly recommended that the name be a valid MVS PDS member name or data set name. Other names may be difficult to manipulate with the AR370 utility.** △

### Update to Optional Modifier Characters

In the section titled "Optional Modifier Characters," add the following entry to the list of characters:

| | |
|---|---|
| **w** | Used in conjunction with the **replace** (**r**) command to allow truncation of member names at 18 characters. |

## Update to the SYSARWRK Data Definition

In Appendix 2, "The AR370 Archive Utility," under the section titled, "AR370 JCL requirements," add the following text at the end of the section:

If SYSARWRK is not specified, a temporary work data set will be used that is defined with the minimum size that is allowed. To avoid running out of space in this temporary data set, allocate SYSARWRK with adequate space to handle your data set. SYSARWRK will be used then instead of the default work data set.

## Enhancements to the All-Resident C Programs

In Chapter 10, "All-Resident C Programs," update the the section titled, "Restrictions," with the following information:

- Replace the fourth item in the list of restrictions with text that reads:

  Under an extended architecture system, all-resident programs may not be linked with RMODE(ANY). However, they may be linked RMODE(SPLIT), in which case all of the load module except for certain portions of the run-time library can be loaded above the 16-megabyte line. Note that RMODE(SPLIT) is suitable only for AMODE(31) applications.
- Add the following item to the list of restrictions.

  The use of the **NOSMPXIVEC** link-time (COOL) option is not permitted.

## Update to the _branch, _bbwd, and _bfwd Functions

In Chapter 13, "Inline Machine Code Interface," in the section titled, "Functions," update the CAUTIONS sections of the **_branch**, **_bbwd**, and **_bfwd** functions with the text indicated for each function.

**_branch**

*Note:* The **_branch** function should not be used to branch to a label defined in a different built-in code sequence, especially one in a different block. This action may cause a compiler error or incorrect results at execution time. △

**_bbwd**

*Note:* The **_bbwd** function should not be used to branch to a label defined in a different built-in code sequence, especially one in a different block. This action may cause a compiler error or incorrect results at execution time. △

**_bfwd**

*Note:* The **_bfwd** function should not be used to branch to a label defined in a different built-in code sequence, especially one in a different block. This action may cause a compiler error or incorrect results at execution time. △

## Update to the oeabntrap Function Description

In Chapter 14, "Systems Programming with the SAS/C Compiler," in the section titled, "The SPE Library," replace the **oeabntrap** example with the following example:

```
#include <oespe.h>
#include <unistd.h>
#include <lclib.h>
#include <string.h>
#include <setjump.h>
```

```
#include <signal.h>

jmp_buf ABEND_escape;
    /* where to run to after an ABEND */

static int ABEND_trapped;

void trace_out(char *line) {
    /* this function writes a btrace output line to file
       descriptor 2 */
    write(2, line, strlen(line));
    write(2, "\n", 1);
}

void ABEND_handler(int signum) {
    char buf[60];
    sprintf(buf, "Interrupted by signal %d!\n", signum);
    write(2, buf, strlen(buf));
    btrace(&trace_out);
    longjmp(ABEND_escape, 1);
}

int ptrvalid(int *ptr) {
    /* return whether storage addressed by ptr can be read */
    struct sigaction segv_action, prev_action;
    int ok;
    volatile int value;

    if (ABEND_trapped == 0) {
        oeabntrap(TRAP_AUTO);
          /* possibility of error ignored */
        ABEND_trapped = 1;
   }

    if (setjmp(ABEND_escape) != 0) goto failed;
          /* set up retry from handler */
    segv_action.sa_handler = &ABEND_handler;
    sigemptyset(&segv_action.sa_mask);
    segv_action.sa_flags = 0;
    sigaction(SIGSEGV, &segv_action, &prev_action);
          /* we'll try to access the storage even if
             sigaction fails... */
    value = *ptr;    /* force reference to *ptr */
    ok = 1;          /* it must be valid */
    goto complete;
failed:
    ok = 0;          /* the pointer is no good */
complete:
    sigaction(SIGSEGV, &prev_action, 0);
                     /* restore previous SIGSEGV handling */
    return ok;
}
```

## Updates to the pow Function Example

In Chapter 4, “Optimization,” in the section titled, “Using inline functions to generate optimized code,” replace the example code for the **pow** function with the following code:

```
#include <lcdef.h> ❶
#include <math.h> ❷
#undef pow ❸
#define pow(x, p) power(x, p, isnumconst(p)) ❹

static __inline double power(double a, double p, int p_is_constant)
{

      /* Test the exponent to see if it’s   */
      /*  - a compile-time integer constant */
      /*  - a whole number                  */
      /*  - nonnegative                     */
   if (p_is_constant && (int) p == p && (int) p >= 0) { ❺

      int n = p;

           /* Handle the cases for 0 <= n <= 4 directly. */
   ❻    if (n == 0) return 1.0;
         else if (n == 1) return a;
         else if (n == 2) return a * a;
         else if (n == 3) return a * a * a;
         else if (n == 4) return (a * a) * (a * a);

            /* Handle 5 <= n <= 16 by calling power            */
            /* recursively.                                    */
            /* Note that power is invoked directly, specifying */
            /* 1 as the value of the p_is_constant argument.   */
            /* This is because the isnumconst macro returns    */
            /* "false" for the expressions (n/2) and           */
            /* ((n+1)/2), which would defeat the optimization. */
   ❼    else if (n <= 16)
         return power(a, (double)(n/2), 1) *
                      power(a, (double)((n+1)/2), 1);

            /* Handle n > 16 via a loop.  The loop below       */
            /* calculates (a ** (2 ** x))                      */
            /* for 2 <= x <= n and sums the results for each   */
            /* power of 2 that has the corresponding bit set   */
            /* in n.                                           */
   ❽    else {
         double prod = 1.0;
         for (; n != 0; a *= a, n >>= 1)
            if (n & 1) prod *= a;
         return prod;
      }
   }

      /* Finally, if p is negative or not a whole number, */
      /* call the library pow function.  The pow macro    */
      /* is defeated by surrounding the name "pow" with   */
      /* parentheses.                                     */
   else ❾
      return (pow)(a, p);
}
```

## Update to the description of the long long Data Type

In Chapter 2, “Source Code Conventions,” replace the tenth and eleventh paragraghs of the section titled, “Description of the long long Data Type,” with the following text.

One way in which the SAS/C **`long long`** implementation differs from the C99 standard is in the interpretation of decimal constants. According to C99, a decimal constant that is too large to fit into a **`long`**, such as **`3000000000`** on the mainframe, should be assumed to be a **`long long`**. According to the 1989 ANSI/ISO standard, the type of this constant is **`unsigned long`**. SAS/C continues to treat any such constant within the range of **`unsigned long`** as **`unsigned long`**, to avoid changing the meaning of existing programs. This behavior is compatible with most compilers for other platforms that support a **`long long`** type.

The **`C99SUBSET`** option will cause unsuffixed decimal constants to be interpreted with C99 rules, that is, they can never be unsigned. You can use the standard C language suffixes to control the type of a decimal constant. For example, **`3000000000UL`** will have a type of **`unsigned long`** regardless of compiler options, but **`3000000000`** will have a type that depends on compiler option settings: **`long long (C99SUBSET, NOHUGEPTRS)`**, **`long (HUGEPTRS)`**, or **`unsigned long (default options)`**.

## Update to the rtconst Option Description

In Chapter 7, “Linking C Programs,” under the section titled, “COOL Options,” add the following text to the description of the **`rtconst`** option.

Usually the run-time constant section can be removed without affecting normal program execution. However, a very large function compiled with the **`archlevel(b)`** option (or greater) may require the run-time constant CSECT to be present to execute correctly. COOL will preserve the run-time constant section for a compilation including such a function, even if **`rtconst`** has been specified.

## Update to the enforce, mention, and suppress Option Descriptions

In Chapter 6, “Compiler Options,” in the section titled, “Option Descriptions,” the descriptions of the **`enforce`**, **`mention`**, and **`suppress`** options need to be updated to remove the information stating that “Only warnings in the range 0–199 and 300–499 are affected.”

- Replace the first paragraph of the description of **`enforce`** with the following text:

  treats one or more warning conditions as error conditions. Each warning condition is identified by its associated message number. Conditions whose numbers have been specified are treated as errors, and the compiler return code is set to 12 instead of 4.

- Replace the first paragraph of the description of **`mention`** with the following text:

  specifies that the warnings whose numbers are specified as *n1*, *n2*, and so on, are not to be suppressed. See also **`suppress`**.

- Replace the second paragraph of the description of **`suppress`** with the following text:

Each warning condition is identified by its associated message number, *n*. Conditions whose numbers have been specified are suppressed. No message is generated, and the compiler return code is changed.

## Updates to the #pragma Options

In Chapter 6, "Compiler Options," in the section titled, "The #pragma options statement," add the following options to the list of **#pragma options**:

**armode**

**indep**

**sname(value)**

## Update to the ENXREF COOL Option Description

In Chapter 7, "Linking C Programs," in the section titled, "COOL Options," replace the third paragraph of the **enxref** description with the following text.

You can use the **references** option (**-Areferences** under USS) to modify the behavior of the **sname**, **cid**, or **linkid** cross-reference listing so that unresolved external references are included in the cross-references listing. You must use the **references** option in conjunction with one or more of these listing options: **sname**, **cid**, or **linkid**. For example:

```
enxref(sname references)
```

specifies an **sname** cross-reference listing that includes cross-references for external symbols that are declared but not defined.

## Updates to the COOL Options Tables

### COOL Options (Short Forms)

The following table contains corrections to the table labeled, "COOL Options Equivalents," in Chapter 7, "Linking C Programs."

**Table 1.12** COOL Options Equivalents

| Long Form | Short Form |
|---|---|
| **auto** | **-a** |
| **clet** | **-m** |
| **clet(all)** | **-m** |
| **clet(noex)** | **-mn** |
| **continue** | **-zc** |
| **cxx** | **-cxx** |
| **dupsname** | **-zd** |
| **endisplaylimit** | **-yn*nnn*** |
| **enexit** | **-xt** |
| **enexitdate** (*xxx*) | **-xt*xxx*** |

| Long Form | Short Form |
|---|---|
| `enxref (cid)` | `-xxx` |
| `enxref (linked)` | `-xxe` |
| `enxref (references)` | `-xxy` |
| `enxref (sname)` | `-xxs` |
| `extname` | `-xn` |
| `files`(*xxx*) | `-f`*xxx* |
| `gmap` | `-yg` |
| `inceof` | `-zi` |
| `libe` | `-b` |
| `lineno` | `-l` |
| `list` | `-yl` |
| `noenxref` | `-!xx` |
| `output` *fileid* | `-o`*fileid* |
| `pagesize`(*nn*) | `-s`*nn* |
| `prem` | `-p` |
| `print` | `-h` |
| `prmap` | `-yp` |
| `rtconst` | `-r` |
| `smpjclin` | `-sj` |
| `smponly` | `-sxo` |
| `smpxivec` | `-sx` |
| `term` | `-t` |
| `upper` | `-u` |
| `verbose` | `-zv` |
| `warn` | `-w` |
| `xfnmkeep` | `-xf` |
| `xsymkeep` | `-xe` |

## COOL Options

The following table contains corrections to the table labeled, “COOL Options,” in Chapter 7, “Linking C Programs.”

**Table 1.13** COOL Options

| Option | TSO | CMS | OS/390 Batch | USS |
|---|---|---|---|---|
| `-Agather` | | | | X |
| `-Ainsert` | | | | X |
| `allresident` | X | X | | X |
| `arlib` | X | | | |

| Option | TSO | CMS | OS/390 Batch | USS |
|---|---|---|---|---|
| `auto` | X | X | X | |
| `-Bep` | | | | X |
| `-Blib` | | | | X |
| `cics` | X | X | | X |
| `cicsvse` | X | X | | X |
| `clet` | X | X | X | X |
| `clet(all)` | X | X | X | X |
| `clet(noex)` | X | X | X | X |
| `continue` | X | X | X | X |
| `cxx` | X | X | | |
| `dupsname` | X | X | X | X |
| `endisplaylimit` | X | X | X | X |
| `enexit` | X | X | X | X |
| `enexitdata` | X | X | X | X |
| `entry` | X | | | |
| `enxref` | X | X | X | X |
| `extname` | X | X | X | X |
| `files` | | | X | |
| `genmod` | | X | | |
| `global` | | X | | |
| `gmap` | X | X | X | X |
| `gos` | X | X | | |
| `inceof` | X | X | X | X |
| `-l` | | | | X |
| `-L` | | | | X |
| `lib` | X | | | |
| `libe` | | X | | |
| `lineno` | X | X | X | X |
| `list` | X | X | X | X |
| `lked` | | X | | |
| `lkedname` | X | | X | |
| `load` | X | | | X |
| `loadlib` | X | | | |
| `nocool` | X | | X | |
| `output` | | X | | |
| `pagesize` | X | X | X | X |
| `prem` | X | X | X | X |

| Option | TSO | CMS | OS/390 Batch | USS |
|---|---|---|---|---|
| **print** | X | X | | X |
| **prmap** | X | X | X | X |
| **rtconst** | X | X | X | X |
| **smpjclin** | X | X | X | X |
| **smponly** | X | X | X | X |
| **smpxivec** | X | X | X | X |
| **spe** | X | X | | X |
| **start** | | X | | |
| **term** | X | X | X | |
| **upper** | X | X | X | X |
| **verbose** | X | X | X | X |
| **warn** | X | X | X | X |
| **xfnmkeep** | X | X | X | X |
| **xsymkeep** | X | X | X | X |

## Update to the _O_SNAME Symbol

In Chapter 6, "Compiler Options," in the section titled, "Preprocessor Symbols," add the following text to the explanation of the **`_O_SNAME`** option that follows the table labeled, "Preprocessor Symbols."

If you use the compiler **`sname`** option to set the **`_O_SNAME`** preprocessor symbol, it has an affect similar to using a **`#define`** preprocessor directive to define the symbolic name **`_O_SNAME`**. You cannot actually use a **`#define`** directive to define **`_O_SNAME`**. You must use the compiler **`sname`** option. Attempting to set the **`_O_SNAME`** symbol with a **`#define`** preprocessor directive will result in an LSCC133 error.

## Update to the <resident.h> Documentation

In Chapter 10, "All-Resident C Programs," in the section titled, "Using **`<resident.h>`**," remove the note at the end of the section that reads,

*Note:* The **`dollars`** compiler option must be used when compiling a C++ source file that contains **`<resident.h>`**. △

and replace it with the following text.

The **`dollars`** compiler option is no longer required when compiling a C++ source file that includes **`<resident.h>`**.

## Updates to the SPE Functions

In Chapter 14, "Systems Programming with the SAS/C Compiler," in the section titled "The SPE Library," add the following functions to the table of SPE functions.

| | | |
|---|---|---|
| **gethostbyaddr*** | **gethostbyname*** | **htonl** |
| **htons** | **IARV64** | **inet_addr** |
| **inet_lnaof** | **inet_makeaddr** | **inet_netof** |
| **inet_network** | **inet_ntoa** | **ntohl** |
| **ntohs** | **osdltok** | **osgttok** |
| **ossttok** | **WTP** | |

After the section titled “HFS Access,” create a new section titled “TCP/IP Access,” and move the last paragraph of the section titled “HFS Access” into the new section. The paragraph to be moved reads:

SPE supports access to USS integrated sockets using the standard UNIX socket interface functions such as **socket**, **accept**, **read**, **write**, and so on. Note that only integrated sockets can be accessed. Starting with z/OS 1.2, the TCP/IP resolver functions; **gethostbyname** and **gethostbyaddr** are supported. However, other resolver type functions such as **res_init** and **getservbyname** are not supported because USS does not provide system calls for these functions.

CHAPTER
2

# SAS/C Library Changes in Release 7.50

## Introduction

This chapter provides a complete description of the changes and enhancements to the *SAS/C Library Reference, Volume 1* and *SAS/C Library Reference, Volume 2* for Release 7.50.

## Release 7.50 Enhancements to the SAS/C Library

### Floating-Point Support

SAS/C 7.50 has enhanced its support of floating-point in two ways. It has added support for IEEE (binary) floating-point arithmetic, and it has implemented new floating-point functions from the C99 standard. An overview of IEEE floating-point support can be found in “Floating-Point Changes for SAS/C Release 7.50” on page 2.

Floating-point support affects a number of different areas in the run-time library, which are documented in a number of sections of this chapter. Sections of this chapter relevant to floating-point processing are as follows:

- “Updates to Mathematical Functions” on page 43 gives an overview of using the mathematical library in release 7.50, including ways in which the processing of functions from previous releases has changed.
- “New Math Functions” on page 53 describes the mathematical functions that have been added to SAS/C for this release.
- “New nan, nanf, and nanl Functions” on page 98 describes new functions that can be used to generate IEEE NaNs (*not a number*) values.
- “Updates to the SIGFPE Signal” on page 100 describes changes to the processing of the SIGFPE signal to support IEEE and the C99 math library.
- “Update to float.h Header File” on page 95 lists the SAS/C 7.50 **`float.h`** header file, which has been enhanced to support IEEE floating-point.
- “Updates to the fprintf Function” on page 120 and “Updates to the fscanf Function” on page 124 describe new format modifiers for the **`printf`** and **`scanf`** families of functions to allow a program to specify whether floating-point arguments are hexadecimnal or binary. These sections also describe changes and enhancements to the output of the **`printf`** family, and the valid input to the **`scanf`** family.
- “Updates to the strtod Function” on page 129 describes changes to the processing of strtod and related functions to support IEEE floating-point.
- “Updates to Function Categories” on page 173 gives information on how various library functions have changed due to IEEE support.
- “Updates to Signal-Handling Functions” on page 177 describes the new SIGBFPE signal, which has been added for errors occurring in IEEE computations.

## 64-Bit Support

SAS/C 7.50 supports programs which execute in the z/Architecture 64-bit addressing mode. An overview of this support can be found in “64-Bit Support” on page 5.

64-bit support affects a number of different areas in the run-time library, which are documented in a number of sections of this chapter. Sections of this chapter relevant to floating-point processing are as follows:

- “New Multiple Heap Functions” on page 102 describes new functionality in heap management to support allocation of 64-bit addressable memory.
- “Huge Pointer Functions” on page 104 describes functions that have been enhanced to support 64-bit pointers.
- “Updates to the fprintf Function” on page 120 and “Updates to the fscanf Function” on page 124 describe new format modifiers for the **`printf`** and **`scanf`** families of functions to allow a program to specify whether pointer arguments are 31-bit or 64-bit.
- “New Library Functions” on page 130 describes the IARV64 function, which allows 64-bit addressible memory to be allocated directly from the operating system.

# Updates to Mathematical Functions

In Chapter 2, “Function Categories,” replace the section titled, “Mathematical Functions,”with the following text.

The SAS/C library of mathematical functions includes the full complement of math functions from the ANSI/ISO C standard, plus additional functions commonly implemented on UNIX systems. It also includes the math functions defined by the ISO C99 standard. All functions are provided for both hexadecimal and binary floating point inputs. Some functions are not applicable to hexadecimal floating point, and may return meaningless results when called in that environment.

When a math function is called, the version called is determined by the manner in which the calling function was compiled. If the **`bfp`** option was used, the IEEE function is called; otherwise, the traditional mainframe function is called. If necessary, the argument is converted to the proper format first, assuming an appropriate header file was included.

## Math Header Files

The following header files are useful to users of the mathematical library:

**`fenv.h`**
defines symbols, types, and functions supporting inspection and modification of the floating-point environment. These facilities are generally useful only with binary floating point. This is a standard C99 header file.

**`fenvtrap.h`**
defines functions supporting control of floating-point trapping. These facilities are generally useful only with binary floating point.

**`float.h`**
defines preprocessor symbols specifying the characteristics of System/390 floating point. This is an ANSI/ISO standard header file.

**`lcfloat.h`**
defines non-standard preprocessor symbols defining constants, such as infinities and NaNs, of various types.

**`lcmath.h`**
defines non-standard mathematical preprocessor symbols and library functions.

**`math.h`**
defines preprocessor symbols and functions supporting the standard mathematical library. The is a standard ANSI/ISO header file. It includes symbols and function prototypes from C99. These can be removed by defining the preprocessor symbol **`_SASC_HIDE_C99MATHLIB`** before including **`math.h`**.

**`tgmath.h`**
defines macro versions of most of the mathematical functions. The macros are *type-generic*, that is, the same function name can be used for **`float`**, **`double`**, or **`long double`** arguments. See "Float and Long Double Functions" on page 53 for further information.

## Preprocessor Symbols

The following lists contain the preprocessor symbols defined by the math-related header files.

The **`fenv.h`** header file contains the following preprocessor symbols:

Exception bits

**`FE_INVALID`**

**`FE_DIVBYZERO`**

**FE_OVERFLOW**

**FE_UNDERFLOW**

**FE_INEXACT**

**FE_ALL_EXCEPT**

Rounding modes

**FE_TONEAREST**

**FE_TOWARDZERO**

**FE_UPWARD**

**FE_DOWNWARD**

Default floating point environment

**FE_DFL_ENV**

*Note:* If the **bfp** option is not used, **FE_DFL_ENV** is the only symbol that is defined. △

The **float.h** header file contains the following preprocessor symbols:

**FLT_EVAL_METHOD**

**FLT_RADIX**

**FLT_ROUNDS**

**FLT_MANT_DIG**

**FLT_DIG**

**FLT_MIN_EXP**

**FLT_MIN_10_EXP**

**FLT_MAX_EXP**

**FLT_MAX_10_EXP**

**FLT_MAX**

**FLT_EPSILON**

**FLT_MIN**

**DBL_MANT_DIG**

**DBL_DIG**

**DBL_MIN_EXP**

**DBL_MIN_10_EXP**

**DBL_MAX_EXP**

**DBL_MAX_10_EXP**

**DBL_MAX**

**DBL_EPSILON**

**DBL_MIN**

**LDBL_MANT_DIG**

**LDBL_DIG**

**LDBL_MIN_EXP**

**LDBL_MIN_10_EXP**

**LDBL_MAX_EXP**

**LDBL_MAX_10_EXP**

**LDBL_MAX**

**LDBL_EPSILON**

**LDBL_MIN**

*Note:* See the C standards documents for details on the meanings of these symbols. △

The **lcfloat.h** header file contains the following preprocessor symbols:

**FLT_INFINITY**
specifies a **float** infinity.

**FLT_QNAN**
specifies a **float** quiet not-a-number (NaN).

**FLT_SNAN**
specifies a **float** signaling NaN.

**DBL_INFINITY**
specifies a **double** infinity.

**DBL_QNAN**
specifies a **double** quiet NaN.

**DBL_SNAN**
specifies a **double** signaling NaN.

**LDBL_INFINITY**
specifies a **long double** infinity.

**LDBL_QNAN**
specifies a **long double** quiet NaN.

**LDBL_SNAN**
specifies a **long double** signaling NaN.

*Note:* Since hexadecimal floating-point does not support NaNs or infinities, these symbols have little utility in a hexadecimal floating-point environment. △

The **lcmath.h** header file contains the following constant representations:

**Table 2.1** Constant Values Declared in lcmath.h

| **Constant** | **Representation** |
|---|---|
| DOMAIN | matherr exception code - domain error |
| SING | exception - singularity |
| OVERFLOW | exception - overflow |
| UNDERFLOW | exception - underflow |
| TLOSS | exception - total loss of significance |
| PLOSS | exception - partial loss of significance |
| M_PI | The constant $\pi$ |
| M_PI_2 | $\pi/2$ |
| M_PI_4 | $\pi/4$ |
| M_1_PI | $1/\pi$ |
| M_2_PI | $2/\pi$ |
| M_E | The constant e |
| HUGE | The largest finite double |
| TINY | The smallest non-zero positive double |

| Constant | Representation |
|---|---|
| LOGHUGE | log(HUGE) |
| LOGTINY | log(TINY) |

The **`math.h`** header file contains the following preprocessor symbols:

**`HUGE_VALF`**
the largest **`float`**, possibly infinite

**`HUGE_VAL`**
the largest **`double`**, possibly infinite

**`HUGE_VALL`**
the largest **`long double`**, possibly infinite

**`INFINITY`**
positive infinity (compile-time error if not **`BFP`**)

**`NAN`**
a float quiet NaN (not defined for HFP)

**`FP_ZERO`**
number classification macro

**`FP_NORMAL`**
number classification macro

**`FP_SUBNORMAL`**
number classification macro

**`FP_INFINITE`**
number classification macro

**`FP_NAN`**
number classification macro

**`FP_FAST_FMA`**
fast multiply and add double support - **`BFP`** only

**`FP_FAST_FMAF`**
fast multiply and add float support - **`BFP`** only

**`FP_FAST_FMAL`**
fast multiply and add long double support - **`BFP`** only

**`FP_ILOGB0`**
result of **`ilogb(0)`**

**`FP_ILOGBNAN`**
result of **`ilogb(NaN)`**

**`MATH_ERRNO`**
math function error handling constant

**`MATH_ERREXCEPT`**
math function error handling constant

**`math_errhandling`**
math function error handling constant

## Type Definitions

The math header files include definitions of the following types:

```
<fenv.h>
    fenv_t         /* represents a floating-point environment */
    fexcept_t      /* represents floating-point exception information */

<lcmath.h>
    struct exception  /* A structure containing error information */

<math.h>
    float_t        /* effective type of float (see C99) */
    double_t       /* effective type of double (see C99) */
```

## Function Categories

Because of the size of the math library, it is useful to divide it into several sublibraries of functions with similar characteristics. The categories and the functions they contain are listed below. Each function name may be followed by a notation indicating whether the function is defined for arguments of type **float** or **long double**. The notations are as follows:

| | |
|---|---|
| (f) | A **variant** function accepts **float** arguments. The name is formed by adding an **f** to the end of the base function name. |
| (g) | A type-generic version of the function is defined in **tgmath.h.** |
| (l) | A **variant** function accepts long double arguments. The name is formed by adding an **l** to the end of the base function name. |
| (m) | The function is a macro, and accepts arguments of any floating-point type. |

## Floating-point Environment Functions

These functions enable you to access and manipulate the floating- point environment, such as exception flags and rounding modes. In general, these functions are useful only when IEEE floating-point is used. If you do not use the **bfp** option in your compilation, these functions have no effect and do not return meaningful results.

Any compilation which uses these functions must include a pragma of the following form

```
    #pragma STDC FENV_ACCESS ON
```

in an enclosing scope. If this pragma is not used, the effects of accessing or modifying the floating-point environment via these functions is undefined. The FENV_ACCESS pragma may reduce optimization, and therefore should be used only when necessary.

These functions are defined in the header file **fenv.h** except for **fegettrapenable** and **fesettrapenable**, which are defined in **fenvtrap.h**. None of these functions should be used without including the appropriate header file.

The floating-point environment functions are:

| | |
|---|---|
| **feclearexcept** | clear floating-point exception flags |
| **fegetenv** | extract the floating-point environment |
| **fegetexceptflag** | extract floating-point exception flags |
| **fegetround** | get the floating-point rounding mode |
| **fegettrapenable** | get the floating-point trap status |

| | |
|---|---|
| **`feholdexcept`** | clear and hold floating-point exceptions |
| **`feraiseexcept`** | raise a floating-point exception |
| **`fesetenv`** | modify the floating-point environemnt |
| **`fesetexceptflag`** | set floating-point exception flags |
| **`fesetround`** | set the floating-point rounding mode |
| **`fesettrapenable`** | set the floating-point trap status |
| **`fetestexcept`** | test for floating-point exceptions |
| **`feupdateenv`** | update the floating-point environment |

## Type Classification Functions

These functions are defined in the header file **`math.h`**. With the exception of **`copysign`**, all of these functions are defined as macros that accept any floating-point argument type. These functions have limited utility with traditional mainframe floating point.

The type classification functions are:

| | |
|---|---|
| **`copysign (f,g,l)`** | propagate sign from one value to another |
| **`fpclassify (m)`** | classify a floating-point value |
| **`isfinite (m)`** | test for finite floating-point value |
| **`isinf (m)`** | test for infinite floating-point value |
| **`isnan (m)`** | test for NaN |
| **`isnormal (m)`** | test for normal (non-zero, normalized) floating-point value |
| **`signbit (m)`** | return the sign of a floating-point number |

## Comparison Functions

These functions are defined in the header file **`math.h`**. They are defined as generic macros, and they take any floating-point type for arguments. They differ from the standard C comparison operators because they do not raise the invalid floating-point exception if an argument is a NaN. Thus, for traditional mainframe floating-point, which has no NaNs, they are completely equivalent to the standard operators. Because these functions are implemented as macros, it is necessary to include **`math.h`** to use them.

The comparison functions are:

**`isgreater (m)`**
compare for greater than

**`isgreaterequal (m)`**
compare for greater than or equal

**`isless (m)`**
compare for less than

**`islessequal (m)`**
compare for less than or equal

**`islessgreater (m)`**

compare for less or greater than

**isunordered (m)**
test for unordered

## Low-level Functions

These functions are defined in **math.h** or **lcmath.h**. Unlike the transcendental functions, these functions perform low-level floating-point operations that are often closely connected with their representation in memory. All of these functions are defined in the C99 standard except for **_ldexp**, whose prototype is in the non-standard header file **lcmath.h**.

The low-level functions are:

| | |
|---|---|
| **_ldexp** | fast ldexp implementation |
| **ceil (f,g,l)** | floating-point ceiling |
| **fabs (f,g,l)** | floating-point absolute value |
| **fdim (f,g,l)** | floating-point positive difference |
| **floor (f,g,l)** | floating-point floor |
| **fma (f,g,l)** | floating-point multiply and add |
| **fmax (f,g,l)** | floating-point maximum |
| **fmin (f,g,l)** | floating-point minimum |
| **fmod (f,g,l)** | floating point modulus |
| **frexp (f,g,l)** | split into fraction and exponent |
| **ilogb (f,g,l)** | integer floating-point exponent |
| **ldexp (f,g,l)** | scale by power of 2 |
| **llrint (f,g,l)** | convert floating-point to long long |
| **llround (f,g,l)** | round floating-point to long long |
| **logb (f,g,l)** | floating-point exponent |
| **lrint (f,g,l)** | convert floating-point to long |
| **lround (f,g,l)** | round floating-point to long |
| **modf (f,l)** | split into integer and fraction |
| **nearbyint (f,g,l)** | return nearest integer |
| **nextafter (f,g,l)** | return next floating-point value |
| **nexttoward (f,g,l)** | return next floating-point value |
| **remainder (f,g,l)** | floating-point remainder |
| **remquo (f,g,l)** | floating-point remainder and partial quotient |

| | |
|---|---|
| **`rint (f,g,l)`** | convert to floating-point integer |
| **`round (f,g,l)`** | round to floating-point integer |
| **`scalbln (f,g,l)`** | scale by power of radix (long exponent) |
| **`scalbn (f,g,l)`** | scale by power of radix (int exponent) |
| **`trunc (f,g,l)`** | truncate to floating-point integer |

## Transcendental Functions

These functions are the functions traditionally known as math functions. They have mathematical definitions with no direct relationship to their hardware representations.

Most of these functions are defined in the C99 standard. The functions, **`gamma`**, **`j0`**, **`j1`**, **`jn`**, **`y0`**, **`y1`**, and **`yn`** are non-standard functions frequently implemented in UNIX systems. The prototypes for these functions are in the non-standard header file **`lcmath.h`**.

| | |
|---|---|
| **`acos (f,g,l)`** | arc cosine |
| **`acosh (f,g,l)`** | arc hyperbolic cosine |
| **`asin (f,g,l)`** | arc sine |
| **`asinh (f,g,l)`** | arc hyperbolic sine |
| **`atan (f,g,l)`** | arc tangent |
| **`atan2 (f,g,l)`** | arc tangent of quotient |
| **`atanh (f,g,l)`** | arc hyperbolic tangent |
| **`cbrt (f,g,l)`** | cube root |
| **`cos (f,g,l)`** | cosine |
| **`cosh (f,g,l)`** | hyperbolic cosine |
| **`erf (f,g,l)`** | error function |
| **`erfc (f,g,l)`** | complementary error function |
| **`exp (f,g,l)`** | exponential |
| **`exp2 (f,g,l)`** | binary exponential (2 ** x) |
| **`expm1 (f,g,l)`** | exponential minus 1 |
| **`gamma`** | log gamma function |
| **`hypot (f,g,l)`** | hypoteneuse |
| **`j0`** | Bessel function of the first kind, order **`0`** |
| **`j1`** | Bessel function of the first kind, order **`1`** |
| **`jn`** | Bessel function of the first kind, order **`n`** |
| **`lgamma (f,g,l)`** | log gamma function |
| **`log (f,g,l)`** | logarithm base e |
| **`log10 (f,g,l)`** | logarithm base 10 |

| | |
|---|---|
| **`log1p (f,g,l)`** | logarithm of argument plus 1 |
| **`log2 (f,g,l)`** | logarithm base 2 |
| **`pow (f,g,l)`** | power function |
| **`sin (f,g,l)`** | sine |
| **`sinh (f,g,l)`** | hyperbolic sine |
| **`sqrt (f,g,l)`** | square root |
| **`tan (f,g,l)`** | tangent |
| **`tanh (f,g,l)`** | hyperbolic tangent |
| **`tgamma (f,g,l)`** | gamma function |
| **`y0`** | Bessel function of the second kind, order **`0`** |
| **`y1`** | Bessel function of the second kind, order **`1`** |
| **`yn`** | Bessel function of the second kind, order **`n`** |

## Math Function Error Handling

Most of the transcendental math functions are subject to errors. Errors may be broadly characterized as domain errors, where a function is called in a way that is not mathematically meaningful, for example, **`sqrt(-5.0)`**, and range errors, where the function result cannot be represented, for example, **`exp(10000.0)`**. The transcendental math library handles these errors in a uniform way that the following list provides details for.

- The transcendental functions will not trap, even if **`fesettrapenable`** has been used to enable trapping. When IEEE floating-point is used, the function **`feupdateenv`** can be used after a math function call to cause error trapping if that is what you want.
- When hexadecimal floating point is used, an error in a math function is indicated by setting **`errno`**. The function return value will depend on the function and the error, but will often be **`HUGE_VAL`** or **`-HUGE_VAL`**.
- When binary floating point is used, an error in a transcendental math function is indicated by setting the IEEE status flags appropriately. **`errno`** is also set. The function return value will depend on the function, the error and the rounding mode, and will often be a NaN or an infinity.
- If an error occurs in a transcendental math function, a library diagnostic message will generally be written to **`stderr`**. This can be controlled by defining an exit routine which can optionally suppress the message and change the function return value. The exit routine is called **`_matherr`** for hexadecimal floating point, and **`_matherb`** for binary floating point. See the descriptions of these functions in *SAS/C Library Reference, Volume 1* for more details.
- Passing a NaN to a math function will generally cause a NaN to be returned. No diagnostic message will be written in this case.
- An IEEE mathematical function may set the IEEE inexact bit, but this is not considered an error. The bit may be set even when the result is exact, for example, **`for log10(1000.0)`**.

The above rules apply specifically to the transcendental functions. The low-level functions are oriented towards performance rather than error detection, and may not abide by all these rules. In particular, a low-level function might trap, might fail without producing a diagnostic or setting errno, and might produce a diagnostic without calling **`_matherr`** or **`_matherb`**. For IEEE floating-point, these low-level functions will set the floating-point status bits correctly, even when they do not write a diagnostic or set **`errno`**.

## Float and Long Double Functions

For most of the functions discussed above, the most commonly used form is one that accepts double arguments and returns double results. Depending on the function, the C99 standard defines two different ways that these functions may be generalized to accept **`float`** or **`long double`** arguments.

Some functions are defined as macros, which can accept any floating-point type. For example,

```
float fl;
double d;
long double ld;

if (isnormal(fl) /* float argument */ &&
    isnormal(d)  /* double argument */ &&
    isnormal(ld)) /* long double argument */ ...
```

The macros produce undefined results if they are passed an argument that does not have floating-point type. A call such as **`isnormal(4)`** is not valid.

The remaining functions are defined as families of functions. They comprise a base function accepting double arguments and related functions accepting float arguments and long double arguments. For example,

```
ld = asinf(fl) /* float argument */ +
     acos(d)   /* double argument */ +
     atan2l(ld, ld+1);  /* long double arguments */
```

Because these are functions defined by prototypes in **`math.h`**, they can be called with arguments which fail to match the expected type; such arguments will be converted, or diagnosed as incorrect if conversion is impossible. For example, the following expression

```
sqrt(fl) /* argument converted to double */ *
cbrtf(6) /* argument converted to float */
```

is valid and well-defined.

Because use of the traditional function names without any suffixes can add clarity and readability to code, C99 also added the **`tgmath.h`** header file. When this header file is included, most of the mathematical functions are redefined as type-generic macros, which select the proper function based on the type of their arguments. Thus, if **`tgmath.h`** is included, the expression **`acos(f)`** will invoke the float arc-cosine function, **`acosf`**, without converting the argument to double.

Note that, in many cases, the float and long double versions of a function can call the double version. If a call to a **`float`** or **`long double`** function version is incorrect, the generated diagnostic might reference the double function name, depending on where the error was detected.

## New Math Functions

Add the following function descriptions to Chapter 6, “Function Descriptions” in SAS/C Library Reference, Volume 1.

The C99 portability term indicates that the function is defined by C99, not by C89.

## acosh

**Compute the inverse hyperbolic cosine**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

double acosh(double x);
float acoshf(float x);
long double acoshl(long double x);
```

### DESCRIPTION

**acosh** computes the inverse hyperbolic cosine of the argument **x**, expressed by the following relation:

$$y = \cosh^{-1}(x)$$

**x** must not be less than 1.0.

The function name **acoshf** should be used for **float** arguments, and **acoshl** for **long double** arguments. Alternately, if the header file **tgmath.h** is included, **acosh** may be used with any numeric argument type.

### RETURN VALUE

**acosh** returns the inverse hyperbolic cosine of the argument **x**, provided that this value is defined and expressible. The value returned will be positive or zero, except in the case of a domain error.

### DIAGNOSTICS

An error message is written to the standard error file (**stderr**) by the runtime library if **x** is less than 1.0. In this case, **acosh** returns 0.0 for **HFP**, or a NaN for **BFP**.

### RELATED FUNCTIONS

**asinh**, **atanh**, **_matherb**, **_matherr**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## asinh

**Compute the inverse hyperbolic sine**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

double asinh(double x);
float asinhf(float x);
long double asinhl(long double x);
```

### DESCRIPTION

**`asinh`** computes the inverse hyperbolic sine of the argument **`x`**, expressed by the following relation:

$$y = \sinh^{-1}(x)$$

The function name **`asinhf`** should be used for **`float`** arguments, and **`asinhl`** for **`long double`** arguments. Alternately, if the header file **`tgmath.h`** is included, **`asinh`** may be used with any numeric argument type.

### RETURN VALUE

**`asinh`** returns the inverse hyperbolic sine of the argument x.

### RELATED FUNCTIONS

**`acosh`**, **`atanh`**

### SEE ALSO

“Mathematical Functions” in Chapter 2, “Function Categories”

## atanh

**Compute the inverse hyperbolic tangent**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

double atanh(double x);
float atanhf(float x);
long double atanhl(long double x);
```

### DESCRIPTION

**atanh** computes the inverse hyperbolic sine of the argument **x**, expressed by the following relation:

$$y = \tanh^{-1}(x)$$

The function name **atanhf** should be used for **float** arguments, and **atanhl** for **long double** arguments. Alternately, if the header file **tgmath.h** is included, **atanh** may be used with any numeric argument type.

### RETURN VALUE

**atanh** returns the inverse hyperbolic tangent of the argument **x**, provided that this value is defined and expressible.

### DIAGNOSTICS

An error message is written to the standard error file (**stderr**) by the runtime library if **x** is outside the valid domain. In this case, **atanh** returns plus or minus HUGE_VAL for HFP, or a NaN for BFP. In BFP, **atanh(-1.0)** evaluates to negative infinity, and **atanh(+1.0)** to positive infinity.

If an error occurs in **atanh**, the **_matherr** or **_matherb** routine is called. You can supply your own version of**_matherr** or **_matherb** to suppress the diagnostic message or modify the value returned.

### RELATED FUNCTIONS

**acosh**, **asinh**, **_matherb**, **_matherr**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## cbrt

**Compute the Cube Root**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

double cbrt(double x);
float cbrtf(float x);
long double cbrtl(long double x);
```

### DESCRIPTION

The **cbrt** function takes the cube root of its argument x.

The function name **cbrtf** should be used for **float** arguments, and **cbrtl** for **long double** arguments. Alternately, if the header file **tgmath.h** is included, **cbrt** may be used with any numeric argument type.

### RETURN VALUE

**cbrt** returns the cube root of its argument.

### RELATED FUNCTIONS

**sqrt**, **pow**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## copysign

**Propagate the sign of a floating-point number to another**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

double copysign(double x, double y);
float copysignf(float x, float y);
long double copysignl(long double x, long double y);
```

### DESCRIPTION

The **copysign** function generates a floating-point result with the magnitude of the first argument and the sign of the second argument. In other words, it propagates the sign bit of the second argument to the first.

The function name **copysignf** should be used for **float** arguments, and **copysignl** for **long double** arguments. Alternately, if the header file **tgmath.h** is included, **copysign** may be used with any numeric argument type.

### RETURN VALUE

The value of the first argument, with the sign of the second.

### RELATED FUNCTIONS

**signbit**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## exp2

**Compute the base 2 exponential function**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

double exp2(double x);
float exp2f(float x);
long double exp2l(long double x);
```

### DESCRIPTION

**exp2** computes the the base 2 exponential function of its argument **x**. This function is expressed by the following relation:

$$y = 2^{x}$$

The function name **exp2f** should be used for **float** arguments, and **exp2l** for **long double** arguments. Alternately, if the header file **tgmath.h** is included, **exp2** may be used with any numeric argument type.

### RETURN VALUE

**exp2** returns the base 2 exponential function of its argument, provided this value is expressible.

### DIAGNOSTICS

If the result of **exp2** is too large to represent, it returns **HUGE_VAL** in hexadecimal floating point, and either infinity or the largest finite value in binary floating point, depending on the rounding mode. In this case, the run-time library writes an error message to the standard error file (**stderr**).

If an error occurs in **exp2**, the **_matherr** or **_matherb** routine is called. You can supply your own version of **_matherr** or **_matherb** to suppress the diagnostic message or modify the value returned.

### RELATED FUNCTIONS

**exp**, **log2**, **_matherb**, **_matherr**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## expm1

**Compute the exponential function minus 1**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

double expm1(double x);
float expm1f(float x);
long double expm1l(long double x);
```

### DESCRIPTION

**exmp1** computes the exponential function of its argument **x**. This function is expressed by the relation:

$$y = e^{x} - 1$$

where **e** is the base of natural logarithms, 2.7128...

The function name **expm1f** should be used for **float** arguments, and **expm1l** for **long double** arguments. Alternately, if the header file **tgmath.h** is included, **expm1** may be used with any numeric argument type.

### RETURN VALUE

**expm1** returns the exponential function of its argument minus 1, provided this value is expressible.

### DIAGNOSTICS

If the result of **expm1** is too large to represent, it returns **HUGE_VAL** in hexadecimal floating point, and either infinity or the largest finite value in binary floating point, depending on the rounding mode. In this case, the run-time library writes an error message to the standard error file (**stderr**).

If an error occurs in **expm1**, the **_matherr** or **_matherb** routine is called. You can supply your own version of **_matherr** or **_matherb** to suppress the diagnostic message or modify the value returned.

### RELATED FUNCTIONS

**exp**, **log1p**, **_matherb**, **_matherr**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## fdim

**Compute the floating point positive difference**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

double fdim(double x, double y);
float fdimf(float x, float y);
long double fdiml(long double x, long double y);
```

### DESCRIPTION

**`fdim`** computes the positive difference of its arguments **`x`** and **`y`**. The positive difference is defined as **`x-y`** if **`x`** is greater than **`y`**, and 0 otherwise, unless either argument is a NaN, in which case the result is a NaN.

The function name **`fdimf`** should be used for **`float`** arguments, and **`fdiml`** for **`long double`** arguments. Alternately, if the header file **`tgmath.h`** is included, **`fdim`** may be used with any numeric argument type.

### RETURN VALUE

**`fdim`** returns the positive difference of **`x`** and **`y`**, as defined above.

### IMPLEMENTATIONS

For binary floating-point, **`fdim`** is implemented as a built-in function unless it is undefined by a **`#undef`** statement.

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## feclearexcept

**Clear floating point exceptions**

**Portability:** C99

### SYNOPSIS

```
#include <fenv.h>

void feclearexcept(int excepts);
```

## DESCRIPTION

**feclearexcept** is used to clear specified exception flags in the current floating-point environment. The argument specifies the flags to be cleared as the sum of one or more of the exception names defined in **fenv.h**.

## RETURN VALUE

**feclearexcept** has no return value.

## CAUTIONS

**feclearexcept** has no effect when called from a function whose default floating-point format is hexadecimal, that is, one compiled without the **bfp** option.

A program which calls **feclearexcept** must use the standard FENV_ACCESS pragma in an enclosing scope, or the effects are unpredictable.

## RELATED FUNCTIONS

**fegetexceptflag**, **feraiseexcept**, **fesetexceptflag**, **fetestexcept**

## SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## fegetenv

**Store the current floating point environment**

**Portability:** C99

## SYNOPSIS

```
#include <fenv.h>

void fegetenv(fenv_t *envp);
```

## DESCRIPTION

**fegetenv** is used to store the current floating-point environment in the location addressed by the **envp** argument. The floating-point environment includes the exception bits, the rounding mode, and the definition of the exceptions for which trapping is enabled.

## RETURN VALUE

**fegetenv** has no return value.

## CAUTIONS

**fegetenv** stores a dummy value as the environment when called from a function whose default floating-point format is hexadecimal, that is, one compiled without the **bfp** option.

A program which calls **fegetenv** must use the standard FENV_ACCESS pragma in an enclosing scope, or the effects are unpredictable.

### RELATED FUNCTIONS

**feholdexcept**, **fesetenv**, **feupdateenv**

### SEE ALSO

“Mathematical Functions” in Chapter 2, “Function Categories”

## fegetexceptflag

**Extract floating point exception flags**

**Portability:** C99

### SYNOPSIS

```
#include <fenv.h>

void fegetexceptflag(fexcept_t *flags, int excepts);
```

### DESCRIPTION

**fegetexceptflag** is used to extract the exception flag settings in the current floating-point environment. The flags argument points to the location where the flags should be stored; the **excepts** argument is the sum of one or more floating-point exception names indicating which flags are to be extracted.

For SAS/C, the flags pointer addresses a character which contains the sum of the requested flags which were set in the current environment. If you depend on this format, it will make your program non-portable.

### RETURN VALUE

**fegetexceptflag** has no return value.

### CAUTIONS

**fegetexceptflag** has no effect when called from a function whose default floating-point format is hexadecimal, that is, one compiled without the **bfp** option.

A program which calls **fegetexceptflag** must use the standard FENV_ACCESS pragma in an enclosing scope, or the effects are unpredictable.

### RELATED FUNCTIONS

**feclearexcept**, **feraiseexcept**, **fesetexceptflag**, **fetestexcept**

### SEE ALSO

“Mathematical Functions” in Chapter 2, “Function Categories”

## fegetround

**Get the floating point rounding mode**

**Portability:** C99

### SYNOPSIS

```
#include <fenv.h>

int fegetround(void);
```

### DESCRIPTION

**`fegetround`** is used to obtain the rounding mode from the current floating-point environment.

### RETURN VALUE

**`fegetround`** returns an integer representing the current rounding mode. The integer will have one of the values **`FE_TONEAREST`**, **`FE_TOWARDZERO`**, **`FE_UPWARD`** or **`FE_DOWNWARD`**, defined in **`fenv.h`**. If the rounding mode cannot be meaningfully determined, a negative value is returned.

### CAUTIONS

**`fegetround`** returns -1 when called from a function whose default floating-point format is hexadecimal, that is, one compiled without the **`bfp`** option.

A program which calls **`fegetround`** must use the standard FENV_ACCESS pragma in an enclosing scope, or the effects are unpredictable.

### RELATED FUNCTIONS

**`fsetround`**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## fegettrapenable

**Extract floating point trap enablement flags**

**Portability:** C99

### SYNOPSIS

```
#include <fenvtrap.h>
```

```
     int fegettrapenable(void);
```

### DESCRIPTION

**fegettrapenable** is used to extract the trap enablement flags from the current floating-point environment.

### RETURN VALUE

**fegettrapenable** returns an integer, which is the sum of the floating-point flags for which traps are enabled.

### CAUTIONS

**fegettrapenable** returns -1 when called from a function whose default floating-point format is hexadecimal, that is, one compiled without the **bfp** option.

A program which calls **fegettrapenable** must use the standard FENV_ACCESS pragma in an enclosing scope, or the effects are unpredictable.

### RELATED FUNCTIONS

**fesettrapenable**

### SEE ALSO

“Mathematical Functions” in Chapter 2, “Function Categories”

## feholdexcept

**Install a non-trapping floating point environment**

**Portability:** C99

### SYNOPSIS

```
     #include <fenv.h>

     int feholdexcept(fenv_t *envp);
```

### DESCRIPTION

**feholdexcept** is used to save the current floating-point environment and establish a new environment in which no floating-point exceptions are indicated, and in which trapping is not enabled. The existing floating-point environment is stored in the location addressed by the ENVP argument. The floating-point environment includes the exception bits, the rounding mode, and the definition of the exceptions for which trapping is enabled.

### RETURN VALUE

**feholdexcept** returns zero if it was successful, or a non-zero value if it was unable to establish a non-trapping environment.

### CAUTIONS

**feholdexcept** returns -1 when called from a function whose default floating-point format is hexadecimal, that is, one compiled without the **bfp** option.

A program which calls **feholdexcept** must use the standard FENV_ACCESS pragma in an enclosing scope, or the effects are unpredictable.

### RELATED FUNCTIONS

**fegetenv**, **fesetenv**, **feupdateenv**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## feraiseexcept

**Raise floating point exception**

**Portability:** C99

### SYNOPSIS

```
#include <fenv.h>

void feraiseexcept(int excepts);
```

### DESCRIPTION

**feraiseexcept** raises the floating-point exceptions indicated by the excepts argument, which should be the sum of one or more of the exception names defined by **fenv.h**. An exception is raised by setting the exception bit in the floating-point status. If trapping is enabled for that exception, the exception will be trapped and a signal will be raised.

### RETURN VALUE

**feraiseexcept** has no return value.

### CAUTIONS

**feraiseexcept** returns -1 when called from a function whose default floating-point format is hexadecimal, that is, one compiled without the **bfp** option.

A program which calls **feraiseexcept** must use the standard FENV_ACCESS pragma in an enclosing scope, or the effects are unpredictable.

### RELATED FUNCTIONS

**feclearexcept**, **fegetexceptflag**, **fesetexceptflag**, **fetestexcept**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## fesetenv

**Set floating point exception flags**

**Portability:** C99

### SYNOPSIS

```
#include <fenv.h>

void fesetenv(const fenv_t * envp);
```

### DESCRIPTION

**fesetenv** is used to replace the current floating-point environment by one stored in the location addressed by the ENVP argument. All elements of the floating-point environment are replaced, including the exception bits, the rounding mode, and the exceptions for which trapping is enabled.

### RETURN VALUE

**fesetenv** has no return value.

### CAUTIONS

**fesetenv** has no effect when called from a function whose default floating-point format is hexadecimal, that is, one compiled without the **bfp** option.

A program which calls **fegetenv** must use the standard FENV_ACCESS pragma in an enclosing scope, or the effects are unpredictable.

### RELATED FUNCTIONS

**fegetenv**, **feholdexcept**, **feupdateenv**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## fesetexceptflag

**Set floating point exception flags**

**Portability:** C99

### SYNOPSIS

```
#include <fenv.h>

void fesetexceptflag(const fexcept_t *flags, int excepts);
```

## DESCRIPTION

**fesetexceptflag** is used to set the exception flag settings in the current floating-point environment. The **excepts** argument is the sum of one or more floating-point exception names indicating which flags are to be set. The **flags** argument addresses the location from which the new settings of the flags should be taken. In normal usage, the memory addressed by flags will have been initialized by a previous call to **fegetexceptflag**.

For SAS/C, the **flags** pointer addresses a character which contains the sum of the requested flags which were set in the current environment. If you depend on this format, it will make your program non-portable.

## RETURN VALUE

**fesetexceptflag** has no return value.

## CAUTIONS

**fesetexceptflag** has no effect when called from a function whose default floating-point format is hexadecimal, that is, one compiled without the **bfp** option.

A program which calls **fesetexceptflag** must use the standard FENV_ACCESS pragma in an enclosing scope, or the effects are unpredictable.

## RELATED FUNCTIONS

**feclearexcept**, **fegetexceptflag**, **feraiseexcept**, **fetestexcept**

## SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

---

# fesetround

**Set the floating point rounding mode**

**Portability:** C99

## SYNOPSIS

```
#include <fenv.h>

int fesetround(int mode);
```

## DESCRIPTION

**fesetround** is used to change the rounding mode in the current floating-point environment. The mode argument must be one of the values **FE_TONEAREST**, **FE_TOWARDZERO**, **FE_UPWARD** or **FE_DOWNWARD**, defined in **fenv.h**.

## RETURN VALUE

**fesetround** returns zero if the request was successful, or a non-zero value if the request could not be honored.

## CAUTIONS

**fesetround** returns -1 when called from a function whose default floating-point format is hexadecimal, that is, one compiled without the **bfp** option.

A program which calls **fesetround** must use the standard FENV_ACCESS pragma in an enclosing scope, or the effects are unpredictable. The pragma must also be present in any code with runs with a rounding mode other than the default, **FE_TONEAREST**.

## RELATED FUNCTIONS

**fegetround**

## SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

---

## fesettrapenable

**Define floating point trap enablement**

**Portability:** C99

### SYNOPSIS

```
#include <fenvtrap.h>

int fesettrapenable(int traps);
```

### DESCRIPTION

**fesettrapenable** is used to define which floating-point exceptions will cause trapping. The **traps** argument should be zero for no trapping or the sum of one or more of the exception names defined in **fenv.h**.

### RETURN VALUE

**fesettrapenable** has no return value.

### CAUTIONS

**fesettrapenable** returns -1 when called from a function whose default floating-point format is hexadecimal, that is, one compiled without the **bfp** option.

A program which calls **fesettrapenable** must use the standard FENV_ACCESS pragma in an enclosing scope, or the effects are unpredictable.

### RELATED FUNCTIONS

**fegettrapenable**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## fetestexcept

**Test floating point exception flags**

**Portability:** C99

### SYNOPSIS

```
#include <fenv.h>

int fetestexcept(int excepts);
```

### DESCRIPTION

**fetestexcept** is used to test the exception flag settings in the current floating-point environment. The **excepts** argument is the sum of one or more floating-point exception names indicating which flags are to be tested.

### RETURN VALUE

**fetestexcept** returns the sum of the exception names for the subset of the requested exceptions which are set in the current environment.

### CAUTIONS

**fetestexcept** has no meaning when called from a function whose default floating-point format is hexadecimal, that is, one compiled without the **bfp** option. If called, it returns zero.

A program which calls **fegetenv** must use the standard FENV_ACCESS pragma in an enclosing scope, or the effects are unpredictable.

### RELATED FUNCTIONS

**feclearexcept**, **fegetexceptflag**, **feraiseexcept**, **fesetexceptflag**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## feupdateenv

**Restore floating point environment and propagate exceptions**

**Portability:** C99

### SYNOPSIS

```
#include <fenv.h>
```

```
void feupdateenv(const fenv_t * envp);
```

## DESCRIPTION

**feupdateenv** temporarily saves the floating-point status from the current floating-point environment. It then establishes a new floating-point environment from the location addressed by **envp**, and raises the exceptions saved from the previous environment. An exception is raised by setting the exception bit in the floating-point status. If trapping is enabled for that exception, the exception will be trapped and a signal will be raised. If multiple exceptions are raised, the order in which they are raised is undefined, except that **overflow** or **underflow** must precede **inexact**.

## RETURN VALUE

**feupdateenv** has no return value.

## CAUTIONS

**feupdateenv** returns -1 when called from a function whose default floating-point format is hexadecimal, that is, one compiled without the **bfp** option.

A program which calls **feupdateenv** must use the standard FENV_ACCESS pragma in an enclosing scope, or the effects are unpredictable.

## USAGE NOTES

A typical scenario for use of this function would be to call **feholdexcept** to reset the exception flags and prevent trapping, then to perform a calculation with assurance that trapping will not occur, and then to call **feupdateenv** to restore the previous environment, possibly trapping for exceptions which occurred during the calculation.

## RELATED FUNCTIONS

**fegetenv**, **feholdexcept**, **fesetenv**

## SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

---

## fma

**Compute floating-point multiply and add functions**

**Portability:** C99

## SYNOPSIS

```
#include <math.h>

double fma(double x, double y, double z);
float fmaf(float x, float y, float z);
long double fmal(long double x, long double y, long double z);
```

## DESCRIPTION

**fma** computes **x*y+z** as if it was a single operation rounded according to the current rounding mode. For binary floating point, the performance of **fma** will be better than the performance of the equivalent expression, so long as a **#undef** is not used to undefine **fma**. For hexadecimal floating point, fma performs considerably worse than the defining expression above. The symbol FP_FAST_FMA can be tested to portably determine if the use of **fma** is advantageous.

The function name **fmaf** should be used for **float** arguments, and **fmal** for **long double** arguments. Alternately, if the header file **tgmath.h** is included, **fma** may be used with any numeric argument type.

## RETURN VALUE

**fma** returns **x*y+z**, computed as a single operation, if it is expressible in the return type.

## IMPLEMENTATIONS

For binary floating-point, **fma** is implemented as a built-in function unless it is undefined by a **#undef** statement.

## DIAGNOSTICS

For binary floating point, if the **fma** function is not undefined, any floating-point exceptions produced during its execution will be raised in the normal fashion, perhaps with trapping. No diagnostic message will be generated, and **errno** will not be set. If the built-in **fma** is undefined, overflow or underflow will cause a diagnostic message to be generated, and trapping will not occur.

The error behavior for hexadecimal floating point is the same as for binary when the built-in function is undefined, that is, errors will be diagnosed, and **errno** will be set.

## SEE ALSO

“Mathematical Functions” in Chapter 2, “Function Categories”

---

# fpclassify

**Classify floating-point number**

**Portability:** C99

## SYNOPSIS

```
#include <math.h>

int fpclassify(floating x);
```

## DESCRIPTION

**fpclassify** determines the class of numbers to which its argument belongs. The argument may be of any floating point type.

### RETURN VALUE

**fpclassify** returns one of the values **FP_ZERO**, **FP_INFINITE**, **FP_NORMAL**, **FP_SUBNORMAL**, or **FP_NAN** to indicate a zero, infinite, normal, subnormal (or denormalized), or invalid (NaN) value.

### USAGE NOTES

When called from a function compiled without the **bfp** option, **fpclassify** will always return either **FP_ZERO** or **FP_NORMAL**.

### RELATED FUNCTIONS

**isfinite**, **isinf**, **isnan**, **isnormal**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## ilogb

**Extract the floating-point exponent**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

int ilogb(double x);
int ilogbf(float x);
int ilogbl(long double x);
```

### DESCRIPTION

**ilogb** extracts the exponent of a floating-point number. For hexadecimal floating point, the exponent represents a power of sixteen; for binary, it represents a power of two.

The function name **ilogbf** should be used for **float** arguments, and **ilogbl** for **long double** arguments. Alternately, if the header file **tgmath.h** is included, **ilogb** may be used with any numeric argument type.

### RETURN VALUE

**ilogb** returns the exponent of the argument as an integer. **ilogb(0.0)** returns the value **FP_ILOGB0**, and **ilogb(NaN)** returns **FP_ILOGBNAN**. **ilogb(infinity)** returns the largest integer value.

### RELATED FUNCTIONS

**frexp**, **logb**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## isfinite

**Test for finite floating-point number**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

int isfinite(floating x);
```

### DESCRIPTION

**`isfinite`** tests whether its argument is a finite number, that is, not infinite, and not a NaN.

### RETURN VALUE

**`isfinite`** returns zero if its argument is not finite, and a non-zero value otherwise.

### USAGE NOTES

When called from a compilation compiled without the **`bfp`** option, **`isfinite`** always returns non-zero.

### RELATED FUNCTIONS

**`fpclassify`**, **`isinf`**, **`isnan`**, **`isnormal`**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## isgreater

**Floating-point greater than comparison**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>
```

```
int isgreater(floating x, floating y);
```

### DESCRIPTION

**isgreater** returns whether its first argument, **x**, is greater than its second argument, **y**. No floating point exception is raised if an argument is a NaN.

### RETURN VALUE

1 if **x** is greater than **y**, and zero otherwise.

### RELATED FUNCTIONS

**isgreaterequal**, **isless**, **islessequal**, **islessgreater**, **isunordered**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## isgreaterequal

**Floating-point greater than or equal to comparison**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

int isgreaterequal(floating x, floating y);
```

### DESCRIPTION

**isgreaterequal** returns whether its first argument, **x**, is greater than or equal to its second argument, **y**. No floating point exception is raised if an argument is a NaN.

### RETURN VALUE

1 if **x** is greater than or equal to **y**, and zero otherwise.

### RELATED FUNCTIONS

**isgreater**, **isless**, **islessequal**, **islessgreater**, **isunordered**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## isinf

**Test for infinite floating-point number**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

int isinf(floating x);
```

### DESCRIPTION

**`isinf`** tests whether its argument is a infinite number, that is, plus or minus infinity.

### RETURN VALUE

**`isinf`** returns zero if its argument is not infinite, and a non-zero value otherwise.

### USAGE NOTES

When called from a compilation compiled without the **`bfp`** option, **`isinf`** always returns zero.

### RELATED FUNCTIONS

**`fpclassify`**, **`isfinite`**, **`isnan`**, **`isnormal`**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## isless

**Floating-point less than comparison**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

int isless(floating x, floating y);
```

### DESCRIPTION

**`isless`** returns whether its first argument, **`x`**, is less than its second argument, **`y`**. No floating point exception is raised if an argument is a NaN.

### RETURN VALUE

1 if **`x`** is less than **`y`**, and zero otherwise.

### RELATED FUNCTIONS

**isgreater**, **isgreaterequal**, **islessequal**, **islessgreater**, **isunordered**

### SEE ALSO

“Mathematical Functions” in Chapter 2, “Function Categories”

## islessequal

**Floating-point less than or equal to comparison**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

int islessequal(floating x, floating y);
```

### DESCRIPTION

**islessequal** returns whether its first argument, **x**, is less than or equal to its second argument, **y**. No floating point exception is raised if an argument is a NaN.

### RETURN VALUE

1 if **x** is less than or equal to **y**, and zero otherwise.

### RELATED FUNCTIONS

**isgreater**, **isgreaterequal**, **isless**, **islessgreater**, **isunordered**

### SEE ALSO

“Mathematical Functions” in Chapter 2, “Function Categories”

## islessgreater

**Floating-point less than or greater than comparison**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

int islessgreater(floating x, floating y);
```

### DESCRIPTION

**`islessgreater`** returns whether its first argument, **`x`**, is greater than or less than its second argument, **`y`**. No floating point exception is raised if an argument is a NaN.

### RETURN VALUE

1 if **`x`** is greater than or less than **`y`**, and zero otherwise.

### RELATED FUNCTIONS

**`isgreater`**, **`isless`**, **`islessequal`**, **`islessgreater`**, **`isunordered`**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## isnan

**Test for floating-point NaN**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

int isnan(floating x);
```

### DESCRIPTION

**`isnan`** tests whether its argument is a NaN.

### RETURN VALUE

**`isnan`** returns zero if its argument is not a NaN, and a non-zero value otherwise.

### USAGE NOTES

When called from a compilation compiled without the **`bfp`** option, **`isnan`** always returns zero.

### RELATED FUNCTIONS

**`fpclassify`**, **`isfinite`**, **`isinf`**, **`isnormal`**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## isnormal

**Test for normal floating-point number**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

int isnormal(floating x);
```

### DESCRIPTION

**isnormal** tests whether its argument is a normal floating-point number, that is, not zero, not infinite, not denormalized, and not a NaN.

### RETURN VALUE

**isnormal** returns zero if its argument is a normal floating-point number, and a non-zero value otherwise.

### USAGE NOTES

When called from a compilation compiled without the **bfp** option, **isnormal** always returns zero for a zero argument and non-zero otherwise.

### RELATED FUNCTIONS

**fpclassify**, **isfinite**, **isinf**, **isnan**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## isunordered

**Floating-point unordered comparison**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

int isunordered(floating x, floating y);
```

### DESCRIPTION

**isunordered** returns whether its two arguments, **x** and **y**, may be compared. It returns non-zero if either argument is a NaN (Nan's cannot be compared), or returns zero if neither argument is a NaN. No floating point exception is raised if an argument is a NaN.

### RETURN VALUE

1 if **x** and **y** are unordered, and zero otherwise.

### RELATED FUNCTIONS

**isgreater**, **isgreaterequal**, **isless**, **islessequal**, **islessgreater**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## llrint

**Convert floating-point to long long integer**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

long long llrint(double x);
long long llrintf(float x);
long long llrintl(long double x);
```

### DESCRIPTION

**llrint** converts its argument to an integral value using the current rounding mode and returns it as a **long long** integer.

The function name **llrintf** should be used for **float** arguments, and **llrintl** for **long double** arguments. Alternately, if the header file **tgmath.h** is included, **llrint** may be used with any numeric argument type.

### RETURN VALUE

**llrint** returns its argument converted to a **long long** integer. If the result is outside the range of **long long** integer, the return value is unpredictable.

### IMPLEMENTATION

**llrint** is implemented as a built-in function unless it is undefined by a **#undef** statement.

### RELATED FUNCTIONS

**ceil**, **floor**, **llround**, **lrint**, **lround**, **nearbyint**, **rint**, **round**, **trunc**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## llround

**Round floating-point to long long integer**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

long long llround(double x);
long long llroundf(float x);
long long llroundl(long double x);
```

### DESCRIPTION

**llround** rounds its argument to the nearest integral value and returns it as a **long long** integer. If two results are equally near, **lround** rounds away from zero.

The function name **llroundf** should be used for **float** arguments, and **llroundl** for **long double** arguments. Alternately, if the header file **tgmath.h** is included, **llround** may be used with any numeric argument type.

### RETURN VALUE

**llround** returns its argument rounded to the nearest integer.

### IMPLEMENTATION

**llround** is implemented as a built-in function unless it is undefined by a **#undef** statement.

### RELATED FUNCTIONS

**ceil**, **floor**, **llrint**, **lrint**, **lround**, **nearbyint**, **rint**, **round**, **trunc**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## log1p

**Logarithm plus 1 function**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

double log1p(double x);
float log1pf(float x);
long double log1pl(long double x);
```

### DESCRIPTION

**log1p** computes the natural logarithm of one plus its argument **x**. This function is expressed by the relation:

```
y = ln(1 + x)
```

**x** must be greater than -1.0.

The function name **log1pf** should be used for **float** arguments, and **log1pl** for **long double** arguments. Alternately, if the header file **tgmath.h** is included, **log1p** may be used with any numeric argument type.

### RETURN VALUE

**log1p** returns the natural logarithm of one plus its argument, provided this value is defined and expressible.

### DIAGNOSTICS

If the argument of **log1p** is invalid, it returns **-HUGE_VAL** in hexadecimal floating point, or a NaN in binary floating point. In this case, the run-time library writes an error message to the standard error file **stderr**. **log1p(-1.0)** returns negative infinity in binary floating point.

### RELATED FUNCTIONS

**expm1**, **log**, **log1p**, **_matherb**, **_matherr**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## log2

**Compute the logarithm base 2**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

double log2(double x);
float log2f(float x);
long double log2l(long double x);
```

## DESCRIPTION

**log2** computes the binary (base 2) logarithm of its argument **x**. **x** must be greater than zero.

The function name **log2f** should be used for **float** arguments, and **log2l** for **long double** arguments. Alternately, if the header file **tgmath.h** is included, **log2** may be used with any numeric argument type.

## RETURN VALUE

**log2** returns the binary (base 2) logarithm of its argument, provided this value is defined and expressible.

## DIAGNOSTICS

If the argument of **log2** is invalid, it returns **-HUGE_VAL** in hexadecimal floating point, or a NaN in binary floating point. In this case, the run-time library writes an error message to the standard error file **stderr**. **log2(0.0)** returns negative infinity in binary floating point.

If an error occurs in **log2**, the **_matherr** or **_matherb** routine is called. You can supply your own version of **_matherr** or **_matherb** to suppress the diagnostic message or modify the value returned.

## RELATED FUNCTIONS

**exp2**, **log**, **_matherb**, **_matherr**

## SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## logb

**Extract the floating-point exponent**

**Portability:** C99

## SYNOPSIS

```
#include <math.h>

double logb(double x);
float logbf(float x);
long double logbl(long double x);
```

## DESCRIPTION

**logb** extracts the exponent of a floating-point number. For hexadecimal floating point, the exponent represents a power of sixteen; for binary, it represents a power of two.

*Note:* The value returned by **logb** is an exact integer, even though its type is double. △

The function name **logbf** should be used for **float** arguments, and **logbl** for **long double** arguments. Alternately, if the header file **tgmath.h** is included, **logb** may be used with any numeric argument type.

### RETURN VALUE

**logb** returns the exponent of the argument as a floating-point value. **logb(0.0)** returns minus infinity (or **-HUGE_VAL** for HFP), and **logb(NaN)** returns a NaN. **logb(infinity)** returns positive infinity.

### RELATED FUNCTIONS

**frexp**, **ilogb**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## lrint

**Convert floating-point to long integer**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

long lrint(double x);
long lrintf(float x);
long lrintl(long double x);
```

### DESCRIPTION

**lrint** uses the current rounding mode to convert its argument to an integral value and returns it as a long integer.

The function name **lrintf** should be used for **float** arguments, and **lrintl** for **long double** arguments. Alternately, if the header file **tgmath.h** is included, **lrint** may be used with any numeric argument type.

### RETURN VALUE

**lrint** returns its argument converted to a **long** integer. If the result is outside the range of **long** integer, the return value is unpredictable.

### IMPLEMENTATION

**lrint** is implemented as a built-in function unless it is undefined by a **#undef** statement.

### RELATED FUNCTIONS

**ceil**, **floor**, **llrint**, **llround**, **lround**, **nearbyint**, **rint**, **round**, **trunc**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## lround

**Round floating-point to long integer**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

long lround(double x);
long lroundf(float x);
long lroundl(long double x);
```

### DESCRIPTION

**lround** rounds its argument to the nearest integral value and returns it as a **long** integer. If two results are equally near, **lround** rounds away from zero.

The function name **lroundf** should be used for **float** arguments, and **lroundl** for **long double** arguments. Alternately, if the header file **tgmath.h** is included, **lround** may be used with any numeric argument type.

### RETURN VALUE

**lround** returns its argument rounded to the nearest integer.

### IMPLEMENTATION

**lround** is implemented as a built-in function unless it is undefined by a **#undef** statement.

### RELATED FUNCTIONS

**ceil**, **floor**, **llrint**, **llround**, **lrint**, **nearbyint**, **rint**, **round**, **trunc**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## _matherb

**Handle Math Function Error (Binary Floating Point)**

**Portability:** C99

## SYNOPSIS

```
#include <lcmath.h>

int _matherb(struct exception *x);
```

## DESCRIPTION

**_matherb** is called whenever an IEEE transcendental function detects an error. **_matherb** receives an exception block as an argument describing the error. This structure is defined in **<lcmath.h>**:

```
struct exception {
   int type;          /* error type */
   __near char *name; /* name of failed function */
   double arg1;       /* first argument */
   double arg2;       /* second argument */
   double retval;     /* proposed return value */
   int flterr;        /* if non-0, failing function returns float */
};
```

The error type names defined in **<lcmath.h>** are:

| Error Type | Definition |
|---|---|
| DOMAIN | domain error |
| SING | singularity |
| OVERFLOW | overflow |
| UNDERFLOW | underflow |
| TLOSS | total loss of significance |
| PLOSS | partial loss of significance |

## RETURN VALUE

If **_matherb** returns 0, a diagnostic message is written to the standard error file (**stderr**). If **_matherb** returns a nonzero value, the diagnostic message is suppressed, and the calling function will return the value stored in **retval** to its caller.

## PORTABILITY

Traditional UNIX C compilers support the functionality of **_matherb** using the name **matherr**. Unfortunately, using the name matherr conflicts with the ANSI Standard. However, the header file **lcmath.h** contains the following macro:

```
#define matherr _matherb
```

If you include this header file, use the name that is compatible with traditional UNIX C compilers.

## IMPLEMENTATION

The standard version of **_matherb** supplied in the library places the appropriate error number into the external integer **errno** and returns zero. When **_matherb** is called, the function that detected the error places its proposed return value into the exception structure. The zero return code indicates that the proposed return value should not be changed.

Supply your own version of **_matherb** if desired. On particular errors, it may be desirable to cause the function detecting the error to return a value other than its usual default. You can accomplish this by storing a new return value in **retval** of the exception structure and then returning a nonzero value from **_matherb**, which forces the function to pick up the new value from the exception structure. If a nonzero value is returned, a diagnostic message is not printed for the error.

### RELATED FUNCTIONS

**_matherr**, **quiet**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## nearbyint

**Convert to floating-point integer**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

double nearbyint(double x);
float nearbyintf(float x);
long double nearbyintl(long double x);
```

### DESCRIPTION

**nearbyint** converts its argument to an integral value using the current rounding mode, and returns it as a floating-point value. The *inexact* floating-point exception is not raised.

The function name **nearbyintf** should be used for **float** arguments, and **nearbyintl** for **long double** arguments. Alternately, if the header file **tgmath.h** is included, **nearbyint** may be used with any numeric argument type.

### RETURN VALUE

**nearbyint** returns its argument converted to an integer.

### IMPLEMENTATION

**nearbyint** is implemented as a built-in function unless it is undefined by a **#undef** statement.

### USAGE NOTES

For hexadecimal floating-point, the **nearbyint** function is identical to the **rint** function.

### RELATED FUNCTIONS

**`ceil`**, **`floor`**, **`llrint`**, **`llround`**, **`lrint`**, **`lround`**, **`rint`**, **`round`**, **`trunc`**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## nextafter

**Compute the next floating-number in a specific direction**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

double nextafter(double x, double y);
float nextafterf(float x, float y);
long double nextafterl(long double x, long double y);
```

### DESCRIPTION

**`nextafter`** returns the next representable value after **`x`** in the direction of **`y`**. That is, if **`y > x`**, then the value returned is the next double greater than **`x`**; if **`y < x`**, the the value returned is the next double less than **`x`**; and if **`y == x`**, the value returned is **`y`**.

The function name **`nextafterf`** should be used for **`float`** arguments, and **`nextafterl`** for **`long double`** arguments. Alternately, if the header file **`tgmath.h`** is included, **`nextafter`** may be used with any numeric argument type.

### RETURN VALUE

**`nextafter`** returns the next value after **`x`** (see DESCRIPTION).

### RELATED FUNCTIONS

**`nexttoward`**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## nexttoward

**Compute the next floating-number in a specific direction**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

double nexttoward(double x, double y);
float nexttowardf(float x, float y);
long double nexttowardl(long double x, long double y);
```

### DESCRIPTION

**nexttoward** returns the next representable value after **x** in the direction of **y**. That is, if **y > x**, then the value returned is the next double greater than **x**; if **y < x**, the the value returned is the next double less than **x**; and if **y == x**, the value returned is **y**.

The function name **nexttowardf** should be used for **float** arguments, and **nexttowardl** for **long double** arguments. Alternately, if the header file **tgmath.h** is included, **nexttoward** may be used with any numeric argument type.

### RETURN VALUE

**nexttoward** returns the next value after **x**. See DESCRIPTION.

### RELATED FUNCTIONS

**nextafter**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## remainder

**Compute the floating-point remainder function**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

double remainder(double x, double y);
float remainderf(float x, float y);
long double remainderl(long double x, long double y);
```

### DESCRIPTION

**remainder** produces the remainder when the first argument, **x**, is divided by the second argument, **y**. Both arguments are assumed to be exact, and the division is performed exactly. The remainder is computed using a rounded integer quotient. If the fractional part of the quotient is 0.5, the quotient is rounded to an even value.

The function name **remainderf** should be used for **float** arguments, and **remainderl** for **long double** arguments. Alternately, if the header file **tgmath.h** is included, **remainder** may be used with any numeric argument type.

### RETURN VALUE

**remainder** returns the remainder resulting from division of **x** by **y**, if it is defined and representable.

### RELATED FUNCTIONS

**fmod**, **remquo**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## remquo

**Compute the floating-point remainder and a partial quotient**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

double remquo(double x, double y, int *q);
float remquof(float x, float y, int *q);
long double remquol(long double x, long double y, int *q);
```

### DESCRIPTION

**remquo** produces the remainder when the first argument, **x**, is divided by the second argument, **y**. Both arguments are assumed to be exact, and the division is performed exactly. The remainder is computed using a rounded integer quotient. If the fractional part of the quotient is 0.5, the quotient is rounded to an even value.

The initial bits of the rounded integer quotient are stored in the integer addressed by the argument, **q**. The sign of the value stored is the correct sign of the quotient, and at least the last three bits of the quotient are correct.

*Note:* With SAS/C, all 31 quotient bits are correct, but the C99 specification only guarantees three bits. △

The function name **remquof** should be used for **float** arguments, and **remquol** for **long double** arguments. Alternately, if the header file **tgmath.h** is included, **remquo** may be used with any numeric argument type.

### RETURN VALUE

**remquo** returns the remainder resulting from division of **x** by **y**, if it is defined and representable.

### RELATED FUNCTIONS

**fmod**, **remainder**

### SEE ALSO

“Mathematical Functions” in Chapter 2, “Function Categories”

## rint

**Convert to floating-point integer**

**Portability:** C99

### SYNOPSIS

```
    #include <math.h>

double rint(double x);
float rintf(float x);
long double rintl(long double x);
```

### DESCRIPTION

**rint** converts its argument to an integral value using the current rounding mode, and returns it as a floating-point value. For binary floating point, The *inexact* floating-point exception is raised if the argument differs from the result.

The function name **rintf** should be used for **float** arguments, and **rintl** for **long double** arguments. Alternately, if the header file **tgmath.h** is included, **rint** may be used with any numeric argument type.

### RETURN VALUE

**rint** returns its argument, converted to an integer.

### IMPLEMENTATION

**rint** is implemented as a built-in function unless it is undefined by a **#undef** statement.

### RELATED FUNCTIONS

**ceil**, **floor**, **llrint**, **llround**, **lrint**, **lround**, **nearbyint**, **round**, **trunc**

### SEE ALSO

“Mathematical Functions” in Chapter 2, “Function Categories”

## round

**Round to floating-point integer**

**Portability:** C99

### SYNOPSIS

```
     #include <math.h>

  double round(double x);
  float roundf(float x);
  long double roundl(long double x);
```

### DESCRIPTION

**round** rounds its argument to the nearest integral value, and returns it as a floating-point value. If two results are equally near, **round** rounds away from zero.

The function name **roundf** should be used for **float** arguments, and **roundl** for **long double** arguments. Alternately, if the header file **tgmath.h** is included, **round** may be used with any numeric argument type.

### RETURN VALUE

**round** returns its argument, rounded to the nearest integer.

### IMPLEMENTATION

**round** is implemented as a built-in function unless it is undefined by a **#undef** statement.

### RELATED FUNCTIONS

**ceil**, **floor**, **llrint**, **llround**, **lrint**, **lround**, **nearbyint**, **rint**, **trunc**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## scalbn

**Scale a floating point number**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

double scalbn(double x, int n);
float scalbnf(float x, int n);
long double scalbnl(long double x, int n);
```

### DESCRIPTION

**scalbn** is used to scale a floating-point number by a power of the floating-point base (16 for hexadecimal floating point, 2 for binary floating point). The result is:

$$x * b^n$$

where **b** is the floating-point base.

The function name **scalbnf** should be used for **float** arguments, and **scalbnl** for **long double** arguments. Alternately, if the header file **tgmath.h** is included, **scalbn** may be used with any numeric argument type.

### RETURN VALUE

**scalbn** returns the argument, **x**, scaled by the floating-point base raised to the power **n**, if that value is expressible.

### IMPLEMENTATION

For binary floating-point, **scalbn** is implemented as a built-in function unless it is undefined by a **#undef** statement.

### USAGE NOTES

**scalbn** resembles the **ldexp** function, but differs in using a power of the floating-point base rather than the constant 2 as the scale factor.

### RELATED FUNCTIONS

**ilogb**, **ldexp**, **logb**, **scalbln**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## scalbln

**Scale a floating point number**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

double scalbln(double x, long n);
float scalblnf(float x, long n);
long double scalblnl(long double x, long n);
```

### DESCRIPTION

**scalbln** is used to scale a floating-point number by a power of the floating-point base (16 for hexadecimal floating point, 2 for binary floating point). The result is:

$$x * b^n$$

where **b** is the floating-point base.

The function name **scalblnf** should be used for **float** arguments, and **scalblnl** for **long double** arguments. Alternately, if the header file **tgmath.h** is included, **scalbln** may be used with any numeric argument type.

### RETURN VALUE

**scalbln** returns the argument, **x**, scaled by the floating-point base raised to the power **n**, if that value is expressible.

### IMPLEMENTATION

For binary floating-point, **scalbln** is implemented as a built-in function unless it is undefined by a **#undef** statement.

### USAGE NOTES

**scalbln** resembles the **scalbn** function, but differs in using the type **long** for the exponent.

### RELATED FUNCTIONS

**ilogb**, **ldexp**, **logb**, **scalbn**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## signbit

**Test sign of floating-point number**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

int signbit(floating x);
```

### DESCRIPTION

**signbit** determines whether its argument is positive or negative by testing the sign bit. Note that zeroes and NaNs may have the sign bit set.

### RETURN VALUE

**signbit** returns 0 if its argument is positive (the sign bit is off), or non-zero if its argument is negative (the sign bit is on).

### RELATED FUNCTIONS

**copysign**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## tgamma

**Compute the gamma function**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

double tgamma(double x);
float tgammaf(float x);
long double tgammal(long double x);
```

### DESCRIPTION

**tgamma** computes the **gamma** function of its argument, **x**. The value returned by **tgamma** is defined by this equation:

$$tgamma(x) = \int_0^\infty t^{x-1} e^{-t} \, dt$$

The function name **tgammaf** should be used for **float** arguments, and **tgammal** for **long double** arguments. Alternately, if the header file **tgmath.h** is included, **tgamma** may be used with any numeric argument type.

### RETURN VALUE

**tgamma** returns the **gamma** function of its argument, if it is defined and representable.

### DIAGNOSTICS

If the argument of **tgamma** is invalid, it returns **HUGE_VAL** in hexadecimal floating point, or a NaN in binary floating point. If the result of **tgamma** overflows, it returns **HUGE_VAL** in hexadecimal floating point, and either an infinity or the maximum finite value, depending on the rounding mode in binary floating-point. In these cases, the run-time library writes an error message to the standard error file (**stderr**).

### RELATED FUNCTIONS

**gamma**, **lgamma**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## trunc

**Truncate to floating-point integer**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>

double trunc(double x);
float truncf(float x);
long double truncl(long double x);
```

### DESCRIPTION

**`trunc`** converts its argument to an integral value, rounding towards zero, and returns it as a floating-point value.

The function name **`truncf`** should be used for **`float`** arguments, and **`truncl`** for **`long double`** arguments. Alternately, if the header file **`tgmath.h`** is included, **`trunc`** may be used with any numeric argument type.

### RETURN VALUE

**`trunc`** returns its argument converted to an integer.

### IMPLEMENTATION

**`trunc`** is implemented as a built-in function unless it is undefined by a **`#undef`** statement.

### RELATED FUNCTIONS

**`ceil`**, **`floor`**, **`llrint`**, **`llround`**, **`lrint`**, **`lround`**, **`nearbyint`**, **`rint`**, **`round`**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories"

## Update to float.h Header File

In Chapter 2, "Function Categories," in the section titled, "Mathematical Functions," replace the contents of the **`float.h`** header file with the following information:

```
/*
*
```

```
* This header file defines several esoteric attributes of the
* 370 floating-point implementation.  The attributes apply to
* the default floating-point format, as specified by a compiler
* option.
*
*/

#if _O_BFP == 0

#define FLT_RADIX 16         /* hardware float radix */
#ifndef _SASC_HIDE_C99MATHLIB
#define DECIMAL_DIG 17       /* precision of decimal<->hex conversion */
#define FLT_EVAL_METHOD 0    /* expressions evaluated as requested type */
#endif
#define FLT_ROUNDS 0         /* float addition does not round */

#define FLT_MANT_DIG 6       /* hex digits in float mantissa */
#define DBL_MANT_DIG 14      /* hex digits in double mantissa */
#define LDBL_MANT_DIG 14     /* hex digits in long double mantissa */

#define FLT_DIG 6            /* float decimal precision */
#define DBL_DIG 15           /* double decimal precision */
#define LDBL_DIG 15          /* long double decimal precision */

#define FLT_MIN_EXP -64      /* minimum exponent of 16 for float */
#define DBL_MIN_EXP -64      /* minimum exponent of 16 for double */
#define LDBL_MIN_EXP -64     /* minimum exponent of 16 for long double */

#define FLT_MIN_10_EXP -78   /* minimum float power of 10 */
#define DBL_MIN_10_EXP -78   /* minimum double power of 10 */
#define LDBL_MIN_10_EXP -78  /* minimum long double power of 10 */

#define FLT_MAX_EXP 63       /* maximum exponent of 16 for float */
#define DBL_MAX_EXP 63       /* maximum exponent of 16 for double */
#define LDBL_MAX_EXP 63      /* maximum exponent of 16 for long double */

#define FLT_MAX_10_EXP 75    /* maximum float power of 10 */
#define DBL_MAX_10_EXP 75    /* maximum double power of 10 */
#define LDBL_MAX_10_EXP 75   /* maximum long double power of 10 */

#define FLT_MAX .7237005e76F             /* maximum float */
                                         /* == 0x.FFFFFFp252F */
#define DBL_MAX .72370055773322621e76    /* maximum double */
                                         /* == 0x.FFFFFFFFFFFFFFp252 */
#define LDBL_MAX .72370055773322621e76L  /* maximum long double */

#define FLT_EPSILON .9536743e-6F             /* smallest float x such
                                                that 1.0 + x != 1.0 */
                                             /* == 0x1.0p-20F */
#define DBL_EPSILON .22204460492503131e-15   /* smallest double x such
                                                that 1.0 + x != 1.0 */
                                             /* == 0x1.0p-52 */
#define LDBL_EPSILON .22204460492503131e-15L /* smallest long double x such
                                                that 1.0 + x != 1.0 */
```

```
#define FLT_MIN .5397606e-78F            /* minimum float */
                                         /* == 0x1.0p-260F */
#define DBL_MIN .53976053469340279e-78   /* minimum double */
                                         /* == 0x1.0p-260 */
#define LDBL_MIN .53976053469340279e-78L /* minimum long double */

#else

#ifndef FLT_ROUNDS
int __builtin_fltrnds(void);
#endif

#define FLT_RADIX 2          /* hardware float radix */
#ifndef _SASC_HIDE_C99MATHLIB
#define DECIMAL_DIG 17       /* precision of decimal<->bin conversion */
#define FLT_EVAL_METHOD 0    /* expressions evaluated as requested type */
#endif
#define FLT_ROUNDS __builtin_fltrnds()  /* actual runtime round mode */

#define FLT_MANT_DIG 24      /* binary digits in float mantissa */
#define DBL_MANT_DIG 53      /* binary digits in double mantissa */
#define LDBL_MANT_DIG 53     /* binary digits in long double mantissa */

#define FLT_DIG 6            /* float decimal precision */
#define DBL_DIG 15           /* double decimal precision */
#define LDBL_DIG 15          /* long double decimal precision */

#define FLT_MIN_EXP -125     /* minimum exponent of 2 for float */
#define DBL_MIN_EXP -1021    /* minimum exponent of 2 for double */
#define LDBL_MIN_EXP -1021   /* minimum exponent of 2 for long double */

#define FLT_MIN_10_EXP -37   /* minimum float power of 10 */
#define DBL_MIN_10_EXP -307  /* minimum double power of 10 */
#define LDBL_MIN_10_EXP -307 /* minimum long double power of 10 */

#define FLT_MAX_EXP 128      /* maximum exponent of 2 for float */
#define DBL_MAX_EXP 1024     /* maximum exponent of 2 for double */
#define LDBL_MAX_EXP 1024    /* maximum exponent of 2 for long double */

#define FLT_MAX_10_EXP 38    /* maximum float power of 10 */
#define DBL_MAX_10_EXP 308   /* maximum double power of 10 */
#define LDBL_MAX_10_EXP 308  /* maximum long double power of 10 */

#define FLT_MAX 3.402823466e38F          /* maximum float */
                                         /* == 0x1.FFFFFEp127F */
#define DBL_MAX 1.7976931348623157e308   /* maximum double */
                                         /* == 0x1.FFFFFFFFFFFFFp1023 */
#define LDBL_MAX 1.7976931348623157e308L /* maximum long double */

#define FLT_EPSILON 1.1920929e-7F    /* smallest float x such that
                                        1.0 + x != 1.0 */
                                     /* == 0x1.0p-23F  */
#define DBL_EPSILON 2.220446049250313le-16  /* smallest double x such
```

```
                                                  that 1.0 + x != 1.0 */
                                               /* == 0x1.0p-52  */
#define LDBL_EPSILON 2.2204460492503131e-16L /* smallest longdbl x such
                                                  that 1.0 + x != 1.0 */

#define FLT_MIN 1.17549435e-38F           /* minimum normalized float */
                                          /* == 0x1.0p-126 */
#define DBL_MIN 2.2250738585072014e-308   /* minimum normalized double*/
                                          /* == 0x1.0p-1022 */
#define LDBL_MIN 2.2250738585072014e-308L /* minimum normalized ldbl  */

#endif
```

## New nan, nanf, and nanl Functions

In Chapter 6, “Function Descriptions,” add the following entry to the list of functions.

## nan, nanf, nanl

**Return NaN**

**Portability:** C99

### SYNOPSIS

```
#include <math.h>
double nan(const char *tagp);
float nanf(const char *tagp);
long double nanl(const char *tagp);
```

### DESCRIPTION

The functions **nan**, **nanf**, and **nanl** convert a character sequence pointed to by **tagp** to double, float, and long double NaN representation, respectively. The character sequence is an array of chars that indicates the bit pattern for the NaN. Valid values differ depending on whether one is using Binary (IEEE) floating point numbers (that is, compiled with the **bfp** option) or the traditional, IBM hexadecimal floating point (HFP) numbers. See “Implementation” for more information.

### RETURN VALUE

For Binary floating point (BFP) the functions, **nan**, **nanf**, and **nanl** return a double, float, and long double NaN representation, respectively, with a fractional bit pattern based on the character sequence pointed to by the **tagp** argument. If **tagp** points to a NULL string, then the default quiet NaN is returned. If the first character in the character sequence string is invalid, a diagnostic is issued, errno is set to EARG, and the default quiet NaN is returned. If the character sequence begins with a *quiet* or *signaling* NaN prefix letter, then any following hex digits in the character sequence must be valid for that type of NaN. That is, the most significant fraction bit must be represented correctly. Otherwise, a diagnostic is issued, errno is set to EARG, and the default quiet or signaling NaN as indicated by the prefix letter is returned. If any other characters in the character sequence string is not valid, the conversion stops at that point and a diagnostic is issued, **`errno`** is set to **`EARG`**, and a NaN whose fraction bits have been described by the string argument up to the point of the invalid character is returned. See "Implementation" for a description of the character sequence string.

For Hexadecimal floating point (HFP) the functions, **`nan`**, **`nanf`**, and **`nanl`** return a double, float, and long double zero representation, respectively, with the bits set in the sign and exponent fields based on the character sequence pointed to by the **`tagp`** argument. If **`tagp`** points to a NULL string, then the default negative zero value, only sign bit set, is returned. If the first character in the character sequence string is invalid, a diagnostic is issued, errno is set to EARG, and the default negative zero is returned. If any other characters in the character sequence string is not valid, the conversion stops at that point and a diagnostic is issued, **`errno`** is set to **`EARG`**, and a zero value whose sign and exponent bits that have been described by the string argument up to the point of the invalid character is returned. See "Implementation" for a description of the character sequence string.

## IMPLEMENTATION

A binary NaN is represented by all Exponent bits being set to one and at least one Fraction bit set to one. The most significant fraction bit determines whether the NaN is a signaling NaN or a quiet NaN. If the most significant fraction bit is set, then it is a quiet NaN. The value of the character sequence for a binary floating point NaN can be:

- **`""`** equals a NULL string, the default quiet NaN.
- **`"Q"`** or **`"q"`** equals the default quiet NaN.
- **`"S"`** or **`"s"`** equals the default signaling NaN.
- **`"xxxxxxxx"`** equals hex digits representing the fraction bits starting with the most significant bit. Trailing zeros maybe omitted. The hex digits may be prefixed optionally by **`"0x"`**.
- **`"l_xxxxxx"`** equals where l is either a **`"Q"`** or **`"q"`**, for a quiet NaN or a **`"S"`** or **`"s"`**, for a signaling NaN, followed by an underscore and hex digits representing the fraction bits starting with the most significant bit. Trailing zeros maybe omitted.

Technically, the traditional Hex Floating Point does not support NaNs. However, there are certain bit patterns that are not valid numbers, or rather are treated as if they had the value of zero, that have traditionally been used to represent missing or invalid values. These missing values have one or more bits set in the sign and/or exponent fields and all zeros in the fraction field. The **`nan`**, **`nanf`**, and **`nanl`** functions have been implemented for HFP to return these values. Therefore, the value of the character sequence for Hexadecimal floating point numbers is a string consisting of two hexadecimal characters, optionally prefixed by **`0x`**, that represent the bit values for the sign and exponent of the return value. By default, that is, when the **`nan`**, **`nanf`**, or **`nanl`** functions are called with a NULL string, a missing value of negative zero is returned, that is, only the sign bit is set. Otherwise, these functions will return a value based on the hex string pointed to by **`tagp`**. For example, calling **`nan("0xaa")`** will return a floating point zero value with the bit pattern **`0xaa00000000000000`**.

*Note:* Because these values are equivalent to zero for HFP, that is, **`0.0 == nan("0xaa")`** is TRUE, this does not violate the ANSI C99 standard, which states that any implementations of these functions that do not support NaNs must return zero. △

## EXAMPLE

Some results from calling BFP **`nan`**:

```
call         bit pattern returned (hex)
_________    ____________________

nan("")   =   0x7ff8000000000000
-nan("")  =   0xfff8000000000000
nan("Q")  =   0x7ff8000000000000
-nan("Q") =   0xfff8000000000000
nan("q")  =   0x7ff8000000000000
-nan("q") =   0xfff8000000000000

nan("S")  =   0x7ff4000000000000
-nan("S") =   0xfff4000000000000
nan("s")  =   0x7ff4000000000000
-nan("s") =   0xfff4000000000000

nan("q_8fff")  =    0x7ff8fff000000000
-nan("q_8fff") =  0xfff8fff000000000
nan("8fff")    =  0x7ff8fff000000000
-nan("8fff")   =  0xfff8fff000000000
nan("0x8fff")  =  0x7ff8fff000000000
-nan("0x8fff") =  0xfff8fff000000000

nan("s_7fff")  =  0x7ff7fff000000000
-nan("s_7fff") =  0xfff7fff000000000
nan("7fff")    =  0x7ff7fff000000000
-nan("7fff")   =  0xfff7fff000000000
nan("0x7fff")  =  0x7ff7fff000000000
-nan("0x7fff") =  0xfff7fff000000000
```

Some results from calling HFP **nan**:

```
call         bit pattern returned (hex)
_________    ____________________

nan("")       =  0x8000000000000000
nan("0x03") =  0x0300000000000000
nan("03")    =  0x0300000000000000
nan("0xff") = 0xff00000000000000
nan("ff")    =  0xff00000000000000
```

### RELATED FUNCTIONS

**strtod**, **strtof**, **strtold**

### SEE ALSO

"Mathematical Functions" in Chapter 2, "Function Categories."

## Updates to the SIGFPE Signal

In Chapter 5, "Signal-Handling Functions," replace the **SIGFPE** description with the following entry.

## SIGFPE

**General Computational Error**

The **SIGFPE** signal is raised whenever a computational error occurs. These errors include hexadecimal floating-point overflow or underflow, integer or hexadecimal floating point division by zero, and binary floating-point traps such as overflow and inexact. Note that integer overflow never causes a signal; when integer overflow occurs, the result is reduced to 32 bits or 64 bits, depending on type, by discarding the most significant bits and is then interpreted as a signed integer.

### Default handling

The default handling for **SIGFPE** is to raise a more specific signal. One of the signals **SIGBFPE**, **SIGFPDIV**, **SIGFPOFL**, **SIGFPUFL** or **SIGIDIV** is raised. Note that **SIGBFPE** is raised for any binary floating-point (IEEE) trap. **SIGFPDIV**, **SIGFPOFL** and **SIGFPUFL** are raised only for hexadecimal floaating point. Handling of the more specific signal depends on whether a handler has been defined for it. Refer to the descriptions of each of these signals for more details.

### Ignoring the signal

If your program ignores **SIGFPE**, the result of the computation that raises **SIGFPE** is undefined, unless the computation causes an underflow. For hexadecimal floating-point underflows, the result is set to zero.

### Information returned by siginfo

If you call **siginfo** after a **SIGFPE** signal occurs, **siginfo** returns a pointer to a structure of type **FPE_t**. This structure is defined as:

```
typedef struct {
   int int_code; /* interrupt code */
   union {
      __near int *intv; /* 4-byte integer result */
      __near long long *llongv; /* 8-byte integer result */
      __near __hexfmt float *floatv; /* hex format float result */
      __near __hexfmt double *doublev; /* hex format double result */
      __near __binfmt float *bfloatv; /* bin format float result */
      __near __binfmt double *bdoublev;/* bin format double result */
   } result;
   __near char *EPIE: /* pointer to hardware program check info */
   __near double *fpregs; /* contents of floating regs 0,2,4,6 */
   __near double *fpregs16; /* contents of all floating regs */
   int res_size; /* size of result (0, 4 or 8) */
   int bfptrap; /* BFP exception bits for trap */
} FPE_t;
```

The **int_code** field contains the number of the more specific signal associated with this error, which will be one of **SIGBFPE**, **SIGFPDIV**, **SIGFPOFL**, **SIGFPUFL**, or **SIGIDIV**. The **result** field is a pointer to the result of the computation that raises the signal. If you want to continue processing, you can change the value that **result** points to. Note that the **result** field may be zero if a floating-point exception occurs in an instruction which has no result or an integer result.

The **EPIE** field is a pointer to a control block containing hardware information available at the time the signal occurs. (This information includes program status word and registers.) For information on the **EPIE** format, see the IBM publication *MVS/XA Supervisor Services and Macro Instructions*. The **EPIE** pointer always addresses a memory block mapped like an OS/390 **EPIE**, even in environments such as CICS where a different format is used internally.

The **fpregs** field is a pointer to an array of four doubles, containing the contents of floating point registers 0, 2, 4 and 6 at the time of the signal. The **fpregs16** field is a pointer to an array of doubles containing the contents of all 16 floating point registers. (On systems with only 4 floating-point registers, the undefined registers are stored as zeroes.) Note that the register contents may be either in hexadecimal or binary format, and that a cast may possibly be needed to interpret them correctly.

The **reg_size** field indicates the size of the result, 4 for an **int** or **float** result, 8 for a **long long** or **double** result, and 0 for a case where no result pointer was stored.

he **bfptrap** field is meaningful only when **int_code** is **SIGBFPE**. Then, it contains one or more of the floating-point bits defined in **fenv.h**. For instance, if **bfptrap** is **FE_OVERFLOW+FE_INEXACT**, it indicates that the error was a combination of an overflow condition with the inexact condition.

### Notes on defining a handler

If you define a handler for **SIGFPE** , you can determine what type of error caused the signal by testing the **int_code** field of the information returned by **siginfo**. You can also use this information to reset the result of the computation by changing the value that **result** points to. Refer to the example in the description of the **siginfo** function for an illustration of this technique.

### USS Considerations

When **SIGFPE** is managed by USS, the default action for **SIGFPE** is abnormal process termination, and **SIGFPE** is never converted into another signal. If you want to handle one or more of the **SIGBFPE**, **SIGFPDIV**, **SIGFPOFL**, **SIGFPUFL**, or **SIGIDIV** signals specific to SAS/C, you must define **SIGFPE** as a signal managed by SAS/C.

## New Multiple Heap Functions

Several new functions have been added allow a program to make use of more than one heap. All programs contain at least 2 heaps; a library heap and a standard heap. Library routines that need dynamic memory will allocate from the library heap. Normally, this memory is internally used and not accessable to the user. The standard heap is where memory will be allocated from by calls to **malloc**, **pool**, **calloc** etc. This heap can be predefined with attributes other than the default. At program start, the standard heap is also the default heap. Additional heaps can be created and declared as the default heap by *pushing* the new heap onto the heap stack. Calls to **malloc calloc**, **free**, **pool**, **realloc** (For brevity, the rest of this description will use **malloc** as the general term for getting heap memory.) will allocate memory from whatever is the current default heap. Additional functions are available to get memory from a specific heap without making it the default heap.

A heap is represented by a heap ID, which is an opaque token returned by the **hpcreate** function. **hpcreate** is passed a heap attribute structure, which defines the required behavior and properties of the heap. **multheap.h** defines the various attributes that can be set for a heap. These include location in memory (that is, above or below the bar/line) subpool number, initial and overflow sizes, whether the heap can be shared between subtasks and more. For above the bar heaps (heaps with virtual addresses greater than 2G), the maximum guaranteed size of the heap is 2G, although the heap may reach a hard limit of 4G. If more than 2G of contiguous memory is needed, then the **IARV64** function should be used.

The attributes of the standard heap can be set by defining an external variable **_heapdef** and initializing it with the required attributes. For example, to have the standard heap be placed above the 2G bar, specify:

```
extern _heap_attr _heapdef = {0, 0, 0, 0, 0, 0, HPLOC_64}
```

Attributes specified in this way override those set through previous methods (the **_heap** variable and the **=/mmm** runtime option).

---

## _heap_attr

The **_heap_attr** structure in **multheap.h** defines the attributes of a heap. This structure is passed to **hpcreate** to create a new heap with these characteristics.

**int version;**
version number set by call to hpattr.

**int flags;**
Miscellaneous flags to represent heap attributes.

**HPATTR_SHARED**
Indicates that the heap can be shared between cooperating subtasks. Memory can be passed from one subtask to another and the receiver can then free the memory. Care must be taken however to serialize access to the memory. If more than one subtask updates the memory at the same time, results are unpredictable. The library serializes the allocation and deallocation of this memory.

**HPATTR_ZERO**
Indicates that the heap will be initialized to all zeros before it is passed to the user program.

**HPATTR_NOUSAGE**
Indicates that the heap will not be included in any usage report. See **=usage** for more information.

**HPATTR_NOABEND**
Indicates that eyecatchers in heap control blocks will not be checked. Normally, if the eyecatchers are overlayed or storage pointers are corrupted, the library will an ABEND in the range of U1205-U1209. Setting this attribute will let the program proceed without abending in the event of a storage overlay. This option should only be used for programs where an ABEND would have a significantly worse consequence than running with bad data.

**HPATTR_RETAIN**
Indicates that the heap will not be returned to the operating system until a call is made to **hpdestroy**.

**HPATTR_SIGOK**
Indicates that the heap will be eligible for use in signal handling code. If **malloc** is called from a signal handler and the default heap does not have this indication, then the most recent heap created with this indication will be

used for allocation. The non-shared heap list is searched before looking at the shared heaps.

**`char *area;`**
If not NULL, then initial allocations will be taken from the memory addressed by **`area`**. This in effect allows a heap within a heap. There is no protection against the owner of this area freeing the storage so care must be taken when using this field. The envisioned use of this would be for a number of callers to be able to use separate heaps without the overhead of getting the memory directly from the operating system. Rather, a large allocation would be done for all of the callers and multiple virtual heaps would be created from that area.

**`unsigned int initial;`**
Size of the initial heap allocation. It must be greater than 0.

**`unsigned int overflow;`**
Size for needed additional allocations. The default is 4K. If it is set to 0, then no secondary allocations will be performed. That is, any **`malloc`**s beyond the initial amount will fail.

**`int subpool;`**
In MVS, the subpool (if it is below the bar) to allocate from. If it is not specified, 13 will be assumed unless the heap is to be shared, in which case the default subpool is 131.

**`char amode;`**
Intended addressing mode of the user. Determines the location of the allocated memory in the address space. Only **`HPLOC_64`** will get memory above the 2G bar.

**`HPLOC_BELOW`**
Memory will be allocated below the 16M line.

**`HPLOC_ANY`**
Memory will be allocated above the 16M line if possible, otherwise from below the 16M line.

**`HPLOC_64`**
Memory will be above the 2G bar.

**`HPLOC_RES`**
Memory will be allocated below the 16M line for callers running AMODE24. And above the line for callers running AMODE31 or AMODE64. If memory is not available above the line, an attempt will be made to allocate memory below the line.

## Huge Pointer Functions

The following library functions accept **`__huge`** pointer arguments from or return **`__huge`** pointers to callers compiled with the **`HUGEPTRS`** option. Note that while many **`stdio.h`** functions are on the list, only pointer arguments which address input or output data are **`__huge`**. Arguments which are FILE pointers, or pointers to control data such as **`fpos_t`** remain **`__near`**.

| | | | | |
|---|---|---|---|---|
| **`afread`** | **`afreadh`** | **`afread0`** | **`afwrite`** | **`afwriteh`** |
| **`afwrite0`** | **`atoi`** | **`atol`** | **`atoll`** | **`bsearch`** |
| **`calloc`** | **`fgets`** | **`format`** | **`fprintf`** | **`fputs`** |
| **`fread`** | **`fscanf`** | **`fwrite`** | **`gets`** | **`hpalloc`** |

| | | | | |
|---|---|---|---|---|
| **hpcalloc** | **hpfree** | **hppoolcreate** | **hprealloc** | **IARV64** |
| **malloc** | **memcasecmp** | **memchr** | **memcmp** | **memcpy** |
| **memfil** | **memlwr** | **memmove** | **memscan** | **memscntb** |
| **memset** | **memupr** | **memxlt** | **nan** | **nanf** |
| **nanl** | **nanl** | **pdel** | **pfree** | **pool** |
| **printf** | **puts** | **qsort** | **realloc** | **scanf** |
| **snprintf** | **sprintf** | **sscanf** | **stcpm** | **stcpma** |
| **strcasecmp** | **strcat** | **strchr** | **strcmp** | **strcpy** |
| **strcspn** | **strlen** | **strlwr** | **strncasecmp** | **strncat** |
| **strncmp** | **strncpy** | **strpbrk** | **strrchr** | **strrcspn** |
| **strrpbrk** | **strrspn** | **strscan** | **strscntb** | **strspn** |
| **strstr** | **strtod** | **strtof** | **strtok** | **strtol** |
| **strtold** | **strtoll** | **strtoull** | **strupr** | **strxlt** |
| **TPUT** | **TPUT_ASID** | **TPUT_USERID** | **TGET** | **vformat** |
| **vsnprintf** | **vsprintf** | **vsscanf** | **WTO** | **WTOR** |
| **WTP** | | | | |

## hpalloc

**Allocate Memory from Heap**

### SYNOPSIS

```
#include <multheap.h>
void *hpalloc(_heap_id heapid, size_t size);
```

### DESCRIPTION

**hpalloc** allocates a block of dynamic memory of the size specified by **size** from a heap indicated by **heapid**, which was returned by **hpcreate** when it was initialized.

### RETURN VALUE

**hpalloc** returns the address of the first character of the new block of memory. The allocated block is suitably aligned for storage of any type of data.

### ERRORS

User ABEND 1205 or 1206 may occur if memory management data areas are overlaid.

## DIAGNOSTICS

If adequate memory is not available, the **heapid** is invalid, or if 0 bytes are requested, NULL is returned.

## IMPLEMENTATION

See **malloc** for a description of normal memory allocation. **hpalloc** is, in effect, the same as calling **hppush(heapid)**, **malloc(size)**, **hppop()**.

## EXAMPLE

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <multheap.h>
char *source, *copy;
main()
{
   source = "A simple line for the hpalloc example ";

   /* Allocate space for a copy for source. */
   copy = hpalloc(_HEAP_STD, strlen(source) + 1);

   /* Copy if there is space. */
   if (copy)
   {
      strcpy(copy,source);
      puts(copy);
   }
   else puts("hpalloc failed to allocate memory for copy.");
}
```

## RELATED FUNCTIONS

**malloc**, **pool**, **hppoolcreate**

## SEE ALSO

"Memory Allocation Functions" in Chapter 2, "Function Categories" of the *SAS/C Library Reference, Volume 1.*

## hpattr

### Initialize _heap_attr Structure

## SYNOPSIS

```
#include <multheap.h>
void hpattr(_heap_attr __near *attr);
```

## DESCRIPTION

**`hpattr`** initializes a **`_heap_attr`** structure. After initialization, members of the structure should be set to indicate qualities of the heap to be created. By default, the version, subpool and overflow members are set to default values. At Version 1, the subpool default is 13 and the overflow size is 4K. See "_heap_attr" on page 103 for more information on the flags and fields of the **`_heap_attr`** structure.

## RETURN VALUE

**`hpattr`** does not return a value.

## EXAMPLE

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <multheap.h>
_heap_attr attr;
main()
{
   _heap_id id1, id2;

// create a user heap

   hpattr(&attr);
   attr.amode = HPLOC_BELOW;
   attr.subpool = 14;
   attr.flags = HPATTR_SIGOK + HPATTR_NOABEND;
   attr.initial = 1000;
   id1 = hpcreate(&attr, "first");
   if (!id1)
   {
      WTP("error in hpcreate.\n");
      return -1;
   }
   WTP("id1 = %04hX\n", id1);

// create second heap with slightly different
// parms.

   attr.amode = HPLOC_ANY;
   attr.subpool = 15;
   attr.initial = 3000;
   id2 = hpcreate(&attr, "second");
   if (!id2)
   {
      WTP("error in hpcreate.\n");
      return -1;
   }
   WTP("id2 = %04hX\n", id2);
}
```

## RELATED FUNCTIONS

**`hpcreate`**

## SEE ALSO

"Memory Allocation Functions" in Chapter 2, "Function Categories" of the *SAS/C Library Reference, Volume 1.*

## hpcalloc

**Allocate and Clear Memory from Heap**

### SYNOPSIS

```
#include <multheap.h>
void *hpcalloc(_heap_id heapid, size_t n,
size_t size);
```

### DESCRIPTION

**`hpcalloc`** allocates a block of dynamic memory from the heap indicated by **`heapid`** to contain **`n`** elements of the size specified by size . The block is cleared to binary zeroes before return.

### RETURN VALUE

**`hpcalloc`** returns the address of the first character of the new block of memory. The allocated block is suitably aligned for storage of any type of data.

### ERRORS

User ABEND 1205 or 1206 may occur if memory management data areas are overlaid.

### DIAGNOSTICS

If adequate memory is not available, the **`heapid`** is invalid, or if 0 bytes are requested, NULL is returned.

### IMPLEMENTATION

See **`calloc`** for a description of normal memory allocation. hpcalloc in effect is the same as calling **`hppush(heapid)`**, **`calloc(size)`**, **`hppop()`**.

### EXAMPLE

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <multheap.h>

double *identity(_heap_id heapid, int size) {
   double *matrix;
   int i;
```

```
      matrix = hpcalloc(heapid, sizeof(double), size*size);
      if (matrix == NULL) return(NULL);
      for (i = 0; i < size; ++i)
         matrix[size*i + i] = 1.0;
      return matrix;
   }
```

## RELATED FUNCTIONS

**`calloc`**, **`pool`**, **`hppoolcreate`**

## SEE ALSO

"Memory Allocation Functions" in Chapter 2, "Function Categories" of the *SAS/C Library Reference, Volume 1.*

# hpcreate

**Create New Heap**

## SYNOPSIS

```
#include <multheap.h>
_heap_id hpcreate(_heap_attr __near *ha,
                  char __near *name);
```

## DESCRIPTION

**`hpcreate`** creates a new heap based on attributes from the structure pointed to by **`ha`**. **`name`** can either be NULL or point to a unique null-terminated string (truncated to 16 bytes) that will name the heap for diagnostic purposes.

## RETURN VALUE

**`hpcreate`** returns the ID of the newly created heap. This ID is used to specify the heap to be used by other multi-heap functions.

## DIAGNOSTICS

If the heap attribute structure pointed to by **`ha`** is invalid, NULL is returned. Possible reasons for rejecting a heap attribute structure include the following: the version number was invalid (possibly indicating an uninitialized structure), a subpool other than 131 was specified for a shared heap, or the name was associated with another heap. NULL may also be returned if **`hpcreate`** was unable to allocate memory for control information.

## IMPLEMENTATION

Only control information is allocated during **`hpcreate`**. Heap storage will not be allocated from the heap until **`hpalloc`**, **`hpcalloc`**, or another multiheap storage function is invoked.

## EXAMPLE

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <multheap.h>
_heap_attr attr;
main()
{
   _heap_id id1;

// create a user heap - shared and above
//  the bar

   hpattr(&attr);
   attr.flags = HPATTR_SHARED;
   attr.amode = HPLOC_64;
   attr.subpool = HPSUBPOOL_DEFAULT;
   attr.initial = 1000000000;
   attr.overflow = 0;
   id1 = hpcreate(&attr, "Above The Bar1");
   if (!id1)
   {
      WTP("error in hpcreate.\n");
      return -1;
   }
   WTP("id1 = %04hX\n", id1);

}
```

## RELATED FUNCTIONS

**hpalloc**, **hpcalloc**, **hppoolcreate**

## SEE ALSO

"Memory Allocation Functions" in Chapter 2, "Function Categories" of the *SAS/C Library Reference, Volume 1.*

---

## hpdestroy

**Destroy Heap**

## SYNOPSIS

```
#include <multheap.h>
void *hpdestroy(_heap_id heapid);
```

## DESCRIPTION

**hpdestroy** destroys a heap and returns all its memory back to the operating system. Any further use of memory allocated from the heap will be invalid.

## RETURN VALUE

**hpdestroy** returns a code indicating success (0) or failure (-1) of the operation.

## IMPLEMENTATION

The control information assocated with the heap is freed. Be sure not to call **hpdestroy** for a shared heap that may be in use by another task.

## EXAMPLE

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <multheap.h>
main()
{
  _heap_attr attr;
  _heap_id id1;
  DCB_t *input, *output;
  DECB_t input_DECB, output_DECB;
  char *buf;
  int count = 0;
  int err;

  input = osbdcb(0);
  memcpy(input->DCBDDNAM, "INPUT ", 8);
  if (osbopen(input, "input")) {
    puts("Input open failed.");
    exit(16);
  }
  output = osbdcb(out_exlst);

  /* Copy output file characteristics from input. */
  output->DCBRECFM = input->DCBRECFM;
  output->DCBLRECL = input->DCBLRECL;
  output->DCBBLKSI = input->DCBBLKSI;
  memcpy(output->DCBDDNAM, "OUTPUT ", 8);
  if (osbopen(output, "output")) {
    puts("Output open failed.");
    osbclose(input, "", 1);
    exit(16);
  }

// Get a below the line heap for use in BSAM I/O

  hpattr(&attr);
  attr.amode = HPLOC_BELOW;
  attr.flags = HPATTR_SIGOK + HPATTR_NOABEND;
  attr.initial = 4000;
  id1 = hpcreate(&attr, "heap below");
  if (!id1) {
    Log("error in hpcreate.\n");
    return -1;
  }
```

```
   buf = hpalloc(id1, input->DCBBLKSI);

   for (;;) {
     buf = hpalloc(id1, input->DCBBLKSI);
     osread(input_DECB, input, buf, 0);
     if ((err = oscheck(input_DECB)) != 0) {
       if (err != -1) puts("Input error.");
       break;
     }
     oswrite(output_DECB, output, buf, 0);
     if (oscheck(output_DECB) != 0) {
       if (full) puts("Output file full.");
       else puts("Output error.");
       break;
     }
     ++count;
   }

   /* Rather than freeing all the records, just free the
   entire heap. */

   hpdestroy(id1);
 }
```

## RELATED FUNCTIONS

**hpcreate**, **hppoolcreate**

## SEE ALSO

"Memory Allocation Functions" in Chapter 2, "Function Categories" of the *SAS/C Library Reference, Volume 1.*

## hpextract

### Retrieve Heap ID

### SYNOPSIS

```
#include <multheap.h>
_heap_id hpextract(void);
```

### DESCRIPTION

**hpextract** returns the ID of the current (default) heap. This ID can be used to verify which heap is about to be used for **malloc** or **free**.

### RETURN VALUE

**hpextract** returns the ID of the current (default) heap. If this ID is the same as **_HEAP_STD**, then the default heap is the same as the standard heap. This is always true at entry to the main function. The default heap can be changed by calls to **hppush** and **hppop**.

## IMPLEMENTATION

For a program using more than one coprocess, it is possible that the default heap can be different for each coprocess. Whenever a coprocess switch occurs, if the new coprocess is using a different heap than the old coprocess, then the default heap is switched as well.

## EXAMPLE

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <multheap.h>
main()
{
_heap_id id1;

...


  id1 = hpextract();
  // if the default heap has been changed
  if (id != _HEAP_STD)
  {
    hppush(_HEAP_STD); // switch to standard heap
  }

  link = calloc(1, 20);

  if (id != _HEAP_STD)
  {
    hppop; // restore current heap
  }

...
>
}
```

## RELATED FUNCTIONS

**hpalloc**, **hpcalloc**, **hpfree**, **hpdestroy**, **hppush**

## SEE ALSO

“Memory Allocation Functions” in Chapter 2, “Function Categories” of the *SAS/C Library Reference, Volume 1.*

# hpfree

**Free a Block of Memory from Heap**

## SYNOPSIS

```
#include <multheap.h>
void hpfree(_heap_id heapid, void *block);
```

## DESCRIPTION

**hpfree** frees a block of dynamic memory from a heap indicated by **heapid**.

## RETURN VALUE

**hpfree** has no return value.

## ERRORS

User ABEND 1206, 1207, or 1208 may occur if memory management data areas are overlaid. User ABEND 1208 will probably occur if the block pointer is invalid; that is, if it does not address a previously allocated area of memory that has not already been freed.

## IMPLEMENTATION

In most cases if an entire page of memory is unused after a free call, the page is returned to the operating system (unless the pointer is in the heap's initial allocation). For memory above the bar, the memory is not returned until the heap is freed by a call to **hpdestroy**.

## EXAMPLE

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <multheap.h>
main()
{
char *source, *copy;

   source = "A simple line for the hpfree example ";
   /* Allocate space for a copy for source. */
   copy = hpalloc(_HEAP_STD, strlen(source) + 1);
   /* Copy if there is space. */
   if (copy)
   {
     strcpy(copy,source);
     puts(copy);
   }
     else puts("hpalloc failed to allocate memory for copy.");

   hpfree(_HEAP_STD, copy);
}
```

## RELATED FUNCTIONS

**hpalloc**, **hpcalloc**, **hppoolcreate**

### SEE ALSO

"Memory Allocation Functions" in Chapter 2, "Function Categories" of the *SAS/C Library Reference, Volume 1.*

## hppoolcreate

**Allocate a Storage Pool from Heap**

### SYNOPSIS

```
#include <multheap.h>
int hppoolcreate(_heap_id heapid, POOL_t *p,
                 unsigned eltsize, unsigned initial,
                 unsigned extend)
```

### DESCRIPTION

**hppoolcreate** creates a storage pool from which elements of a given size can be quickly allocated and freed. **hppoolcreate** differs from **pool**, in that it will allocate it's elements from a given heap. The arguments are as follows:

| | |
|---|---|
| **heapid** | is an ID returned from a call to **hpcreate**. |
| **p** | is a pointer to a **POOL_t** structure. |
| **eltsize** | is the size of the elements to be allocated. |
| **initial** | is the number of elements the pool is to contain initially. |
| **extend** | is the number by which the pool is extended if all elements are allocated. |

If **initial** is 0, the **hppoolcreate** routine computes a convenient initial number of elements. If **extend** is 0, it is set equal to **initial**.

In a situation that requires allocation of many items of the same size, using a storage pool is more efficient than using **malloc** in terms of execution time. It also can be more efficient in terms of storage usage if the **initial** and **extend** values are reasonably chosen.

### RETURN VALUE

The return value is 1 if a pool is successfully created, or 0 if it is not. If a pool is created, its address and other information is stored in the area addressed by the second argument to **hppoolcreate**.

### ERRORS

User ABEND 1205 or 1206 may occur if memory management data areas are overlaid.

### DIAGNOSTICS

The **POOL_t** pointer is set to 0 if there is no storage available for the new pool.

## IMPLEMENTATION

See **pool** for a description of normal pool allocation. **hppoolcreate** is identical except for the fact that the heap can be chosen to be other than the current (default) heap.

## EXAMPLE

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <multheap.h>

static POOL_t word_pool;

main()
{

   _heap_id id1;
   _heap_attr attr;
   int success;

// create a user heap ¼ shared and above
//  the bar

   hpattr(&attr);
   attr.flags = HPATTR_SHARED;
   attr.amode = HPLOC_64;
   attr.subpool = HPSUBPOOL_DEFAULT;
   attr.initial = 1000000000;
   attr.overflow = 0;
   id1 = hpcreate(&attr, "Above The Bar1");
   if (!id1)
   {
      WTP("error in hpcreate.\n");
      return -1;
   }
   WTP("id1 = %04hX\n", id1);

      /* Allocate a pool of binary tree elements */
      /* to hold some "words". */
   success = hppoolcreate(id1, &word_pool, sizeof(word_t), 100, 100)
   if (!success) {
      puts("Can’t allocate word pool.");
      exit(4);
   }
}
```

## RELATED FUNCTIONS

**malloc**, **pool**, **hppoolcreate**

## SEE ALSO

“Memory Allocation Functions” in Chapter 2, “Function Categories” of the *SAS/C Library Reference, Volume 1*.

## hppop

**Restore Previous Heap as Current Heap**

### SYNOPSIS

```
#include <multheap.h>
int hppop(void);
```

### DESCRIPTION

**hppop** removes the heap specified by the previous call to **hppush** from the heap stack, restoring the previous default heap. If no heaps are on the heap stack, a warning will be produced.

### RETURN VALUE

**hppop** returns 0 if the prior heap is restored and -1 if there are no currently pushed heaps.

### IMPLEMENTATION

Each coprocess (including the implicit coprocess that represents the main program) has its own *heap stack*, which is maintained independently.

### EXAMPLE

See the example for "hpextract" on page 112.

### RELATED FUNCTIONS

**hppush**, **hpcreate**

### SEE ALSO

"Memory Allocation Functions" in Chapter 2, "Function Categories" of the *SAS/C Library Reference, Volume 1.*

## hppush

**Save and Change Current Heap**

### SYNOPSIS

```
#include <multheap.h>
int *hppush(_heap_id heapid, int autopop);
```

### DESCRIPTION

**hppush** changes the default heap to the heap identified by **heapid**. If **autopop = HPPUSH_AUTOPOP**, when the function that calls **hppush** returns, the heap will be reset to its state on entry to the calling function.

### RETURN VALUE

**hppush** returns 0 if the function succeeds, -1 if the heapid was invalid, or -2 if insufficient memory was available to complete the operation.

### ERRORS

User ABEND 1205 or 1206 may occur if memory management data areas are overlaid.

### IMPLEMENTATION

Using **hppush** to change the default heap allows code that calls **malloc**, **calloc**, and so forth to make use of a different heap than that which is provided as the standard heap. The **autopop** specification HPPUSH_AUTOPOP may be useful in specifying a default heap to be used only for the duration of a specific function and its descendants. It is especially useful if the function has many different return statements, or if it may be terminated by **longjmp** from a signal handler.

### EXAMPLE

See the example for “hpextract” on page 112.

### RELATED FUNCTIONS

**hppop**, **hpcreate**

### SEE ALSO

“Memory Allocation Functions” in Chapter 2, “Function Categories” of the *SAS/C Library Reference, Volume 1*.

## hprealloc

**Change the Size of an Allocated Memory Block in Heap**

### SYNOPSIS

```
#include <multheap.h>
void *hprealloc(_heap_id heapid, void *p,
size_t size);
```

### DESCRIPTION

**hprealloc** decreases or increases the size of a memory block previously allocated, possibly moving it to another location and heap, or to another heap. The new storage will be allocated from the heap specified by **`heapid`**. **`p`** points to the previously allocated memory block. **`size`** is the size of the new block. The contents of the old block are preserved in the new block after reallocation, unless the old size is greater than the new size. If the old size is greater, the unwanted extra bytes are lost. When the new size is larger than the old size, the contents of the new block that follow the data from the old block are unpredictable.

## RETURN VALUE

**`hprealloc`** returns the address of the first character of the new block of memory. The allocated block is suitably aligned for storage of any type of data. If a new memory block cannot be allocated, the contents of the location that **`p`** points to are not changed, and **`hprealloc`** returns NULL.

## ERRORS

User ABEND 1205 or 1206 may occur if memory management data areas are overlaid.

## DIAGNOSTICS

If adequate memory is not available, the **`heapid`** is invalid, or if 0 bytes are requested, NULL is returned.

## IMPLEMENTATION

See **`hpalloc`** for a description of normal memory allocation.

## EXAMPLE

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <multheap.h>


char **table, **temp, *item;
unsigned table_size, max_elem;
_heap_id id1;
_heap_attr attr;

  hpattr(&attr);
  attr.initial = 4000;
  id1 = hpcreate(&attr, "realloc heap");

   /* Determine if table size is too small.             */
if (max_elem >= table_size) {
   table_size *= 2;    /* Double table size.            */

      /* Allocate more space for table. */
   temp = hprealloc(id1, (char*)table, table_size*sizeof(char*));

      /* If reallocation is successful, copy address of */
      /*  new area to table.                            */
   if (temp)
      table = temp;
   else {
      puts("Item table overflow");
      exit(16);
   }
}
```

### RELATED FUNCTIONS

**hpalloc**, **hpcreate**

### SEE ALSO

"Memory Allocation Functions" in Chapter 2, "Function Categories" of the *SAS/C Library Reference, Volume 1.*

## Updates to the fprintf Function

In Chapter 6, "Function Descriptions," replace the **fprintf** function description with the following entry.

## fprintf

**Write Formatted Output to a File**

### SYNOPSIS

```
#include <stdio.h>
 int fprintf(FILE *f, const char *format, var1, var2, ...);
```

### DESCRIPTION

**fprintf** writes output to the stream associated with the **FILE** object addressed by **f** under the control of the string addressed by **format**. The argument list following **format** may contain one or more additional arguments whose values are to be formatted and transmitted.

**format** points to a string that contains ordinary characters (not including **%**) that are sent without modification to the file and 0 or more conversion specifications. *Conversion specifications* begin with the **%** character. The **%** character may be followed by these specifications:

- zero or more modifier flags
- an optional minimum field width specified by a decimal integer
- an optional precision in the form of a period (.) followed by a decimal integer
- a conversion modifier (**~b**, **~h**, **~n**, **~f**, **~l**)
- an optional **hh**, **h**, **l**, **ll**, **j**, **z**, **t**, or **L**
- one of the characters **d**, **i**, **o**, **u**, **x**, **X**, **f**, **e**, **E**, **g**, **G**, **c**, **s**, **n**, **p**, or **V** that specifies the conversion to be performed.

Here are the modifier flags:

| | |
|---|---|
| – | left-justifies the result of the conversion within the field. |
| **+** | always precedes the result of a signed conversion with a plus sign or minus sign. |
| **space** | precedes the result of a signed conversion with a space or a minus sign. (If both **space** and **+** are used, the **space** flag is ignored.) |
| **#** | uses an alternate form of the conversion. This flag affects the **o** and **x** (or **X**) integer-conversion specifiers and all of the floating-point conversion specifiers. For **o** conversions, the **#** flag forces the result to have a leading 0. For **x** (or **X**) conversion, the result of the conversion is prefixed with **0x** (or **0X**). For **e**, **E**, **f**, **g**, and **G** conversions, the **#** flag causes the result of the conversion to always have a decimal indicator. For **g** and **G** conversions, the **#** indicates that trailing 0s are *not* to be removed. |
| **0** | for **d**, **i**, **o**, **u**, **x**, **X**, **e**, **E**, **f**, **g**, and **G** conversions, leading **0**s are used to pad the field width. (If both **-** and **0** are used, the **0** flag is ignored.) For **d**, **i**, **o**, **u**, **x**, and **X** conversions, the **0** flag is ignored if a precision is specified. |
| @ | for conversions that specify or dereference a pointer (**p**, **s**, **n**, **V**), treat pointers as **__far**. |

The field width specifies the minimum number of characters in the converted value. If the value has fewer characters than that specified by the field width, it is padded on the left (or right, if the – flag is used). By default, the pad character is a blank.

The precision specifies the minimum number of digits to appear for the **d**, **i**, **o**, **u**, **p**, **x,** and **X** conversions. For the **e**, **E,** and **f** conversions, the precision specifies the number of digits to appear after the decimal indicator. For the **g** and **G** conversions, the precision specifies the maximum number of significant digits to appear. Finally, the precision specifies the maximum number of characters to be used in the **s** conversion.

If the precision is explicitly given as 0 and the value to be converted is 0, no characters are written. If no precision is specified, the default precision is 0. The actual width of the field is the wider of that specified by the field width and that specified by the precision.

Precision might be followed by the conversion modifer specified as a tilde (~) followed by a character. The following are acceptable conversion modifier values:

| | |
|---|---|
| **~b** | argument is binary floating point (IEEE), **__binfmt**. |
| **~h** | argument is hex floating point, **__hexfmt**. |
| **~n** | argument pointer is a **__near** pointer. |
| **~f** | argument pointer is a **__far** pointer. |
| **~l** | argument pointer is a **__hugeptr** pointer. |

An * may be used for either the field width, the precision, or both. If used, the value of the field width or precision is supplied by an **int** argument. This argument appears in the argument list before the argument to be converted. A negative value for the field width is taken as a **-** (left-justify) flag followed by a positive field width. A negative value for the precision is ignored.

An **hh** before a **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier indicates that the conversion applies to a **char** or **unsigned char**. An **hh** before an **n** conversion specifier indicates that the conversion applies to a pointer to a **char**.

An **h** before a **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier indicates that the conversion applies to a **short int** or **unsigned short int**. An **h** before an **n** conversion specifier indicates that the conversion applies to a pointer to a **short int**.

An **l**, **z**, or **t** before a **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier indicates that the conversion applies to a **long int** or an **unsigned long int**. An **l** before an **n** conversion specifier indicates that the conversion applies to a pointer to a **long int**. An **L** before an **e**, **E**, **f**, **g**, or **G** conversion specifier indicates that the conversion applies to a **long double**.

An **ll**, or **j** before a **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier indicates that the conversion applies to a **long long int** or **unsigned long long int**. An **ll**, or **j** before an **n** conversion specifier indicates that the conversion applies to a pointer to a **long long int**.

An **L** before an **e**, **E**, **f**, **g**, or **G** conversion specifier indicates that the conversion applies to a **long double**.

The type of conversion to be performed is specified by one of these characters:

| | |
|---|---|
| **a, A** | converts the corresponding **double** argument to hexadecimal notation in the style **[-]0x.hhhp+dd**, where there is one hexadecimal digit (which is nonzero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-point character and the number of hexadecimal digits after it is equal to the precision. If the precision is zero and the # flag is not specified, no decimal point character appears. The exponent always contains at least one digit, and only as many additional digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent is zero. |
| **c** | converts the corresponding **int** argument to **unsigned char** and writes the character. |
| **d**, **i** | converts the corresponding **int** argument to decimal notation. |
| **e**, **E** | converts the corresponding **double** argument to the form **[-] d.ddde± dd** or **[-] d.dddE± dd**. The precision has the same effect as with the **f** conversion. The exponent will be a minimum of two digits. |
| **f** | converts the corresponding **double** argument to the form **[-] ddd.ddd**. The precision indicates the number of digits after the decimal indicator. If no precision is given, the default is 6. If the precision is given as 0, a decimal indicator is not used. If a decimal indicator is used, at least one digit appears before it. |
| **g**, **G** | converts the **double** argument using the **f** or **e** (or **E**) format. The precision specifies the number of significant digits in the converted result. An **e** conversion is used if the exponent is greater than the precision or is less than –3. Unless the **#** (alternate form) flag is used, trailing 0s are removed. The decimal indicator appears only if followed by a digit. |
| **n** | writes a number into the string addressed by the corresponding **int *** argument. The number written is the number of characters written to the output stream so far by this call to **fprintf**. |
| **o** | converts the corresponding **unsigned int** argument to octal notation. |
| **p** | converts the **void *** argument to a sequence of printable characters. In this implementation, **p** is converted as if **x** were specified. |

| | |
|---|---|
| **s** | writes characters from the string addressed by the corresponding **`char *`** argument until a terminating null character ('\0') is encountered or the number of characters specified by the precision have been copied. The null character, if encountered, is not written. |
| **u** | converts the corresponding **`unsigned int`** argument to decimal notation. |
| **V** | is the same as the **`%s`** conversion specifier, except that it expects the corresponding argument to be a pointer to a PL/I or Pascal format varying-length character string. See the *SAS/C Compiler Interlanguage Communication Feature User's Guide* for more information on this conversion specifier. |
| **x**, **X** | converts the corresponding **`unsigned int`** argument to hexadecimal notation. The letters abcdef are used for **x** conversion and ABCDEF for **X** conversion. |

A **`%`** character can be written by using the sequence **`%%`** in the format string. The **`fprintf`** formats are described in more detail in the ISO/ANSI C standard.

In support of installations that use terminals with only uppercase characters, this implementation of **`fprintf`** accepts any of the lowercase format characters in uppercase. Use of this extension renders a program nonportable.

## CAUTION

Binary floating point (**`BFP`**) NaN values will always have a minimum of 10 characters with a maximum of 22 characters. Trailing zeros are stripped. The following are two examples of maximum output:

| | |
|---|---|
| Negative Quiet NaN | **`-nan(q_0ffffffffffff)`** |
| Positive Signaling NaN | **`+nan(S_00000000000001)`** |

## RETURN VALUE

**`fprintf`** returns the number of characters transmitted to the output file.

## DIAGNOSTICS

If there is an error during output, **`fprintf`** returns a negative value.

## PORTABILITY

The **`%V`** format is an extension and is not portable. The
~ and **`@`** modifiers are not portable, either.

## IMPLEMENTATION

The format string can also contain multibyte characters. For details on how **`fprintf`** handles multibyte characters in the format string and in conversions, see Chapter 11, "Multibyte Character Functions," in the *SAS/C Library Reference, Volume 2*.

**`fprintf`** can only produce up to 512 characters per conversion specification, except for %s and %V conversions, which are limited to 16 megabytes.

### EXAMPLE

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

main()
{
  int i;
  double x;
  FILE *sysout;

      /* Print a columnar table of logs and square roots to an */
      /*  MVS SYSOUT data set.                                 */
   sysout = fopen("dsn:sysout=a", "w");
   if (!sysout) abort();
   fprintf(sysout, " x %10s  log(x) %10s  sqrt(x)\n\n", " ", " ");

      /* Print heading. */
   for(i = 1; i <= 20; ++i {
         x = i;
         fprintf(sysout, "%3d%10s%10.5f%10s%10.5f\n",
                 i, " ", log(x), " ", sqrt(x));
   }
   exit(EXIT_SUCCESS);
}
```

### RELATED FUNCTIONS

**fscanf**, **printf**, **sprintf**, **vfprintf**

### SEE ALSO

- Chapter 3, "I/O Functions" in *SAS/C Library Reference, Volume 1*
- "I/O Functions" in Chapter 2, "Function Categories," in *SAS/C Library Reference, Volume 1*

## Updates to the fscanf Function

In Chapter 6, "Function Descriptions," replace the **fscanf** function description with the following entry.

## fscanf

**Read Formatted Input from a File**

**Portability:** ISO/ANSI C conforming, UNIX compatible

### SYNOPSIS

```
#include <stdio.h>
```

```
int fscanf(FILE *f, const char *format, loc1, loc2, ...);
```

## DESCRIPTION

**`fscanf`** reads formatted input from the **`FILE`** designated by **`f`** according to the format specified by the string **`format`**. Following the format in the argument list may be one or more additional pointers (**`loc1, loc2,..., locn`**), addressing storage where the input values are stored.

**`format`** points to a string that contains zero or more of the following:

- □ white-space characters
- □ regular characters (not including %)
- □ conversion specifications.

The format string contains format specifiers or characters to be matched from the input. Format items have the following form:

```
%[*][ width][~type] [h | l | L | hh | z | t | ll | j]form
```

The specifiers have the following meanings:

- □ An asterisk (*) indicates that an input item is processed according to the format, but its value is not stored.
- □ If a value for **`width`** is present, **`width`** specifies the maximum width of the input item.
- □ A tilde (~) indicates a non-portable, SAS/C argument type specifier (type). The following argument type specifiers are supported:
  - □ **`~b`** specifies that the corresponding argument is a default pointer to a binary floating point (BFP) number.
  - □ **`~h`** specifies that the corresponding argument is a default pointer to a hexadecimal floating point (HFP) number.
  - □ **`~n`** specifies that the corresponding argument is a near pointer. This specifier may optionally be followed by a period and one of the following letters, **`b`**, **`h`**, **`n`**, **`f`**, or **`l`** to indicate that the corresponding argument is a near pointer to a binary or hexadecimal floating point number, that is, **`~n.b`** or **`~n.h`**, or a near pointer to a near, far, or huge pointer, that is, **`~n.n`**, **`~n.f`**, or **`~n.l`**. Note that the latter three specifiers are valid only with the **`p`** form. See the list item for **`form`** below.
  - □ **`~f`** specifies that the corresponding argument is a far pointer. This specifier may optionally be followed by a period and one of the following letters, **`b`**, **`h`**, **`n`**, **`f`**, or **`l`** to indicate that the corresponding argument is a far pointer to a binary or hexadecimal floating point number, that is , **`~f.b`** or **`~f.h`**, or a far pointer to a near, far, or huge pointer, that is , **`~f.n`**, **`~f.f`**, or **`~f.l`**. Note that the latter three specifiers are valid only with the **`p`** form. See the list item for **`form`** below.
  - □ **`~l`** specifies that the corresponding argument is a huge (long) pointer. This specifier may optionally be followed by a period and one of the following letters, **`b`**, **`h`**, **`n`**, **`f`**, or **`l`** to indicate that the corresponding argument is a huge pointer to a binary or hexadecimal floating point number, that is , **`~l.b`** or **`~l.h`**, or a far pointer to a near, far, or huge pointer, that is , **`~l.n`**, **`~l.f`**, or **`~l.l`**. Note that the latter three specifiers are valid only with the **`p`** form. See the list item for **`form`** below.
  - □ **`~.{n | f | l}`** specifies that the corresponding argument is a default pointer to one of the pointer types indicated by the letter following the period. The letters, **`n`**, **`f`**, and **`l`**, following the period indicate that the corresponding

argument is a default pointer to a near, far, or huge pointer. This is valid only with the **p** form. See the list item for **form** below.

- An optional letter has the following meanings:
  - An **hh** before a **d**, **i**, or **n** conversion specifier indicates that the corresponding argument is a pointer to **char** instead of **int**.
  - An **h** before a **d**, **i**, or **n** conversion specifier indicates that the corresponding argument is a pointer to **short int** instead of **int**.
  - An **l**, **z**, or **t** before a **d**, **i**, or **n** conversion specifier indicates that the corresponding argument is a pointer to **long int** instead of **int**.
  - An **ll**, or **j** before a **d**, **i**, or **n** conversion specifier indicates that the corresponding argument is a pointer to **long long int** instead of **int**.
  - An **hh** before an **o**, **u**, or **x** conversion specifier indicates that the corresponding argument is a pointer to **unsigned char** instead of **unsigned int**.
  - An **h** before an **o**, **u**, or **x** conversion specifier indicates that the corresponding argument is a pointer to **unsigned short int** instead of **unsigned int**.
  - An **l**, **z**, or **t** before an **o**, **u**, or **x** conversion specifier indicates that the corresponding argument is a pointer to **unsigned long int** instead of **unsigned int**.
  - An **ll**, or **j** before an **o**, **u**, or **x** conversion specifier indicates that the corresponding argument is a pointer to **unsigned long long int** instead of **unsigned int**.
  - An **l** before an **a**, **e**, **f**, or **g** conversion specifier indicates that the corresponding argument is a pointer to **double** instead of **float**.
  - An **L** before an **a**, **e**, **f**, or **g** conversion specifier indicates that the corresponding argument is a pointer to **long double** instead of **float**.
- **form** is one of the following characters, defining the type of the corresponding target object and the expected format of the input:

| | |
|---|---|
| **a, A, e, E, g, or G** | matches a floating-point number. The corresponding argument should be **float ***. |
| **c** | matches a sequence of characters specified by width. If no width is specified, one character is expected. A null character is not added. The corresponding argument should point to an array large enough to hold the sequence. |
| **d** | matches an optionally signed decimal integer whose format is the same as expected for the subject sequence of **strtol** with **base=10**. The corresponding argument should be **int ***. |
| **i** | matches an optionally signed decimal integer, which may be expressed in decimal, in octal with a leading 0, or in hexadecimal with a leading 0x. The corresponding argument should be **int ***. |
| **n** | indicates that no input is consumed. The number of characters read from the input stream so far by this call to **fscanf** is stored in the object addressed by the corresponding **int *** argument. |
| **o** | matches an optionally signed octal integer. The corresponding argument should be **unsigned int ***. |
| **p** | matches a pointer in the format written by the **%p printf** format. This implementation treats **%p** like **%x**. The corresponding argument should be **void ****. |

| | |
|---|---|
| **s** | matches a sequence of nonwhite-space characters. A terminating null character is automatically added. The corresponding argument should point to an array large enough to hold the sequence plus the terminating null character. |
| **u** | matches an optionally signed integer. The corresponding argument should be **`unsigned int *`**. |
| **x**, **X** | matches a hexadecimal integer. The corresponding argument should be **`unsigned int *`**. |
| [ ] or < > | matches a string comprised of a particular set of characters. A terminating-null character is automatically added. The corresponding argument should point to an array large enough to hold the sequence plus the terminating-null character. Note that you cannot use the two-character sequences ([ ] and < >) to replace the brackets in a **`fscanf`** format. |

The format string is a C string. With the exception of the **`c`** and **`[`** or **`<`** specifiers, white-space characters in the format string cause white-space characters in the input to be skipped. Characters other than format specifiers are expected to match the next nonwhite-space character in the input. The input is scanned through white space to locate the next input item in all cases except the **`c`** and **`[    ]`** specifiers, where the initial scan is bypassed. The **`s`** specifier terminates on any white space.

The **`fscanf`** formats are described in more detail in the ISO/ANSI C standard. As an extension, uppercase characters may also be used for the format characters specified in lowercase in the previous list.

## RETURN VALUE

**`fscanf`** returns **`EOF`** if end of file (or an input error) occurs before any values are stored. If values are stored, it returns the number of items stored; that is, the number of times a value is assigned with one of the **`fscanf`** argument pointers.

## DIAGNOSTICS

**`EOF`** is returned if an error occurs before any items are matched.

## IMPLEMENTATION

The format string can also contain multibyte characters. For details on how **`fscanf`** treats multibyte characters in the format string and in conversions, see Chapter 11 in the *SAS/C Library Reference, Volume 2*.

Because square brackets do not exist on some 370 I/O devices, the library allows the format **`%[xyz]`** to be replaced by the alternate form **`%<xyz>`**. This is not a portable format.

## EXAMPLE

This example writes out the data stored in lines to a temporary file and reads them back with **`fscanf`**:

```
#include <stdio.h>
#include <stdlib.h>

static char *lines[] = {
   "147.8 pounds\n"
```

```
      "51.7 miles\n",
      "4.3 light-years\n",
      "10000 volts\n",
      "19.5 gallons\n"
   };

   main()
   {
      FILE *tmpf;
      int i;
      __binfmt float amount;  // Declare a binary floating point number
      char unit[20];
      int count;

      tmpf = tmpfile();
      if (!tmpf){
            puts("Couldn't open temporary file.");
            exit(EXIT_FAILURE);
      }

      for (i = 0; i < sizeof(lines)/sizeof(char *); ++i){
         fputs(lines[i], tmpf);
      }
      rewind(tmpf);
      for(;;){

   // Note: auto variables (stack allocated) are always 31-bit addressable.
   //       As such the use of '~n' and the cast to a __near pointer are
   //       not really necessary but are shown here as an example of
   //       combining two SAS/C specifiers; i.e., near(~n) and binary(~b)

         count = fscanf(tmpf, "%~n.bf %s",
                        (__binfmt float __near *)&amount, unit);
         if (feof(tmpf)) break;
         if (count < 2){
            puts("Unexpected error in input data.");
            exit(EXIT_FAILURE);
         }
            printf("amount = %~bf, units = \"%s\"\n", amount, unit);
      }
      fclose(tmpf);
   }
```

## RELATED FUNCTIONS

**fprintf**, **scanf**, **sscanf**

## SEE ALSO

- Chapter 3, "I/O Functions" in the *SAS/C Library Reference, Volume 1*
- "I/O Functions" in Chapter 2, "Function Categories" in the *SAS/C Library Reference, Volume 1*

## Updates to the strtod Function

In Chapter 6, “Function Descriptions,” replace the **strtod** function description with the following entry.

## strtod, strtof, strtold

**Convert a String to Double**

**Portability:** ISO/ANSI C conforming, UNIX compatible

### SYNOPSIS

```
#include <stdlib.h>
double strtod(const char *str, char **end);
float strtof(const char *str, char **end);
long double strtold(const char *str, char **end);
```

### DESCRIPTION

**strtod**, **strtof**, and **strtold**, expect a floating-point number in C syntax, with these specifications:

- a decimal point may be omitted
- a + or - sign may precede the number
- no type suffix (**F** or **L**) is allowed
- can include one of the character sequences **INF** or **INFINITY**, ignoring case
- can include one of the character sequences **NAN** or **NAN(character sequence)**, ignoring case in the **NAN** part. See the **nan** function description for information about character sequence.

If the **end** value is not **NULL**, ***end** is modified to address the first character of the string that is not consistent with the floating-point syntax above. However, if no initial segment of the string can be interpreted as a floating-point number, **str** is assigned to ***end**.

### RETURN VALUE

**strtod**, **strtof**, and **strtold** functions return the floating-point value (**double**, **float**, and **long double**) represented by the character string up to the first unrecognized character. If no initial segment of the string can be interpreted as a floating-point number, 0.0 is returned.

### DIAGNOSTICS

If the floating-point value is outside the range of valid floating-point numbers, **errno** is set to ERANGE. In this case, **±HUGE_VAL** (defined in <**math.h**>) is returned if the correct value is too large, or 0.0 if the correct value is too close to 0.

### EXAMPLE

```
#include <stdlib.h>
#include <ctype.h>
#include <stdio.h>

main()
{
   double number;
   char *input, *stopchar;
   char string[20];

   puts("Enter a string to be converted to double:");
   input = gets(string);

      /* Skip space characters.            */
   while(isspace(*input)) ++input;

      /* Convert from character to double. */
   number = strtod(input, &stopchar);
      /* Determine if string is valid.     */
   if (stopchar == input)
      printf("Invalid float number: %s\n", input);

      /* Check for characters afterwards.  */
   else if (*stopchar && !isspace(*stopchar))

   printf("Extra characters after value ignored: %s\n", stopchar);
   printf("The entered string was converted to: %g\n", number);
}
```

### RELATED FUNCTIONS

**nan**, **strtol**

### SEE ALSO

"String Utility Functions" in Chapter 2, "Function Categories."

## New Library Functions

## IARV64

**64-Bit Virtual Storage Allocation**

**Portability:** SASC

### SYNOPSIS

```
#include  <osiarv64.h>
int IARV64(char __near *request, ...);
```

## DESCRIPTION

The **IARV64** function implements the functionality of the **IARV64** assembler macro. The **request** argument is the address of a null-terminated string. The remainder of the argument list is a list of keywords followed, in most cases, by an argument specifying a value for the keyword. The list is terminated by the **_Iend** keyword. The **request** argument must be one of the following:

**GETSTOR**
Create a memory object.

**DETACH**
Free one or more memory objects.

**PAGEFIX**
Fix physical pages within one or more memory objects.

**PAGEUNFIX**
Unfix physical pages within one or more memory objects.

**PAGEOUT**
Notify the system that data within physical pages of one or more memory objects will not be used in the near future.

**PAGEIN**
Notify the system that data within physical pages of one or more memory objects are needed in the near future.

**DISCARDDATA**
Discard data within physical pages of one or more memory objects.

**CHANGEGUARD**
Request that a specified range in a memory object be changed from guard area to usable area or from usable area to guard area.

**LIST**
Requests a list of memory objects.

The supported keywords and their associated data are as follows. All references to pointer operands should be considered **__near** unless specifically stated as being **__huge**.

- The **_Ireason** keyword is equivalent to the Assembler **RSNCODE=**, which identifies an optional output parameter to hold the reason code from register 0. The next argument should be a pointer to an **int** where the reason code will be placed.
- The **_Icond** keyword is equivalent to the Assembler **COND=**, which specifies whether the request is conditional or uncoditional. The next argument should be the string **YES** for a conditional request, or **NO** for an unconditional request.
- The **_Isegments** keyword is equivalent to the Assembler **SEGMENTS=**, which specifies the size of the memory object in megabytes. The next argument should be a **long long** integer.
- The **_Ikey** keyword is equivalent to the Assembler **KEY=**, which specifies the storage key to be assigned to the memory object. The next argument should be a unsigned character with bits 0-3 containing the key to be used. Bits 4-7 are ignored.
- The **_Ifprot** keyword is equivalent to the Assembler **FPROT=**, which idicates whether the memory should be fetch-protected. The next argument should be the string **YES** to fetch-protect memory, or **NO** to indicate that the memory will not be fetch-protected.
- The **_Isvcdumprgn** keyword is equivalent to the Assembler **SVCDUMPRGN=**, which idicates whether the memory object should be included in an SVC dump or LIST

request. The next argument should be the string **YES** to include the memory object (**SDATA=RGN** must be set in the dump options), or **NO** to indicate that the memory object will not be included in an SVC dump or LIST request.

- The **_Icontrol** keyword is equivalent to the Assembler **CONTROL=**, which indicates whether the memory object must be freed by an authorized caller. The next argument should be the string **AUTH** to indicate that only an authorized caller can free the object (and that the object is eligible for **PAGEFIX** and **PAGEUNFIX**), or **UNAUTH** to indicate that the object cannot be freed by an unauthorized caller.
- The **_Iusertkn** keyword is equivalent to the Assembler **USERTKN=**, which identifies a user token to be identified with a memory object. The next argument should be a **long long** integer.
- The **_Iguardsize** keyword is equivalent to the Assembler **GUARDSIZE=**, which specifies the length in megabytes of the guard area to be created at the high or low end of the memory object. The next argument should be an **unsigned int**.
- The **_Iguardloc** keyword is equivalent to the Assembler **GUARDLOC=**, which indicates at which end of the memory object the guard area should be placed. The next argument should be the string **HIGH** to place the guard area at the top of the memory object, or **LOW** to place the guard area at the bottom of the memory object.
- The **_Ittoken** keyword is equivalent to the Assembler **TTOKEN=**, which identifies a task to be assigned ownership of the memory object. The next argument should be a pointer to a 16 byte array. The array should be set from the result of a call to the assembler **TCBTOKEN** macro.
- The **_Iorigin** keyword is equivalent to the Assembler **ORIGIN=**, which identifies where the address of a returned memory object will be placed. The next argument should be a **__near** pointer to a **__huge** pointer (type **__huge * __near ***) where the address should be stored. Note that the caller does not have to be compiled with the **hugeptrs** option. The returned **__huge** pointer is not dereferenced by this function. If the **_Iguardloc** parameter indicates that the guard area is **LOW**, then the returned **__huge** pointer must be bumped past the guard area before it is dereferenced.
- The **_Imatch** keyword is equivalent to the Assembler **MATCH=**, which specifies whether an operation should be performed on a single memory object or all memory objects associated with a given usertoken. The next argument should be the string **SINGLE** to indicate only one memory object, or **USERTOKEN** to indicate all memory objects matching the usertoken indicated in the **_Iusertkn** parameter.
- The **_Imemobjstart** keyword is equivalent to the Assembler **MEMOBJSTART=**, which identifies the address of the first byte of a memory object. The next argument should be a **__huge** pointer to a memory object.
- The **_Iowner** keyword is equivalent to the Assembler **OWNER=**, which specifies whether a task other than the owning task (or a task specifying the correct **TTOKEN**) may free a memory object. The next argument should be the string **YES** to indicate that the task freeing the memory object must be the owner, or specify the correct **TTOKEN** for the memory object, or **NO** to indicate that the task freeing the memory object doesn not have to be the owner or provide the correct **TTOKEN**. Only supervisor state programs or programs running with a **PSW** key between 0-7 inclusive may use this parameter.
- The **_Iranglist** keyword is equivalent to the Assembler **RANGLIST=**, which points to a list of 16 byte entries describing memory ranges. The number of entries is specified by **_Inumrange**. The next argument should be a **__huge** pointer to an array of entries consisting of starting address (itself a **__huge** pointer which must address a byte on a physical page boundry) and a **long long** containing the number of pages for a range.

- □ The **_Inumrange** keyword is equivalent to the Assembler **NUMRANGE=**, which identifies the number of entries contained in the associated **Iranglist** list. The next argument should be an **int** containing the number of entries.
- □ The **_Iclear** keyword is equivalent to the Assembler **CLEAR=**, which indicates whether the storage specified in a **DISCARDDATA** request will be cleared to binary zeros. The next argument should be the string **YES** to indicate that the memory should be cleared, or **NO** to indicate that the memory will be indeterminate after the **DISCARDDATA** request has completed.
- □ The **_Iconvert** keyword is equivalent to the Assembler **CONVERT=**, which specifies in the **CHANGEGUARD** request whether the memory will be converted from or to a guard area. The next argument should be the string **TOGUARD** to indicate that memory should be converted from virtual storage to guard area, or **FROMGUARD** to indicate that the memory should be converted to virtual storage from the guard area. How the memory object was created determines whether the guard area is at the top or bottom of the memory object.
- □ The **_Iconvertsize** keyword is equivalent to the Assembler **CONVERTSIZE=**, which specifies the size in megabytes of memory to be transferred to or from the guard area. The next argument should be an **int** containing the number of megabytes to be transferred.

The following parameters may only be specified in an authorized program.

- □ The **_Ilong** keyword is equivalent to the Assembler **LONG=**, which indicates whether the expected duration of a **PAGEFIX** is long or short. If the time is expected to be more than a few seconds, then **YES** should be used. The next argument should be the string **YES** to indicate that the **PAGEFIX** is expected to be of long duration, or **NO** to indicate that the **PAGEFIX** is expected to be of short duration.
- □ The **_Ialetvalue** keyword is equivalent to the Assembler **ALETVALUE=**, which specifies an **ALET** for the address space to be pagefixed. The only valid **ALET**s are 0 (primary) and 2 (home). The next argument should be an **int** containing either 0 or 2.
- □ The **_Iv64listptr** keyword is equivalent to the Assembler **V64LISTPTR=**, which points to a buffer to hold the returned information from a **LIST** request. The next argument should be a **__near** pointer to a buffer where the list information will be returned. **osiarv64.h** contains C structures converted from the **IAXV64WA** assembler **DSECT** that can be used to map the returned list.
- □ The **_Iv64listlength** keyword is equivalent to the Assembler **V64LISTLENGTH=**, which specifies the size of the buffer to contain the results from a **LIST** request. The next argument should be an **int** containing the length of the buffer.
- □ The **_Iv64select** keyword is equivalent to the Assembler **V64SELECT=**, which indicates whether the **LIST** request is for all or just a subset of the allocated memory objects. The next argument should be the string **YES** to indicate that the **LIST** request is only for a subset of the allocated memory objects, or **NO** to indicate that the **LIST** request should return all allocated memory object data.
- □ The **_Iend** keyword indicates the end of the list of keywords.

## RETURN VALUE

**IARV64** returns the value in **R15** after calling the **IARV64** OS macro. If the parms are in error though, it returns -1.

## IMPLEMENTATION

The **IARV64** function is implemented by the source module **L$UIARV**.

## EXAMPLE 1

This SPE example gets one megabyte of above the bar memory (compile with the **HUGEPTRS** option):

```
#include <osmain.h>
#include  <osiarv64.h>
#include   <oswto.h>

char *string1; //(this pointer is __huge due to compile option)
int rc;

void osmain()
{


     rc = IARV64("GETSTOR", _Isegments, 1ll,
              _Iorigin, (__near *) &string1,  _Iend);


   WTP("GETSTOR rc = %i", rc);
   strcpy(string1, "hello world.");
      WTP("string1 = %~hpX, '%s'", string1, string1);
}
```

## EXAMPLE 2

This example gets some storage in system keys and then displays a list of memory objects via the **LIST** request. This example is valid only if it is compiled with **hugeptrs**.

```
int num, x, reason;
long long adr;
char list[255];
struct V64WAHEADER *pHeader;
struct V64WAENTRY *pEntry;
char *string1;


_ldregs(R1, 0x0000000C);
_ossvc(107);              // switch to supervisor state
rc = IARV64("GETSTOR", _Isegments, 1ll,
            _Ikey, 0x20,
            _Iorigin, (__near *) &string1,  _Iend);
WTP("GETSTOR rc = %i", rc);
rc = IARV64("GETSTOR", _Isegments, 5000ll,
            _Ikey, 0,
            _Iorigin, (__near *) &string1,  _Iend);
rc = IARV64("LIST", _Iv64listptr, (__near *) list,
            _Ireason, (__near *) &reason,
            _Iv64listlength, sizeof(list), _Iend);
_ldregs(R1, 0x00000004);
_ossvc(107);              // return to problem state
if (rc == 0)
{
   pHeader = list;
   WTP("V64WARETURNCODE = %i", pHeader->V64WARETURNCODE);
```

```
      WTP("V64WANUMDATAAREAS = %i", pHeader->V64WANUMDATAAREAS);
      pEntry = (struct V64WAENTRY *)
               ((char *) pHeader + V64WAHEADER_LEN);
      x = 1;
      memcpy(&num, &pHeader->V64WANUMDATAAREAS, 4);
      while (num--)
      {
         WTP("x = %i", x++);
         WTP(" Storage key = %02hhX", pEntry->v64waflag);
         memcpy(&adr, &pEntry->v64wastart64, 8);
         WTP(" Start: %016llX", adr);
         memcpy(&adr, &pEntry->v64waend64, 8);
         WTP("   End: %016llX", adr);
         pEntry++;
      }
   }
```

## RELATED FUNCTIONS

**hpcreate**, **hpalloc**

## osdltok

**Delete a token associated with a given name**

**Portability:** SASC

### SYNOPSIS

```
#include <osnamtok.h>
int osdltok(char __near *name);
```

### DESCRIPTION

The **osdltok** function uses a **name** (up to 16 characters) to delete the associated token. Although **IEANTDL** can be called in assembler with the bytes of the name consisting of any value, this function expects the name to be a null-terminated character string. The actual name passed to the **IEANTDL** service will consist of the character string truncated to 16 bytes, or padded with blanks. IBM has reserved names starting with the letters A through I (uppercase) and the null character ('\0'). The level of the name/token will be 2 (home).

### RETURN VALUE

**osdltok** returns either -1 if the service cannot be called or the return value from the service.

### IMPLEMENTATION

The **osdltok** function is implemented by the source module **L$UNMTK**.

## EXAMPLE

This SPE example retrieves the address of storage that had previously been allocated in another task and then frees it.

```
#include <osnamtok.h>
#include <getmain.h>
#include <oswto.h>

char *name = "Z Global";

typedef struct
{
     char eye[16];   // eyecatcher
     long long lock;   // lockword
     char *data;    // data chain
} GLOBAL, *PGLOBAL;


void osmain()
{
  PGLOBAL pGlobal;
     char token[16];
    int rc;

    if ((rc = osgttok(name, token)) != 0)
    {
        WTP("osgttok failed.");
         return;
    }
    memcpy(&pGlobal, token, 4); // only the first 4 bytes of
                       //  the token are used.
    if (strcmp(pGlobal->eye, "Z Global EYE")
    {
       WTP("Global storage corrupt.");
       return;
    }

    // now delete the token

    if ((rc = osdltok(name)) != 0)
    {
       WTP("osdltok failed.");
        return;
    }

    // and free the common storage

    FREEMAIN(pGlobal, sizeof(GLOBAL), 131, UNCOND);

}
```

## RELATED FUNCTIONS

**ossttok**, **osgttok**

## osgttok

**Retrieve a token associated with a given name**

**Portability:** SASC

### SYNOPSIS

```
#include  <osnamtok.h>
int osgttok(char __near *name, char __near *token);
```

### DESCRIPTION

The **osgttok** function uses a name (up to 16 characters) to retrieve a **token** (16 bytes binary) using the **IEANTRT** service. Although **IEANTRT** can be called in assembler with the bytes of the name consisting of any value, this function expects the name to be a null-terminated character string. The actual name passed to the **IEANTRT** service will consist of the character string truncated to 16 bytes, or padded with blanks. IBM has reserved names starting with the letters A through I (uppercase) and the null character (even though this function cannot create a name beginning with the null character.) The token parameter should be the address of an array of 16 bytes. This array will be overlaid by the call to osgttok with the token retrieved. The level of the name/token will be 2 (home).

### RETURN VALUE

**osgttok** returns either -1 if the service cannot be called or the return value from the service.

### IMPLEMENTATION

The **osgttok** function is implemented by the source module **L$UNMTK**.

### EXAMPLE

This SPE example retrieves the address of storage that had previously been allocated in another task and then frees it.

```
#include <osnametok.h>
#include <getmain.h>
#include <oswto.h>

char *name = "Z Global";

typedef struct
{
     char eye[16];   // eyecatcher
     long long lock;   // lockword
     char *data;    // data chain
} GLOBAL, *PGLOBAL;


void osmain()
```

```
{
  PGLOBAL pGlobal;
     char token[16];
    int rc;

    if ((rc = osgttok(name, token)) != 0)
    {
        WTP("osgttok failed.");
         return;
    }
    memcpy(&pGlobal, token, 4); // only the first 4 bytes of
                       //  the token are used.
    if (strcmp(pGlobal->eye, "Z Global EYE")
    {
       WTP("Global storage corrupt.");
       return;
    }

    // now delete the token

    if ((rc = osdltok(name)) != 0)
    {
       WTP("osdltok failed.");
        return;
    }

    // and free the common storage

    FREEMAIN(pGlobal, sizeof(GLOBAL), 131, UNCOND);

}
```

## RELATED FUNCTIONS

**ossttok**, **osdltok**

## ossttok

**Save a token associated with a given name**

**Portability:** SASC

### SYNOPSIS

```
#include  <osnamtok.h>
int ossttok(char __near *name, char __near *token);
```

### DESCRIPTION

The **ossttok** function associates a name (up to 16 characters) with a token (16 bytes binary) using the **IEANTCR** service. Although **IEANTCR** can be called in assembler with the bytes of the name consisting of any value, this function expects the name to be a null-terminated character string. The actual name passed to the **IEANTCR** service will consist of the character string truncated to 16 bytes, or padded with blanks. IBM has reserved names starting with the letters A through I (uppercase) and the null character (even though this function can not create a name beginning with the null character.) The token parameter should be the address of an array of 16 bytes. The level of the name/token will be 2 (home) and the persistence will be 0 (IEANT_NOPERSIST).

## RETURN VALUE

**ossttok** returns either -1 if the service cannot be called or the return value from the service.

## IMPLEMENTATION

The **ossttok** function is implemented by the source module **L$UNMTK**.

## EXAMPLE

This SPE example gets some memory from **subpool 131** and creates a name/token pair for use in other tasks accessing the memory.

```
#include  <osnamtok.h>
#include  <getmain.h>
#include  <oswto.h>

char *name = "Z Global";

typedef struct
{
     char eye[16];   // eyecatcher
     long long lock;  // lockword
     char *data;   // data chain
} GLOBAL, *PGLOBAL;


void osmain()
{
  PGLOBAL pGlobal;
     char token[16];
    int rc;

    if ((rc = GETMAIN_U(sizeof(GLOBAL), 131, LOC_ANY, &pGlobal)) != 0)
    {
        WTP("Getmain error.");
        return;
    }

    memset(pGlobal, 0, sizeof(GLOBAL);

    strcpy(pGlobal->eye, "Z Global EYE");

    memset(token, 0, 16);
    memcpy(token, &pGlobal, 4); // copy address as token
                       //  remaining 12 bytes of token
```

```
                           //  are currently unused.

      if ((rc = ossttok(name, token)) != 0)
      {
         WTP("ossttok failed.");
          return;
      }

   }
```

## RELATED FUNCTIONS

**osgttok**, **osdltok**

---

## WTP

**Write to Programmmer**

**Portability:** SASC

### SYNOPSIS

```
   #include <oswto.h>
   int WTP(char *format, ...);
```

### DESCRIPTION

The **WTP** function implements the functionality of the OS/390 assembler **WTO** macro with options set to indicate routing to the programmer. The **format** argument is the address of a null-terminated string. This can optionally be followed by one or more variables to be output according to the format characters in the input string. The **vformat** function is used to convert the variable list. See **vformat** for a list of acceptable conversion characters.

### RETURN VALUE

**WTP** returns 0 if the **WTP** macro was successful. If the **WTP** macro fails, it returns the return code from the macro, which will be a positive value. **WTP** might return -2 if there was not enough memory to perform the **WTP**.

### IMPLEMENTATION

The **WTP** function is implemented by the source module **L$UWTO**.

### EXAMPLE

This example uses **WTP** to send two single-line programmer's messages:

```
      #include <oswto.h>
      int iLine;

      iLine = 20;
```

```
WTP("Error discovered at line: %i", iLine);
WTP("Aborting...");
```

## New Message Exit Facility

In Chapter 2, "Function Categories," in the section titled, "Diagnostic Control Functions," add the following text after the first paragraph.

Additionally, the functions **wmi_set** and **wmi_del** allow you to define exits which control the processing of library diagnostics. You can define an exit to selectively suppress messgaes, to add text to messages, and to capture messages for your own processing, for example, writing them to a log file. Additionally, the functions **wmifilte** and **wmifiltn** are simple message exits to suppress specific messages, either by message number or by **errno** value. See the function writeups for more detailed information on writing library message exits.

Add the following diagnostic control functions to the existing list:

| | |
|---|---|
| **wmi_del** | cancel a previously requested message exit |
| **wmi_set** | specify a message exit |
| **wmifilte** | filter library messages by errno value |
| **wmifiltn** | filter library messages by message number. |

Add the following functions, **wmi_del**, **wmifilte**, **wmifiltn**, and **wmi_set**, to Chapter 6, "Function Descriptions."

## wmi_del

**Delete a message exit**

**Portability:** SAS/C

### SYNOPSIS

```
#include <wmi.h>
int wmi_del(wmi_token token);
```

### DESCRIPTION

**wmi_del** is called to cancel a message exit defined by **wmi_set**. The token argument should contain the value returned by **wmi_set** when the exit was defined.

### RETURN VALUE

**wmi_del** returns **0** if the exit was successfully deleted, or a non-zero value if there was an error, for instance, if the token value does not represent a defined exit.

### EXAMPLE

See the example for **wmi_add**.

## RELATED FUNCTIONS

**wmi_set**, **wmifilte**, **wmifiltn**

## SEE ALSO

"Diagnostic Control Functions" in Chapter 2, "Function Categories."

## wmifilte, wmifiltn

**Selectively suppress diagnostic messages**

**Portability:** SAS/C

### SYNOPSIS

```
#include <wmi.h>
enum wmi_outcome wmifilte(int msgnum, int errnoval,
                          struct wmi_msg_info *info, void *arg);
enum wni_outcome wmifiltn(int msgnum, int errnoval,
                          struct wmi_msg_info *info, void *arg);
```

### DESCRIPTION

**wmifilte** and **wmifiltn** are two message exit routines supplied by the library to allow selective suppression of library messages by **errno** value or by message number. These functions are not intended to be called directly by the application program. Rather, they are intended to be defined (with the **wmi_set** function) as message exits with an argument specifying the messages to be suppressed. A typical use of **msgfiltn** or **msgfilte** in this way might look like the following:

```
int errorlist[] = { /* list of error or message numbers */ };
wmi_token filter;

filter = wmi_set(&wmifiltn, (void *) &errorlist);
/* perform processing */
wmi_del(filter);
```

The second argument passed to **wmi_set** for these functions should point to an array of integers, terminated by a zero entry. For **wmifilte**, the list contains one or more **errno** values. For **wmifiltn**, it should contain one or more message numbers.

### IMPLEMENTATION

These functions are very simple applications of the SAS/C message exit facility. Their source code is as follows:

```
enum wmi_outcome wmifilte(int msgnum, int errnoval, struct wmi_msg_info *info,
                          void *arg)
{
   int *list;

   list = (int *) arg;
   while(*list)
      if (*list++ == errnoval)
         return WMI_SUPPRESS;
   return WMI_PASS;
}

enum wmi_outcome wmifiltn(int msgnum, int errnoval, struct wmi_msg_info *info,
                          void *arg)
{
   int *list;

   list = (int *) arg;
   while(*list)
      if (*list++ == msgnum)
         return WMI_SUPPRESS;
   return WMI_PASS;
}
```

## EXAMPLE

The following code calls the **fopen** function, suppressing any message for the **errno ENFOUND**.

```
#include <wmi.h>
#include <stdio.h>
#include <errno.h>

char *filename;
int ENFOUND_list[] = { ENFOUND, 0 };
wmi_token filt_token;

filt_token = wmi_set(&wmifilte, &ENFOUND_list);
fopen(filename, "r");
wmi_del(filt_token);
```

The following is similar code that is specific to OS/390, and which deletes messages indicating *file not found* by message number.

```
#include <wmi.h>
#include <stdio.h>

char *filename;
int not_found_list[] = { 500, 503, 504, 509, 544, 557, 878, 0 };
wmi_token filt_token;

filt_token = wmi_set(&wmifiltn, &_found_list);
fopen(filename, "r");
wmi_del(filt_token);
```

## RELATED FUNCTIONS

**wmi_set**, **wmi_del**

## SEE ALSO

“Diagnostic Control Functions” in Chapter 2, “Function Categories.”

## wmi_set

**Establish a warning message exit**

**Portability:** SAS/C

### SYNOPSIS

```
#include <wmi.h>

typedef enum wmi_outcome (*wmi_exittype)
            (int msgtnum, int errnoval, struct wmi_msg_info *info,
             void *arg);     /* This typedef is in wmi.h */

wmi_token wmi_set(wmi_exittype exitptr, void *arg);
```

### DESCRIPTION

**wmi_set** is called to define a warning message exit. The exit will be called, with a few exceptions, whenever the library begins processing a runtime warning message. The exit may request that the message be suppressed or that it be forced to print. It may also augment the message with additional text, or request that the message be *captured*, which means that the lines of the message will be passed to another routine (a *capture exit*) for further processing such as logging. Once established, a message exit remains in effect until **wmi_del** is called to delete it.

The **exitptr** argument to **wmi_set** is a function pointer specifying the message exit. The **arg** parameter is an arbitrary pointer value (which may be NULL) that is passed to the message exit each time it is called. **wmi_set** returns a value of type **wmi_token**, which identifies this specific exit request. This value should be passed to **wmi_del** to delete the exit. If **wmi_set** fails, it returns 0.

The rules for how message exits are called are as follows:

When a message is generated, any message exits are called in the reverse order of their definition, that is, the most recently defined exit is called first. Message exits are not called in any of the following situations:

- □ The message number is between 000 and 099 (these numbers are reserved for messages with special requirements).
- □ The message has been suppressed due to use of the **quiet** function, or previous failures in message processing indicate possible damage to the C environment.

In general, a message is passed to each defined exit in turn. However, as described below, once an exit is able to completely suppress a message, the message is not passed to any other pending exits.

When a message exit is called, it is passed four parameters:

- □ The SAS/C message number (**msgnum**)
- □ The SAS/C errno value (**errnoval**)
- □ The argument value specified when the exit was defined (**arg**)
- □ A pointer to additional information about the message (**info**)

Note that **errnoval** may be 0, indicating an informational message that does not set **errno**, or it may be -1, indicating an error-level condition that will cause program termination after processing of the message is complete.

The **info** argument is a pointer to a structure of type **wmi_msg_info**, which is defined in **wmi.h** as follows:

```
struct wmi_msg_info {
   char *newtext;
   wmi_captype capture;
   void *captarg;
   enum { WMI_INFO = 1, WMI_WARN = 2, WMI_ERR = 4, WMI_HUSH = 256,
          WMI_SUPPRESSED = 512, WMI_FORCED = 1024 }
      msgtype;
   char recursion;
};
```

The meaning and usage of the fields of the info structure are as follows:

**newtext** may be set by the exit to request that text be added to the message. This can be useful for adding information such as time of day or transaction ID to the message. The maximum length of the additional text is 110 characters. If the text begins with a new line character, it is printed on a separate line after the library's description of the problem. If the text does not start with a new line, it is printed on the first message line printed by the library unless it does not fit, in which case it appears on the next line. Except for an initial new line, the new text must not contain any control characters, including new lines.

The **capture** field may be set by the exit to a function pointer defining a routine which is to *capture* the message. Each line of the message will be passed to the capture routine after other library processing of the message has completed. Note that it is possible to both suppress and capture a message. In this case, the message will not be printed by the library, but will be passed to the capture routine. If a message is captured but not suppressed, the message is both printed by the library and passed to the capture exit.

The **captarg** field may be set by the exit to a value of type **void*** to be passed to the capture routine. If a capture routine is specified but no argument is required, **captarg** should be set to NULL.

The **msgtype** field of the **info** structure provides additional information about the message. This field contains one or more flag bits, each of which is associated with a symbolic name. The bits and their meanings are as follows:

**WMI_INFO**
The message is informational, that is, it does not set **errno**, and the program is not informed of the associated condition.

**WMI_WARN**
The message is a warning message that sets **errno**, and after which execution will continue.

**WMI_ERR**
The message is an error message. After processing the message the program will be terminated.

**WMI_HUSH**
The message has been suppressed internally by the library. Unless an exit forces the message to be printed, it will be suppressed. Messages for which this flag is set indicate situations for which a message is not usually required.

**WMI_SUPPRESSED**
The message has been suppressed by the library or by another exit.

**WMI_FORCED**
The message has been forced to print by the library or by another exit. If both **WMI_SUPPRESSED** and **WMI_FORCED** are set, **WMI_FORCED** has precedence.

The **recursion** field is an integer which is set to either 0 or 1. If recursion is set to 1, the diagnostic message was generated during capture of a message requested by this exit. When recursion is set to 1, an exit may want to avoid capture processing to prevent infinite loops.

When a message exit has completed processing, it must return a value of type **enum wmi_outcome**, which specifies the disposition of the message. One of the following values must be returned:

**WMI_PASS**
The exit makes no change in the disposition of the message. That is, it neither suppresses it nor forces it to print. Returning **WMI_PASS** does not prevent either augmenting the message with additional text or capturing it.

**WMI_SUPPRESS**
The exit wants to suppress the message. The request may not be honored, if the library or another exit has forced processing. Note that an exit is permitted to both capture and suppress a message. If a message is suppressed by an exit, has not been forced to print, and has not been captured, the message will not be passed to any remaining exits.

**WMI_FORCE**
The exit wants to force the message to print.

A message exit should be careful not to generate any library diagnostics itself. To prevent infinite loops, a message exit is never called for a message which was generated while it was running. If a diagnostic message is generated, and not completely suppressed, during the execution of a message exit, the exit is terminated via **longjmp**, and does not complete. Because the use of **longjmp** with C++ is problematic, you should avoid writing message exits in C++ if there is any chance of generating a diagnostic within the exit.

Unlike message exits, capture exits are called at the end of library message processing. If a message is to be printed as well as captured, it will have been written to **stderr** before the capture routine is called.

A capture exit should have the following prototype:

```
enum wmi_capture_outcome capture_name(char *text,
                                      struct wmi_capture_info *info,
                                      void *arg);
```

The **text** argument is a line of the message that ends with a new line character. A library message normally generates at least three lines of output, and a capture routine is called separately for each line. The **arg** argument contains the value specified for the **captarg** field of the **wmi_msg_info** structure by the message exit which requested capture. The **info** argument addresses a structure of type **wmi_captype_info** which contains additional information about the message and the situation, as described below.

The text passed to a capture exit for a message includes any text inserted by message exits. It does not include a traceback, even if the runtime option **=btrace** is specified. The capture exit can generate a traceback itself using the **btrace()** function if necessary. Note that normally a traceback generated from a capture exit will be the same as one at the time the message was generated, but it can be different for messages which are generated by a message exit routine.

It is possible that a capture exit might itself generate a diagnostic. For instance, if a capture exit's purpose is to keep a log of messages, the log might fill up, or suffer an output error. Because of the timing of calls to capture exits, the library can be more tolerant of messages generated there than of ones from message exits. For a message generated from a capture exit, when a message exit is called, the exit's recursion flag is set, which allows the exit to choose not to capture this message. However, even if the exit chooses to capture this message, the message is placed on a capture queue rather than causing a recursive call to the capture routine. This allows the capture exit to possibly recover from the condition and process the recursive message normally.

To allow a capture exit to deal with having generated a message itself, the **`wmi_capture_info`** structure passed to the exit contains a field named **`recursion`**. The field is initialized to zero. If a diagnostic is generated from the capture exit, and is not completely suppressed, the recursion flag is set to 1. This allows the capture exit to be aware of the situation. Whether or not the recursion flag has been set changes the effect of some of the return values from a capture exit. This is intended to protect the library from problems caused by capture exits which do not test for recursions.

When a capture exit has completed processing, there are three basic actions it can request. It can inform its caller that it has failed, in which case no more lines of this message will be captured. It can inform its caller that it was successful, in which case it wil be called again for each remaining line of the message. Finally, it can request a restart from its caller. In this case, it is called again for each line of the message, starting with the first line. This is useful if, for instance, due to an error, a capture exit opens a new log file, and wants all lines of the message to appear in that file.

The mapping of the **`wmi_capture_info`** structure addressed by the **`info`** argument to the capture exit is as follows:

```
struct wmi_capture_info {
   unsigned char recursion;
   unsigned char recovering;
   unsigned char fatal;
};
```

The recursion flag is zero on entry to the capture exit, but is set to 1 if an unsuppressed diagnostic message is generated while the capture exit is executing. The recovering flag indicates whether or not the previous call to the exit for this message generated a recursive diagnostic, 0 for the normal case, or 1 where the previous call generated a diagnostic. The recovering flag can allow an exit to detect cases where it is generating warnings repeatedly, and avoid looping in this situation. The fatal flag is used to indicate whether the current message represents a fatal error, after which program execution will terminate. If the capture routine generates a diagnostic processing a fatal error, further capture processing may be bypassed to avoid the possibility of further errors.

A capture exit must return to its caller a value of type **`enum wmi_capture_outcome`**. One of the following four symbolic values must be returned:

**`WMIC_FAIL`**
: indicates that the capture exit has failed. No further lines of this message will be captured.

**`WMIC_OK`**
: indicates the capture exit has succeeded. If **`WMIC_OK`** is returned when the recursion flag has been set, the return value is treated as **`WMIC_FAIL`**.

**`WMIC_RECOVER`**
: indicates that, although a diagnostic was generated from the capture exit, the problem has been recovered, and capture processing for this message should continue. If **`WMIC_RECOVER`** is returned and the recursion flag has not been set, the return value is treated as **`WMIC_FAIL`**.

**`WMIC_RESTART`**
: requests that processing of this message resume with the first line of the message. **`WMIC_RESTART`** may be returned whether or not the recursion flag is set.

*Note:* Although the library attempts to prevent infinite loops in exit processing, they can still occur. Some caution is recommended in dealing with problems which occur in message or capture exits to prevent loops from taking place. △

## RETURN VALUE

**wmi_add** returns a token which represents the exit request. The token can be passed to **wmi_del** to delete the exit. **wmi_add** returns a zero token to indicate that the exit could not be defined.

## EXAMPLES

This example shows how to define a message exit to add the date and time of day to each library diagnostic.

*Note:* In this example the buffer containing the message must be static or external. If the buffer is defined as **auto**, the memory for the text will be deallocated when the exit returns to its caller, with unpredictable results. △

```
#include <stdio.h>
#include <time.h>
#include <assert.h>
#include <wmi.h>

static enum wmi_outcome time_exit(int, int, struct wmi_msg_info *, void *);

   main() {
      wmi_token time_exit_token;

      time_exit_token = wmi_set(&time_exit, NULL);
      assert(time_exit_token != 0);
      /* perform main program processing */
      wmi_del(time_exit_token);
      exit(0);
   }

   static enum wim_outcome time_exit(
      int msgnum,
      int errnoval,
      struct wmi_msg_info *info,
      void *arg)
   {
      static char timebuf[111];
      time_t now;
      int result;

      time(&now);
      result = strftime(timebuf, 111, "Date=%d%b%Y Time=%H:%M:%S",
                        localtime(&now));
      if (result > 0)
         info->newtext = timebuf;
      return WMI_PASS;
   }
```

The next example shows how to use a message exit to capture message lines and write them to a log file. If an error occurs writing to the log, a new log file is opened. Most of the management of the log file is encapsulated as subroutines.

The following example assumes that if a message is generated in the **write_to_log()** function, but the function does not indicate an error, then the message did not indicate a significant problem. Whether this is a reasonable assumption or not would depend on the code in **`write_to_log`**.

```
#include <stdio.h>
#include <assert.h>
#include <wmi.h>

static enum wmi_outcome msgexit(int, int, struct wmi_msg_info *, void *);
static enum wmi_capture_outcome capture_line(char *,
                                             struct wmi_capture_info *,
                                             void *);

FILE *logfile;

   main() {
      wmi_token token;

      open_log_file();
      token = wmi_set(&msgexit, NULL);
      assert(token != 0);
      /* perform main program processing */
      wmi_del(token);
      exit(0);
   }

   static enum wmi_outcome msgexit(
      int msgnum,
      int errnoval,
      struct wmi_msg_info *info,
      void * arg)
   {
      if (info->recursion)
         return WMI_PASS;      /* don't capture msg from exit */
      if (info->msgtype & WMI_HUSH)
         return WMI_PASS;      /* don't capture insiginificant messages */
      info->capture = &capture_line;
      info->captarg = 0;
      if (info->msgtype & WMI_ERR) return WMI_PASS;
      else WMI_SUPPRESS;   /* don't print message unless fatal */
   }

   static enum wmi_capture_outcome capture_line(
      char *text,
      struct wmi_capture_info *info,
      void *arg)
   {
      int error_flag;
      int restart = 0;

      error_flag = write_to_log(text);
      if (error_flag != 0) {
         /* don't try to recover in perilous situations */
         if (info->recovering || info->fatal)
            return WMIC_FAIL;
         error_flag = open_new_log();
```

```
            /* give up if we can't open a good log file */
            if (error_flag != 0)
               return WMIC_FAIL;
            else restart = 1;
         }
         if (restart) return WMIC_RESTART;
         else if (info->recursion) return WMIC_RECOVER;
         else return WMIC_OK;
      }
```

See the examples for the **wmifilte** and **wmifiltn** functions, which are generalized exit functions that can be used to suppress specific library messages, for more information.

### RELATED FUNCTIONS

**wmi_del**, **wmifilte**, **wmifiltn**, **quiet**, **longjmp**

### SEE ALSO

"Diagnostic Control Functions" in Chapter 2, "Function Categories."

## New hstrerror Function

Add the following function description to Chapter 18, "Socket Function Reference," in *SAS/C Library Reference, Volume 2*.

## hstrerror

**Map h_error number to Message String**

**Portability:** SASC

### SYNOPSIS

```
#include <netdb.h>
char *hstrerror( int errnum );
```

### DESCRIPTION

**hstrerror** maps the **h_error** number in **errnum** to an error message string.

### RETURN VALUE

**hstrerror** returns a pointer to a message describing the **h_error** number.

### EXAMPLE

```
#include <sys/types.h>
#include <stdlib.h>
```

```
#include <sys/socket.h>
#include <netinet.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>

main(int argc, char *argv[])
{
   struct hostent *hp;
   struct in_addr ip_addr;
   /* Verify a "hostname" parameter was supplied */
   if  (argc <1 || *argv[1] == '\0')
   exit(EXIT_FAILURE);

   /* call gethostbyname() with a host name. */
   /* gethostbyname() returns a pointer */
   /* to a hostent struct or NULL. */
   hp = gethostbyname(argv[1]);
   if  (!hp)
   {
      printf("gethostbyname failed. \"%s\"\n", hstrerror(h_errno) );
      printf("%s was not resolved\n", argv[1]);
      exit(EXIT_FAILURE);
   }
}
```

### RELATED FUNCTIONS

**herror**, **perror**, **strerror**

## New oe2errno Function

## oe2errno

**Convert USS errno value to SAS/C errno value**

**Portability:** SASC

### SYNOPSIS

```
#include <lclib.h>
#include <errno.h>

int oe2errno( int ussrc );
```

### DESCRIPTION

**oe2errno** converts an IBM UNIX System Services (USS) **errno** value to the equivalent SAS/C **errno** value.

### RETURN VALUE

**oe2errno** returns the equivalent SAS/C **errno** value, if applicable. Otherwise it returns -1.

### RELATED FUNCTIONS

**perror**, **strerror**

## New String Functions

**memcasecmp**, **strcasecmp**, and **strncasecmp** are new string functions for Release 7.50. The following updates and additions need to be made to *SAS/C Library Reference, Volume 1*.

- Add **memcasecmp**, **strcasecmp**, and **strncasecmp** to the table labeled, “String Utility Functions.”In Chapter 2, “Function Categories.”
- Add the following function descriptions to Chapter 6, “Function Descriptions.”

## memcasecmp

**Compare Two Blocks of Memory, ignoring differences in case**

**Portability:** SASC

### SYNOPSIS

```
#include <lcstring.h>

int memcasecmp( const void *ptr1, const void *ptr2, size_t n );
```

### DESCRIPTION

**memcasecmp** compares, while ignoring differences in case, two blocks of memory specified by **ptr1** and **ptr2**. The number of bytes to be compared is **n**. The null character is treated like any other character and participates in the comparison. The comparison is performed using the standard EBCDIC collating sequence with the exception that differences in case are ignored. Note that case differences are determined in a locale- specific manner.

### RETURN VALUE

**memcasecmp** returns 0 if the two blocks are equal, an integer less than 0 if the first block is less than the second, or an integer greater than 0 if the first block is greater than the second.

### IMPLEMENTATION

**memcasecmp** is functionally equivalent to **memcmp**, except that each byte is converted to lowercase before the comparison is made.

## EXAMPLE

```
#include <lcstring.h>

int main(void)
{
   int ch;
   int num;
   char lwr_alpha[] = "abcdefghijklmnopqrstuvwxyz";
   char upr_alpha[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   char mix_alpha[] = "AbCdEfGhIjKlMnOpQrStUvWxYz";

   for (num = 1; num <= strlen(lwr_alpha); num++)
   {
     ch = memcasecmp(lwr_alpha, mix_alpha, num);
     if ( ch != 0 )
     {
        exit(EXIT_FAILURE);
     }
   }
   for (num = 1; num <= strlen(upr_alpha); num++)
   {
     ch = memcasecmp(upr_alpha, mix_alpha, num);
     if ( ch != 0 )
     {
        exit(EXIT_FAILURE);
     }
   }
   exit(EXIT_SUCCESS);
}
```

## RELATED FUNCTIONS

**memcmp**, **strcasecmp**, **strcmp**, **strncasecmp**, **strncmp**

## SEE ALSO

“String Utility Functions” in Chapter 2, “Function Categories.”

## strcasecmp

**Compare Two Null-Terminated strings, ignoring differences in case**

**Portability:** SAS/C

## SYNOPSIS

```
#include <strings.h>

int strcasecmp( const char *str1, const char *str2 );
```

## DESCRIPTION

**strcasecmp** compares, while ignoring differences in case, two character strings specified by **str1** and **str2**. The comparison is performed using the standard EBCDIC collating sequence with the exception that differences in case are ignored. The return value has the same relationship to 0 as **str1** has to **str2**. If the two strings are equal up to the point at which one terminates (that is, contains a null character), the longer string is considered greater. Note that case differences are determined in a locale-specific manner.

## RETURN VALUE

**strcasecmp** returns 0 if the two strings are equal, an integer less than 0 if **str1** compares less than **str2**, or an integer greater than 0 if **str1** compares greater than **str2**. No other assumptions should be made about the value returned by **strcasecmp**.

## CAUTION

If one of the arguments of **strcasecmp** is not properly terminated, a protection or addressing exception may occur.

## IMPLEMENTATION

**strcasecmp** is functionally equivalent to **strcmp**, except that each byte is converted to lowercase before the comparison is performed.

## EXAMPLE

```
#include <strings.h>
int main(void)
{
   int ch;
   char lwr_alpha[] = "abcdefghijklmnopqrstuvwxyz";
   char upr_alpha[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   char mix_alpha[] = "AbCdEfGhIjKlMnOpQrStUvWxYz";

   ch = strcasecmp(lwr_alpha, mix_alpha);
   if ( ch != 0 )
   {
      exit(EXIT_FAILURE);
   }

   ch = strcasecmp(upr_alpha, mix_alpha);
   if ( ch != 0 )
   {
      exit(EXIT_FAILURE);
   }

   exit(EXIT_SUCCESS);
}
```

## RELATED FUNCTIONS

**strcmp**, **memcasecmp**, **memcmp**, **strncasecmp**, **strncmp**

## SEE ALSO

"String Utility Functions" in Chapter 2, "Function Categories."

## strncasecmp

**Compare Portions of two strings, ignoring differences in case**

**Portability:** SAS/C

### SYNOPSIS

```
#include <strings.h>

int strncasecmp( const char *str1, const char *str2, size_t maxlen );
```

### DESCRIPTION

**strncasecmp** compares, while ignoring differences in case, two character strings specified by **str1** and **str2**. The comparison is performed using the standard EBCDIC collating sequence with the exception that differences in case are ignored. The return value has the same relationship to 0 as **str1** has to **str2**. If the two strings are equal up to the point at which one terminates, that is, contains a null character, the longer string is considered greater. If **maxlen** characters are inspected from each string and no inequality is detected, the strings are considered equal. Note that case differences are determined in a locale- specific manner.

### RETURN VALUE

**strncasecmp** returns 0 if the two strings are equal, an integer less than 0 if **str1** compares less than **str2**, or an integer greater than 0 if **str1** compares greater than **str2**, within the first **maxlen** characters. No other assumptions should be made about the value returned by **strncasecmp**.

### CAUTION

If the **maxlen** value is specified as 0, a result of 0 is returned. If the value is a negative integer, it is interpreted as a very large unsigned integer value. This may cause a protection or addressing exception, but this is unlikely because comparsion ceases as soon as unequal characters are found.

### IMPLEMENTATION

**strncasecmp** is functionally equivalent to **strncmp**, except that each byte is converted to lowercase before the comparison is performed.

### EXAMPLE

```
#include <strings.h>

int main(void)
{
   int ch;
   size_t num;
   char lwr_alpha[] = "abcdefghijklmnopqrstuvwxyz";
   char upr_alpha[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   char mix_alpha[] = "AbCdEfGhIjKlMnOpQrStUvWxYz";

   for (num = 1; num <= strlen(lwr_alpha); num++)
   {
     ch = strncasecmp(lwr_alpha, mix_alpha, num);
     if ( ch != 0 )
     {
        exit(EXIT_FAILURE);
     }
   }
   for (num = 1; num <= strlen(upr_alpha); num++)
   {
     ch = strncasecmp(upr_alpha, mix_alpha, num);
     if ( ch != 0 )
     {
        exit(EXIT_FAILURE);
     }
   }
   exit(EXIT_SUCCESS);
}
```

### RELATED FUNCTIONS

**strcmp**, **memcasecmp**, **memcmp**, **strcasecmp**, **strcmp**

### SEE ALSO

“String Utility Functions” in Chapter 2, “Function Categories.”

## New I/O Functions

**vfscanf**, **vscanf**, and **vsscanf** are new I/O functions for Release 7.50. The following updates and additions need to be made to *SAS/C Library Reference, Volume 1.*

- □ Add **vfscanf**, **vscanf**, and **vsscanf** to the list of I/O functions in Chapter 2, “Function Categories.”
- □ Add **vfscanf**, **vscanf**, and **vsscanf** to the table labeled, “I/O Functions”in the section titled, “Formatted I/O Functions,” in Chapter 3, “I/O Functions.”
- □ Add the following function descriptions to Chapter 6, “Function Descriptions.”

## vfscanf

**Read Formatted Input from file**

**Portability:** C99

## SYNOPSIS

```
#include <stdarg.h>
#include <lcio.h>

int vfscanf( FILE *f, const char *format, va_list arg );
```

## DESCRIPTION

**vfscanf** reads formatted input from the FILE designated by **f** according to the format specified by the string format. Following the format in the argument list is a pointer to a list of arguments, designated by **arg**, of type **va_list** as defined in the header **<stdarg.h>**.

The string pointed to by **format** is in the same form as that used by **fscanf**. See the **fscanf** description for detailed information concerning the formatting conventions.

## RETURN VALUE

**vfscanf** returns **EOF** if end of file (or an input error) occurs before any values are stored. If values are stored, it returns the number of items stored; that is, the number of times a value is assigned to a member of the variable argument list.

## IMPLEMENTATION

**vfscanf** is functionally equivalent to **fscanf**, except that the argument list has been replaced by a variable argument list, **va_list**, as defined in the header **<stdarg.h>**.

## EXAMPLE

```
#include <stdarg.h>
#include <lcio.h>
#include <stdlib.h>

int DoScan(FILE *f, const char *fmt, ...);

static char *lines[] = {
   "147.8 pounds\n"
   "51.7 miles\n",
   "4.3 light-years\n",
   "10000 volts\n",
   "19.5 gallons\n"
};


main()
{
   FILE *tmpf;
   int i;
   __binfmt float amount;  // Declare a binary floating point number
   char unit[20];
   int count;

   tmpf = tmpfile();
   if (!tmpf)
   {
         puts("Couldn’t open temporary file.");
         exit(EXIT_FAILURE);
   }
   for (i = 0; i < sizeof(lines)/sizeof(char *); ++i)
   {
      fputs(lines[i], tmpf);
   }
   rewind(tmpf);

   for(;;)
   {
      count = DoScan(tmpf, "%~bf %s", &amount, unit);
      if (feof(tmpf)) break;
      if (count < 2)
      {
         puts("Unexpected error in input data.");
         exit(EXIT_FAILURE);
      }
         printf("amount = %.1~bf, units = \"%s\"\n", amount, unit);
   }
   fclose(tmpf);
}

int DoScan(FILE *f, const char *fmt, ...)
{
   int n = 0;
   va_list args;

   va_start(args, fmt);
   n = vfscanf(f, fmt, args);
   va_end(args);

   return n;
}
```

## RELATED FUNCTIONS

**vfprintf**, **fscanf**, **va_arg**, **va_start**, **va_end**

## SEE ALSO

- “I/O Functions” in Chapter 2, “Function Categories”
- Chapter 3, "I/O Functions"

## vscanf

**Read Formatted Input from the Standard Input Stream**

**Portability:** C99

## SYNOPSIS

```
#include <stdarg.h>
#include <lcio.h>
```

```
int vscanf( const char *format, va_list arg );
```

## DESCRIPTION

**vscanf** reads formatted input from **stdin** according to the format specified by the string format. Following the format in the argument list is a pointer to a list of arguments, designated by **arg**, of type **va_list** as defined in the header **<stdarg.h>**.

The string pointed to by **format** is in the same form as that used by **fscanf**. See the **fscanf** description for detailed information concerning the formatting conventions.

## RETURN VALUE

**vfscanf** returns **EOF** if end of file (or an input error) occurs before any values are stored. If values are stored, it returns the number of items stored; that is, the number of times a value is assigned to a member of the variable argument list.

## IMPLEMENTATION

**vscanf** is functionally equivalent to **scanf**, except that the argument list has been replaced by a variable argument list, **va_list**, as defined in the header **<stdarg.h>**.

## EXAMPLE

```
#include <stdarg.h>
#include <lcio.h>
#include <stdlib.h>

int DoScan(const char *fmt, ...);

double point[40];

main()
{
   int index = 0;
   double sum = 0.0;
   double avg;
   int nopoints;
   int stdn_fn = 0;

   /* If stdin is the terminal, fileno(stdin) is always 0. */
   if (isatty(stdn_fn))
      /* Tell user to enter data points; maximum = 39.        */
      puts("Enter data points (EOF to indicate end of list).");

   for(;;)
   {
      /* Read number; check for end of file.                  */
      if (DoScan("%le", &point[index]) == EOF)
         break;
      sum += point[index];
      ++index;
   }
   nopoints = index;
   avg = sum / nopoints;
   printf("%d points read.\n", nopoints);
   printf("%lf = average.\n", avg);
}

int DoScan(const char *fmt, ...)
{
   int n = 0;
   va_list args;

   va_start(args, fmt);
   n = vscanf(fmt, args);
   va_end(args);

   return n;
}
```

## RELATED FUNCTIONS

**vfprintf**, **scanf**, **va_arg**, **va_start**, **va_end**

## SEE ALSO

- "I/O Functions" in Chapter 2, "Function Categories"
- Chapter 3, "I/O Functions"

## vsscanf

**Read Formatted Input from a String**

**Portability:** C99

### SYNOPSIS

```
#include <stdarg.h>
#include <lcio.h>

int vsscanf( const char *source, const char *format, va_list arg );
```

### DESCRIPTION

**vsscanf** reads formatted input text from the string addressed by **source**. No file input is performed. Following the format in the argument list is a pointer to a list of arguments, designated by **arg**, of type **va_list** as defined in the header **<stdarg.h>**.

The string pointed to by **format** is in the same form as that used by **fscanf**. See the **fscanf** description for detailed information concerning the formatting conventions.

### RETURN VALUE

**vfscanf** returns **EOF** if end of file (or an input error) occurs before any values are stored. If values are stored, it returns the number of items stored; that is, the number of times a value is assigned to a member of the variable argument list.

## IMPLEMENTATION

**vsscanf** is functionally equivalent to **sscanf**, except that the argument list has been replaced by a variable argument list, **va_list**, as defined in the header **<stdarg.h>**.

## EXAMPLE

```
#include <stdarg.h>
#include <lcio.h>
#include <stdlib.h>

int DoScan(const char *s, const char *fmt, ...);

static char *lines[] =
{
   "147.8 pounds\n",
   "51.7 miles\n",
   "4.3 light-years\n",
   "10000 volts\n",
   "19.5 gallons\n",
   ""
};

int main()
{
   int i;
   float amount;
   char unit[20];
   int count;


   for (i = 0 ;; i++)
   {
      count = DoScan(lines[i], "%f %s", &amount, unit);

      if (count == EOF)
         break;

      if (count < 2)
      {
         puts("Unexpected error in input data.");
         exit(EXIT_FAILURE);
      }

      printf("amount = %.1f, units = \"%s\"\n", amount, unit);
   }
   exit(EXIT_SUCCESS);
}


int DoScan(const char *s, const char *fmt, ...)
{
   int n = 0;
   va_list args;
```

```
    va_start(args, fmt);
    n = vsscanf(s, fmt, args);
    va_end(args);

    return n;
}
```

## RELATED FUNCTIONS

**vsprintf**, **sscanf**, **va_arg**, **va_start**, **va_end**

## SEE ALSO

- "I/O Functions" in Chapter 2, "Function Categories"
- Chapter 3, "I/O Functions"

## New opcmd Function

Add the following entry for **opcmd** to the list of descriptions in Chapter 6, "Function Descriptions."

## opcmd

**Initialize Operator Command Interface**

**Portability:** SASC

### SYNOPSIS

```
#include <lcsignal.h>
void opcmd(unsigned int cibctr);
```

### DESCRIPTION

**opcmd** requests that a SIGOPER signal be generated whenever the MVS operator issues a **modify** or **stop** command for the program. The SIGOPER signal will also be raised if the operator included a parameter in the **start** command that started the program. This is in addition to any parm that may have been placed in the started task PROC which is handled as usual by passing as an argument to **main**. It is not necessary for the program to be run as a started task to use this facility, but only started tasks will receive **start** parms.

**cibctr** specifies the number of Command Input Buffers (CIBs) to be chained before rejecting **modify** commands. This value can be from 0 to 255. If set to 0, no **modify** commands will be accepted, but STOP commands will still be processed.

The SIGOPER signal is asynchronous, so it is discovered only when a function is called or returns.

### RETURN VALUE

None.

## PORTABILITY

**opcmd** is not portable.

## EXAMPLE

This example waits for operator commands and shuts down when the **stop** command is received:

```
#include <oswto.h>
#include <stdlib.h>
#include <signal.h>
#include <lcsignal.h>
static void OperMsg(int);
static int shutdown = 0;

void main(void)
{
   signal(SIGOPER, &OperMsg); /* Catch SIGOPER signal.             */
   opcmd(5);                  /* enable operator command service   */
   while(!shutdown)
     sigpause(0);
}

/*-----------------------------------------------------------------+
| SIGOPER handler                                                  |
+-----------------------------------------------------------------*/
static void OperMsg(int signum)
{
OPER_t *info;

   info = siginfo();
   switch (info->type)
   {
     case OPER_start:
       WTP("Start parm received: %s\n", info->request);
       break;
     case OPER_stop:
       WTP("Stop received.\n");
       shutdown = 1;
       break;
     case OPER_modify:
       WTP("Modify received: %s\n", info->request);
       break;
   }
   signal(SIGOPER, &OperMsg); /* Catch SIGOPER signal.         */
   return;
}
```

## SEE ALSO

SIGOPER in Chapter 5, “Signal-Handling Functions”

# New Coverage Support Feature

The coverage feature provides information about which lines of source code written in C (C source code) were executed during a given compilation. The coverage feature has four parts:

- Part one is implemented by the compiler.
- Part two is implemented by COOL.
- Part three is the application itself.
- Part four is the data extraction routine.

The compiler provides the first part of the coverage feature by generating additional inline code that flags each line of code when it is executed. The flag is located in a data element that is also created by the compiler and added to the generated object.

COOL implements the second part of the line coverage feature by producing a global list that addresses all the compiler-generated coverage data elements encountered in its input.

The third part of the coverage feature is the application upon which coverage analysis is to be performed.

The fourth part of the coverage feature is a C routine that you provide, named **__cvgtrm**, that extracts and processes the coverage data elements at the end of program execution.

## The Compiler Part of the Coverage Feature

When the **coverage** option is specified, the compiler activates the coverage feature by creating an internal data element that is used to track which lines of C source code are executed. You must also specify the compiler's extended name option, **extname**, for COOL to correctly process the data element created by the compiler.

*Note:* The object created by the compiler when the coverage option is specified will not be reentrant, regardless of whether you specify the **rent** option. △

The following table describes the nine components in the data element that are created by the compiler.

**Table 2.2**

| Field Name | Type | Boundary | Description |
|---|---|---|---|
| Reserved | **int** | Fullword | |
| **FELNO** | **int** | Fullword | Line number of first line of executable C source code. |
| **LELNO** | **int** | Fullword | Line number of last line of executable C source code. |
| Reserved | **int** | Fullword | |
| **SOURCE_FN_SIZE** | **int** | Fullword | Number of characters in the **SOURCE_FN**. |
| Reserved | **int** | Fullword | |

| Field Name | Type | Boundary | Description |
|---|---|---|---|
| **`EXE_INDICATOR`** | **`char[?]`** | Fullword | Variable Length; each element represents a C souce code line. Element 0 is FELNO, and the last character is LELNO. |
| **`SOURCE_FN`** | **`char[?]`** | Fullword | A string that contains the input C source filename. It starts on the first fullword boundary that follows the **`EXE_INDICATOR`**. |
| **`SNAME`** | **`char[8]`** | Char | Section name of the compilation unit. |
| **`SOURCE_COMPDATE`** | **`char[17+1]`** | Char | A string containing the creation date of the object. |
| **`SOURCE_SNAME`** | **`char[7+1]`** | Char | A string containing the section name of the compilation unit. The maximum length is 7 bytes. |

The header file **`cvg.h`** defines structure type **`USAGE_DATA`** which maps the line coverage table. Because fields **`SOURCE_FN`**, **`SOURCE_COMPDATE`** and **`SOURCE_SNAME`** are at variable offsets from the start of the structure, they are accessed through macros rather than as structure members. For instance, if **`ptr`** is the address of the line coverage table, **`ptr->FELNO`** addresses the **`FELNO`** field, but **`SOURCE_SNAME(ptr)`** is used to reference the **`SOURCE_SNAME`** data.

When the coverage option is specified, the compiler searches for the first C source code line (or any portion of a C source code line) in the compilation unit that generates executable code and flags it as the first line of C source code. All previous lines are ignored. All subsequent lines (including white space and comment lines) are flagged as lines that generate executable code or lines that do not generate executable code.

The **`EXE_INDICATOR`** is an array of characters. Each element of the array represents a C source code line. The first element of the array (**`FELNO`**) represents the first C source code line that has been flagged as a line that generates executable code within the compilation unit. Each subsequent element represents a C source code line, regardless of whether it has been flagged as a line that generates executable code or not. The last element (**`LELNO`**) represents the last C source code line that has been flagged as a line that generates executable code.

At compile time, elements of the array are initialized to 0x01 if the line has been flagged as a line that generates executable code and 0x02 if not. At run-time, the 0x01 elements are changed to 0x00 when the corresponding line is executed.

The C source filename is provided in **`SOURCE_FN`**, which follows the **`EXE_INDICATOR`** array and starts on the next fullword boundary. The length of the filename that contains the C source code is in **`SOURCE_FN_SIZE`**. The section name for the compilation unit immediately follows the **`SOURCE_FN`** and is an 8-byte null-terminated string.

Immediately following **`SOURCE_FN`** is **`SOURCE_COMPDATE`**, which is the date and time on which the object was created. **`SOURCE_COMPDATE`** is a null terminated string which is a total of 18 bytes long (including null terminator). The format is **`mm/dd/yy hh:mm:ss`**.

Following **SOURCE_COMPDATE** is the **SOURCE_SNAME**, which is a null-terminated string, with a maximum length of 8 bytes (including the null-terminator).

## The COOL Part of the Coverage Feature

Objects compiled with the coverage option must be prelinked with COOL by using COOL's coverage option. COOL constructs the line coverage table pointer lists that are used by the data extraction routine. See "Extracting Coverage Data" on page 166 for more information on the data extraction routine. The data extraction routine object should be prelinked with the object module that contains the application's primary entry point, along with any other application objects.

COOL creates a list of pointers to line coverage tables found in each object. A pointer to this list will be passed to the data extraction routine as an argument.

*Note:* The object created by COOL will not be reentrant, even if you specify the **RENT** option at link time. △

## The Coverage Feature Application

With the exception of the data extraction routine, all routines that require coverage data must be compiled with the **coverage** and **extname** options. There are no special or unique coding requirements for the coverage feature.

## Extracting Coverage Data

To extract coverage data, create a replacement for the dummy library routine called **__cvgtrm**. The routine you write should be in a source file separate from the application, and it must be coded in C because **__cvgtrm** is called after destructors have been called for all C++ objects. **__cvgtrm** should not be compiled with the **coverage** option because any data that is generated will not be correct.

As noted in "The COOL Part of the Coverage Feature" on page 166, **__cvgtrm** should be linked with the application's object that contains the primary entry point. It should not be included with any objects that are part of dynamically loaded modules. A dynamically loadable module is one that contains a **_dynamn** function.

**__cvgtrm** is called once for the primary entry-point module during library termination. For dynamically-loaded modules, **__cvgtrm** is called when the module is explicitly unloaded (by using **unloadm**), or during library termination when the module is unloaded by the library.

The library will pass the following three parameters on each call to **__cvgtrm**:

```
void __cvgtrm    (char *sname_ep,
                  char *load_module,
               struct USAGE_DATA *** cvg_list
                 );
```

Where the parameters have the characteristics described in the following table.

**Table 2.3**

| Data Name | Description |
|---|---|
| **`char * pgmname`** | Pointer to the string with the name of the application program, if it can be determined. This is the same value that is passed to **`main`** as the first element of the **`argv`** array. |
| **`char * load_module`** | Pointer to the name of the load module to which this coverage data applies. For the initial load module of an application (the one containing the **`main`** function), **`load_module`** will be the string `""`. |
| **`struct USAGE_DATA *** cvg_list`** | Pointer to a list of pointers to line coverage tables. |

## JCL for Coverage Sample

The following JCL builds the application which produced the sample output in "Sample Output" on page 172:

```
//* Compile the Sample C Main cvgmain.c
//* Note: 'coverage' option specified
//CVGMAIN   EXEC LC370C,
//             PARM.C='EXTNAME,SNAME(CVGSAMP),COVERAGE'
//C.SYSIN      DD   DISP=SHR,DSN=COVERAGE.C(CVGMAIN)
//C.SYSLIN     DD   DISP=SHR,DSN=COVERAGE.OBJ(CVGMAIN)
//* Compile the subroutine called by the C main
//* Note: 'coverage' option specified
//CVGMAIN2  EXEC LC370C,
//             PARM.C='EXTNAME,SNAME(CVGMN2),COVERAGE'
//C.SYSIN      DD   DISP=SHR,DSN=COVERAGE.C(CVGMAIN2)
//C.SYSLIN     DD   DISP=SHR,DSN=COVERAGE.OBJ(CVGMAIN2)
//* Compile the module dynamically loaded by the C Main
//* Note: 'coverage' option specified
//CVGDYNM   EXEC LC370C,
//             PARM.C='EXT,SNAME(CVGDYNM),COVERAGE'
//C.SYSIN      DD   DISP=SHR,DSN=COVERAGE.C(CVGDYNM)
//C.SYSLIN     DD   DISP=SHR,DSN=COVERAGE.OBJ(CVGDYNM)
//* Compile the Data Extraction function called by the library
//* as __cvgrtm()  Note: NO 'coverage' option specified!
//CVGDUMP   EXEC LC370C,
//             PARM.C='EXTNAME,SNAME(CVGDUMP)'
//C.SYSIN      DD   DISP=SHR,DSN=COVERAGE.C(CVGDUMP1)
//C.SYSLIN     DD   DISP=SHR,DSN=COVERAGE.OBJ(CVGDUMP1)
//* Linkedit the sample application
//* Note: 'coverage' option specified
//LKED      EXEC LC370LR,PARM.LKED=('LIST,MAP,COVERAGE')
//LKED.OBJINPT DD DISP=SHR,DSN=COVERAGE.OBJ
//LKED.SYSLMOD DD DISP=SHR,DSN=COVERAGE.LOAD
//LKED.SYSIN   DD *
  INCLUDE OBJINPT(CVGDUMP1)
  INCLUDE OBJINPT(CVGMAIN)
  INCLUDE OBJINPT(CVGMAIN2)
  ENTRY   MAIN                   Standard Entry Point
  NAME    CVGMAIN(R)             Member Name
/*
//* Linkedit the dynamically loaded module
//* Note: 'coverage' option specified
//LKED      EXEC LC370LR,PARM.LKED=('LIST,MAP,COVERAGE')
//LKED.SYSLIN  DD DISP=SHR,DSN=COVERAGE.OBJ(CVGDYNM)
//LKED.OBJINPT DD DISP=SHR,DSN=COVERAGE.OBJ
//LKED.SYSLMOD DD DISP=SHR,DSN=COVERAGE.LOAD
//LKED.SYSIN   DD *
  INCLUDE OBJINPT(CVGDYNM)
  ENTRY   #DYNAMNR               Standard Entry Point
  NAME    CVGDYNM(R)             Member Name
/*
```

## Sample MAIN (cvgmain.c);

```
#include <stdio.h>
int main()
{                                    /* Executed, start up.   */

/*==================================    Non-Executable.
   'coverage' defines two types of C source code
   lines, executable, and all others.  All others
   can be white-space lines, comment lines, or lines
   of code that do not result in executable code. ============*/

int exitrc =0;                       /* Executed, initialized.*/
int notableentry;                    /* Non-executable.       */
int (*ep)(void);                     /* Non-executable.       */
/* No doubt this is a comment line.     Non-executable.       */
notableentry=1;                      /* Executed.             */
if (notableentry = 1) printf("=1\n");/* Executed.             */
if (notableentry = 1)                /* Executed.             */
  printf("=1\n");                    /* Executed.             */
if (notableentry > 1) printf(">3\n");/* Executed, if () only. */
if (notableentry > 1)                /* Executed.             */
 {                                   /* Non-executable.       */
  printf(">1\n");                    /* Not executed.         */
  notableentry=3;                    /* Not executed.         */
  printf("notableentry = 3\n");      /* Not executed.         */
 };                                  /* Non-executable.       */
Label:                               /* Non-executable        */
printf("Hello World\n");             /* Executed.             */
loadm("CVGDYNM", &ep);               /* Executed.             */
if (ep != NULL)                      /* Executed.             */
 {                                   /* Non-executable.       */
  (*ep)();                           /* Executed.             */
  unloadm(ep);                       /* Executed.             */
 };                                  /* Non-executable.       */
cvgmain2();                          /* Executed.             */
exit(exitrc);                        /* Executed.             */
}                                    /* Not executed.  A      */
                                     /* return was added by   */
                                     /* the compiler because  */
                                     /* exit() was called     */
                                     /* instead of return().  */
```

## Sample Subroutine called by MAIN (cvgmain2.c);

```
int cvgmain2()
{                                    /* Executed, start up.   */
 printf("main2\n");                  /* Executed.             */
 return(0);                          /* Executed.             */
}
```

## Sample Dynamically-Loaded Module (cvgdynm.c);

```
int _dynamn()
{                                    /* Executed, start up.   */
  printf("_dynamn is here\n");       /* Executed.             */
  return(0);                         /* Executed.             */
}                                    /* Non-executable.       */
```

## Sample Coverage Routine (cvgdump.c);

```
#include <stdio.h>
#include <string.h>
#include <cvg.h>

/*-------------------------------------------------------------------+
| Track number of data elements processed.                           |
+-------------------------------------------------------------------*/
static int ndx = 0;
static void rewind_usage(){
  ndx = 0;
}
/*-------------------------------------------------------------------+
| Extract pointer to line coverage data                              |
+-------------------------------------------------------------------*/
static USAGE_DATA * next_usage(USAGE_DATA **CVGTBL[]){
   USAGE_DATA * result = 0;
   if ( CVGTBL[ndx] ) {
      result = *CVGTBL[ndx];
      ndx++;
   }
   return result;
}
/*-------------------------------------------------------------------+
| Open the source file for a given data element.                     |
+-------------------------------------------------------------------*/
static FILE *opnf(char *name){
   FILE *fp;
   quiet(1);
   fp = fopen(name,"r");
   quiet(0);
   return fp;
```

```
}
/*-----------------------------------------------------------------------+
| Process data elements.                                                 |
+-----------------------------------------------------------------------*/
void __cvgtrm   (char * pgm_name, char * load_module,
             USAGE_DATA *** cvg_list){
    FILE *fp;
    char buffer[4096];
    int i, j, data_size;
    USAGE_DATA *udp;
    fprintf(stderr,"\n\npgm_name: %s  load_module: %s\n",
            pgm_name, load_module);
    fprintf (stderr,"\nLegend: "
       "(*=executed; .=not_executed, blank=comment/white space"
          "/non-executable)\n\n");
    rewind_usage();
    while ( udp = next_usage(cvg_list) ) {
        data_size = DATA_SIZE(udp);
        fprintf(stderr,
                "SOURCE_FN      : %s    SOURCR_FN_SIZE : %d  \n"
                "SOURCE_COMPDATE: %s  \nSOURCE_SNAME   : %s  \n",
                 SOURCE_FN(udp), udp->SOURCE_FN_SIZE,
                 SOURCE_COMPDATE(udp),SOURCE_SNAME(udp));

        fp = opnf(SOURCE_FN(udp));
        if (fp == NULL) {
            strcpy (buffer, "Unavailable\n");
            return;
        }
        fprintf (stderr, "  Line Covered  Source \n");
        fprintf (stderr, "  --------------------\n");
        /* Print source up to FELNO                     */
        for (i = 1; i < udp->FELNO; i++){
            if (fp)
                fgets (buffer, sizeof buffer, fp);
            fprintf (stderr, "%5d:             ", i);
            fputs (buffer, stderr);
        }
        /* Print source from FELNO through LELNO        */
        for (j = 0; j < DATA_SIZE(udp); j++) {
            if (fp)
                fgets (buffer, sizeof buffer, fp);
            switch (udp->EXE_INDICATOR[j]) {
            case 0:
                fprintf (stderr, "%5d:    *        ", i + j);
                break;
            case 1:
                fprintf (stderr, "%5d:    .        ", i + j);
                break;
            case 2:
                fprintf (stderr, "%5d:             ", i + j);
                break;
            default:
                printf ("ERROR: illegal code %d\n",
                udp->EXE_INDICATOR[j]);
                return;
            }
            fputs (buffer, stderr);
        }
        /* Print source from LELNO+12 through EOF O      */
        if (fp) {
            while (fgets (buffer, sizeof buffer, fp)) {
                fprintf (stderr, "%5d:             ", i + j);
                fputs (buffer, stderr);
                i++;
            }
            fclose (fp);
        }
    }
}
```

## cvg.h Header

```
#ifndef __IncCVG
#define __IncCVG
#include <string.h>
/*-------------------------------------------------------------------+
| SAS/C Line Coverage Feature                                        |
|                                                                    |
| Structure mapping data elements through first byte of coverage     |
| data, after that point the length is variable.                     |
|                                                                    |
| There are 3 elements that are not defined in the structure,        |
| source filename, object creation date, and sname. There are        |
| macros below to assist with locating each of them correctly.       |
+------------------------------------------------------------------*/
typedef struct usage_data {
    int reserved0;
    int FELNO;               /* First executable line number       */
    int LELNO;               /* Last executable line number        */
    int reserved1;
    int SOURCE_FN_SIZE;      /* Length of source filename          */
    int reserved2;
    char EXE_INDICATOR[1];   /* Variable length character array    */
}USAGE_DATA;

#define DATA_SIZE(udp) ((udp->LELNO - udp->FELNO) + 1)

#define SOURCE_FN(udp) \
      ( udp->EXE_INDICATOR+((DATA_SIZE(udp) + 3) & ~ 3))

#define SOURCE_COMPDATE(udp) \
      ( SOURCE_FN(udp) + (udp->SOURCE_FN_SIZE) )

#define SOURCE_SNAME(udp) \
      ( SOURCE_FN(udp) + udp->SOURCE_FN_SIZE + \
        strlen(SOURCE_COMPDATE(udp)) + 1 \
      )
```

```
#endif
```

## Sample Output

```
pgm_name:,cvgsamp  load_module: CVGDYNM

Legend: (*=executed; .=not_executed, blank=comment/white space/non-executable)

SOURCE_FN      : //DSN:USER.DEV.C(CVGDYNM)    SOURCR_FN_SIZE : 28
SOURCE_COMPDATE: 10/09/02 10:05:31
SOURCE_SNAME   : CVGDYNM
  Line Covered  Source
  --------------------
    1:             int _dynamn()
    2:    *        {                                 /* Executed, start up.   */
    3:    *          printf("_dynamn is here\n");    /* Executed.             */
    4:    *          return(0);                      /* Executed.             */
    5:             }                                 /* Non-executable.       */
pgm_name: cvgsamp ,load_module:,

Legend: (*=executed; .=not_executed, blank=comment/white space/non-executable)

SOURCE_FN      : //DSN:USER.DEV.C(CVGMAIN)    SOURCR_FN_SIZE : 28
SOURCE_COMPDATE: 10/09/02 10:05:28
SOURCE_SNAME   : CVGSAMP
  Line Covered  Source
  --------------------
    1:             #include     2:             int main()
    3:    *        {                                 /* Executed, start up.   */
    4:
    5:             /*==================================    Non-Executable.
    6:                'coverage' defines two types of C source code
    7:                lines, executable, and all others.  All others
    8:                can be white-space lines, comment lines, or lines
    9:                of code that do not result in executable code. ============*/
   10:
   11:    *        int exitrc =0;                    /* Executed, initialized.*/
   12:             int notableentry;                 /* Non-executable.       */
   13:             int (*ep)(void);                  /* Non-executable.       */
   14:             /* No doubt this is a comment line.    Non-executable.       */
   15:    *        notableentry=1;                   /* Executed.             */
   16:    *        if,(notableentry,= 1) printf("=1\n");/* Executed.          */
   17:    *        if (notableentry = 1)             /* Executed.             */
   18:    *          printf("=1\n");                 /* Executed.             */
   19:    *        if (notableentry > 1) printf(">3\n");/* Executed, if () only. */
   20:    *        if (notableentry > 1)             /* Executed.             */
   21:              {                               /* Non-executable.       */
   22:    .          printf(">1\n");                 /* Not executed.         */
   23:    .          notableentry=3;                 /* Not executed.         */
   24:    .          printf("notableentry = 3\n");   /* Not executed.         */
   25:              };                              /* Non-executable.       */
   26:             Label:                            /* Non-executable        */
   27:    *        printf("Hello World\n");          /* Executed.             */
   28:    *        loadm("CVGDYNM", &ep);            /* Executed.             */
   29:    *        if (ep != NULL)                       /* Executed.             */
   30:              {                                   /* Non-executable.       */
   31:    *          (*ep)();                           /* Executed.             */
   32:    *          unloadm(ep);                       /* Executed.             */
   33:              };                                  /* Non-executable.       */
   34:    *        cvgmain2();                           /* Executed.             */
   35:    *        exit(exitrc);                         /* Executed.             */
   36:    .        }                                     /* Not executed.  A      */
   37:                                                  /* return was added by   */
   38:                                                  /* the compiler because  */
   39:                                                  /* exit() was called     */
   40:                                                  /* instead of,return()., */

SOURCE_FN      : //DSN:USER.DEV.C(CVGMAIN2)    SOURCR_FN_SIZE : 29
SOURCE_COMPDATE: 10/09/02 10:05:29
SOURCE_SNAME   : CVGMN2
  Line Covered  Source
  --------------------
    1:             int cvgmain2()
    2:    *        {                                     /* Executed, start up.   */
    3:    *         printf("main2\n");                   /* Executed.             */
    4:    *         return(0);                           /* Executed.             */
    5:             }
```

# Release 7.50 Changes to the SAS/C Library Reference, Volume 1

## Updates to Function Categories

In Chapter 2, “Function Categories,” update the information for the different types of functions with the following text.

### Math Functions

Update the math functions, for example, **exp**, **atan**, with the following enhancements.

- □ All of the math functions support calls from functions compiled with the **bfp** option.
- □ All of the math functions support float and long double callers if the name is suffixed with the letter **f** or **l**.
- □ All of the math functions other than **fmod** are declared as macros in **tgmath.h**, and can be used in a type-generic fashion if that header file is included.
- □ Math functions which are documented as calling **_matherr** in case of errors will call **_matherb** instead if invoked from a caller compiled with the **bfp** option.
- □ The values returned for domain and range errors may differ for **bfp** callers from the documented values. In general, a NaN will be returned for domain errors. Overflow will produce an infinity or the largest possible finite number, depending on the rounding mode. Underflow will produce a denormalized result or a zero depending on the magnitude of the correct answer and the rounding mode. A singularity will produce an infinite result.
- □ Except in a few degenerate cases, for example, **pow(1.0, NaN)**, any function called with an argument that is a NaN will return a NaN.

Other changes or enhancements to existing functions worth noting are:

- The **sqrt** function is now built-in if binary floating point is used, or if the compiler option **archlevel(c)** is used.
- The function name **lgamma** is provided as a synonym for **gamma**. The **lgamma** name is preferred because it more accurately describes the function.
- The function prototypes for **fmax**, **fmin**, **hypot**, **erf** and **erfc** were previously defined only in **lcmath.h**. Because these functions are defined by the C99 standard, their prototypes are now also in **math.h**.

### String Utility Functions

Add the following functions to the list of string functions.

| | |
|---|---|
| **nan** | convert a string to an IEEE double NaN |
| **nanf** | convert a string to an IEEE float NaN |
| **nanl** | convert a string to an IEEE long double NaN |

### Timing Functions

The following paragraph:

The resolution and accuracy of time values vary from implementation to implementation. Timing functions under traditional UNIX C compilers return a value of type **long**. The library implements **time_t** as a **double** to allow more accurate time measurement. Keep this difference in mind for programs ported among several environments.

Should read:

The resolution and accuracy of time values vary from implementation to implementation. Timing functions under traditional UNIX C compilers return a value of type **long**. The library implements **time_t** and **clock_t** as hex format doubles to allow more accurate time measurement. This is permitted by the ANSI/ISO standards, but is unusual, as most systems define these types as integral types. This can lead to problems in programs ported from other environments. Also note that programs which use the **bfp** option may receive diagnostics if they attempt to evaluate expressions with both a **time_t** or **clock_t** operand and one or more double operands. Casting the **time_t** or **clock_t** value to a **__binfmt** double is recommended to solve such problems.

## Updates to the SAS/C Functions

The following sections contain updates to the SAS/C functions in Chapter 6, "Function Descriptions."

## llmax

Replace the SYNOPSIS and EXAMPLE sections with the following new sections:

### SYNOPSIS

```
#include <lclib.h>

long long int llmax(long long int s, long long int r);
```

### EXAMPLE

```
#include <lclib.h>
#include <stdio.h>
```

```
main()
{
long long int num1, num2; /* numbers to be compared */
long long int result; /* holds the larger of num1 and num2 */

puts("Enter num1 & num2 : ");
scanf("%lld %lld", &num1, &num2);
result = llmax(num1, num2);
printf("The larger number is %lld\n", result);
}
```

## llmin

Replace the SYNOPSIS and EXAMPLE sections with the following new sections:

### SYNOPSIS

```
#include <lclib.h>

long long int llmin(long long int s, long long int r);
```

### EXAMPLE

```
#include <lclib.h>
#include <stdio.h>

main()
{
long long int num1, num2; /* numbers to be compared */
long long int result; /* holds the smaller of num1 and num2 */

puts("Enter num1 & num2 : ");
scanf("%lld %lld", &num1, &num2);
result = llmin(num1, num2);
printf("The smaller number is %lld\n", result);
}
```

## storck

In the "DESCRIPTION" section of the **storck** function entry, make the following changes:

- Change the last list-item term, STROCK_STACK_REPT, to STORCK_STACK_REPT.
- Change the OS/390 Batch default value, **DDN:DBGSTG**, to **DDN:STGRPT**.

## Updates to Multi-Volume Seeks Support

In Chapter 3, "I/O Functions," add the following information for multi-volume seeks support.

- In the section titled, "Library access methods," in the information on the **rel** access method Under OS/390, add the following text after the list:

Under OS/390, datasets are allowed to extend to multiple volumes.

- □ In the section titled, “File positioning with standard I/O (fseek and ftell),” add the following entry to the table labeled, “OS/390 Files with Restricted Positioning:”

**Table 2.4** OS/390 Files with Restricted Positioning

| File Type | Restrictions |
|---|---|
| multivolume disk or tape file | For non-rel datasets: only rewind supported if not opened for append; only seek to the end of file supported if opened for append |

## New WTP Function

In Release 7.50, the **`WTP`** macro has been changed to a function. Because of this change, several modifications need to be made to the *SAS/C Library Reference, Volume 1*.

- □ In Chapter 1, “Introduction to the SAS/C Library,” in the section titled “Implementation of Functions,” delete **`WTP`** from the list of SAS/C functions implemented as macros.
- □ In Chapter 6, “Function Descriptions,” in the **`wto`** function description, delete the reference to the **`WTP`** macro, delete the sample showing the **`WTP`** macro, and add **`WTP`** to the list of related functions.
- □ In Chapter 6, “Function Descriptions,” add the function description for **`WTP`** at “WTP” on page 140 to the list of functions.

## Updates to SAS/C I/O Questions and Answers

In Chapter 3, “I/O Functions,” in the section titled, “SAS/C I/O Questions and Answers,” replace the section titled, “Sharing an Output PDS” with the following text.

**`Q.`** When I open a PDS member for output, the **`fopen`** call fails if another user has the PDS allocated, even if it is allocated as SHR. How can I write to the PDS if it shared with another user?

**`A.`** If more than one user writes to the same PDS at the same time, the results are unpredictable. Generally, both members will be damaged. For this reason, when a PDS member, or any other OS/390 data set, is opened for output, the library allocates the data set to OLD to make sure that no one else writes to it at the same time. In some cases, this may be overprotective, but it prevents file damage from unintended simultaneous access. In cases where the PDS will only be updated via SAS/C or ISPF, or other applications that conform to the ISPF enqueing mechanism, you can specifiy **`share=ispf`** as an amparm when you open the member to force the library to open the dataset as shared.

*Note:* With a PDSE, it is possible to simultaneously write to distinct members. Even with a PDSE, the effects are unpredictable if the same member is opened by more than one user for output at the same time. △

## Updates to Signal-Handling Functions

### Update to USS Supported Signals

In Chapter 5, "Signal-Handling Functions," in the section titled "Supported Signals," add the following entry to the list of signals managed exclusively by USS:

| | |
|---|---|
| **`SIGDUMP`** | request for **`SYSMDUMP`** |

### Updates to Information on Signals

In Chapter 5, "Signal-Handling Functions," add the following information for the new **`SIGBFPE`** and **`SIGOPER`** signals and for updates to other signals.

- In the section titled, "Synchronous Signals," add **`SIGBFPE`** to the list of sychronous signals.
- In the section titled, "Asynchronous Signals," add **`SIGOPER`** to the list of signals.
- In the section titled, "Supported Signals," add **`SIGBFPE`** and **`SIGOPER`** to the list of signals managed exclusively by SAS/C.
- Add the following entries to the table labeled, "Summary of Information from siginfo."

**Table 2.5** Summary of Information from siginfo

| Signal | Information Returned by `siginfo` for Signals Raised Naturally |
|---|---|
| **`SIGBFPE`** | pointer to structure of type **`FPE_t`** |
| **`SIGOPER`** | pointer to structure of type **`OPER_t`** |

- In the section titled, "Default Signal Handling," add the following entries to the table labeled, "Summary of Default Actions."

**Table 2.6** Summary of Default Actions

| Signal | SAS/C Library Default Action (`SIG_DFL` Handler) | USS Default Action (`SIG_DFL` Handler) |
|---|---|---|
| **`SIGBFPE`** | ABEND with 0C7 | not supported |
| **`SIGOPER`** | ABEND with user code 1225 | ends the process |

- In the section titled, "Ignoring Signals," add the following entry to the table labeled, "Summary of Ignoring Signals."

**Table 2.7** Summary of Ignoring Signals

| Signal | SAS/C Library Ignored Signals (`SIG_IGN` Handler) | USS Ignored Signals (`SIG_IGN` Handler) |
|---|---|---|
| **`SIGBFPE`** | program continues; result of computation undefined | not supported |

- In the section titled, "Signal Descriptions," add the following updates:

- Change the first sentence of the **SIGFPDIV** description to read:

  The **SIGFPDIV** signal is raised when the second operand of a hexadecimal format floating point division is zero, and default handling is in effect for **SIGFPE**.
- Change the first sentence of the **SIGFPOFL** description to read:

  The **SIGFPOFL** signal is raised when the magnitude of the result of a hexadecimal format floating-point computation exceeds the maximum supported by the hardware and default handling is in effect for **SIGFPE** .
- Change the first sentence of the **SIGFPUFL** description to read:

  The **SIGFPUFL** signal is raised when the magnitude of the result of a hexadecimal format floating-point computation exceeds the maximum supported by the hardware and default handling is in effect for **SIGFPE** .

### New SIGBFPE and SIGOPER Signals

In the section titled, “Signal Descriptions,” add the following entries for **SIGBFPE** and **SIGOPER**to the list of descriptions.

## SIGBFPE

**Binary Floating Point Error**

The **SIGBFPE** signal is raised when a binary floating-point exception occurs for which trapping is enabled, and default handling is in effect for **SIGFPE**. If you have specified a handler for **SIGFPE** (either **SIG_IGN** or a function you define), **SIGBFPE** is not raised.

### Default handling

If the **SIGBFPE** signal is raised, and default handling is in effect, the program abnormally terminates with an ABEND code of 0C7.

### Ignoring the Signal

If your program ignores **SIGBFPE**, program execution continues, but the results of the failed expression are unpredictable. Note that the exception bits for the failure may not be reflected in the floating-point environment.

### Information returned by siginfo

If you call **siginfo** after a **SIGBFPE** signal occurs, **siginfo** returns a pointer to a structure of type **FPE_t**. Refer to the description of **SIGFPE** for a discussion of this structure.

### Notes on defining a handler

If you define a handler for **SIGBFPE**, you can change the result of the computation by using the information returned by **siginfo**. Refer to the example in the descrption of the **siginfo** function for an illustration of this technique.

## SIGOPER

**Operator Communication**

**Portability:** SASC

### Description

SIGOPER is an asynchronous signal. The SIGOPER signal is raised when the operator issues a STOP or MODIFY command or if a parm was included in a START command that initiated program execution.

*Note:* Operator commands will only raise the SIGOPER signal after a program call to the **opcmd** function to enable this interface. If the SAS/C program was invoked from an operator **START** command, the signal for the **START** command will generally be raised on return from the call to **opcmd**. △

Because SIGOPER is an asynchronous signal, the SAS/C library discovers the signal only when you call a function, when a function returns, or when you issue a call to **sigchk**.

### Default handling

By default, SIGOPER causes the program to abnormally terminate with a user ABEND code of 1225.

### Ignoring the signal

It is possible, but not particularly useful, to ignore SIGOPER.

### Information returned by siginf

When **siginfo** is called in a handler for SIGOPER, it returns a pointer to an **OPER_t** structure. It is defined as:

```
typedef struct {
    unsigned char type;   /* Type of request                        */
#define OPER_start  0x01  /* START                                  */
#define OPER_stop   0x02  /* STOP                                   */
#define OPER_modify 0x03  /* MODIFY                                 */
    char request[105];    /* text from operator - null terminated   */
    char unused[22];      /* future use                             */
   } OPER_t;
```

**type** contains an indicator showing which operator command was issued. For START and MODIFY, the **request** field will contain a null-terminated string containing the parameter the operator entered.

# Release 7.50 Changes to the SAS/C Library Reference, Volume 2

## Updates to Header Files in Function Examples

In Chapter 20, "POSIX Function Reference," replace the examples for the **chpriority**, **getpgid**, and **getsid** functions with the examples that are provided here.

### chpriority

```
/*---------------------------+
| POSIX/UNIX header files    |
+----------------------------*/
#include <sys/types.h>
#include <unistd.h>
#include <sys/resource.h>
/*---------------------------+
| ISO/ANSI header files      |
+----------------------------*/
#include <stdio.h>
#include <stdlib.h>
/*---------------------------+
| Name:     main             |
| Returns: exit(EXIT_SUCCESS)|
| or exit(EXIT_FAILURE)      |
+----------------------------|
+---------------------------*/
int main()
{
/* generic return code      */
   int rc;
/* which kind of process    */
/* id to use                */
   int kind;
 /* process id              */
   pid_t id;
/* is oeprty relative or    */
/* absolute                 */
   int form;
/* process priority         */
/* (version 1)              */
   int prty1;

/* process priority         */
/* (version 2)              */
   int prty2;

/* process id from getpid() */
   pid_t pid;

/* process group id from     */
/* getpgid()                 */
   pid_t pgid;
```

```
/* process user id from      */
/* getuid()                  */
   uid_t uid;
/* get the user id for this   */
/* process                    */
   uid = getuid();
/* get the process id for     */
/* this process               */
   pid = getpid();
/* get the group process id   */
/* for this process  */
   pgid = getpgid(pid);
   if (pgid == -1)
   {
    perror("Call to getpgid failed");
    exit(EXIT_FAILURE);
   }
   printf("      The process id: %d\n",
     (int)pid);
   printf("The process group id: %d\n",
     (int)pgid);
   printf(" The process user id: %d\n",
     (int)uid);
   /*------------------------------*/
   /* Get the process priority     */
   /* using the process id         */
   /*------------------------------*/
   printf("\nGet the Process Priority using the Process ID\n");

/* the id arg is the pid of a process */
   kind = PRIO_PROCESS;
   /* version 1 */
   id = (id_t)pid;
/* Set errno to zero for    */
/* error handling           */
   errno = 0;
   prty1 = getpriority(kind, id);
/* --------------------------------*/
/* Test for Error                  */
/* Note:                           */
/* getpriority() may return a '-1'  */
/* return code for either a         */
/* failure rc, or when the priority */
/* is in-fact '-1'.  To distinguish */
/* between the two conditions,      */
/* check the errno                  */
/* value for a non-zero value.      */
/*---------------------------------*/
 if (prty1 == -1 && errno != 0)
 {
   perror("Call to getpriority failed");
   exit(EXIT_FAILURE);
   }
   else
   {
      printf("The process priority (pid:version 1): %d\n", prty1);

   }

/* version 2 */
/* 0 implies current processs id  */
   id = (id_t)0;
/* Reset errno to zero for        */
/* error handling                 */
   errno = 0;
   prty2 = getpriority(kind, id);
/* Test for Error */
   if (prty2 == -1 && errno != 0)
   {
      perror("Call to getpriority failed");
      exit(EXIT_FAILURE);
   }
   else
   {
     printf("The process priority (pid:version 2): %d\n", prty2);

   }
/*-----------------------------*/
/* Get the process priority    */
/* using the group process id  */
/*-----------------------------*/
   printf("\nGet the Process Priority using the Group Process ID\n");

/* the id arg is the group process id */
   kind = PRIO_PGRP;
/* version 1 */
   id = (id_t)pgid;
/* Set errno to zero for error handling      */
   errno = 0;
   prty1 = getpriority(kind, id);
/* Test for Error */
   if (prty1 == -1 && errno != 0)
   {
      perror("Call to getpriority failed");
      exit(EXIT_FAILURE);
   }
   else
   {
      printf("The process priority (gpid:version 1): %d\n", prty1);

   }
/* version 2 */
/* 0 implies current group processs id       */
   id = (id_t)0;
/* Reset errno to zero for error handling    */
   errno = 0;
   prty2 = getpriority(kind, id);
/* Test for Error */
```

```
   if (prty2 == -1 && errno != 0)
   {
      perror("Call to getpriority failed");
      exit(EXIT_FAILURE);
   }
   else
   {
      printf("The process priority (gpid:version 2): %d\n", prty2);
   }
 /*---------------------------------------------*/
 /* Get the process priority using the          */
 /* process User id                             */
 /*---------------------------------------------*/
 printf("\nGet the Process Priority of the User ID\n");

/* the id arg is the user id of the process  */
   kind = PRIO_USER;
/* version 1 */
   id = (id_t)uid;
/* Set errno to zero for error handling      */
   errno = 0;
   prty1 = getpriority(kind, id);
   /* Test for Error */
   if (prty1 == -1 && errno != 0)
   {
      perror("Call to getpriority failed");
      exit(EXIT_FAILURE);
   }
   else
   {
      printf("The process priority (uid:version 2):  %d\n", prty1);
   }

/* version 2 */
/* Reset errno to zero for error handling    */
   errno = 0;

 /* 0 implies current process user id        */
   id = (id_t)0;
   prty2 = getpriority(kind, id);
   /* Test for Error */
   if (prty2 == -1 && errno != 0)
   {
      perror("Call to getpriority failed");
      exit(EXIT_FAILURE);
   }
   else
   {
      printf("The process priority (uid:version 2):  %d\n", prty2);
   }
/*-----------------------------------------------*/
/* Set the process priority using the            */
/* process id                                    */
 /*----------------------------------------------*/
 printf("\nSet the Process Priority using the Process ID\n");
/* the id arg is the pid of a process        */
   kind = PRIO_PROCESS;
/* an id of 0 implies current processs id    */
   id = (id_t)0;
/* Reset errno to zero for error handling    */
   errno = 0;
/* Set process priority to 5                 */
   prty1 = 5;
   rc = setpriority(kind, id, prty1);
   /*-----------------------------------------------*/
   /* Test for Error                                */
   /* Note: UNIX System Services  sites must enable */
   /*       the use of the setpriority() function.  */
   /*       If the use of setpriority() has not     */
   /*       beenenabled, any use of setpriority()   */
   /*       will fail with errno set to ENOSYS.     */
   /*                                               */
   /*-----------------------------------------------*/
   if (rc == -1)
   {
      perror("Call to setpriority failed");
      exit(EXIT_FAILURE);
   }
   else
   {
      prty2 = getpriority(kind, id);
      /* Test for Error */
      if (errno != 0)
      {
         perror("Call to getpriority failed");
         exit(EXIT_FAILURE);
      }
      printf("The process priority is now (pid):  %d\n", prty2);
   }
   /*-------------------------------------------*/
   /* Set the process priority using the group   */
   /* process id                                 */
   /*-------------------------------------------*/
 printf("\nSet the Process Priority using the Group Process ID\n");
/* the id arg is the group id of the process */
   kind = PRIO_PGRP;
/* 0 implies current group processs id       */
   id = (id_t)0;
/* Reset errno to zero for error handling    */
   errno = 0;
/* Set process priority to 10                */
   prty1 = 10;
   rc = setpriority(kind, id, prty1);
   /* Test for Error */
   if (rc == -1)
   {
      perror("Call to setpriority failed");
      exit(EXIT_FAILURE);
   }
   else
   {
      prty2 = getpriority(kind, id);
      /* Test for Error */
      if (errno != 0)
      {
         perror("Call to getpriority failed");
         exit(EXIT_FAILURE);
      }
      printf("The process priority is now (gpid):  %d\n", prty2);
   }
 /*--------------------------------------------*/
 /* Set the process priority using the         */
 /*process User id                             */
 /*--------------------------------------------*/
  printf("\nSet the Process Priority of the User ID\n");
/* the id arg is the user id of the process  */
   kind = PRIO_USER;
/* an id of 0 implies current user id        */
   id = (id_t)0;
/* Reset errno to zero for error handling    */
   errno = 0;
/* Set process priority to 15                */
   prty1 = 15;
   rc = setpriority(kind, id, prty1);
/* Test for Error */
   if (rc == -1)
   {
      perror("Call to setpriority failed");
      exit(EXIT_FAILURE);
   }
   else
   {
      prty2 = getpriority(kind, id);
      /* Test for Error */
      if (errno != 0)
      {
         perror("Call to getpriority failed");
         exit(EXIT_FAILURE);
      }
printf("The process priority is now (uid): %d\n", prty2);
   }
/*--------------------------------------------*/
/* Change the process priority using the     */
/* process id                                */
/*--------------------------------------------*/
printf("\nChange the Process Priority using the Process ID\n");
/* the id arg is the pid of a process        */
   kind = PRIO_PROCESS;
/* an id of 0 implies current processs id    */
   id = (id_t)0;

/* Reset errno to zero for error handling    */
   errno = 0;
/* change using "absolute"                    */
/* priority - equivalent to setpriority()    */
   form = CPRIO_ABSOLUTE;
   printf("\tChange using CPRIO_ABSOLUTE\n");
/* Change process priority to 3              */
   prty1 = 3;
   rc = chpriority(kind, id, form, prty1);
/*-----------------------------------------------*/
/* Test for Error                                */
/* Note: UNIX System Services  sites must enable  */
/*       the use of the chpriority() function.    */
/*       If the use of chpriority() has not been  */
/*       enabled, any use of chpriority() will    */
/*       fail with errno set to ENOSYS.           */
/*                                                */
/*-----------------------------------------------*/
   if (rc == -1)
   {
      perror("Call to chpriority failed");
      exit(EXIT_FAILURE);
   }
   else
   {
      prty2 = getpriority(kind, id);
      /* Test for Error */
      if (errno != 0)
      {
         perror("Call to getpriority failed");
         exit(EXIT_FAILURE);
      }
  printf("The process priority is now (pid): %d\n", prty2);
   }
/* change using "relative" priority */
   form = CPRIO_RELATIVE;
   printf("\tChange using CPRIO_RELATIVE\n");
/* Bump process priority up by 2            */
   prty1 = 2;
   rc = chpriority(kind, id, form, prty1);
   /* Test for Error */
   if (rc == -1)
   {
      perror("Call to chpriority failed");
      exit(EXIT_FAILURE);
   }
   else
   {
      prty2 = getpriority(kind, id);
      /* Test for Error */
      if (errno != 0)
      {
       perror("Call to getpriority failed");
       exit(EXIT_FAILURE);
      }
   printf("The process priority is now (pid):  %d\n", prty2);
   }
/*---------------------------------------------*/
/* Change the process priority using the       */
/* group process id                            */
/*---------------------------------------------*/
 printf("\nChange the Process Priority using the Group Process ID\n");
/* the id arg is the group id of the process */
   kind = PRIO_PGRP;
/* 0 implies current group processs id       */
    id = (id_t)0;
/* Reset errno to zero for error handling    */
   errno = 0;
   /* change using "absolute"                  */
   /* priority - equivalent to setpriority()  */
   form = CPRIO_ABSOLUTE;
   printf("\tChange using CPRIO_ABSOLUTE\n");
 /* Change process priority to 7              */
   prty1 = 7;
   rc = chpriority(kind, id, form, prty1);
   /* Test for Error */
   if (rc == -1)
   {
      perror("Call to chpriority failed");
      exit(EXIT_FAILURE);
   }
   else
   {
      prty2 = getpriority(kind, id);
      /* Test for Error */
      if (errno != 0)
      {
         perror("Call to getpriority failed");
         exit(EXIT_FAILURE);
      }
  printf("The process priority is now (gpid): %d\n", prty2);
   }
/* change using "relative" priority */
   form = CPRIO_RELATIVE;
   printf("\tChange using CPRIO_RELATIVE\n");
/* Bump process priority up by 3            */
   prty1 = 3;
   rc = chpriority(kind, id, form, prty1);
   /* Test for Error */
   if (rc == -1)
   {
      perror("Call to chpriority failed");
      exit(EXIT_FAILURE);
   }
   else
   {
      prty2 = getpriority(kind, id);
      /* Test for Error */
      if (errno != 0)
      {
         perror("Call to getpriority failed");
         exit(EXIT_FAILURE);
      }
 printf("The process priority is now (gpid):  %d\n", prty2);
   }
/*--------------------------------------*/
/* Change the process priority using    */
/* the process User id                  */
/*--------------------------------------*/
printf("\nChange the Process Priority of the User ID\n");
/* the id arg is the user id of the process  */
   kind = PRIO_USER;
/* 0 implies current group processs id       */
   id = (id_t)0;
/* Reset errno to zero for error handling    */
   errno = 0;
/* change using "absolute"                  */
/*  priority - equivalent to setpriority()  */
   form = CPRIO_ABSOLUTE;
   printf("\tChange using CPRIO_ABSOLUTE\n");
/* Change process priority to 11           */
   prty1 = 11;
   rc = chpriority(kind, id, form, prty1);
   /* Test for Error */
   if (rc == -1)
   {
      perror("Call to chpriority failed");
      exit(EXIT_FAILURE);
   }
   else
   {
      prty2 = getpriority(kind, id);
      /* Test for Error */
      if (errno != 0)
      {
       perror("Call to getpriority failed");
         exit(EXIT_FAILURE);
      }
printf("The process priority is now (uid):  %d\n", prty2);
   }
/* change using "relative" priority */
   form = CPRIO_RELATIVE;
   printf("\tChange using CPRIO_RELATIVE\n");
/* Bump process priority up by 4            */
   prty1 = 4;
   rc = chpriority(kind, id, form, prty1);
   /* Test for Error */
   if (rc == -1)
   {
      perror("Call to chpriority failed");
      exit(EXIT_FAILURE);
   }
   else
```

```
      {
         prty2 = getpriority(kind, id);
         /* Test for Error */
         if (errno != 0)
         {
         perror("Call to getpriority failed");
          exit(EXIT_FAILURE);
         }
         printf("The process priority is now (uid):  %d\n", prty2);
      }
      exit(EXIT_SUCCESS);
   }  /* end of main() */
```

## getpgid

```
/*-------------------------------------+
| POSIX/UNIX header files              |
+-------------------------------------*/
#include <sys/types.h>
#include <unistd.h>
/*-------------------------------------+
| ISO/ANSI header files                |
+-------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
/*-------------------------------------+
| Name:      main                      |
| Returns:   exit(EXIT_SUCCESS) or
| exit(EXIT_FAILURE)                   |
+-------------------------------------*/
int main()
{
/* current process id from getpid()     */
   pid_t pid;

/* process group id from getpgid()      */
   pid_t pgid;
/*-----------------------------------------*/
/* Get the group process id of the current  */
/* process                                  */
/* Note: Both version 1 and version 2 are   */
/*       equivalent to calling the          */
/*       getpgrp() function                 */
/*-----------------------------------------*/
printf("\nGet the Group Process ID of the Current Process\n");
/* version 1                                 */
/* Set errno to zero for error handling      */
   errno = 0;
/* get the process id for this process       */
   pid = getpid();
/* get the group process id for this process  */
   pgid = getpgid(pid);
/* Test for Error */
   if (pgid == -1)
   {
      perror("Call to getpgid failed");
      exit(EXIT_FAILURE);
   }
   else
   {
      printf("      The process id: %d\n", (int)pid);
      printf("The process group id:%d\n", (int)pgid);
   }

/* version 2                                 */
/* Reset errno to zero for error handling    */
   errno = 0;
/* 0 implies current processs id             */
   pid = 0;
/* get the group process id for this process  */
   pgid = getpgid(pid);
/* Test for Error */
   if (pgid == -1)
   {
      perror("Call to getpgid failed");
      exit(EXIT_FAILURE);
   }
   else
   {
 printf("The process group id: %d\n", (int)pgid);
   }
   exit(EXIT_SUCCESS);
}  /* end of main() */
```

## getsid

```
/*-------------------------------------+
| POSIX/UNIX header files              |
+-------------------------------------*/
#include <sys/types.h>
#include <unistd.h>
/*-------------------------------------+
| ISO/ANSI header files                |
+-------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
/*---------------------------------------+
| Name:      main                        |
| Returns:   exit(EXIT_SUCCESS)          |
|            or exit(EXIT_FAILURE)       |
+ -------------------------------------*/
int main()
{
/* current process id from getpid()     */
   pid_t pid;
/* process group id from getsid()       */
   pid_t sid;
/*----------------------------------------*/
/* Get the Session Leader id for the     */
/* Current Process                       */
/*-----------------------------------------*/
printf("\nGet the Session Leader ID of the Current Process\n");
/* version 1 */
/* Set errno to zero for error handling    */
   errno = 0;
/* get the process id for this process     */
   pid = getpid();
/* get the session leader id for           */
/* this process                            */
   sid = getsid(pid);
/* Test for Error */
   if (sid == -1)
   {
      perror("Call to getsid failed");
      exit(EXIT_FAILURE);
   }
   else
   {
printf("       The process id: %d\n", (int)pid);
printf("The Session Leader id: %d\n", (int)sid);
   }

/* version 2                               */
/* Reset errno to zero for error handling  */
   errno = 0;
 /* 0 implies current processs id          */
   pid = 0;
/* get the session leader id for           */
/* this process                            */
   sid = getsid(pid);
/* Test for Error */
   if (sid == -1)
   {
      perror("Call to getsid failed");
      exit(EXIT_FAILURE);
   }
   else
   {
 printf("       The process id: %d\n",
  (int)pid);
 printf("The Session Leader id: %d\n",
  (int)sid);
   }

   exit(EXIT_SUCCESS);
}  /* end of main() */
```

## chown

Replace the **#include** statements in the **chown** example with the **#include** statements that are provided here.

```
#include <sys/types.h>
#include <grp.h>
#include <stdlib.h>
```

```
#include <stdio.h>
#include <unistd.h>
```

## New oeattach Example

In Chapter 20, "POSIX Function Reference," replace the **oeattach** sample code under the "Example" heading with the following code:

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <signal.h>
#include <lclib.h>
main()
{
   int pipefds[2];
   pid_t pid;
   char *const parmList[] = {"/bin/ls", "-l", "/u/userid/dirname",
                            NULL };
   char lsout[200];           /* buffer for out ls output          */
   int amt;
   int status;                /* status code from ls               */
   fclose(stdout);            /* avoid stdio interfering with fd 1 */
   fclose(stdin);             /* avoid stdio interfering with fd 0 */
   close(STDOUT_FILENO);      /* make sure fd's 0 & 1 are avail    */
   close(STDIN_FILENO);
   pipe(pipefds);             /* create both ends of a pipe        */
   if (pipefds[0] == STDOUT_FILENO) {
      fprintf(stderr,
              "Warning: Input fileno is 1, should not occur.\n");
      pipefds[0] = dup(pipefds[0]);
      close(STDOUT_FILENO);
   }
   if (pipefds[1] != STDOUT_FILENO) {
      fprintf(stderr,
              "Warning: Output fileno is not 1, should not occur.\n");
      dup2(pipefds[1],STDOUT_FILENO);
                              /* make write end of pipe fd 1       */
      close(pipefds[1]);      /* close write end                   */
   }
  pid = oeattach("/bin/ls", parmList);
                             /* run ls command as subtask         */
  close(STDOUT_FILENO);      /* close write end of pipe in parent */
  for(;;) {                  /* read from the pipe                */
     amt = read(pipefds[0], lsout, sizeof(lsout));
     if (amt <= 0) break;
     fwrite(lsout, 1, amt, stderr);  /* write ls output to stderr */
  }
  wait(&status);             /* wait for ls to complete           */
  close(pipefds[0]);         /* close pipe input end              */
  if (WIFEXITED(status)) exit(WEXITSTATUS(status));
  else             /* if ls failed, use kill to fail the same way */
     kill(0, WTERMSIG(status));
}
```

## Update to loadm Function Description

In Chapter 1, “Dynamic-Loading Functions,” in the **`loadm`** function description, in the section titled, “Example 1.2 dynamic loading modules with multiple functions,” replace the example code in “STEP I. Put the following declarations a common header file and name it DYNTABLE:” with the following example:

```
int func1(void)
int func2(void)

struct funcdef {           /* structure definition for functions     */
   int (*func1)();
   int (*func2)();
                            /* More functions can go here.            */
};

typedef struct funcdef *fptrtable;  /* pointer to list of funcdefs    */
```

## Updates to the osdynalloc Function

In Chapter 3, “MVS Low-Level I/O Functions,” update the **`osdynalloc`** function description with the following information.

Replace the corresponding lines in the table labeled, “SVC 99 Request Block Keywords,” with the following entries:

**Table 2.8** SVC 99 Requst Block Keywords

| Identifier | RB Field | Value | Description | Notes |
|---|---|---|---|---|
| msgbelow<br>elsto | S99LSTO | none | Allocate SVC 99 messages below the 16 meg line | |
| msgerror<br>msgrc | none | int* | Message processing error code (returned in register 15 by IEFDB476) | |

Replace the corresponding lines in the table labeled, “Dynamic Allocation Keywords,” with the following entries:

**Table 2.9** Dynamic Allocation Keywords

| Identifier | JCL Equiv | SVC 99Key | Value | Format | Description | Notes |
|---|---|---|---|---|---|---|
| subsysattr<br>ssattr<br>ssatt | none | DALSSATT | char* | Res Word | Subsystem attributes | (6) |
| subsysparm<br>ssparm<br>ssprm | SUBSYS=<br>(,parm... ) | DALSSPRM | char[68] | Multiple | Subsystem Parameters | (4) |

Add the following lines to the table labeled, “Dynamic Allocation Keywords:”

**Table 2.10** Dynamic Allocation Keywords

| Identifier | JCL Equiv | SVC 99Key | Value | Format | Description | Notes |
|---|---|---|---|---|---|---|
| bufl<br>buflen<br>bufln<br>dcbbufl<br>dcbbuflen<br>dcbbufln | DCB=BUFL= | DALBUFL | int | | Buffer length | |
| fdat<br>filedata | FILEDATA= | DALFDAT | char* | Res Word | TEXT/ BINARY for HFS file | |
| ovaff<br>overaff<br>overrideaff | | DALOVAFF | none | | Override affinity | |
| retclienttok<br>retctk<br>rtctoken<br>rtctk<br>rtctoken | none | DALRTCTK | char(*) | [81] | Return JES client token | |
| ssreq | none | DALSSREQ | char[5] | | Subsystem request | |
| uncnt<br>unitcount | none | DALUNCNT | int | | Device count | |
| unit | UNIT= | DALUNIT | char[9] | | Unit name | |

Replace the corresponding lines in the table labeled, “Dynamic Allocation Inquiry Keywords,” with the following entries:

**Table 2.11** Dynamic Allocation Inquiry Keywords

| Identifier | JCL Equiv | SVC 99Key | Value | Format | Description | Notes |
|---|---|---|---|---|---|---|
| path | PATH= | DINRPATH | char[256] | | HFS filename for which information is needed | (1) |
| retpathcdisp<br>rtpathcdisp<br>rpathcdisp<br>retpcdisp<br>rtpcdisp<br>rpcdisp<br>retpcds<br>rtpcds<br>rpcds<br>retcnds<br>rtcnds<br>rcnds | return PATHDISP=(n,c) | DINRPCDS | int* | Encoded | Return HFS file abnormal disposition | |

Add the following lines to the table labeled, “Dynamic Allocation Inquiry Keywords:”

**Table 2.12** Dynamic Allocation Keywords

| Identifier | JCL Equiv | SVC 99Key | Value | Format | Description | Notes |
|---|---|---|---|---|---|---|
| retfdat<br>retfiledata<br>rtfdat<br>rfiledata<br>rtfdat<br>rtfiledata | return FILEDATA= | DINRFDAT | int* | Encoded | Return TEXT/ BINARY attribute of HFS file | |
| rettyp<br>rettype<br>rttyp<br>rttype | none | DINRTTYP | char* | Encoded | Return data set type | |

| Identifier | JCL Equiv | SVC 99Key | Value | Format | Description | Notes |
|---|---|---|---|---|---|---|
| retpathndisp<br>retpndisp<br>retpnds<br>rpathndisp<br>rpndisp<br>rpnds<br>rtpathndisp<br>rtpndisp<br>rtpnds | return PATHDISP= (n, ) | DINRPNDS | int* | Encoded | Return HFS file disposition | |
| retvolume<br>retvolser<br>retvol<br>rtvolume<br>rtvolser<br>rtvol | return VOL=SER= | DINRTVOL | char(*) | [7] | Return first volume serial | |

## Updates to the Socket Functions

In Chapter 18, "Socket Function Reference," update the following sections as indicated:

- Update the second paragraph of the **selectecb** description to read:

  The **ecblist** argument is the address of an array of structures, each of which represents one or more contiguous **_ecblist** structures. Each structure contains two members, a count of the number of ECBs, and the address of an array of ECBs. The count may be zero, in which case the ECB array address is ignored.

- Add the following note after the last paragraph of the **setsocketopt** description:

  *Note:* **TCP_NODELAY** is not supported for non-integrated sockets. This is an IBM restriction, and is not a SAS/C restriction. However, **TCP_NODELAY** is supported for **OE** (integrated) sockets, which is the default for SAS/C Release 7.00. For releases prior to 7.00, you can use the **setsockimp("OE")** function to get integrated sockets. △

## Update to getdtablesize Example

In Chapter 18, "Socket Function Reference," replace the code in the "Example" section with the following code:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>

main()
{
   int maxsockets;
   fd_set *readset, *writeset, *exceptset);
   int ready;
```

```
      /* get maximum number of sockets                          */
   maxsockets = getdtablesize();

     /* allocate storage for fd sets (8 bits per byte, 4 byte int) */
   readset   = calloc((maxsockets+31)/32, 4);
   writeset  = calloc((maxsockets+31)/32, 4);
   exceptset = calloc((maxsockets+31)/32, 4);


      /* allocate storage for fd sets (8 bits per byte)             */
   .
   .
   .
   ready = select(maxsockets, readset, writeset, exceptset, NULL);

      /* wait for socket activity                                   */
   .
   .
   .
}
```

## New <sys/un.h> Header File

In Chapter 15, "The BSD UNIX Socket Library," add the following information for the <sys/un.h> header file:

- In the section titled, "Header Files," add the following entry for the **<sys/un.h>** header file to the list of header files:

  **17 <sys/un.h>**

- After the section titled "<sys/socket.h>" add a section titled "<sys/un.h>" that contains the following text:

  The <sys/un.h> header file contains the definitions for UNIX domain sockets. <sys/socket.h> must be included in your code before <sys/un.h>.

  **sockaddr_un** as illustrated in the following example, is the UNIX domain socket address structure.

```
struct   sockaddr_un {
   unsigned char sun_len;      /* sockaddr len including null */
   sa_family_t sun_family;     /* AF_UNIX / AF_LOCAL          */
   char sun_path[104];         /* path name                   */
};
```

  The family type constant AF_LOCAL has replaced the equivalent, historical constant AF_UNIX in more recent practice. However, both constants are defined in the header file <sys/socket.h>, and either one may be used.

CHAPTER

3

# SAS/C Cross-Platform Changes for Release 7.50

## Introduction

This chapter provides a complete description of the changes and enhancements to the *SAS/C Cross-Platform Compiler and C++ Development System User's Guide* for Release 7.50.

## Release 7.50 Enhancements to the SAS/C Cross-Platform Compiler

### New Cross-Platform Compiler Options

#### New Options Summary

Table 3.1 on page 200 summarizes all the new cross-platform compiler options. This table is an extension of the table labeled, "Compiler Options" in Chapter 3, "Compiling C and C++ Programs."

The option specifications are listed in the first column of the table. The second column indicates whether the option can be negated. An exclamation point (!) means that the option can be negated. A plus sign (+) means that the option cannot be negated.

**Table 3.1** Compiler Options

| Option | Negation | Default | Description |
|---|---|---|---|
| **-Karchlevel=let** | ! | -Knoarchlevel | Specifies 390 architecture. |
| **-Kasynsig** | ! | -Kasynsig | Specifies the detection of asynchronous signals when exiting from a function. |
| **-Kbfp** | ! | -Knobfp | Specifies IEEE floating point format. |
| **-Kcoverage** | ! | -Knocoverage | Activates the COVERAGE feature of the compiler. |
| **-Kc99subset** | ! | -Knoc99subset | See description. |
| **-Khugeptrs** | ! | -Knohugeptrs | Specifies 64-bit addressing. |
| **-Khxref** | ! | -Knohxref | Prints references in standard header files in the cross-reference listing. |
| **-Kiccp** | ! | | Call the CICS traslator before compiling. |
| **-Kixref** | ! | -Knoixref | Prints references in user header files in the cross-reference listing. |
| **-Kmpsafe** | ! | -Knompsafe | Specifies safe asynchronous signals for nonMP configurations. |
| **-Knofriendinject** | ! | **-Kfriendinject** | For C++ compilations: Disable the creation of visible names for friend class and function declarations. |
| **-Koptions** | ! | -Koptions | Specifies options listing. |
| **-Koldforscope** | ! | **-Knooldforscope** | For C++ compilations: Use the old scoping rules for variables declared in the initialization clause of a **for** statement. |
| **-Kstkbelow** | ! | -Knostkbelow | Allocates stack below the 16M line. |
| **-Kstmap** | ! | | Requests a map of structure elements. |
| **-Kwarn** | ! | -Knowarn | Causes the compiler warning messages to be printed. |

## New Options

**-Karchlevel**

allows you to request code generation for a specific level of the 390 architecture. By specifying this option, you can exploit newer features of recent processors, but you should be aware that the generated code will fail if you run it on a processor that does not have the indicated feature.

The **archlevel** option specifies an architectural level by a single-letter code. Four codes are supported currently, with the following meanings.

| | |
|---|---|
| **a** | The processor supports the **logical string assist** facility. This facility allows the compiler to generate better code for **string.h** built-in functions such as **strlen**, **strcpy**, and **strcmp**. |
| **b** | The processor supports the immediate and relative instruction set, as well as the compare and move extended facility. This allows the compiler to generate improved code in many areas. |
| **c** | The processor supports the floating-point extensions feature. This feature allows the compiler to generate improved code for floating-point computations, and is required to use IEEE floating-point. Note that use of **archlevel(c)** is advantageous for programs which use the traditional 390 hex format for floating-point as well as for IEEE applications. |
| **d** | The processor supports the z/Architecture. This feature allows the compiler to exploit 64-bit registers for programs which use the long long data type, and is a prerequisite for 64-bit addressing support. |

*Note:* The codes are cumulative so that, for instance, specifying an architecture level of **c** indicates presence of all features defined for levels **a** and **b**,as well as **c**. △

If no architecture level is specified, the compiler assumes that none of the above architectural features can be used. However, if the **bfp** option is specified, an **archlevel** of **c** is assumed by default, and if the **hugeptrs** option is specified, the **archlevel** is assumed to be **d**.

**-Kasynsig**

specifies that extra code should be generated when necessary to detect asynchronous signals on exit from a function. If the program does not use any asynchronous signals (such as SIGINT, SIGALRM, or any POSIX signal), you can improve performance slightly by specfying **noasynsig**.

If **noasynsig** is specified but the program responds to asynchronous signals, detection of these signals by the program may be delayed, causing the handler to be called later than would otherwise be expected.

**-Kbfp**

specifies that the default floating point format is binary (IEEE). Note that the **BFP** option implies the **ARCHLEVEL(C)** option.

**-Kcoverage**

activates the COVERAGE feature of the compiler, which provides information on which lines of code written in C were executed during a compilation. This adds additional code and a data element for tracking execution to the object. The information about which lines of code were executed are made available at runtime by the **__cvgtrm** routine, which the user can provide.

**-Kc99subset**

**-Kc99subset** enables the following new features of the ISO C99 standard:

- Variable declarations may occur anywhere within a compound block and in the first clause of a **for** statement.
- The treatment of large unsuffixed decimal constants is C99 compliant in the determination of type, for example, 4000000000 is a 64 bit signed integer instead of a 32 bit unsigned integer.
- The predefined identifier **__func__** is made available for each function. Whenever referenced **__func__** is treated as if it were declared at the beginning of the function as follows:

```
static const char __func__[] = function-name;
```

If the identifier is not used, the declaration will be deleted and no space will be wasted.

- Preprocessor macros may have a variable number of arguments, for example,

```
#define LOGIT(...) fprintf(logfile, __VA_ARGS__)
LOGIT("x was %d, but y was %d", x, y);
```

- **inline** is a keyword. This is equivalent to the SAS/C **__inline** keyword except that
  - Only one definition may occur in a compilation unit.
  - An actual external definition will be created if the function is declared with external linkage.
- **restrict** is a keyword. The optimizer does not use the information provided by the use of this qualifier; consequently, its use will not result in better code genration. The keyword is enabled as a convenience for porting code written for C99 to SAS/C.

*Note:* **-Kc99subset** is ignored for C++ compilations. △

**-Khugeptrs**
specifies that the object code is intended to execute in 64-bit addressing mode. When **hugeptrs** is specified, the default pointer type is **__huge**, and the size of signed and unsigned long data is 8 bytes. Note that the **hugeptrs** option implies the **ARCHLEVEL(D)** option.

Though the **hugeptrs** option is valid under CMS, the object code generated with this option cannot be executed under CMS because of operating system limitations.

**-Khxref**
prints references in standard header files in the cross-reference listing. See **hlist** for a description of header files.

**-Kiccp**
calls the CICS translator before compiling. When **-Kiccp** is specified, the CICS translator is called regardless of the file extension.

Prior to the implementation of the **-Kiccp** option, it was necessary to use a **.ccp** extension to indicate that the CICS translator should be called. Now that the **-Kiccp** is available, you can use one of the following commands to call the CICS translator:

```
sascc370 -Kiccp test.cxx
sascc370 -Kiccp test.c
```

where:

- The **.cxx** extension causes the C++ translator to be called after the CICS translator.
- The **.c** extension causes the C compiler to be called after the CICS translator.

The old behavior is still functional, and you can use the **.ccp** extension to indicate that the CICS translator should be called.

**-Kixref**
lists references in user **#include** files.

**-Kmpsafe**
specifies that extra code should be generated to assure correct behavior when a SAS/C asynchronous signal is detected on a different processor in an MP configuration than the one executing the SAS/C program. An example would be a

user-added asynchronous signal which is generated by a subtask of the SAS/C program. **mpsafe** causes a slight performance penalty in the function epilog, so it should be used only when the object code may be used in the presence of such asynchronous signals.

If **nompsafe** is specified or defaulted, an asynchronous signal generated on another processor may be ignored or may cause an ABEND if it occurs while the function is returning.

**-Knofriendinject**
specifies that the translator should make the names of of **friend** class and function names visible in the enclosing non-class scope of the class containing the declaration. This is called *friend name injection*. The C++ standard requires that friend names not be injected. However older code may require name injection. The **-Kfriendinject** option is the default for compatability with older code. However this default may change in a future release.

The **-Knofriendinject** option is equivalent to the SAS/C C++ Development System's **nofriendinject** option.

**-Koldforscope**
specifies that the scope of a variable defined in the initialization clause of a **for** statement will follow the old C++ rules concerning scoping. The new scoping rules in the C++ standard specify that the scope of a variable defined in the **for** loop initialization clause only includes the for statement and its associated loop body.

The **-Koldforscope** option is equivalent to the SAS/C C++ Development System's **oldforscope** option.

**-Koptions**
generates an options listing. The options listing contains all options in effect for the compilation.

**-Kstkbelow**
causes the stack frame for functions in this compilation to be allocated below the 16M line. If the **_stkabv** external variable has been set to indicate the the program wants to have the stack above the line, but certain functions cannot tolerate this (for example, stack variables will be passed to system services that run only AMODE=24), then **stkbelow** can be specified to force the **auto** variables of such functions to be allocated below the line. For assembler routines, the **STKBELOW=YES** option of the CENTRY macro will accomplish the same result. For best results, compile only those functions that require a stack below the line with **stkbelow**.

**-Kstmap**
requests that a map of structure elements and their offsets be generated in the cross-reference for each structure tag enclosed. Specifying the **-Kstmap** option implies the **-Kxref** option.

**-Kwarn**
causes compilation warning messages to be printed. **nowarn** suppresses warning messages.

## New cool Options

### cool Options Summary

Table 3.2 on page 204 lists the new cool options. This table is an extension of the table labeled, “cool Options,” in Chapter 6, “Prelinking C and C++ Programs.”

**Table 3.2** cool Options

| sascc370 Option | Negation | Default | cool Option | Description |
|---|---|---|---|---|
| **-Aenexitdata=***dll* | **+** | | **-xt***dll* | Under Windows, specifies the name of a DLL that generates external symbols that are used by COOL for extended processing. |
| **-Asevere** | **!** | **-Anosevere** | **-we** | Causes COOL to assign the same level of importance to warnings as it does to errors. |

## cool Options Description

**-Aenexitdata=***dll*
Under Windows, specifies the name of a Dynamic-Link Library that generates external names that are used by COOL to resolve extended names. See "COOL Extended Names Processing" in Appendix 7, "Extended Names" of the *SAS/C Compiler and Library User's Guide*.

**severe** (**-Asevere** under USS)
causes COOL to assign the same level of importance to warnings as it does to errors. If COOL returns a warning, the COOL return code is the same as if COOL had returned an error; however, the message the user receives for a warning remains the same as before. It indicates only that COOL has returned a warning, not an error.

## New Alternative Code Page Feature

In Chapter 3, "Compiling C and C++ Programs," following the section titled, "External Compiler Variables," add a new heading of the same level with the title, "Alternative Code Page Feature," that contains the following text.

With release 7.50, the SAS/C and C++ cross-platform compiler allows you to specify alternative codepages for the translation of characters used in your source code. Two codepage tables are required: one giving the ASCII to EBCDIC translation and a second giving the EBCDIC to ASCII translation. The codepage tables must be placed in files named **atoe.codepage** and **etoa.codepage**. The location of the directory containing these files is specified with the **_SASC_CODEPAGE_PATH** environment variable.

The alternative codepages enable you to control the way the ASCII characters in your source code are translated into EBCDIC by the compiler. For example, you can use them to control the way the ASCII '$' is translated in the following example:

```
#include <stdio.h>
void main(void)
{
 int ascii_char = '$';

 printf("ebcdic_char = %c\n", ascii_char);
 printf("ebcdic_char = %d\n", ascii_char);
}
```

If you compile this on the cross-platform compiler, the ASCII '$' **0x24** will be translated into the EBCDIC '$' **0x5B**. By default the '$' will be displayed at run-time on your EBCDIC mainframe even though you compiled it on an ASCII machine.

This default behavior is what you usually require; however, there are situations in which you want to control the ASCII to EBCDIC translation performed by the compiler. You do this by supplying alternative codepages.

The following figures, atoe.codepage and etoa.codepage, provide examples of alternative codepages. Note that the codepage tables consist of a 16X16 array of hexadecimal digits where position determines the replacement value and comments are delineated by semicolons.

**Example Code 3.1** atoe.codepage

```
00 01 02 03 37 2D 2E 2F 16 05 15 0B 0C 0D 0E 0F     ; 00 ;
10 11 12 13 3C 3D 32 26 18 19 3F 27 1C 1D 1E 1F     ; 10 ;
40 5A 7F 7B 5B 6C 50 7D 4D 5D 5C 4E 6B 60 4B 61     ; 20 ;
F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 7A 5E 4C 7E 6E 6F     ; 30 ;
7C C1 C2 C3 C4 C5 C6 C7 C8 C9 D1 D2 D3 D4 D5 D6     ; 40 ;
D7 D8 D9 E2 E3 E4 E5 E6 E7 E8 E9 AD E0 BD 5F 6D     ; 50 ;
79 81 82 83 84 85 86 87 88 89 91 92 93 94 95 96     ; 60 ;
97 98 99 A2 A3 A4 A5 A6 A7 A8 A9 C0 4F D0 A1 07     ; 70 ;
20 21 22 23 24 25 06 17 28 29 2A 2B 2C 09 0A 1B     ; 80 ;
30 31 1A 33 34 35 36 08 38 39 3A 3B 04 14 3E FF     ; 90 ;
41 AA 4A B1 9F B2 6A B5 BB B4 9A 8A B0 CA AF BC     ; A0 ;
90 8F EA FA BE A0 B6 B3 9D DA 9B 8B B7 B8 B9 AB     ; B0 ;
64 65 62 66 63 67 9E 68 74 71 72 73 78 75 76 77     ; C0 ;
AC 69 ED EE EB EF EC BF 80 FD FE FB FC BA AE 59     ; D0 ;
44 45 42 46 43 47 9C 48 54 51 52 53 58 55 56 57     ; E0 ;
8C 49 CD CE CB CF CC E1 70 DD DE DB DC 8D 8E DF     ; F0 ;
```

**Example Code 3.2** etoa.codepage

```
00 01 02 03 9C 09 86 7F 97 8D 8E 0B 0C 0D 0E 0F     ; 00 ;
10 11 12 13 9D 0A 08 87 18 19 92 8F 1C 1D 1E 1F     ; 10 ;
80 81 82 83 84 85 17 1B 88 89 8A 8B 8C 05 06 07     ; 20 ;
90 91 16 93 94 95 96 04 98 99 9A 9B 14 15 9E 1A     ; 30 ;
20 A0 E2 E4 E0 E1 E3 E5 E7 F1 A2 2E 3C 28 2B 7C     ; 40 ;
26 E9 EA EB E8 ED EE EF EC DF 21 24 2A 29 3B 5E     ; 50 ;
2D 2F C2 C4 C0 C1 C3 C5 C7 D1 A6 2C 25 5F 3E 3F     ; 60 ;
F8 C9 CA CB C8 CD CE CF CC 60 3A 23 40 27 3D 22     ; 70 ;
D8 61 62 63 64 65 66 67 68 69 AB BB F0 FD FE B1     ; 80 ;
B0 6A 6B 6C 6D 6E 6F 70 71 72 AA BA E6 B8 C6 A4     ; 90 ;
B5 7E 73 74 75 76 77 78 79 7A A1 BF D0 5B DE AE     ; A0 ;
AC A3 A5 B7 A9 A7 B6 BC BD BE DD A8 AF 5D B4 D7     ; B0 ;
7B 41 42 43 44 45 46 47 48 49 AD F4 F6 F2 F3 F5     ; C0 ;
7D 4A 4B 4C 4D 4E 4F 50 51 52 B9 FB FC F9 FA FF     ; D0 ;
5C F7 53 54 55 56 57 58 59 5A B2 D4 D6 D2 D3 D5     ; E0 ;
30 31 32 33 34 35 36 37 38 39 B3 DB DC D9 DA 9F     ; F0 ;
```

These sample codepages use the same default translations that are used by the SAS/C library. Note that the library's translation is not affected by the alternative codepage feature described here. The library's translation is controlled by the tables found in ***prefix*.SOURCE(L$USKCS)**. The alternative codepages will affect only the compiler's translation of ASCII to EBCDIC characters.

The **set** command can be used to set the value of the **_SASC_CODEPAGE_PATH** environment variable. For example, the following command can be used to specify the fully qualified path to the directory containing the alternative codepages under Windows:

```
set _SASC_CODEPAGE_PATH=C:\codepages
```

This command will set the **`_SASC_CODEPAGE_PATH`** environment variable to the **`\codepages`** directory on your C drive.

*Note:* Under UNIX, you have to export the **`_SASC_CODEPAGE_PATH`** as well as set it. △

If the alternative codepages are properly formatted and found in the location specified by the **`_SASC_CODEPAGE_PATH`** environment variable, the following notes will be displayed when you compile:

```
NOTE: Environment variable "_SASC_CODEPAGE_PATH" found.
NOTE: Alternative codepages will be loaded from the "C:\codepages" directory
```

If errors are detected while processing the codepages, cautions will be displayed.

## Release 7.50 Changes to the SAS/C Cross-Platform Compiler User's Guide

### Updates to Compiler Options

In Chapter 3, "Compiling C and C++ Programs," in the section titled "Option Summary," in the table labeled "Compiler Options," make the following changes to the indicated entries:

- Add the following text to the description of the **`-Knoinline`** option:

  When used with the C++ translator this option also disables translator inlining.
- Delete the text "For C compilations:" from the descriptions of the following options:

  **`-Kexclude`**

  **`-Khlist`**

  **`-Kilist`**

  **`-Klisting`**

  **`-Kmaclist`**

  **`-Kpagesize`**

  **`-Ktrigraphs`**

  **`-Kxref`**

  **`-Ksource`**

Update the following entries in the list of options in the section titled "Option Descriptions:"

- Add the following text to the end of the first paragraph in the description of **`-Knoinline`**:

  When used with the C++ translator, this option also disables translator inlining. Translator inlining is always disabled with the **`-g`** or **`-Kdebug`** options.
- Delete the text "for C compilations only" from the end of the descriptions of the **`-Kpagesize`**, **`-Kstrict`**, and **`-Ktrigraphs`** options.

### Updates to the ar370 Archive Utility

In Chapter 7, "ar370 Archive Utility," add the following updates to the specified sections.

## Update to the Introduction

In the “Introduction,” replace the second paragraph with the following text:

An **ar370** archive is organized as a collection of members, identified by a member name that resembles a filename. Member names are limited to 18 characters total length. Member names are not significant as far as resolving external references; however, the member name is important for maintaining the archive. For each object file contained in an **ar370** archive, the **ar370** utility records the names of external symbols defined or referenced in the member (including external objects with extended names). This allows **cool** to find the member that defines a particular symbol. No connection is required between an **ar370** member name and the external symbol names defined by the member.

## Update to Optional Modifier Characters

In the section titled “Optional Modifier Characters,” add the **-y** command modifier to the table.

**Table 3.3**

| Modifier Characters | Description |
|---|---|
| **y** | Yes: List the mangled name along with the demangled name. The **y** modifier is meaningful only when used with the **e** (**enumerate**) optional modifier. |

## Update to Combinations of Command and Modifiers

In the section titled, “Combinations of Command and Modifiers,” update the table labeled “Command and Command Modifier Combinations” with the **-y** command modifier as indicated below:

**Table 3.4** Command and Command Modifier Combinations

| Command | Accepted Modifiers and Commands |
|---|---|
| **d** | **e, f, j, q, t, v, y** |
| **m** | **e, f, j, q, t, v, y** and **a \| b** |
| **r** | **e, f, j, q, t, v, y** and **a \| b** |
| **t** | **d, e, f, j, m, r, v, x, y** |
| **x** | **e, f, j, t, v, y** |

## Update to Optional Modifier Characters

In the section titled “Optional Modifier Characters,” add the following entry to the list of characters:

| | |
|---|---|
| **w** | Used in conjunction with the **replace** (**r**) command to allow truncation of member names at 18 characters. |

## Update to the -Aenexit and -Aenexitdata COOL Options

The following text and example code should be added to the information for the **enexit** and **enexitdata** options in Chapter 6, "Prelinking C and C++ Programs," under the section titled, "Cool Options."

For Release 7.50, support for the **-Aenexit** and **-Aenexitdata** options has been enhanced to allow you to use a DLL to create external symbols for extended names when running under Windows.

*Note:* The way COOL selects external symbols for extended names is explained in Appendix 7, "Extended Names," of the *SAS/C Compiler and Library User's Guide*. △

The following **userexitdll.cxx** and **StdAfx.cxx** source files provide a shell for a DLL that functions as the user exit for the **-Aenexit** option. You can use this example source to develop a DLL that generates external symbols that meet the specific needs of your build system. The DLL entry point **clkexit** is defined here in the **userexitdll.cxx** file.

**Example Code 3.3** userexitdll.cxx

```
#include <string.h>
#include <iostream.h>
#include "stdafx.h"
#include "userexitdll.h"

BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
      )
{
    switch (ul_reason_for_call)
 {
  case DLL_PROCESS_ATTACH:
  case DLL_THREAD_ATTACH:
  case DLL_THREAD_DETACH:
  case DLL_PROCESS_DETACH:
   break;
    }
    return TRUE;
}

extern "C" {

   USEREXITDLL_API int clkexit(const char UserData[8],
                const unsigned char *ExtendedName,
                int ExtendedNameLength, int FunctionFlag,
                int OldId, unsigned int *NewId)
  {
     int retcode = 0;

     /* Replace code from here ... */
     if (!strcomp(Userdata, "TESTDATA"))
     *NewId = OldIs +1;
     else
    *NewId = OldId;
     /* ... to here. */
```

```
        return(retcode);
    }
}
```

The **ifdef** block in **userexitdll.h** is the standard way of creating macros that make exporting from a DLL simpler. All files within this DLL are compiled with the **USEREXITDLL_EXPORTS** symbol defined on the command line. This symbol should not be defined on any project that uses this DLL. Any other project whose source files include this file see **USEREXITDLL_API** functions as being imported from a DLL, whereas this DLL sees symbols defined with this macro as being exported.

**Example Code 3.4** userexitdll.h

```
// userexitdll.h
//
#ifdef USEREXITDLL_EXPORTS
#define USEREXITDLL_API __declspec(dllexport)
#else
#define USEREXITDLL_API __declspec(dllimport)
#endif

extern "C" {
USEREXITDLL_API int clkexit(const char UserData[8],
                const unsigned char *Name, int NameLength,
                int FunctionFlag, int OldId, unsigned int *NewId);
}
```

**stdafx.cxx** is the source file that includes just the standard include files.

**Example Code 3.5** stdafx.cpp

```
// userexitdll.pch will be the pre-compiled header
// stdafx.obj will contain the pre-compiled type information

#include "stdafx.h"

// TODO: reference any additional headers you need in STDAFX.H
// and not in this file
```

**stdafx.h** is the include file for standard system include files or project specific include files that are used frequently, but are changed infrequently.

**Example Code 3.6** stdafx.h

```
#if !defined(AFX_STDAFX_H__453C6E62_806B_11D5_87B6_00C04F38FCF0__INCLUDED_)
#define AFX_STDAFX_H__453C6E62_806B_11D5_87B6_00C04F38FCF0__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000


// Insert your headers here
#define WIN32_LEAN_AND_MEAN  // Exclude rarely-used stuff from Windows headers

#include <windows.h>
// TODO: reference additional headers your program requires here
```

```
//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately
   before the previous line.

#endif // !defined(AFX_STDAFX_H__453C6E62_806B_11D5_87B6_00C04F38FCF0__INCLUDED_)
```

You should modify the **`*NewId = OldId`** assignment statement in **`userexitdll.cpp`** to assign the new symbols as required for your application. The example DLL will function as written, but it merely assigns the old value to the new value.

The function parameters for the user exit are described in Appendix 7, "Extended Names," of the *SAS/C Compiler and Library User's Guide*.

To compile the DLL you should create a Win32 Dynamic-Link Library Project with Microsoft Visual C++. You will need to define the **`USEREXITDLL_EXPORTS`** symbol in the preprocessor definitions for the Project Settings of the compilation for the DLL.

The syntax for the **`-Aenexit`** and **`-Aenexitdata`** options is as follows:

```
-Aenexit=dllname

-Aenexitdata=userdata
```

For example:

```
-Aenexit=C:\bin\userexit.dll
```

tells cool to use the **`userexit.dll`** located at **`C:\bin`** to process the external symbols for extended names.

And

```
-Aenexitdata=PKGA
```

passes the string **`PKGA`** to the **`userexit.dll`** as the UserData parameter.

Refer to the *SAS/C Compiler and Library User's Guide* and the *SAS/C Cross-Platform Compiler and C++ Development System User's Guide* for more information about the **`-Aenexit`** and **`-Aenexitdata`** options and extended names processing.

CHAPTER

4

# SAS/C C++ Translator Changes in Release 7.50

## Introduction

This chapter provides a complete description of the changes and enhancements to the *SAS/C C++ Development System User's Guide* for Release 7.50.

## Release 7.50 Changes and Enhancements to the C++ Translator

The following updates apply to *SAS/C C++ Development System User's Guide*.

## Updates to Introduction to the SAS/C C++ Development System

The following updates apply to Chapter 1, “Introduction to the SAS/C C++ Development System.”

### Updates to C++ Language Definition

Make the following changes to the section titled “C++ Language Definition:”

- Delete the following items:
  - Number 8.3.6
  - Number 10.3
  - All the items with numbers beginning with 14.5
  - All the items with numbers beginning with 14.6 except for item 14.6.2
  - All the items with numbers beginning with 14.7.
- Add the following items:

  9.6
  By default bitfields with widths greater than the field unit size are handled as SAS/C non-integer bitfields instead of treating the extra bits as padding. Also widths greater than 32 are not allowed. Use the **`NOBITFIELD`** option to handle bitfield widths as specified in the C++ standard.

  15.4
  Exception specifications are not allowed on parameter or return types.
- Replace the text of item 7.3.1.2 with the following text:

  Friend names are injected into the enclosing namespace unless the **`NOFRIENDINJECT`** option is specified.

### Updates to Improved Conformance with the C++ Standard

Make the following changes to the section titled “Improved Conformance with the C++ Standard.”

- Add the following items:

  9.6
  The **`NOBITFIELD`** option can be used to enforce C++ standard rules about bitfield widths.

  10.3
  Virtual function overrides can have covariant return types.

  14.5
  Nested templates are now fully supported. Members, including templates, of class templates can be defined outside the body of the enclosing template. Template member friend declarations are allowed.

  14.5.4
  Partial specializations of class templates can be declared and used.

  14.5.5
  Template conversion functions are now supported.

  14.5.5.2
  Template partial ordering rules are applied to function overload resolution.
- Replace the note under item 3.4.2 with the following note:

*Note:* The C++ translator injects the names from friend declarations into the containing namespace when defining a class unless the **NOFRIENDINJECT** option is specified. Friend declarations are always visible to name lookups that are argument dependent when the arguments use matching namespaces and classes. △

## Updates to Environmental Elements

Make the following changes to the section titled “Environmental Elements:” Replace the table labeled “Integral Type Sizes” with the following table:

**Table 4.1** Integral Type Sizes

| Type | Length in Bytes | Range |
|---|---|---|
| **bool** | 1 | false, true |
| **char** | 1 | 0 to 255 (EBCDIC character set) |
| **signed char** | 1 | -128 to 127 |
| **short** | 2 | -32768 to 32767 |
| **unsigned short** | 2 | 0 to 65535 |
| **int** | 4 | -2147483648 to 2147483647 |
| **unsigned int** | 4 | 0 to 4294967295 |
| **long (NOHUGEPTRS)** | 4 | -2147483648 to 2147483647 |
| **unsigned long (NOHUGEPTRS)** | 4 | 0 to 4294967295 |
| **long (HUGEPTRS)** | 8 | -9223372038854775808 to 9223372036854775807 |
| **unsigned long (HUGEPTRS)** | 8 | 0 to 18446744073709551615 |
| **long long** | 8 | -9223372038854775808 to 9223372036854775807 |
| **unsigned long long** | 8 | 0 to 18446744073709551615 |
| **wchar_t** | 2 | 0 to 65535 |

Replace the table labeled “Float and Double Type Sizes” with the following table:

**Table 4.2** Float and Double Type Sizes

| Type | Length in Bytes | Range |
|---|---|---|
| **float** | 4 | +/-5.4E-70 to +/-7.2E75 (__hexfmt)<br>+/-1.4E-45 to +/-3.4E38 (__binfmt) |
| **double** | 8 | +/-5.4E-70 to +/-7.2E75 (__hexfmt)<br>+/-4.9E-324 to +/-1.8E308 (__binfmt) |
| **long double** | 8 | +/-5.4E-70 to +/-7.2E75 (__hexfmt)<br>+/-4.9E-324 to +/-1.8E308 (__binfmt) |

## Updates to Predefined Constants

In the section titled “Language Elements,” make the following changes to the section titled “Predefined Constants:”

- Delete the entries for **`c_plusplus`** from the list of predefined constants and the list of macro descriptions.
- Before the note at the end of the section add the following text:

  The translator also defines the preprocessor option symbols described in the *SAS/C Compiler and Library User's Guide* in Chapter 6, "Compiler Options," in the section titled "Preprocessor Symbols."

## Updates to Language Extensions

Make the following changes in the section titled "Language Extensions:"

- Replace the first sentence, which reads:
  This section describes SAS/C extensions to the language described in The C++ Programming Language.
  With the following text:
  This section describes SAS/C extensions to the language described in the ISO C++ standard.
- In the section titled "long long Type Support" delete the last sentence, which reads, "Also, the preprocessor does not yet support **`long long`** values in **`#if`** expressions."
- In the section titled "SAS/C extension keywords," add the following items to the list of keywords:

```
__binfmt     __huge          __norent
__far        __multireturn   __rent
__hexfmt     __near          __setjmp
```

- In the section titled "Floating-point constants in hexadecimal," add the following note at the end of the existing text:

  *Note:* The translator also supports the C99 hexadecimal floating point format. △
- In the section titled "signed long long Support," replace the existing text with text that reads:

  Decimal constants like 4000000000 that are too large to represent as **`signed long`**, and that do not have a **`U`** or **`u`** suffix, are treated as **`unsigned long`**. This is the same treatment that these large decimal constants would receive under the C89 rule. Octal, hexadecimal, and explicitly unsigned constants are not affected.

## Additions to Language Extensions

Add the following sections to the "Language Extensions" section after the subsection titled "C++ iostream Library Constructors:"

Pragma options directive

The tranlator supports the **`#pragma`** options **`copts(...)`** directive, in a manner comparable to the C compiler. The following is a sample directive:

```
#pragma options copts(sname(ABC123))
```

Most of the options that the C compiler allows to be changed through **`#pragma`** options can also be specified for C++. Options like **`PPIX`** that are not supported for C++ compilations are also not supported through **`#pragma`** options in C++.

C99-style floating point constants

The translator supports the C99 hexadecimal floating-point constant format. This provides an alternative way to specify exact values for constants that have floating-point type. The syntax is:

```
0x<hexdecimal-digits>.<hexdecimal digits>P<binary exponent><suffixes>
```

For more details, see the section titled "Specifying floating-point constants in hexadecimal" in Chapter 2, "Source Code Conventions," of the *SAS/C Compiler and Library User's Guide* .

Explicit __binfmt and __hexfmt constant qualifiers
The translator allows a suffix of **B** or **H** to explicit specify that a floating point constant is in **__binfmt** or **__hexfmt** respectively. These suffixes can be used with all the floating point constant formats and follow any **F** or **L** suffixes. Note suffixes must be preceded by a period when they are used with the SAS/C hexadecimal format. For example **0.x4110.FH** is a valid **__hexfmt** constant.

## Updates to Implementation-Defined Behavior

Make the following changes to the section titled "C++ specific behaviors" In the section titled "Implementation-Defined Behavior:"

- Replace the text of the first list item with text that reads:

  There are three accepted linkage strings: **C**, **C++**, and **OS**. **C** linkage for functions means that type information is not encoded in the function's identifier. **C** linkage does not affect the linkage for non-functions. **OS** is like **C** except that **__ibmos** is implicitly applied to non-member function declarators. Remember that the linkage for main is always **C**.
- Add the following items to the list:
  - The dynamic type of the object returned by a **typeid** expression is **const std::type_info**.
  - The representation of member pointers is unchanged by **reinterpret_cast**. Otherwise for valid casts the **reinterpret_cast** operator has semantics matching the cast operator in C code. The **reinterpret_cast** operator does not perform adjustments based on class inheritance.
  - The underlying type of an enumeration is **signed int** if all the enumeration constant values fit. Otherwise the underlying type is **unsigned int**.
  - Plain **char** bitfields are unsigned. A **bool** bitfield has the same layout as an unsigned **char** field. A **wchar_t** bitfield has the same layout as an **unsigned short** field. An **enumeration** bitfield has the same layout as a bitfield of the underlying type of the enumeration except that the field will be unsigned if all declared enumeration constant values are non-negative.
  - When binding a class **rvalue** to a compatible **const** reference, the reference is bound directly to the **rvalue** object if the class has a nontrivial destructor or nontrivial copy constructor. Otherwise the **rvalue** is copied to a temporary and the reference is bound to the temporary. The temporary copy, however, can be elided by the rules in section 12.8 of the C++ standard.
  - The maximum depth for recursive template instantiations is 50.
  - When an exception is unhandled, the stack is unwound before the call to **std::terminate()**.
  - The C++ standard library is safe for multi-tasking. However the routines are not guaranteed to be reentrant for interrupt handling purposes.

## Update to the Anachronisms Section

Add the following item to the list in the section titled "Anachronisms:"

- For compatibility, the names for friend class and function declarations are made visible in an enclosing scope of the class containing the friend declaration. This *name injection* was eliminated in the C++ standard and can be disabled with the **`NOFRIENDINJECT`** option.

## Updates to Translator Options

The following updates apply to Chapter 3, “Translator Options.”

### Updates to the Translator Options Table

In Chapter 3, “Translator Options,” in the section titled, “Option Summary,” replace the entries for **`asciiout`**, **`INLIne`**, and **`tronly`** in the table labeled, “Translator Options,” with the following entries.

**Table 4.3** Translator Options

| Option Name | Default | USS | Environment | Affects Process |
|---|---|---|---|---|
| **`ASCiiout`** | **`NOASCiiout`** | **`-Kasciiout`** | all | T,C |
| **`INLIne`** | see description | **`-Kinline`** | all | G |
| **`TROnly`** | see description | **`-Ktronly`** | all | T |

Add the following entries to the table labeled, “Translator Options.”

**Table 4.4** Translator Options

| Option Name | Default | USS | Environment | Affects Process |
|---|---|---|---|---|
| **`ARChlevel`** | see description | **`-Karchlevel=let`** | all | T,C |
| **`ARMode`** | **`NOARMode`** | **`-Karmode`** | all | T,C |
| **`BFp`** | **`NOBFp`** | **`-Kbfp`** | all | T,C |
| **`FRIendinject`** | **`NOFRIendinject`** | **`-Kfriendinject`** | all | T |
| **`HUgeptrs`** | **`NOHUgeptrs`** | **`-Khugeptrs`** | all | T,C |
| **`OPTIOns`** | **`OPTIOns`** | **`-Koptions`** | all | L |
| **`TErm`** | see description | **`-Kterm`** | all | M |

### Updates to the Translator Options Descriptions

Make the following changes to the list of option descriptions in the section titled “Option Descriptions:”

- Update the entry for the **`tronly`** with the following information:
  - Change the **`tronly`** list-entry term to the following:

    **`tronly`** (**`-Ktronly`**) under USS
  - Add the following information to the **`tronly`** description:

    Under USS, the following command sends the **`tronly`** output to **`hello..c`**:

    ```
    sascc370 -v -c -Ktronly hello.cpp
    ```

Note the two periods (..) between the file's basename and extension. This is to prevent accidental overwriting of a **hello.c** source file. Also note that you have to specify the **-c** option to prevent the **binder** from being called.

The following example specifies an output filename for the **-Ktronly** option:

```
sascc370 -v -c -Ktronly=test.c hello.cpp
```

- Replace the last sentence of the **bitfield** option description that reads:

  This option cannot be negated.

  with text that reads:

  By default the SAS/C rules are used for the interpretation of bitfield widths. The **NOBITFIELD** option can be used to enforce the bitfield rules from the C++ standard instead of applying the SAS/C bitfield rules. With **NOBITFIELD** option, bitfield widths that are larger than the field unit type have a different layout. For example an **unsigned char bitfield** that is 12 bits wide has 12 data bits by the SAS/C rules. With the **NOBITFIELD** option there are 8 data bits, the width of an unsigned char, and 4 padding bits.
- Replace the description text of the **inline** option with text that reads:

  controls inlining in the C++ translator as well as the optimizer. The translator performs inlining by default unless the **DEBUG** option is used. The use of the **DEBUG** option forces **NOINLINE**. The translator can inline functions that require exception handling support, unlike **GO**. Translator inlining can be explicitly disabled with the **NOINLINE** option.
- Add the following entries to the list of option descriptions:

  **archlevel** (**-Karchlevel=let** under USS)
  allows you to request code generation for a specific level of the 390 architecture. See the option description in the *SAS/C Compiler and Library User's Guide* Guide for more information. The default level is **ARCHLEVEL(D)** if the **HUGEPTRS** option is specified and **ARCHLEVEL(C)** if the **BFP** option is specified. Otherwise the default is unspecfied and no new architecture features are assumed.

  **armode** (**-Karmode** under USS
  specifies that code that uses the ESA access registers may be generated. This option is required to compile code that uses far pointers. See "Optimization and Far Pointers" in the *SAS/C Compiler and Library User's Guide*for more information on far pointers and access register mode.

  **bfp** (**-Kbfp** under USS)
  specifies that the default floating point format is binary (IEEE). Note that the **BFP** option implies the **ARCHLEVEL(C)** option.

  **friendinject** (**-Kfriendinject** under USS)
  specifies that the translator should make the names of of friend class and function names visible in the enclosing non-class scope of the class containing the declaration. This is called *friend name injection*. The C++ standard requires that friend names not be injected. However older code may require name injection. The **FRIENDINJECT** option is the default for compatility with older code; however, this default may change in a future release.

  **hugeptrs** (**-Khugeptrs** under USS)
  specifies that the object code is intended to execute in 64-bit addressing mode. When **hugeptrs** is specified, the default pointer type is **__huge**, and the size of **signed** and **unsigned long** data is 8 bytes. Note that the **hugeptrs** option implies the **ARCHLEVEL(D)** option.

**options** (**-Koptions** under USS)
generates an options listing. The options listing contains all options in effect for the compilation.

**term** (**-Kterm** under USS)
directs diagnostic messages to **stderr**. The messages appear in addition to the messages in the listing file, if any.

By default the **term** option is enabled only when no listing is created.

## Updates to Standard Libraries

The following updates apply to Chapter 4, "Standard Libraries."

### Updates to Header Files

- Update "The <exception> header file" with the following changes:
  - In the section titled "class std::exception," under "VIRTUAL MEMBER FUNCTIONS," change the sentence that reads:

    Unless overridden in a derived class this string is **std::exception** or **derived class**.

    to the following:

    Unless overridden in a derived class this string is "**std::exception** or **derived class**".

    Also change the prototype of **xtrace** under "STATIC MEMBER FUNCTIONS (SAS/C EXTENSION)" to the following:

    ```
    static void xtrace( __remote void (*writer)(const __near char *line) = 0 );
    ```

  - In the section titled "class std::bad_exception," under "VIRTUAL MEMBER FUNCTIONS," change the text that reads:

    Unless overridden in a derived class, this string is as follows:

    ```
    std::bad_exception
    ```

    or

    ```
    derived class
    ```

    to the following:

    Unless overridden in a derived class, this function returns the string "**std::bad_exception** or **derived class**".

- Update "The <new> header file" with the following changes:

  In the section titled "class std::bad_alloc," under "VIRTUAL MEMBER FUNCTIONS," change the text that reads:

  Unless overridden in a derived class this string is **std::bad_alloc** or **derived class**.

  to the following:

  Unless overridden in a derived class this string is "**std::bad_alloc** or **derived class**".

- Update "The <typeinfo> header file" with the following changes:
  - In the section titled "class std::type_info," under "NONVIRTUAL MEMBER FUNCTIONS," change the second sentence of the description of **const char* name() const;** to read:

    The function calls a library allocator to allocate working storage.

- In the section titled “class std::bad_cast,” under “VIRTUAL MEMBER FUNCTIONS,” change the text that reads:

  Unless overridden in a derived class this string is **`std::bad_cast`** or **`derived class`**.

  to the following:

  Unless overridden in a derived class this string is “**`std::bad_cast`** or **`derived class`**”.
- In the section titled “class std::bad_typeid,” under “VIRTUAL MEMBER FUNCTIONS,” change the text that reads:

  Unless overridden in a derived class this string is **`std::bad_typeid`** or **`derived class`**.

  to the following:

  Unless overridden in a derived class this string is “**`std::bad_typeid`** or **`derived class`**”.
- In the section titled “class std::__non_rtti,” under “VIRTUAL MEMBER FUNCTIONS,” change the text that reads:

  Unless overridden in a derived class this string is **`std::__non_rtti`** or **`derived class`**.

  to the following:

  Unless overridden in a derived class this string is “**`std::__non_rtti`** or **`derived class`**”.

## Update to C++ Complex Library

In the section titled “C++ Complex Library,” add the following note after the first paragraph.

*Note:* This library performs all computations using **`__hexfmt`** floating-point and expects its operands to be in this format. △

## Updates to I/O Class Descriptions

Update “I/O Class Descriptions” with the following changes:

- In the section titled “class bsamstream, ibsamstream, and obsamstream,” in the “DESCRIPTION,” add the following note after the last paragraph:

  *Note:* These classes are not supported with the **`HUGEPTRS`** option. △
- In the section titled “class fstream, ifstream, and ofstream,” replace the SYNOPSIS with the following code:

```
#include <fstream.h>
class ifstream : public istream
{
public:
   ifstream();
   ifstream(const char __near *name,
            int mode = ios::in,
            const char __near *amparms = "",
            const char __near *am = "");
   virtual ~ifstream();
   void open(const char __near *name,
             int mode = ios::in,
             const char __near *amparms = "",
             const char __near *am = "");
   void close();
   void setbuf(char *p, int len);
   filebuf* rdbuf();
};
class ofstream : public ostream
{
public:
   ofstream();
   ofstream(const char __near *name,
            int mode = ios::out,
            const char __near *amparms = "",
            const char __near *am = "");
   virtual ~ofstream();
   void open(const char __near *name,
             int mode = ios::out,
             const char __near *amparms = "",
             const char __near *am = "");
   void close();
   void setbuf(char *p, int len);
   filebuf* rdbuf();
};
class fstream : public iostream
{
public:
   fstream();
   fstream(const char __near *name, int mode,
           const char __near *amparms = "",
           const char __near *am = "");
   virtual ~fstream();
   void open(const char __near *name, int mode,
             const char __near *amparms = "",
             const char __near *am = "");
   void close();
   void setbuf(char *p, int len);
   filebuf* rdbuf();
};
```

In CONSTRUCTORS, replace the second group of prototypes for constructors with arguments with the following code:

```
ifstream::ifstream(const char __near *name,
int mode = ios::in,
const char __near *amparms = " ",
const char __near *am = " ")

ofstream::ofstream(const char __near *name,
int mode = ios::out,
const char __near *amparms = "",
const char __near *am = "")

fstream::fstream(const char __near *name,
int mode, const char __near *amparms = "",
const char __near *am = "")
```

In MEMBER FUNCTIONS replace the group of prototypes for open functions with the following code:

```
void ifstream::open(const char __near *name,
int mode = ios::in,
const char __near *amparms = " ",
const char __near *am = " ")

void ofstream::open(const char __near *name,
int mode = ios::out,
const char __near *amparms = " ",
const char __near *am = " ")

void fstream::open(const char __near *name,
int mode,
const char __near *amparms = "",
const char __near *am = "")
```

- Make the following changes to the section titled “class ios.”
  - Update the SYNOPSIS with the following changes.
    - After the line of code that reads **`#include <iostream.h>`**, add the following two lines of code:

```
typedef unsigned long _ulong;  // for NOHUGEPTRS compilations
typedef unsigned int  _ulong;  // for HUGEPTRS compilations
```

  - Change the entry for **`static unsigned long ios::bitalloc()`** to

    **`static _ulong ios::bitalloc()`**
    returns an **`unsigned long`** (**`unsigned int`** with the **`HUGEPTRS`** option) with a single, previously unallocated, bit set. This allows you to create additional format flags. This function returns 0 if there are no more bits available. Once the bit is allocated, you can set it and clear it using the **`flags()`** , **`setf()`**, and **`unsetf()`** functions.

- Make the following changes to the section titled “enum format_state.”
  - Replace the SYNOPSIS with the following code:

```
#include <iostream.h>
typedef unsigned long _ulong;  // for NOHUGEPTRS compilations
typedef unsigned int  _ulong;  // for HUGEPTRS compilations

class ios
{
public:

/* See the class ios, enum io_state,
   enum open_mode, and enum seek_dir
   descriptions for more definitions. */

   enum {skipws,

         left,
         right,
         internal,

         dec,
         oct,
         hex,

         showbase,
         showpoint,
         uppercase,
         showpos,

         scientific,
         fixed,

         unitbuf,
         stdio
        };
   static const _ulong basefield;
   static const _ulong adjustfield;
   static const _ulong floatfield;

   _ulong flags();
   _ulong flags(_ulong f);

   _ulong setf(_ulong mask);
   _ulong setf(_ulong setbits,
               _ulong mask);
   _ulong unsetf(_ulong mask);
};
```

- Replace the FORMATTING FUNCTIONS section with the following text:

  The following functions can be used to switch format flags on and off:

  **_ulong ios::flags()**
  returns an **unsigned long** (**unsigned int** with **HUGEPTRS**) representing the current format flags.

  **_ulong ios::flags (_ulong f)**
  turns on all of the format flags specified by **f** and returns an **unsigned long** (**unsigned int** with **HUGEPTRS**) representing the previous flag values.

  **_ulong ios::setf (_ulong mask)**
  turns on only those format flags that are set in **mask** and returns an **unsigned long** (**unsigned int** with **HUGEPTRS**) representing the previous values of those flags. You can accomplish the same task by using the parameterized manipulator, **setiosflags**.

  **_ulong ios::setf (_ulong setbits, _ulong mask)**
  turns on or off the flags marked by mask according to the corresponding values specified by **setbits** and returns an **unsigned long** (**unsigned int** with **HUGEPTRS**) representing the previous values of the bits specified by mask. The EXAMPLES section provides an example of using this function.

  Using **setf(0, mask)** clears all the bits specified by **field**. You can accomplish the same task by using the parameterized manipulator **resetiosflags**.

  **_ulong ios::unsetf (_ulong mask)**
  clears the format flags specified by **mask** and returns an **unsigned long** (**unsigned int** with **HUGEPTRS**) representing the previous flag values.

- Make the following changes to the section titled “class istream.”

- Update the SYNOPSIS with the following changes:
  - After the line of code that reads **`istream& operator >>(unsigned long& l);`**, add the following two lines of code:

    ```
    istream& operator >>(long long& ll);
    istream& operator >>(unsigned long long& ll);
    ```

  - Change the three declarations that read:

    ```
    istream& operator >>(float& f);
    istream& operator >>(double& d);
    istream& operator >>(long double& ld);
    ```

    to

    ```
    istream& operator >>(__hexfmt float& f);
    istream& operator >>(__hexfmt double& d);
    istream& operator >>(__hexfmt long double& ld);

    istream& operator >>(__binfmt float& f);        // if ARCHLEVEL>=C
    istream& operator >>(__binfmt double& d);       // if ARCHLEVEL>=C
    istream& operator >>(__binfmt long double& ld); // if ARCHLEVEL>=C
    ```

- Update the FORMATTED INPUT FUNCTIONS with the following changes:
  - After the line of code that reads **`istream& istream::operator >>(unsigned long& l)`**, add the following two lines of code:

    ```
    stream& istream::operator >>(long long& l)
    istream& istream::operator >>(unsigned long long& l)
    ```

  - Change the following lines of code

    ```
    istream& istream::operator >>(float& f)
    istream& istream::operator >>(double& d)
    istream& istream::operator >>(long double& ld)
    ```

    to

    ```
    istream& istream::operator >>(__hexfmt float& f)
    istream& istream::operator >>(__hexfmt double& d)
    istream& istream::operator >>(__hexfmt long double& ld)
    istream& istream::operator >>(__binfmt float& f)
    istream& istream::operator >>(__binfmt double& d)
    istream& istream::operator >>(__binfmt long double& ld)
    ```

- Add the following note to the end of the "class istream" DESCRIPTION section:

  *Note:* The **`__binfmt`** input functions are only available if the **`ARCHLEVEL`** option is **`C`** or greater. △

- Make the following changes to the section titled "class ostream."
  - Update the SYNOPSIS with the following changes:
    - After the line of code that reads **`ostream& operator <<(unsigned long l);`**, add the following two lines of code:

      ```
      ostream& operator <<(long long ll);
      ostream& operator <<(unsigned long long ll);
      ```

    - Change the lines of code that read:

```
ostream& operator <<(float f);
ostream& operator <<(double d);
```

to

```
ostream& operator <<(__hexfmt float f);
ostream& operator <<(__hexfmt double d);
ostream& operator <<(__hexfmt long double d);
ostream& operator <<(__binfmt float f);       // if ARCHLEVEL>=C
ostream& operator <<(__binfmt double d);      // if ARCHLEVEL>=C
ostream& operator <<(__binfmt long double d); // if ARCHLEVEL>=C
```

- Update the FORMATTED OUTPUT FUNCTIONS section with the following changes:
  - After the lines of code that read:

    ```
    ostream& ostream::operator <<(unsigned
    long l)
    ```

    add the following two lines of code:

    ```
    ostream& ostream::operator <<(long long ll)
    ostream& ostream::operator <<(unsigned long long ll)
    ```

  - Change the lines of code that read:

    ```
    ostream& ostream::operator <<(float f)

    ostream& ostream::operator <<
    (double d)
    ```

    to

    ```
    ostream& ostream::operator <<(__hexfmt float f)

    ostream& ostream::operator <<(__hexfmt double d)

    ostream& ostream::operator <<(__hexfmt long double d)

    ostream& ostream::operator <<(__binfmt float f)

    ostream& ostream::operator <<(__binfmt double d)

    ostream& ostream::operator <<(__binfmt long double d)
    ```

  - Add the following note to the end of the "class ostream" DESCRIPTION section:

    *Note:* The **`__binfmt`** output functions are only available if the **`ARCHLEVEL`** option is **`C`** or greater. △

- Make the following changes to the section titled "class stdiostream."
  - Replace the SYNOPSIS with the following code:

```
#include <stdiostream.h>
class stdiostream : public iostream
{
public:
   stdiostream(__near FILE *file);

   stdiobuf* rdbuf();
```

```
};
```

- ☐ Replace the prototype in the CONSTRUCTORS section that reads:

  ```
  stdiostream::stdiostream(FILE *file)
  ```

  with the following prototype:

  ```
  stdiostream::stdiostream(__near FILE *file)
  ```

- ☐ Delete the entry for **`stdiofile()`** from the MEMBER FUNCTIONS section.
- ☐ Add the following note at the end of the DESCRIPTION:

  *Note:* Although **`streampos`** supports 64-bit integer offsets when the **`HUGEPTRS`** option, only offsets in the 32-bit range are supported by the I/O library. △

## Updates to Buffer Class Descriptions

Update "Buffer Class Descriptions" with the following changes:

- ☐ Add the following note between the two existing notes at the end of the introductory section of text:

  *Note:* None of the classes defined in **`<bsamstr.h>`** are supported by the **`HUGEPTRS`** option. △

- ☐ Make the following changes to the section titled "class filebuf."
  - ☐ Replace the prototype for **`open()`** in the SYNOPSIS with the following code:

    ```
    filebuf* open(const char __near *name,
                  int mode,
                  const char __near *amparms = "",
                  const char __near *am = "");
    ```

  - ☐ Replace the prototype for **`filebuf::open`** in the NONVIRTUAL MEMBER FUNCTIONS section with the following code:

    ```
    filebuf* filebuf::open(const char __near *name,
                           int mode,
                           const char __near *amparms = "",
                           const char __near *am = "")
    ```

  - ☐ Add the following note in the VIRTUAL MEMBER FUNCTIONS section after the description of **`virtual streambuf* filebuf::setbuf`**

    *Note:* In Release 7.50, the value of **`len`** must be less or equal to 2^32-1 bytes. △

- ☐ Make the following changes to the section titled "class stdiobuf."
  - ☐ Replace the SYNOPSIS with the following code:

    ```
    #include <stdiostream.h
    >
    class stdiobuf : public streambuf
    {
    public:
      stdiobuf(__near FILE *file);

      virtual ~stdiobuf( );

      int is_open( );
    ```

```
    __near FILE* stdiofile( );

    streampos seekoff(streamoff offset,
                      seek_dir place,
                      int mode =
                      ios::in|ios::out);
    streampos seekpos(streampos pos,
                      int mode =
                      ios::in|ios::out);
    virtual int sync();
  };
```

□ Change the CONSTRUCTOR prototype

```
stdiobuf::stdiobuf(FILE *file)
```

to

```
stdiobuf::stdiobuf(__near FILE *file)
```

□ Change the NONVIRTUAL MEMBER FUNCTIONS prototype

```
FILE* stdiofile()
```

to

```
__near FILE* stdiofile)
```

## Updates to Header Files, Classes, and Functions

The following updates apply to Appendix 2, "Header Files, Classes, and Functions."

### Update to Stream Classes

In the section titled "Stream Classes," in the table labeled "Header Files, Classes, and Functions for Streams," in the entry for **stdiostream.h**, delete the term **stdiofile()** from the Functions column.

## Updates to Templates

The following updates apply to Appendix 3, "Templates."

### Update to Template Parameters

Make the following changes to the section titled "Template Paremeters:"

□ Change the list item that reads:

data member pointers

to

data or function member pointers

□ Replace the text and example after the list with the following text and example:

Nontype template parameters can use previously declared template parameters in their declaration type. For example,

```
template <class T, T* someObject>
class C { };
```

```
int p;
C <int, &p> c;
template <int I, int (*pArray[I])> class C2; // ok
template <int I, int (*pBigArray[I*100])> // ok
class C3;
```

## Update to Template Arguments

In the section titled “Template Arguments,” change the last list item to read:

□ member pointer constants for nonstatic data or function members specified as:

```
&class_name::member
```

## Updates to Template Declarations and Definitions

Make the following changes In the section titled “Template Declarations and Definitions,” in the section titled “Declaration Rules.”

□ Remove the note that reads:

*Note:* Template friend declarations of template class members are not supported in this release of the SAS/C C++ Development System. △

□ In the paragraph that follows the note, change the sentence that reads:

However, friend template declarations may not be definitions.

to

However, friend template class declarations may not be definitions.

□ Remove the note that reads:

*Note:* Beginning with Release 6.50, template parameters may not have default arguments. △

## Updates to Function Templates

In the section titled “Function Templates,” in the section titled “Deducing Arguments,” add the following item at the end of the bulleted list:

□ **`derived to base conversions`**

When the parameter type is a template ID or pointer to a template ID then deduction will consider derived-to-base conversions to match the template ID.

## Updates to Interpreting C++ Demangled Names

The following updates apply to Appendix 5, “Interpreting C++ Demangled Names.”

In the section titled “Special Conventions Used in Demangled Names,” add the following entries to the table labeled “Keywords and Their Abbreviations.”

**Table 4.5** Keywords and Their Abbreviations

| Keyword | Abbreviation |
|---|---|
| `__binfmt` | `_B` |
| `__far` | `_F` |
| `__huge` | `_H` |

## New Appendix on ARMODE, HUGEPTRS, and Pointer Kinds

After Appendix 5, "Interpreting C++ Demangled Names," add a new appendix titled "ARMODE, HUGEPOINTERS, and Pointer Kinds" that contains the following information.

The C++ translator supports the **HUGEPTRS** and **ARMODE** options as well as the **__near**, **__far**, and **__huge** keywords in pointer and reference declarations. This appendix describes some of the special considerations for use of these SAS/C features with C++.

Pointer operations generally behave the same way with the C++ translator that they do with the C compiler. However the C++ translator is more restrictive about implicit pointer conversions in order to facilitate overloading and avoid unsafe conversions. Far and huge pointers cannot be converted to a different pointer kind without an explicit cast. As a special case pointers to strings and simple pointer expressions that can be determined at compile time to address auto, formal, static, or external objects can be implictly converted to near pointers.

References can be **__near**, **__far**, or **__huge**. By default object references are **__near** unless **HUGEPTRS** is specified, in which case they are **__huge**. Dereferencing a **__huge** reference requires the **HUGEPTRS** option. Dereferencing a **__far** reference requires **ARMODE**. With the **HUGEPTRS** option, only huge const references can bind to non-lvalues.

C++ specific pointer operations have their own restrictions. Conversion of a far pointer to a base class pointer requires the **ARMODE** option. Conversion of a huge pointer to a base class pointer requires the **HUGEPTRS** option. Applying **dynamic_cast** to far pointers or references requires the **ARMODE** option. Similarly, applying **dynamic_cast** to huge pointers or references requires the **HUGEPTRS** option.

Class object layout does not depend on the **HUGEPTRS** or **ARMODE** option except for differences due to user nonstatic data members. However a nonstatic member function always has a default sized **this** pointer. That is, in a **HUGEPTRS** compilation the **this** parameter will be a 64-bit huge pointer instead of a 32 bit near pointer. **ARMODE** does not affect the **this** pointer type. Virtual function calls to objects created in a different mode will cause unexpected results. Mixing **HUGEPTRS** and non-**HUGEPTRS** code is not generally recommended.

The **new** operator always returns a default sized pointer. The return type for an operator new function and the first argument for a delete function must be a default void pointer. The leftmost array size in an array allocation type can be a 64-bit quantity when **HUGEPTRS** is specified.

Enumeration constants and array sizes are limited to 32 bit values, except for the leftmost array size in a declarator for a new expression.

The **std::bad_exception** object thrown by the C++ library when a mismatched exception is encountered expects virtual function calls with 64-bit or 32-bit object pointers according to the mode (**HUGEPTRS** or **NOHUGEPTRS**) of the routine which included **std::bad_exception** in its exception specification.

The AT&T compatible streams and complex library supports **HUGEPTRS**. Note the **cin**, **cout**, **cerr**, and **clog** streams refer to different objects (with different layouts) in

64 and 32-bit modes. The Rogue Wave Standard C++ Library and Tools Library do not support **`HUGEPTRS`**.

CHAPTER 5

# SAS/C Debugger Changes for Release 7.50

## Introduction

This chapter provides a complete description of the changes and enhancements made to the SAS/C Debugger for Release 7.50.

## Incompatibility with Previous Releases of SAS/C

The debugging information produced by Release 7.50 of the SAS/C Compiler has been redesigned. Because of this change, Release 7.50 of the SAS/C Debugger is not compatible with debugging files generated by any releases of the SAS/C Compiler prior to Release 7.00.

## Release 7.50 Enhancements to the SAS/C Debugger

The following enhancements to the SAS/C Debugger have been implemented with Release 7.50:

- Namespace support
- IEEE floating-point support
- 64-bit addressing support.

### Support for Namespaces

A new debugger command, **namespc**, has been added for SAS/C Release 7.50. In Chapter, 14, "Command Directory," add the following entry to the list in the section titled "List of Commands:"

**n{amespc}** display namespaces encountered in the source code.

Add the following entry for **namespc** to the section that contains the descriptions of the commands:

## namespc

Display Namespaces

ABBREVIATION

**n{amespc}**

FORMATS

Format 1: namespc

Format 2: namespc full

Format 3: namespc NAMESPACE-NAME

Format 4: namespc NAMESPACE-NAME TYPE-INTEGER

DESCRIPTION

The **namespc** command produces a listing of all namespaces or selected namespaces. The listing includes a list of identifier, **class/struct/union** and function names within each namespace.

Format 1: produces only an abbreviated list of all namespaces without any details.

Format 2: produces a sorted listing of all namespaces with detailed contents of each listed namespace. These details include all entries recorded in each namespace scope such as Identifier/Variable Name, Function Name, and Class/Struct/Union Name.

Format 3: produces a listing of the defined namespace with detailed contents. These details include all entries recorded in this namespace scope such as Identifier/Variable Name, Function Name, and Class/Struct/Union Name.

The **NAMESPACE-NAME** argument represents a valid namespace name implemented within the source code that is being debugged.

Format 4: produces a listing of the defined namespace with the defined detailed contents. These details include all entries of the same type recorded in this namespace scope such as Identifier/Variable Name, Function Name, and Class/Struct/Union Name where Class/Struct/Union Name is of type 0, Function Name is of type 1, and Identifier/Variable Name is of type 2.

The **NAMESPACE-NAME** argument represents a valid namespace name implemented within the source code that is being debugged. The argument **TYPE-INTEGER** represents one of the following:

0 - Class/Struct/Union names found in the scope of the namespace

1 - Function names found in the scope of the namespace

2 - Identifier/variable names found in the scope of the namespace.

EXAMPLES

Format 1: **namespc** displays the following namespaces listing in the Log window:

```
Namespace(s)
=============
1 <<unnamed>>
2 moreStuff
3 myStuff
4 myStuff::nested
5 myStuff::nested::nestagain
6 std
```

Format 2: **namespc full** displays the following namespaces listing in the Log window:

```
Namespace(s)  Entries recorded in this namespace scope
============= ========================================
<<unnamed>>
       Class/Struct/Union XYZ
       Variable count
       Variable cpanon
Namespace(s)  Entries recorded in this namespace scope
============= ========================================
moreStuff
       Class/Struct/Union XYZ
       Function swapargs
Namespace(s)  Entries recorded in this namespace scope
============= ========================================
myStuff
       Class/Struct/Union XYZ
       Variable count
       Variable cpanon
       Function swapargs
       Function argswap
Namespace(s)  Entries recorded in this namespace scope
============= ========================================
myStuff::nested
       Class/Struct/Union XYZ
       Variable count
       Variable cpanon
       Function swapargs
Namespace(s)  Entries recorded in this namespace scope
============= ========================================
myStuff::nested::nestagain
         Class/Struct/Union XYZ
       Variable count
       Variable cpanon
       Function swapargs
Namespace(s)  Entries recorded in this namespace scope
============= ========================================
std
       Class/Struct/Union __siocb
       Function fopen
       Function fflush
       Variable __io
       Function main
```

Format 3: **namespc myStuff** displays the following namespace listing in the Log window:

```
Namespace(s)  Entries recorded in this namespace scope
============= ========================================
myStuff
       Class/Struct/Union XYZ
       Variable count
       Variable cpanon
       Function swapargs
       Function argswap
```

Format 4: **namespc myStuff 2** displays the following namespace listing in the Log window:

```
Namespace(s)  Entries recorded in this namespace scope
============ ========================================
myStuff
        Variable count
        Variable cpanon
```

Support has been added to several commands to accept namespace expressions, including **assign**, **break**, **browse**, **disable**, **drop**, **enable**, **goto**, **ignore**, **list**, **monitor**, **on**, **print**, **query**, **resume**, **runto**, **scope**, **trace**, **watch**, and **whatis**.

SEE ALSO

**scope** command in Chapter 14, "Command Directory."

## Support for IEEE Floating-Point

The 390 Architecture has been augmented in recent years by the addition of support for IEEE standard floating-point. Programs can now be written using the SAS/C Compiler and the SAS/C Debugger to exploit this support. Traditional mainframe floating-point continues to be supported.

Support has been added to several commands to accept IEEE floating-point expressions, including **assign**, **copy**, **dump**, **monitor**, **print**, **transfer**, **watch**, and **whatis**. With this support, all unary and binary C operators are available for use with IEEE floating-point expressions including the cast operator.

You can use the cast operator to convert from a hexadecimal floating-point type to a binary floating-point type using the **__binfmt** type-name modifier. Similarly, you can convert from a binary floating-point type to a hexadecimal floating-point type using the **__hexfmt** type-name modifier. You cannot mix binary and hexadecimal floating point operands in the same expression. You must use the cast operator to ensure that all floating-point operands are converted to the same format.

The following example illustrates how to convert from binary floating-point to hexadecimal floating-point using the print command:

```
print (__hexfmt float)binaryFloatVariable;
```

When using floating-point constants or displaying the floating-point values in the register window, the SAS/C Debugger will use the compilation options in effect for the module in the current scope. For example, if the module was compiled to use the binary floating-point by using either the SAS/C Compiler **BFP** or **ARCHLEVEL(C)** options, then the SAS/C Debugger will convert floating-point constants or register values into binary floating-point format before using or displaying them.

The SAS/C Debugger will check for exceptions that occur in IEEE floating-point expressions. The following exceptions will be checked for:

underflow

overflow

divide by zero.

## Support for 64-Bit Addressing

In Release 7.50, support has been added to the following debugger commands to accept 64-bit pointers and their use in expressions:

|  |  |  |
|---|---|---|
| **assign** | **monitor** | **storage** |
| **copy** | **print** | **watch** |
| **dump** | **return** | **whatis** |

The Dump, Print, Register, and Watch windows have also been updated to support 64-bit addressing:

*Note:* When you compile with the **HUGEPTRS** option, all pointers are 64-bit and longs become 8 bytes in size. The debugger will honor these rules when using constants in expressions in a module compiled with **HUGEPTRS**. △

The Register window displays 64-bit general purpose register values and all sixteen of the floating-point registers if they are available. The Register window also indicates **AMODE64**. The Dump and Watch windows will accept 64-bit pointer expressions. Note that the Dump window displays only the bottom 8 bytes of the address when **Rel:** is set to *N* (no). It is best to set **Rel:** to *Y* (yes) when dumping memory above the bar.

You can use the cast operator to convert from a 31-bit pointer type to a 64-bit pointer type using the **__huge** type-name modifier. Similarly, you can convert from a 64-bit pointer type to a 31-bit pointer type using the **__near** type-name modifier.

See "64-Bit Support" on page 5 and "64-Bit Support" on page 43 for more information on the 64-bit support that has been added to SAS/C for Release 7.50.

# Release 7.50 Updates to the SAS/C Debugger User's Guide

## Updates to the storage Command

In Chapter 14, "Command Directory," make the following changes to the description of the **storage** command.

- Change all of the instances of message LSCD612 to the following:

```
LSCD458  WARNING: The storage command has found corruptions in a header
for a storage block in either the stack, heap, or free heap, as indicated
by messages which follow.
```

- Change all of the instances of message LSCD622 to the following:

```
LSCD459  WARNING: The storage command has found corruptions in a storage
block in either the stack, heap, or free heap, as indicated by messages
which follow.
```

# CHAPTER 6 SAS/C Diagnostic Messages Changes in Release 7.50

## LSCC Compiler Messages

### 088

**WARNING: Argument type incorrect. Expecting arg-type, found arg-type.**
**ERROR: Argument type incorrect. Expecting arg-type, found arg-type.**

#### Explanation

Although a prototype is in scope for this function, the type of the argument does not match the type in the prototype. The argument is converted to the correct type. This message does not appear if only a simple widening conversion, such as **`char`** to **`int`**, is needed.

### 204

**WARNING: Item is wrong size for conversion to pointer.**

### EXPLANATION

An integer was converted to a pointer type whose size did not match the pointer type. For examplee, a **`char`**, **`short`** or **`long long`** was converted to a **`__near`** pointer. If the integer is a **`char`** or **`short`**, the result is unlikely to address valid data. If the integer is longer than the pointer, the results of the conversion will be correct only if all the extra bits are zeroes.

---

## 240

**ERROR: _stllregs is not supported by the 32-bit code generator.**

### Explanation

The **`_stllregs`** function can only be used in a source file that is compiled with the **`ARCHLEV(D)`** or the **`HUGEPTRS`** option.

---

## 241

**ERROR: Attempt to load 8 byte value into a 4 byte register.**

### Explanation

An attempt was made to use the **`_ldregs`** function to load an 8 byte integer (signed or unsigned **`long long`**) into a register, and neither the **`ARCHLEVEL(D)`** option nor the **`HUGEPTRS`** option was specified.

---

## 242

**ERROR: __label is not defined: [decimal] ([hex])**

### Explanation

A label that was referenced as the target of a call to one of the built-in branch functions was never defined.

---

## 243

**WARNING: Dubious branch between machine code blocks.**

### Explanation

One of the built-in branch functions was used to transfer control between one inline machine code block and another. It is possible that the generated code will fail to preserve one or more registers that is used by the target code sequence. In this case, the results could be incorrect.

### 244

**ERROR: Incorrect number of arguments for branch operation code.**

### Explanation

A call to one of the built-in branch instructions specifies an incorrect number of arguments. **`__builtin_branch`** has three arguments for most operations, but four for BCTG, BXHG or BXLEG.

### 245

**ERROR: _stregs pointer type is not compatible with the corresponding register.**

### Explanation

The _stregs builtin function attempted to store

- a general register by using a pointer to a floating-point type
- a floating point register by using a pointer to a non-floating-point type
- any register by using a pointer to an object of the wrong size (for example, attempting to store a general purpose register by using a pointer to a 16-byte structure).

## LSCD Messages

### 458

**WARNING: The storage command has found corruptions in a header for a storage block in either the stack, heap, or free heap, as indicated by messages which follow.**

### Explanation

The storage command has found corruption in a header for a storage block on the stack, heap, or free heap. A dump of the corrupted storage follows. See the **`storage`** command in the *SAS/C Debugger User's Guide and Reference* for an explanation of the message content.

## 459

**WARNING: The storage command has found corruptions in a storage block in either the stack, heap, or free heap, as indicated by messages which follow.**

### Explanation

The storage command has found corruption in a storage block in the stack, heap, or free heap. A dump of the corrupted storage follows. See the **`storage`** command in the *SAS/C Debugger User's Guide and Reference* for an explanation of the message content.

## LSCL COOL Messages

## 1013

**ERROR: There is a duplicated definition of "symbol" in input object files "file-1" and "file-1."**

### Explanation

Both of the named input object files or CSECTs contain a definition of the external symbol **`symbol`**. This is usually caused by defining an external symbol in more than one compilation in a program that is not re-entrant. Only one compilation in a program should define an external symbol.

### Action

Correct the problem and re-run COOL.

## 1017

**WARNING: The reference to "symbol" in "file-name" is unresolved.**

### Explanation

COOL completed autocall processing but did not find a definition for the external reference symbol found in file *file-name*. This does not necessarily indicate a problem if the definition will be supplied during subsequent linking, or if the reference is never used when the program executes. Otherwise, the reference may be misspelled, or the object file that contains the definition was omitted from the COOL input files.

## 1049

**ERROR: The \"%\" option must specify a length >= \"%\".**

### Explanation

The length that is given to this listing option is less than the minimum required length. To correct this error, give the option a large enough length or omit the option.

## 1050

**ERROR: The reference to extended name \"%\" in \"%\" is unresolved.**

### Explanation

COOL completed autocall processing but did not find a definition for the extended name external reference symbol found in file *file-name*. The reference might be misspelled, or the object file that contains the definition might have been omitted from the COOL input files.

## 1051

**ERROR: Cannot fork a new process for user exit: %.**

**Valid in:** UNIX environment only

### Explanation

COOL attempted to create a new process to execute the user exit, but the attempt failed. Your machine might be too busy or you might have reached the limit for the maximum number of processes that can run on your login.

## 1052

**ERROR: Cannot open pipes for communication with user exit: %.**

**Valid in:** UNIX environment only

### Explanation

COOL attempted to create the pipes necessary for communication with the user exit, but the attempt failed. You might have reached the limit for the maximum number of open files for your login, or there might be too many open files on your machine.

## 1053

**ERROR: User exit may only return status 4 on first invocation.**

### Explanation

A user exit that does not want to be called can return a return code of 4 the first time it is called. In all other cases a return code of 4 from the user exit is an error. This is a programming error in your user exit.

## 1054

**ERROR: Empty line from user exit.**

### Explanation

The user exit returned a blank line instead of a properly formed response to an extended name. This is a programming error in your user exit.

## 1055

**ERROR: The output from the user exit process is invalid: \"%\".**

### Explanation

The user exit returned a line with data in it in response to an extended name, but the line is improperly formed. This is a programming error in your user exit.

## 1056

**ERROR: User exit reports failure with exit code %.**

### Explanation

The user exit has exited but with a nonzero return code.

## 1057

**ERROR: Error communicating with the user exit process.**

### Explanation

An unknown error occurred while COOL was communicating with the user exit.

## 1058

**ERROR: User exit was killed by signal %.**

### Explanation

The user exit did not exit normally and was instead terminated by a signal. This may be due to a programming error in the user exit.

## 1059

**ERROR: User exit returned ID out of range: %.**

### Explanation

A user exit can only return a symbol ID between 0 and 999999.

## 1060

**ERROR: -Xt option must be followed by the path to the User Exit.**

### Explanation

When specifying the use of a user exit, the **-Xt** option must be given a valid path to the desired user exit.

## 1061

**ERROR: Unable to do dynamic load of User Exit: %.**

### Explanation

COOL attempted to load the user exit, but failed. On Windows this can also mean the loaded DLL did not have a **clkexit** function present. Additional status information is available only on Windows.

## 1062

**ERROR: Internal error in file % line %.**

### Explanation

COOL has experienced a fatal error. Call SAS/C technical support.

## 1063

**NOTE: Treating warnings as errors.**

### Explanation

A warning condition has occurred and the **-we** option has been given. Warnings are treated as severely as errors.

## 1064

**Output object is empty.**

### Explanation

After COOL has finished processing any and all input objects, no object was emitted. This may caused by a problem with your input.

## 1065

**Library "libraryname" not found.**

### Explanation

You specified an archive library for COOL to find objects in, but that archive could not be located. Make sure the directory containing the archive is in the search path.

## Updated LSCT C++ Translator Messages

Replace the following messages in Chapter 8, "LSCT C++ Translator Messages," with the text provided.

## 355

**ERROR: Value of an undefined class cannot be used.**

### Explanation

An expression with undefined class type was used by `value`. For example the expression might be an argument passed by value to a function with a formal of undefined class type, it might be a call to a function returning an undefined class type, or it might be an operand to an operator. This error could be resolved by providing a definition of the class prior to the use.

## 371

**ERROR: Cannot dereference pointer to undefined class.**

### Explanation

A class must be defined before a pointer to an object of its type can be dereferenced with the **->** operator.

---

## 382

**ERROR: Missing array size expression.**

### Explanation

One or more dimensions of the given array have not been specified.

---

## 415

**ERROR: Functions cannot return undefined classes.**

### Explanation

If a function is intended to return a class, that class must be defined before the function definition.

---

## 430

**ERROR: Conversion function 'operator type' CONSTRAINT.**

### Explanation

A declaration for a conversion function was found but it violates one of the constraints on conversions. Declarations of conversion functions do not specify the return type in the usual way. The return type is part of the name of the function. Also, conversion functions cannot take arguments.

---

## 431

**ERROR: Constructor function 'class-name' CONSTRAINT.**

### Explanation

A declaration for a constructor function was found but it violates one of the contraints on constructors. Perhaps you have specified a return type for a constructor, or the name used in the declaration of the constructor is not the same as the name of the class. Declarations of constructors cannot specify a return type, not even void.

---

## 460

**ERROR: Constructor function 'class-name' CONSTRAINT.**

### Explanation

A declaration for a constructor function was found but it violates one of the contraints on constructors. Perhaps you have specified a return type for a constructor, or the name used in the declaration of the constructor is not the same as the name of the class. Declarations of constructors cannot specify a return type, not even void.

---

## 504

**WARNING: Constant expression contains a division by zero (0).**
**ERROR: Constant expression contains a division by zero (0).**

### Explanation

Constant expressions cannot contain division by zero. This is an error if a compile-time constant was expected.

---

## 513

**ERROR: Overriding virtual function 'function-name' has different return type.**

### Explanation

An overriding virtual function cannot change the return type unless the new return type is a covariant class pointer or class reference. See section 10.3 in the C++ standard for more details.

## 517

**ERROR: Function declared '__builtin' after first use.**

### Explanation

The function must be declared **`__builtin`** before it is used.

## 522

**ERROR: Keyword 'keyword-name' can only be used on functions.**

### Explanation

A keyword that can be used only in function declarations has been used in a declaration of some other type. For example, the **`virtual`** and **`inline`** keywords can be used only in function declarations.

## 525

**WARNING: Function declared 'inline' after first use.**

### Explanation

A function was used prior to its declaration as an inline function. The translator might not be able to inline the previous uses.

## 595

**ERROR: Floating point value out of range.**

### Explanation

The value of a floating point constant or an operation on floating point constants is either too large or too small to be represented. Frequently this is caused by a multiplication or division of two floating point constants. This message is also generated when an IEEE floating point constant representing NaN (not a number) is used in a constant expression.

## 599

**ERROR: The function declaration 'function-name' requires an explicit parameter list.**

### Explanation

This error can be diagnosed when a function is defined using a function typedef name. The named function was defined with just a function **typedef** name. This is erroneous because the parameter list is not explicitly stated in the function definition. For example, the following would produce this message:

```
typedef int F(int);
F foo
    {
    return 10;
    }
```

This error can also be diagnosed for a member declaration in a specialization of a class template when the declaration has function type but the original template declaration is not a function declaration. See the C++ standard, 14.3.1 paragraph 3.

## 628

**WARNING: Template function not deducible.**

### Explanation

Not all template argument types and values can be determined from the function parameter types. References to this template function in calls must explicitly specify the non-deducible template arguments. This message is diagnosed with the **STRICT** option.

## 631

**ERROR: Scope for 'name' does not match a class template.**

### Explanation

The scope for the name being declared in a template must match a class template or partial specialization. For example:

```
template <class T, class U> class C {
   class Nested { ... };
   ...
};                                          // primary template for C
template <class T> class <T, int> { ... }; // partial specialization

template <class X, class Y> int C<X, Y>::i = 1; // OK
template <class X, class Y> int C<X, Y>::Nested::i = 1; // OK
template <class X, class Y> int C<Y, X>::j = 0; // Error
template <class X, class Y> int C<Y, int>::k = 7; // Error
template <class X> int C<X, int>::j = 2; // OK
template <class X> int C<X, double>::k = 3; // Error
```

In other words, to define a member of the primary class template, the template parameters in effect must match the parameters from the template class declaration and the the argument list for the template scope must use the template parameters in the original order. To define a member of a partial specialization the template parameters in effect must match the parameters from the class partial specialization declaration and the the argument list for the template scope must match the class argument list from the partial specialization,

## 643

**ERROR: Invalid explicit template specialization.**

### Explanation

An explicit template specialization declaration, which begins with **`template <>`**, must refer to a specialization of

- □ a previously declared function template
- □ a previously declared class template
- □ a member function or static data member of a specialization of class template (or a class nested in such a specialization) where the enclosing specialization is not explicitly specialized.

## 644

**ERROR: Template function was explicitly specialized after its first use.**

### Explanation

An explicit template specialization refers to a function or function template that has been previously used. The explicit specialization should be moved before the first use.

## 646

**ERROR: Template class was specialized after it was implicitly instantiated.**

### Explanation

An explicit specialization of an instance of a template class or an explicit specialization of the member class template for the instance was provided after a use of the instance which would have caused it to be implicitly generated from its template. The explicit specialization should be moved before the first use of the specialized class that would cause the class to be implicitly instantiated.

## 662

**ERROR: Name 'name' does not resolve to a ( type | scope | class template )name.**

### Explanation

The template definition for a specialization used a template formal dependent name in a context where a scope, type, or class template name was expected. However the corresponding name in the specialization was not found or did not have the correct type. For example:

```
template <class T>
class C { public: int Z; };

template <class T>
void f( typename C<U>::Z* p );

void testit()
{
   f<int>( 0 ); // Error, C<int>::Z is not a type
}
```

This error may also be detected in the template definition.

## 668

**ERROR: Default arguments are not allowed on template redeclaration 'template-name'.**

### Explanation

Function templates must declare all their default arguments on their first declaration.

---

## 670

**WARNING: Missing type specifier.**

### Explanation

The C++ standard requires that declarations (except for constructors, destructors, and type conversion operators) include a type specifier other than **`const`** or **`volatile`**. The **`implicit int`** rule of C is considered obsolete. Also the **`__binfmt`** and **`__hexfmt`** keywords must be used in combination with the **`float`** or **`double`** keywords.

---

## 696

**ERROR: Default template arguments allowed only for unspecialized class templates.**

### Explanation

Default template arguments are only allowed on the declaration or definition of a primary class template, not on function templates, template member definitions, or class template partial specializations.

---

## 705

**ERROR: 'declaration-name' not previously declared.**

### Explanation

A declaration with a scoped declarator was found. Such a declaration must refer to a previous declaration, but no such declaration was found. The message may also be issued if all the previous declarations where friend declarations and the **`NOFRIENDINJECT`** option is active.

---

## 738

**ERROR: Missing or ambiguous delete operator 'operator-name'.**

### Explanation

A **`new`** expression was encountered and there were either zero or multiple corresponding delete operators found for handling cleanup in cases where the new object initialization throws an exception. **`operator delete`** will not be called to release the memory for the object during exception handling. This message indicates a potential memory leak. Note that the **`operator delete`** name is looked up by the same method used for **`delete`** expressions, so the name may be found in a different scope than the **`operator new`** name. This message is diagnosed with the **`STRICT`** option.

## Deleted LSCT C++ Translator Messages

Delete the following messages from Chapter 8, "LSCT C++ Translator Messages."

### 361

**ERROR: Cannot take the address of a member of virtual base class.**

### 441

**ERROR: Static members 'member-name' of a local class may not be initialized.**

### 461

**ERROR: Destructor function 'destructor-name' not correctly declared.**

### 490

**ERROR: Missing function name in function declaration.**

## 624

**ERROR: Cannot instantiate incomplete template class ‘template-class’.**

## 625

**ERROR: Recursive template instantiation of ‘template-class’.**

## 627

**ERROR: Template instantiation failed for ‘template-name’.**

## 636

**ERROR: Multiple definitions for ‘symbol’.**

## 657

**ERROR: Elaborated name depends on a template parameter.**

## New LSCT C++ Translator Messages

Add the following messages to Chapter 8, “LSCT C++ Translator Messages.”

## 135

**ERROR: Cannot redefine macro ‘macro-name’.**

### Explanation

A **#define** or **#undef** preprocessor directive specified one of the special predefined symbols such as **__FILE__**. These symbols may not be changed directly in source code.

## 189

**WARNING: Archlevel reset to 'option-name' because of the 'option-name' option.**

### Explanation

An **ARCHLEVEL** option was specified that was too low to support another requested option, such as **BFP**. The **ARCHLEVEL** option is reset to the specified value.

## 746

**WARNING: Friend declaration does not declare a template specialization.**

### Explanation

A friend declaration in a template class uses the names of template type formals, but the declaration designates a non-template function. When used with older compilers, such a construct was generally used to refer to a specialization of a template function. However the C++ standard treats the construct as a declaring a non-template function unless the function declarator has explicit template arguments or has an explicit scope specifier. With an explicit scope specifier, a template specialization is found if there was not a previous non-template declaration. For example:

```
template <class T> class C;
template <class T> C<T> operator+( const C<T>&, const C<T>& );
template <class T> C <T> operator-( const C<T>&, const C<T>& );
template <class T> class C {
    // non-template declaration, the warning is issued
    friend C<T> operator+( const C<T>&, const C<T>& );
    // scoped declaration finds a template specialization
    friend C<T> (::operator+)( const C<T>&, const C<T>& );
    // template specialization, using explicit template arguments
    friend C<T> operator-<>( const C<T>&, const C<T>& );
  private:
    // . . .
};
```

The warning is not issued if the friend is defined in the template class body. With the **NOTMPLFUNC** option, the old interpretation of the first friend declaration is used and the warning will not be issued.

## 747

**ERROR: Friend declaration may not declare a partial specialization.**

### Explanation

A friend declaration may not declare a partial specialization of a class template.

## 748

**ERROR: Primary class template must be declared first.**

### Explanation

A primary class template must be declared before any partial specializations of the template.

## 749

**ERROR: Partial specialization matches original class template.**

### Explanation

The arguments for class template partial specialization may not match the arguments for the template. The original class template implicitly has arguments corresponding to the order of its parameter list. For example:

```
template <typename T, typename U>
class C;                 // implicitly C<T, U>
template <typename T, typename U>
class C<T*, U>;          // partial specialization, OK
template <typename T, typename U>
class C<T, U>;           // error, same as primary template
```

## 750

**ERROR: Complex partial specialization argument ‘argument-name’ depends on a template parameter.**

### Explanation

A nontype argument expression for a class template partial specialization must either not use template parameters unless it is the identifier of a template parameter. See the C++ standard, Section 14.5.4, paragraph 9.

## 751

**ERROR: Template formal type 'type-name' for specialized argument 'argument-name' depends on partial specialization parameters.**

### Explanation

The type of the class template formal (of the original template) corresponding to a nontype argument of a class template partial specialization must be determined at the time the partial specialization is declared unless the argument is the name of a template parameter. Otherwise the formal type cannot depend on specialization parameters. See the C++ standard, Section 14.5.4 , paragraph 9.

## 752

**ERROR: Ambiguous partial specialization for 'class-name'.**

### Explanation

The class template specialization corresponds to multiple partial specializations and the partial ordering rules can not identify the best specialization.

## 753

**WARNING: Partial specialization is not deducible.**

### Explanation

No specializations can be generated using the class template partial specialization because not all of the partial specialization template formals are used in a way in the argument list that allows them to be deduced. For example:

```
template <typename T, typename U>
class C;
```

```
template <typename T, typename U>
class C <T, int>          // does not use formal U
{ ...
};
```

Note that template formals also cannot be deduced when they are used inside expressions or as part of the scope of a qualified type name.

## 754

**ERROR: Default arguments may not be declared for specializations of template functions.**

### Explanation

Friend declarations of specializations of function templates, explicit specialization declarations, and explicit instantiation declarations may not declare default arguments. The default arguments used for calls to such functions come from the template declaration.

## 755

**ERROR: Declaration is not a template specialization.**

### Explanation

An explicit specialization declaration was found for a function that was not an instance of a function template or member of an unspecialized class generated from a template, or a function or class template that was not a member of an unspecialized class generated from a template. Note that the explicit specialization syntax is not used for members of explicitly specialized classes. See the C++ standard, section 14.7.3, paragraphs 1 and 5.

## 756

**ERROR: Static data members may not be declared inside an unnamed class.**

### Explanation

Static data members may not be declared in an unnamed class or a class nested in such a class. See the C++ standard, section 9.4.2, paragraph 5.

## 757

**ERROR: Class templates may not declare template members as friends.**

### Explanation

Only classes that are not templates may declare a member of a template class as a friend. See the C++ standard, section 14.5.3, paragraph 6.

## 758

**ERROR: Invalid using declaration.**

### Explanation

The using declaration specified an invalid name, such as a destructor name.

## 759

**ERROR: Type of destructor name does not match its scope: 'type_name1::~type_name2'.**

### Explanation

When a destructor name is specified with a qualified name (for example, **`type_name1::~type_name2`**) the type of the name specified by the scope must match the type specified by the name following the ~.

## 760

**WARNING: Non-placement operator delete is hidden in <class-name>.**
**ERROR: Non-placement operator delete is hidden in <class-name>.**

### Explanation

The deallocation function selected by a delete expression or a virtual destructor definition was a placement delete. A placement delete may not be used in such contexts. See section 12.5, paragraph 4 in the C++ standard. Note that template deallocation functions are always considered to be placement declarations and a member placement delete will hide the global default non-placement delete declaration.

This message is issued when a placement delete class member has been declared, no corresponding non-placement delete member has been declared, and a delete expression using the class is found or a virtual destructor for the class is defined. This message is a warning for virtual destructor definitions but may become an error in the future. Attempting to delete via a virtual destructor which generates the warning could result in no deallocation function being called. The message can be corrected by declaring a non-placement member operator delete.

## 761

**ERROR: The AT option is required to use the '@' operator.**

### Explanation

The **@** operator was found in an expression but the **AT** compiler option was not specified. The **@** operator is a SAS/C extension that supports call by reference for interlanguage communication

## 762

**ERROR: Use of 'feature-name' requires the archlevel(option-name) option.**

### Explanation

The compiler found a use of a feature, such as binary floating point, that is not supported by the architecural level determined from the compiler options. The code must be changed or a higher **ARCHLEVEL** option must be specified.

## 763

**ERROR: Mixed __binfmt and __hexfmt floating point.**

### Explanation

A binary operator was found where both operands have floating point types but different modes (hexadecimal versus binary). The mode for the operation cannot be determined. The operands need to be adjusted so that they have the same format.

## 764

**ERROR: Invalid use of cv-qualified function type.**

### Explanation

A **`const`** or **`volatile`** qualified function type can be used only as the top level type of a **`typedef`** declaration or nonstatic member declation or as the target of a member pointer declarator. Also **`const`** and **`volatile`** qualifiers cannot be applied to a **`typedef`** name of function type.

## 765

**ERROR: Ambiguous base class initializer.**

### Explanation

A base class designated in the member initializer list for a constructor is both a direct non-virtual base and an indirect virtual base. Such a base cannot be explicitly initialized.

## 766

**ERROR: Operation '*' is not supported on a far pointer without the ARMODE option.**

### Explanation

An operation, such as unary `'*'`, was applied to a far or huge pointer or reference. This is not supported either due to the current option settings or to the limitations of the current release.

## 767

**WARNING: Expression requires ARMODE.**

### Explanation

Generally this means that an expression references a default argument that used operations requiring **`ARMODE`** but the current function was not defined **`ARMODE`**. Note that default arguments are evaluated in the context of the function that uses them. The function current being defined will be marked **ARMODE** so that correct code can be generated.

## 768

**ERROR: Multiple member initializers for a union.**

### Explanation

At most one member of a union or anonymous union can be initialized in the initializer list of a constructor function.

## 769

**ERROR: Object size limit exceeded.**

### Explanation

The object size exceeded the limit for its storage class. For objects allocated by a new expression, the limit is 2**32-1 bytes (2**64-1 bytes with the **HUGEPTRS** option).

## 770

**WARNING: Bitfield width larger than type.**

### Explanation

A bitfield declaration specified a field width larger than required by the specified type. For example a **char** bitfield was specified with more than 8 bits. The extra bits are not used to store the value and are treated as padding. This warning is only issued with the **NOBITFIELD** option. Otherwise the SAS/C bitfield rules are applied and all bits are used for the value.

# LSCX Run-time Messages

## Updates to Library Message Processing

In Chapter 9, "LSCX Run-time Messages," in the section titled, "Library Message Processing," replace the last paragraph and the list that follows it with the following text.

You can control the generation and verbosity of library diagnostic messages using any of the following:

□ the **`quiet`** function to suppress these messages, as described in *SAS/C Library Reference, Volume 1*

□ the **`=warning`** run-time option to force library diagnostics to be displayed, even if suppressed by **`quiet`**

□ the **`=btrace`** run-time option to get a list of the active functions (a *traceback*) at the time a diagnostic is generated

□ the **`=rsntrace`** run-time option to print information about recent failures in operating system calls with library messages

□ the **`wmi_add`** function to define one or more exit routines to control the printing of diagnostic messages.

## Updates to Message Types

In Chapter 9, "LSCX Run-time Messages," make the following changes in the section titled, "Message Types:"

1 Replace the example of the form of the library messages with the following example:

```
LSCX[num]  **** [severity] **** ERRNO = [errno value]
   [info added by message exit]
Generated in function-name called from line line-number of function-name,
   offset hex-value
[C++/Extended] name: full-name
message-text
Last failing system call: [name], return code [code], reason code [code],
   info [code]
[info added by message exit]
Interrupted while: context
```

2 Add the following text after the list of severities.

Note that some of the text above will not appear for every message. The **`C++/ extended name`** line appears only if the error occurred in a C++ function or one with an extended name. The **`Last failing system call`** line appears only if the **`=rsntrace`** run-time option is in effect, and a system call has failed since the last run-time diagnostic message. **`info added by message exit`** indicates optional text which could be added by a library message exit (defined with the **`wmi_add`** function).

Replace Message **`025`** with the following message

## 025

**ERROR: Program terminated due to [name] signal.**

**ABEND code:** 1225

### Explanation

A signal occurred for which the default action is termination, and no signal handler was defined. These signals include SIGALRM, SIGIUCV, SIGOPER and most of the UNIX Systems Services signals.

### Action

If abnormal termination is not desired, modify the program to define a handler for the signal.

## 153

**WARNING: SAS/C [function] format string [string] is not supported on this system.**

**Errno value:** ESYS

### Explanation

A `printf` or `scanf` family function was called with a SAS/C type specifier that is not supported by the system environment.

### Action

Change the type specifier.

## 154

**WARNING: Memory allocation failed, floating conversion not performed.**

**Errno value:** ENOMEM

### Explanation

scanf was unable to allocate memory to perform an accurate conversion of an input field to floating-point representation because the input field contains too many digits. The conversion returns a NaN for binary floating-point, or a very small negative number for hexadecimal floating-point.

### Action

Either increase the region size, or specify fewer digits in the floating-point input data.

## 155

**WARNING: The conversion modifier ~l (hugeptr) is not available in your environment.**

**Errno value:** EARG

### Explanation

A **`printf`** family function was called with a conversion specifier of **`~l`** or **`~L`**, indicating a huge pointer. The environment does not support huge pointers. Huge pointers are not supported for callers linked with AMODE24.

### Action

Re-link the application AMODE31 or remove the huge pointer specification.

## 208

**WARNING: [error] occured during computation of [expression].**

**Errno value:** ERANGE

### Explanation

Floating-point overflow or underflow (as indicated in the message) occurred during computation of a mathematical function. The **`errno`** variable is set to **`ERANGE`** and an appropriate value (+/-**`HUGE_VAL`** for overflow or 0.0 for underflow) is returned.

### ACTION

If possible, correct the program or the data. The message can be suppressed using the **`_matherr`** function, as described in the *SAS/C Library Reference, Volume 1.*

## 210

**WARNING: [error] occured during computation of [expression].**

**Errno value:** ERANGE

### Explanation

Floating-point overflow or underflow, as indicated in the message, occurred during computation of a mathematical function. **`errno`** is set to **`ERANGE`**, and appropriate flags are set in the floating-point environment. The function returns an appropriate value, determined by the specific error which occurred and the current rounding mode. (For the default rounding mode, plus or minus infinity is returned for overflow, and a denormalized result or 0 is returned for underflow.)

### Action

If possible, correct the program or the data. The message can be suppressed using the **`_matherb`** function, as described in the *SAS/C Library Reference, Volume 1*.

## 211

**WARNING: [error] occured during comutation of [expression].**

**Errno value:** ERANGE

### Explanation

Floating-point overflow or underflow, as indicated in the message, occurred during computation of a mathematical function. **`errno`** is set to **`ERANGE`**, and appropriate flags are set in the floating-point environment. The function returns an appropriate value, determined by the specific error which occurred and the current rounding mode. For the default rounding mode, plus or minus infinity is returned for overflow, and a denormalized result or 0 is returned for underflow.

### Action

If possible, correct the program or the data. The message can be suppressed using the **`_matherb`** function, as described in the *SAS/C Library Reference, Volume 1*.

## 212

**WARNING: [value] is a singularity of the [name] function. The result is infinite.**

**Errno value:** ERANGE

### Explanation

The indicated value is a singularity of the indicated function. That is, the function value approaches infinity as the argument approaches the singularity. **`errno`** is set to **`ERANGE`**, and the **`divide by 0`** flag is set in the floating-point environment. The function returns an infinite value with an appropriate sign.

### Action

If possible, correct the program or the data. The message can be suppressed using the **`_matherb`** function, as described in the *SAS/C Library Reference, Volume 1*.

## 213

**WARNING: [value] is not a valid argument to the [name] function.**

**Errno value:** EDOM

### Explanation

The indicated function is mathematically undefined for the given argument. **`errno`** is set to **`EDOM`**, and the **`invalid`** flag is set in the floating-point environment. The function returns a NaN.

### Action

If possible, correct the program or the data. The message can be suppressed using the **`_matherb`** function, as described in the *SAS/C Library Reference, Volume 1*.

## 214

**WARNING: [expression] is not mathematically meaningful.**

**Errno value:** EDOM

### Explanation

The indicated function is mathematically undefined for the given arguments. **`errno`** is set to **`EDOM`**, and the **`invalid`** flag is set in the floating-point environment. The function returns a NaN.

### Action

If possible, correct the program or the data. The message can be suppressed using the **`_matherb`** function, as described in the *SAS/C Library Reference, Volume 1*.

## 215

**NOTE: [Total | Partial] loss of significance occurred during evaluation of [expression].**

**Errno value:** unchanged

### Explanation

The function result is less accurate than the arguments. If the message indicates total loss of significance, the result may be completely inaccurate. If it indicates partial loss of significance, the result is approximately correct, but considerably less precise than its input arguments.

### ACTION

If possible, correct the program or the data. Output of this message may indicate serious problems with the correctness of program results, which should be evaluated carefully. The message can be suppressed using the **`_matherb`** function, as described in the *SAS/C Library Reference, Volume 1.*

## 216

**WARNING: [error] occurred during computation of [expression].**

**Errno value:** ERANGE

### Explanation

Floating-point overflow or underflow, as indicated in the message, occurred during computation of a mathematical function. **`errno`** is set to **`ERANGE`**, and appropriate flags are set in the floating-point environment. The function returns an appropriate value, determined by the specific error which occurred and the current rounding mode. For the default rounding mode, plus or minus infinity is returned for overflow, and a denormalized result or 0 is returned for underflow.

### Action

If possible, correct the program or the data. The message can be suppressed using the **`_matherb`** function, as described in the *SAS/C Library Reference, Volume 1.*

## 217

**NOTE: [Total/Partial] loss of significance occurred during evaluation of [expression].**

**Errno value:** unchanged

### Explanation

The function result is less accurate than the arguments. If the message indicates total loss of significance, the result may be completely inaccurate. If it indicates partial loss of significance, the result is approximately correct, but considerably less precise than its input arguments.

### ACTION

If possible, correct the program or the data. Output of this message may indicate serious problems with the correctness of program results, which should be evaluated carefully. The message can be suppressed using the **`_matherb`** function, as described in the *SAS/C Library Reference, Volume 1*.

## 218

**WARNING: [error] occurred during computation of [expression].**

**Errno value:** ERANGE

### Explanation

Floating-point overflow or underflow, as indicated in the message, occurred during computation of a mathematical function. **`errno`** is set to **`ERANGE`**, and appropriate flags are set in the floating-point environment. The function returns an appropriate value, determined by the specific error which occurred and the current rounding mode. For the default rounding mode, plus or minus infinity is returned for overflow, and a denormalized result or 0 is returned for underflow.

### Action

If possible, correct the program or the data. The message can be suppressed using the **`_matherb`** function, as described in the *SAS/C Library Reference, Volume 1*.

## 219

**WARNING: ([value],[value]) is a singularity of the [name] function. Result is infinite.**

**Errno value:** ERANGE

### Explanation

The indicated pair of values is a singularity of the indicated function. That is, the function approaches infinity as the arguments approach the singularity. **`errno`** is set to **`ERANGE`**, and the **`divide by 0`** flag is set in the floating-point environment. The function returns an infinite value with an appropriate sign.

## 220

**WARNING: [value] is not a valid argument to the [function] function.**

**Errno value:** EARG

### Explanation

A string argument **`value`** containing an invalid floating-point number was passed to the **`function`** function.

## 221

**WARNING: [error] occured during computation of [expression].**

**Errno value:** ERANGE

### Explanation

The function result is less accurate than the arguments. If the message indicates total loss of significance, the result may be completely inaccurate. If it indicates partial loss of significance, the result is approximately correct, but considerably less precise than its input arguments.

### ACTION

If possible, correct the program or the data. Output of this message may indicate serious problems with the correctness of program results, which should be evaluated carefully. The message can be suppressed using the **`_matherr`** function, as described in the *SAS/C Library Reference, Volume 1*.

## 231

**WARNING: [error] occurred during computation of [expression].**

**Errno value:** ERANGE

### Explanation

Floating-point overflow or underflow, as indicated in the message, occurred during computation of a mathematical function. **`errno`** is set to **`ERANGE`**, and appropriate flags are set in the floating-point environment. The function returns an appropriate value, determined by the specific error which occurred and the current rounding mode. For the default rounding mode, plus or minus infinity is returned for overflow, and a denormalized result or 0 is returned for underflow.

### Action

If possible, correct the program or the data. The message can be suppressed using the **`_matherb`** function, as described in the *SAS/C Library Reference, Volume 1*.

## 728

**ERROR: Alarm subtask has abended with code [code].**

**Errno value:** ESYS

### Explanation

During processing of the **`sleep`** or **`alarm`** functions the subtask unexpectedly abended with the indicated code.

### Action

Look up the ABEND code in the IBM documentation, *OS/390 V2R10.0 MVS System Codes* and take appropriate action.

## 968

**WARNING: [name] syscall failed: [text].**

**Errno value:** various

### Explanation

An error occurred in a UNIX System Services system call. The error was of a sort that does not ordinarily require a diagnostic message. This message is written only if the **`=warning`** runtime option is in effect, or if a diagnostic message exit directs it to be printed. The **`text`** portion of the message is a brief synopsis of the meaning of the errno value.

### Action

Ordinarily, none is required. However, in the event of mysterious problems with UNIX System Services facilities (especially TCP/IP), this information might clarify the nature of the problem.

## 969

**NOTE: querydub syscall returned [text].**

**Errno value** unchanged

### Explanation

The library called the UNIX System Services **`querydub`** system call to determine whether UNIX System Services could be used. While the system call did not fail, it returned an unusual return code which might represent an environmental problem. The "text" portion of the message is the symbolic name of the **querydub** return code, which can be looked up in the IBM publication *UNIX System Services Assembler Callable Services*. This message is written only if the **=warning** runtime option is in effect, or if a diagnostic message exit directs it to be printed.

## Action

Ordinarily, none is required. However, in the event of mysterious problems with UNIX Systems Services facilities (especially TCP/IP), this information might clarify the nature of the problem.

# Your Turn

If you have comments or suggestions about *SAS/C® 7.50: Changes and Enhancements*, please send them to us on a photocopy of this page, or send us electronic mail.

For comments about this book, please return the photocopy to

SAS Publishing
SAS Campus Drive
Cary, NC 27513
**email:** yourturn@sas.com

For suggestions about the software, please return the photocopy to

SAS Institute Inc.
Technical Support Division
SAS Campus Drive
Cary, NC 27513
**email:** suggest@sas.com